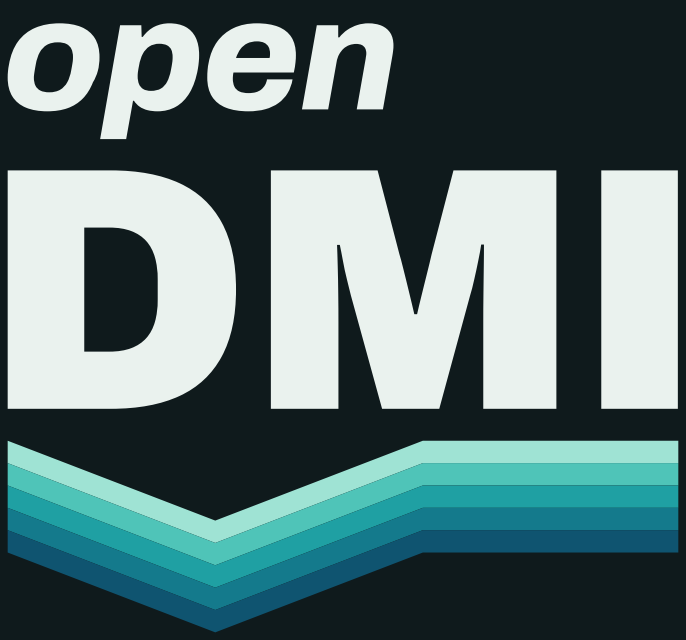


Reference manual

The OpenDMI contributors

Version 0.7.0, 2026-10-21

Copyright © 2025-2026, The OpenDMI contributors.
Free use of this software is granted under the terms of the BSD 3-Clause License.

Copyright © 2025-2026, The OpenDMI contributors

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

Part I: Introduction . . . 1
1. About OpenDMI . . . 2
1.1. Components . . . 2
1.2. License . . . 2
2. About this manual . . . 3
2.1. Organization . . . 3
2.2. Conventions . . . 3
3. DMI and SMBIOS . . . 4
3.1. Entry point . . . 4
3.2. Structures . . . 4
3.3. Types . . . 4
3.4. Handles . . . 5
3.5. Versions . . . 5
4. Features . . . 6
4.1. SMBIOS support . . . 6
4.2. Vendor-specific structures . . . 6
4.3. Platforms . . . 6
4.4. Command line tool . . . 7
4.5. Validation . . . 7
4.6. Library . . . 7
4.7. Other languages and services . . . 8
4.8. Quality . . . 8
5. Terminology . . . 9
6. References . . . 12
6.1. Specifications . . . 12
6.2. Other sources . . . 12
6.3. Related software . . . 12
Part II: Getting started . . . 13
7. Building from sources . . . 14
7.1. Prerequisites . . . 14
7.2. Configuring . . . 14
7.3. Building and testing . . . 15
7.4. Installing . . . 15
7.5. Building with CMake directly . . . 15
8. First steps . . . 17
8.1. Reading the data of the system . . . 17
8.2. Exporting the data . . . 17
8.3. Working with dumps . . . 17
8.4. Checking the data . . . 18
8.5. Structures the vendors define . . . 18
9. Using the library . . . 19
9.1. Finding the library . . . 19

9.2. Linking . . . 19
9.3. Reading the data . . . 20
9.4. Handling errors . . . 21
Part III: Command line tool . . . 22
Tool . . . 23
opendmi . . . 23
Commands . . . 27
dump . . . 27
entry . . . 28
explain . . . 29
export . . . 30
import . . . 33
lint . . . 34
list . . . 47
modules . . . 49
show . . . 51
types . . . 53
Modules . . . 56
Extension modules . . . 56
Part IV: C library . . . 62
`<opendmi/defs.h>` . . . 63
dmi_member . . . 63
`<opendmi/context.h>` . . . 65
dmi_add_extension . . . 65
dmi_close . . . 66
dmi_create . . . 67
dmi_destroy . . . 68
dmi_get_flags . . . 69
dmi_get_log_level . . . 71
dmi_get_logger . . . 72
dmi_get_platform . . . 72
dmi_get_registry . . . 74
dmi_is_open . . . 75
dmi_load . . . 75
dmi_log . . . 77
dmi_open . . . 78
dmi_save . . . 80
dmi_set_log_level . . . 82
dmi_set_logger . . . 83
dmi_set_platform . . . 84
`<opendmi/anonymize.h>` . . . 86
dmi_anonymize . . . 86
`<opendmi/types.h>` . . . 88
dmi_type . . . 88

dmi_type_id . . . 89
<opendmi/platform.h> . . . 91
dmi_platform_clone . . . 91
dmi_platform_create . . . 91
dmi_platform_destroy . . . 93
dmi_platform_in_generations . . . 93
dmi_platform_match . . . 95
dmi_platform_set_product . . . 96
<opendmi/vendor.h> . . . 98
dmi_vendor . . . 98
<opendmi/module.h> . . . 102
dmi_module_find . . . 102
dmi_module_register . . . 104
<opendmi/module/hpe.h> . . . 106
dmi_hpe_platform_detect . . . 106
<opendmi/error.h> . . . 108
dmi_error_clear . . . 108
dmi_error_get_first . . . 108
dmi_error_get_last . . . 109
dmi_error_message . . . 110
dmi_error_peek_first . . . 111
dmi_error_peek_last . . . 112
dmi_error_raise . . . 113
dmi_error_raise_ex . . . 114
<opendmi/trace.h> . . . 116
dmi_trace_argument_null . . . 116
<opendmi/log.h> . . . 119
dmi_log_level . . . 119
dmi_log_message . . . 120
dmi_log_message_va . . . 121
<opendmi/locale.h> . . . 123
dmi_get_locale . . . 123
dmi_locale_setup . . . 123
dmi_message_format . . . 124
dmi_resource_close . . . 126
dmi_resource_message . . . 127
dmi_resource_open . . . 128
dmi_resource_string . . . 130
dmi_set_locale . . . 131
<opendmi/entity.h> . . . 133
dmi_entity_buffer . . . 133
dmi_entity_context . . . 133
dmi_entity_create . . . 134
dmi_entity_data . . . 135

dmi_entity_decode . . . 136
dmi_entity_destroy . . . 137
dmi_entity_encode . . . 138
dmi_entity_handle . . . 139
dmi_entity_info . . . 140
dmi_entity_is_decoded . . . 141
dmi_entity_link . . . 142
dmi_entity_name . . . 143
dmi_entity_offset . . . 144
dmi_entity_property . . . 144
dmi_entity_string_ex . . . 145
dmi_entity_type . . . 146
dmi_entity_type_id . . . 147
dmi_spec_name . . . 148
dmi_type_find . . . 149
dmi_type_name . . . 150
dmi_type_spec . . . 151
`<opendmi/registry.h>` . . . 153
dmi_registry_create . . . 153
dmi_registry_iter_initialize . . . 154
dmi_registry_lookup . . . 156
dmi_registry_resolve . . . 158
dmi_registry_status . . . 161
`<opendmi/filter.h>` . . . 163
dmi_filter_add_type . . . 163
dmi_filter_create . . . 164
dmi_filter_match . . . 166
`<opendmi/attribute.h>` . . . 168
DMI_ATTRIBUTE . . . 168
DMI_ATTRIBUTE_VARIANT . . . 171
dmi_attribute_format . . . 172
dmi_attribute_get_int . . . 175
dmi_attribute_is_unspecified . . . 177
dmi_attribute_name . . . 179
dmi_attribute_resolve . . . 180
dmi_attributes_link . . . 181
dmi_attributes_walk . . . 183
`<opendmi/field.h>` . . . 188
DMI_FIELD . . . 188
DMI_FIELD_ARRAY . . . 190
DMI_FIELD_BITS . . . 192
DMI_FIELD_EXTENDED . . . 193
DMI_FIELD_GROUP . . . 194
DMI_FIELD_SPLIT . . . 196

dmi_field_decode_kilobytes . . . 197
dmi_field_get . . . 199
dmi_fields_decode . . . 201
dmi_fields_decode_into . . . 203
dmi_fields_encode . . . 205
dmi_fields_release . . . 207
`<opendmi/value.h>` . . . 209
dmi_bool_code . . . 209
dmi_unit . . . 209
`<opendmi/buffer.h>` . . . 214
dmi_buffer_assign . . . 214
dmi_buffer_at . . . 214
dmi_buffer_clear . . . 215
dmi_buffer_context . . . 216
dmi_buffer_create . . . 217
dmi_buffer_destroy . . . 218
dmi_buffer_is_empty . . . 218
dmi_buffer_read . . . 219
dmi_buffer_resize . . . 220
dmi_buffer_write . . . 221
`<opendmi/reader.h>` . . . 223
dmi_reader_get_bytes . . . 223
dmi_reader_get_bytes_at . . . 224
dmi_reader_has . . . 225
dmi_reader_initialize . . . 225
dmi_reader_is_done . . . 226
dmi_reader_mark . . . 227
dmi_reader_ref_bytes . . . 228
dmi_reader_remaining . . . 229
dmi_reader_reset . . . 229
dmi_reader_rewind . . . 230
dmi_reader_seek . . . 230
dmi_reader_skip . . . 231
dmi_reader_skip_ex . . . 232
dmi_reader_tell . . . 233
`<opendmi/writer.h>` . . . 234
dmi_writer_initialize . . . 234
dmi_writer_mark . . . 235
dmi_writer_put_bytes . . . 236
dmi_writer_put_bytes_at . . . 236
dmi_writer_reset . . . 237
dmi_writer_rewind . . . 238
dmi_writer_seek . . . 239
dmi_writer_skip . . . 240

dmi_writer_skip_ex . . . 240
dmi_writer_tell . . . 241
<opendmi/decoder.h> . . . 243
dmi_decoder_context . . . 243
dmi_decoder_entity . . . 243
dmi_decoder_get . . . 244
dmi_decoder_get_at . . . 245
dmi_decoder_get_bcd . . . 246
dmi_decoder_get_bcd_at . . . 246
dmi_decoder_get_binary . . . 247
dmi_decoder_get_bytes . . . 248
dmi_decoder_get_bytes_at . . . 249
dmi_decoder_get_string . . . 250
dmi_decoder_get_uuid . . . 251
dmi_decoder_get_uuid_at . . . 251
dmi_decoder_incomplete . . . 252
dmi_decoder_initialize . . . 253
dmi_decoder_reader . . . 254
dmi_decoder_remaining . . . 254
dmi_decoder_skip . . . 255
dmi_decoder_stop . . . 256
<opendmi/encoder.h> . . . 257
dmi_encoder_copy . . . 257
dmi_encoder_entity . . . 257
dmi_encoder_finalize . . . 258
dmi_encoder_finish . . . 259
dmi_encoder_initialize . . . 259
dmi_encoder_peek . . . 261
dmi_encoder_put . . . 262
dmi_encoder_put_at . . . 262
dmi_encoder_put_bcd . . . 263
dmi_encoder_put_bytes . . . 264
dmi_encoder_put_bytes_at . . . 265
dmi_encoder_put_string . . . 266
dmi_encoder_put_uuid . . . 267
dmi_encoder_remaining . . . 268
dmi_encoder_tell . . . 268
dmi_encoder_writer . . . 269
<opendmi/lint.h> . . . 270
dmi_lint . . . 270
dmi_lint_context . . . 273
dmi_lint_issue . . . 274
dmi_lint_rule_name . . . 276
dmi_lint_rules . . . 277

`<opendmi/backend/generic.h>` 280
dmi_generic_find_entry_addr 280
dmi_generic_parse_entry_addr 281
`<opendmi/utils.h>` 283
dmi_alloc 283
dmi_checksum_calc 284
dmi_file_load 285
dmi_ipow32 286
dmi_memory_load 287
`<opendmi/utils/base64.h>` 289
dmi_base64_decode 289
dmi_base64_encode 290
`<opendmi/utils/datetime.h>` 292
dmi_date_parse 292
`<opendmi/utils/endian.h>` 294
dmi_bswap 294
`<opendmi/utils/file.h>` 295
dmi_file_close 295
dmi_file_lock 295
dmi_file_read 296
dmi_file_seek 297
dmi_file_stat 298
dmi_file_tell 298
dmi_file_unlock 299
dmi_file_write 300
`<opendmi/utils/name.h>` 302
DMI_NAMES 302
dmi_name_lookup 303
`<opendmi/utils/string.h>` 306
dmi_asprintf 306
dmi_string_is_placeholder 307
dmi_string_set 308
dmi_string_tolower 310
`<opendmi/utils/uuid.h>` 312
dmi_uuid_decode 312
dmi_uuid_encode 313
`<opendmi/utils/vector.h>` 315
dmi_vector_initialize 315
`<opendmi/utils/version.h>` 319
dmi_version 319
dmi_version_format 320
dmi_version_format_ex 321
`<opendmi/entity/additional-info.h>` 323
dmi_additional_info 323

dmi_additional_info_entry . . . 324
<opendmi/entity/baseboard.h> . . . 326
dmi_baseboard . . . 326
dmi_baseboard_features . . . 328
dmi_baseboard_type . . . 329
<opendmi/entity/battery.h> . . . 332
dmi_battery . . . 332
dmi_battery_chemistry . . . 334
<opendmi/entity/bis-entry-point.h> . . . 337
dmi_bis_entry_point . . . 337
dmi_bis_real_mode_addr . . . 338
<opendmi/entity/cache.h> . . . 339
dmi_cache . . . 339
dmi_cache_assoc . . . 341
dmi_cache_location . . . 343
dmi_cache_mode . . . 344
dmi_cache_sram_type . . . 345
dmi_cache_type . . . 346
<opendmi/entity/chassis.h> . . . 348
dmi_chassis . . . 348
dmi_chassis_element . . . 350
dmi_chassis_security_status . . . 351
dmi_chassis_type . . . 353
dmi_rack_type . . . 357
<opendmi/entity/common.h> . . . 358
dmi_error_correct_type . . . 358
dmi_pci_addr . . . 359
dmi_pci_class . . . 360
dmi_pci_device_id . . . 361
dmi_pci_slot . . . 361
dmi_pci_vendor_id . . . 362
dmi_status . . . 362
<opendmi/entity/cooling-device.h> . . . 365
dmi_cooling_device . . . 365
dmi_cooling_device_type . . . 366
<opendmi/entity/firmware-inventory.h> . . . 369
dmi_firmware_ident . . . 369
dmi_firmware_ident_format . . . 369
dmi_firmware_inventory . . . 370
dmi_firmware_inventory_component . . . 373
dmi_firmware_inventory_chars . . . 374
dmi_firmware_inventory_state . . . 375
dmi_firmware_version . . . 376
dmi_firmware_version_format . . . 377

dmi_firmware_version_number . . . 379
<opendmi/entity/firmware-language.h> . . . 380
dmi_firmware_language . . . 380
dmi_firmware_language_flags . . . 381
<opendmi/entity/firmware.h> . . . 383
dmi_firmware . . . 383
dmi_firmware_features . . . 384
dmi_firmware_features_ex . . . 388
<opendmi/entity/group-assoc.h> . . . 390
dmi_group_assoc . . . 390
dmi_group_assoc_item . . . 391
<opendmi/entity/hardware-security.h> . . . 393
dmi_hardware_security . . . 393
dmi_hardware_security_status . . . 394
<opendmi/entity/ipmi-device.h> . . . 396
dmi_ipmi_addr_type . . . 396
dmi_ipmi_device . . . 397
dmi_ipmi_device_details . . . 399
dmi_ipmi_iface_type . . . 400
dmi_ipmi_intr_polarity . . . 402
dmi_ipmi_intr_trigger_mode . . . 403
dmi_ipmi_register_spacing . . . 404
<opendmi/entity/memory-array-addr.h> . . . 406
dmi_memory_array_addr . . . 406
<opendmi/entity/memory-array.h> . . . 408
dmi_memory_array . . . 408
dmi_memory_array_location . . . 409
dmi_memory_array_use . . . 411
<opendmi/entity/memory-channel.h> . . . 414
dmi_memory_channel . . . 414
dmi_memory_channel_device . . . 415
dmi_memory_channel_type . . . 416
<opendmi/entity/memory-controller.h> . . . 418
dmi_error_correct_caps . . . 418
dmi_error_detect_method . . . 419
dmi_memory_controller . . . 420
dmi_memory_interleave . . . 423
dmi_memory_module_speed . . . 424
dmi_memory_module_voltage . . . 425
<opendmi/entity/memory-device-addr.h> . . . 427
dmi_memory_device_addr . . . 427
<opendmi/entity/memory-device.h> . . . 430
dmi_memory_device . . . 430
dmi_memory_device_form_factor . . . 435

dmi_memory_device_tech ... 437
dmi_memory_device_type ... 439
dmi_memory_device_type_details ... 443
<opendmi/entity/memory-error-32.h> ... 445
dmi_memory_error_32 ... 445
<opendmi/entity/memory-error-64.h> ... 447
dmi_memory_error_64 ... 447
<opendmi/entity/memory-error.h> ... 449
dmi_memory_error_granularity ... 449
dmi_memory_error_operation ... 450
dmi_memory_error_type ... 451
<opendmi/entity/memory-module.h> ... 454
dmi_memory_module ... 454
dmi_memory_module_error ... 455
dmi_memory_module_size ... 456
dmi_memory_module_size_status ... 457
dmi_memory_module_type ... 459
<opendmi/entity/mgmt-controller.h> ... 461
dmi_mgmt_controller ... 461
dmi_mgmt_iface_type ... 462
dmi_mgmt_nhi ... 465
dmi_mgmt_nhi_char ... 466
dmi_mgmt_nhi_device_type ... 467
dmi_mgmt_nhi_format ... 468
dmi_mgmt_nhi_oem ... 469
dmi_mgmt_nhi_pci ... 470
dmi_mgmt_nhi_pci_v2 ... 471
dmi_mgmt_nhi_usb ... 473
dmi_mgmt_nhi_usb_v2 ... 473
dmi_mgmt_proto ... 475
dmi_mgmt_proto_record ... 476
dmi_mgmt_redfish_ip_assignment ... 477
dmi_mgmt_redfish_ip_format ... 478
dmi_mgmt_redfish_over_ip ... 479
<opendmi/entity/mgmt-device-component.h> ... 482
dmi_mgmt_device_component ... 482
<opendmi/entity/mgmt-device-threshold.h> ... 484
dmi_mgmt_device_threshold ... 484
<opendmi/entity/mgmt-device.h> ... 486
dmi_mgmt_device ... 486
dmi_mgmt_device_addr_type ... 487
dmi_mgmt_device_type ... 488
<opendmi/entity/oem-strings.h> ... 491
dmi_oem_strings ... 491

`<opendmi/entity/onboard-device-ex.h>` 493
dmi_onboard_device_ex 493
`<opendmi/entity/onboard-device.h>` 495
dmi_onboard_device 495
dmi_onboard_device_instance 496
dmi_onboard_device_type 497
`<opendmi/entity/oob-remote-access.h>` 500
dmi_oob_connections 500
dmi_oob_remote_access 501
`<opendmi/entity/pointing-device.h>` 503
dmi_pointing_device 503
dmi_pointing_device_iface 504
dmi_pointing_device_type 506
`<opendmi/entity/port-connector.h>` 508
dmi_connector_type 508
dmi_port_connector 512
dmi_port_type 513
`<opendmi/entity/power-controls.h>` 518
dmi_power_controls 518
`<opendmi/entity/power-supply.h>` 520
dmi_power_supply 520
dmi_power_supply_type 522
dmi_range_switching_type 524
`<opendmi/entity/probe.h>` 526
dmi_probe 526
dmi_probe_location 528
`<opendmi/entity/processor-ex.h>` 531
dmi_processor_aarch64_arch_data 531
dmi_processor_aarch64_data 533
dmi_processor_aarch64_data_subtype 534
dmi_processor_amd64_attribute 535
dmi_processor_amd64_data 536
dmi_processor_amd64_temperature_range 537
dmi_processor_amd64_use_condition 538
dmi_processor_arch 539
dmi_processor_ex 540
dmi_processor_revision 542
dmi_processor_rv64_data 543
dmi_processor_specific_block 544
`<opendmi/entity/processor.h>` 545
dmi_processor 545
dmi_processor_arm_id 549
dmi_processor_family 550
dmi_processor_features 570

dmi_processor_id_format . . . 572
dmi_processor_soc_id . . . 573
dmi_processor_status . . . 574
dmi_processor_type . . . 575
dmi_processor_upgrade . . . 576
dmi_processor_voltage_data . . . 584
dmi_processor_voltages . . . 585
dmi_processor_x86_id . . . 586
<opendmi/entity/slot.h> . . . 588
dmi_slot . . . 588
dmi_slot_features . . . 590
dmi_slot_features_ex . . . 592
dmi_slot_height . . . 593
dmi_slot_length . . . 594
dmi_slot_peer_group . . . 596
dmi_slot_type . . . 596
dmi_slot_usage . . . 604
dmi_slot_width . . . 605
<opendmi/entity/string-property.h> . . . 608
dmi_string_property . . . 608
<opendmi/entity/system-boot.h> . . . 610
dmi_boot_status . . . 610
dmi_system_boot . . . 612
<opendmi/entity/system-config.h> . . . 614
dmi_system_config_opts . . . 614
<opendmi/entity/system-event-log.h> . . . 616
dmi_event_log_data_format . . . 616
dmi_event_log_type . . . 617
dmi_system_event_log . . . 620
dmi_system_log_access_method . . . 622
dmi_system_log_header_format . . . 624
dmi_system_log_io_ports . . . 625
dmi_system_log_status . . . 626
dmi_system_log_type_descriptor . . . 627
<opendmi/entity/system-reset.h> . . . 628
dmi_boot_option . . . 628
dmi_system_reset . . . 629
<opendmi/entity/system.h> . . . 631
dmi_system . . . 631
dmi_system_wakeup_type . . . 632
<opendmi/entity/tpm-device.h> . . . 635
dmi_tpm_device . . . 635
dmi_tpm_device_chars . . . 637
dmi_tpm_firmware_version . . . 638

dmi_tpm_firmware_version_format . . . 639
<opendmi/entity/intel/asf.h> . . . 640
dmi_intel_asf . . . 640
<opendmi/entity/intel/amt.h> . . . 642
dmi_intel_amt . . . 642
dmi_intel_amt_oem_caps_1 . . . 643
dmi_intel_amt_oem_caps_3 . . . 645
dmi_intel_amt_oem_caps_4 . . . 646
dmi_intel_amt_terminal . . . 647
<opendmi/entity/intel/fvi.h> . . . 649
dmi_intel_fvi . . . 649
dmi_intel_fvi_item . . . 650
<opendmi/entity/intel/mei.h> . . . 652
dmi_intel_mei . . . 652
dmi_intel_mei_device . . . 654
dmi_intel_me_state . . . 655
dmi_intel_me_error . . . 656
dmi_intel_me_mode . . . 658
dmi_intel_me_sku . . . 659
<opendmi/entity/intel/platform.h> . . . 661
dmi_intel_platform . . . 661
<opendmi/entity/intel/svt.h> . . . 664
dmi_intel_svt . . . 664
dmi_intel_svt_milestone . . . 665
<opendmi/entity/intel/vpro.h> . . . 666
dmi_intel_vpro . . . 666
dmi_intel_vpro_version . . . 668
dmi_intel_vpro_cpu_caps . . . 669
dmi_intel_vpro_mch_caps . . . 670
dmi_intel_vpro_me_caps . . . 672
dmi_intel_vpro_tpm_caps . . . 673
dmi_intel_vpro_bios_caps . . . 674
<opendmi/entity/lenovo/oem.h> . . . 676
dmi_lenovo_oem . . . 676
dmi_lenovo_device_presence . . . 677
dmi_lenovo_bay_io . . . 678
dmi_lenovo_ecp . . . 679
<opendmi/entity/lenovo/records.h> . . . 681
dmi_lenovo_date . . . 681
dmi_lenovo_mtm . . . 681
dmi_lenovo_tpm_info . . . 683
<opendmi/entity/lenovo/tvt.h> . . . 685
dmi_lenovo_tvt . . . 685
<opendmi/entity/acer/devices.h> . . . 687

dmi_acer_devices . . . 687
dmi_acer_device_entry . . . 687
dmi_acer_device_kind . . . 688
<opendmi/entity/acer/hotkeys.h> . . . 691
dmi_acer_hotkeys . . . 691
dmi_acer_hotkey_entry . . . 692
<opendmi/entity/ami/firewire-guid.h> . . . 694
dmi_ami_firewire_guid . . . 694
<opendmi/entity/apple/firmware-volume.h> . . . 696
dmi_apple_firmware_volume . . . 696
dmi_apple_flash_region . . . 697
dmi_apple_firmware_feature . . . 698
dmi_apple_extended_feature . . . 700
dmi_apple_region_type . . . 701
<opendmi/entity/apple/memory-spd-data.h> . . . 703
dmi_apple_memory_spd_data . . . 703
<opendmi/entity/apple/platform-feature.h> . . . 705
dmi_apple_platform_feature . . . 705
dmi_apple_platform_feature_bit . . . 705
<opendmi/entity/apple/processor-bus-speed.h> . . . 708
dmi_apple_processor_bus_speed . . . 708
<opendmi/entity/apple/processor-type.h> . . . 709
dmi_apple_processor_type . . . 709
dmi_apple_processor_class . . . 710
<opendmi/entity/apple/smc-version.h> . . . 712
dmi_apple_smc_version . . . 712
<opendmi/entity/cisco/pci-adapter.h> . . . 713
dmi_cisco_pci_adapter . . . 713
<opendmi/entity/cisco/slot-buses.h> . . . 715
dmi_cisco_slot_buses . . . 715
<opendmi/entity/dell/bios-flags.h> . . . 716
dmi_dell_bios_flags . . . 716
<opendmi/entity/dell/calling-iface.h> . . . 717
dmi_dell_calling_iface . . . 717
dmi_dell_calling_iface_token . . . 718
<opendmi/entity/dell/common.h> . . . 720
dmi_dell_check_type . . . 720
dmi_dell_enable_state . . . 721
dmi_dell_port_security . . . 722
dmi_dell_value_format . . . 723
<opendmi/entity/dell/device-bay.h> . . . 726
dmi_dell_device_bay . . . 726
<opendmi/entity/dell/device-names.h> . . . 728
dmi_dell_device_names . . . 728

dmi_dell_device_name_entry . . . 729
<opendmi/entity/dell/hotkeys.h> . . . 730
dmi_dell_hotkeys . . . 730
dmi_dell_hotkey_entry . . . 730
<opendmi/entity/dell/indexed-io.h> . . . 732
dmi_dell_indexed_io . . . 732
dmi_dell_indexed_io_token . . . 733
<opendmi/entity/dell/infrared-port.h> . . . 735
dmi_dell_infrared_port . . . 735
dmi_dell_infrared_proto . . . 736
<opendmi/entity/dell/memory-ids.h> . . . 739
dmi_dell_memory_ids . . . 739
dmi_dell_memory_id_entry . . . 740
<opendmi/entity/dell/parallel-port.h> . . . 741
dmi_dell_parallel_port . . . 741
dmi_dell_parallel_port_caps . . . 742
dmi_dell_parallel_port_connector_pinout . . . 743
dmi_dell_parallel_port_connector_type . . . 745
<opendmi/entity/dell/protected-area-1.h> . . . 748
dmi_dell_protected_area_1 . . . 748
<opendmi/entity/dell/protected-area-2.h> . . . 750
dmi_dell_protected_area_2 . . . 750
<opendmi/entity/dell/revisions.h> . . . 753
dmi_dell_revisions . . . 753
<opendmi/entity/dell/serial-port.h> . . . 755
dmi_dell_serial_port . . . 755
dmi_dell_serial_port_caps . . . 756
dmi_dell_serial_port_connector_type . . . 758
<opendmi/entity/dell/system-id.h> . . . 761
dmi_dell_system_id . . . 761
<opendmi/entity/dell/token-refs.h> . . . 762
dmi_dell_token_refs_1 . . . 762
dmi_dell_token_refs_2 . . . 762
<opendmi/entity/dell/video-rom.h> . . . 764
dmi_dell_video_rom . . . 764
<opendmi/entity/hpe/backplane.h> . . . 766
dmi_hpe_backplane . . . 766
<opendmi/entity/hpe/common.h> . . . 768
dmi_hpe_flag . . . 768
dmi_hpe_encryption . . . 769
<opendmi/entity/hpe/cru.h> . . . 771
dmi_hpe_cru . . . 771
<opendmi/entity/hpe/device-correlation.h> . . . 773
dmi_hpe_device_correlation . . . 773

dmi_hpe_device_type . . . 776
dmi_hpe_device_location . . . 778
<opendmi/entity/hpe/dimm-attrs.h> . . . 781
dmi_hpe_dimm_attrs . . . 781
<opendmi/entity/hpe/dimm-config.h> . . . 783
dmi_hpe_dimm_config . . . 783
dmi_hpe_interleave_health . . . 784
<opendmi/entity/hpe/dimm-location.h> . . . 787
dmi_hpe_dimm_location . . . 787
<opendmi/entity/hpe/dimm-vendor.h> . . . 790
dmi_hpe_dimm_vendor . . . 790
<opendmi/entity/hpe/drive.h> . . . 792
dmi_hpe_drive . . . 792
dmi_hpe_drive_type . . . 794
dmi_hpe_drive_form_factor . . . 795
dmi_hpe_drive_health . . . 797
<opendmi/entity/hpe/extension-board.h> . . . 799
dmi_hpe_riser . . . 799
dmi_hpe_mhs_riser . . . 800
dmi_hpe_board_type . . . 801
dmi_hpe_riser_position . . . 802
<opendmi/entity/hpe/inventory.h> . . . 804
dmi_hpe_inventory . . . 804
<opendmi/entity/hpe/microcode.h> . . . 806
dmi_hpe_microcode . . . 806
dmi_hpe_microcode_patch . . . 806
<opendmi/entity/hpe/nic.h> . . . 808
dmi_hpe_nic_info . . . 808
dmi_hpe_nic_port . . . 809
dmi_hpe_nic_state . . . 809
<opendmi/entity/hpe/nic-mac.h> . . . 811
dmi_hpe_nic_mac . . . 811
<opendmi/entity/hpe/physical-attrs.h> . . . 813
dmi_hpe_physical_attrs . . . 813
<opendmi/entity/hpe/power-supply.h> . . . 815
dmi_hpe_power_supply . . . 815
dmi_hpe_fru_access . . . 816
<opendmi/entity/hpe/processor.h> . . . 818
dmi_hpe_processor . . . 818
<opendmi/entity/hpe/proliant-info.h> . . . 820
dmi_hpe_proliant_info . . . 820
<opendmi/entity/hpe/rack-locator.h> . . . 822
dmi_hpe_rack_locator . . . 822
<opendmi/entity/hpe/reserved-memory.h> . . . 824

dmi_hpe_reserved_memory . . . 824
dmi_hpe_reserved_memory_entry . . . 824
<opendmi/entity/hpe/rom-info.h> . . . 826
dmi_hpe_rom_info . . . 826
<opendmi/entity/hpe/super-io.h> . . . 828
dmi_hpe_super_io . . . 828
<opendmi/entity/hpe/system-id.h> . . . 830
dmi_hpe_system_id . . . 830
<opendmi/entity/hpe/tcontrol.h> . . . 832
dmi_hpe_tcontrol . . . 832
<opendmi/entity/hpe/trusted-module.h> . . . 833
dmi_hpe_trusted_module . . . 833
dmi_hpe_tm_presence . . . 835
dmi_hpe_tm_disable_reason . . . 836
dmi_hpe_tm_type . . . 837
dmi_hpe_tm_mounting . . . 838
dmi_hpe_tm_fips . . . 839
dmi_hpe_tm_chip . . . 840
dmi_hpe_tm_error . . . 841
<opendmi/entity/hpe/usb-device.h> . . . 843
dmi_hpe_usb_device . . . 843
dmi_hpe_usb_storage_subclass . . . 844
dmi_hpe_usb_storage_proto . . . 846
dmi_hpe_usb_hub_proto . . . 847
<opendmi/entity/hpe/usb-port.h> . . . 849
dmi_hpe_usb_port . . . 849
dmi_hpe_usb_location . . . 850
dmi_hpe_usb_sharing . . . 851
dmi_hpe_usb_speed . . . 853
<opendmi/entity/hpe/version.h> . . . 855
dmi_hpe_version . . . 855
dmi_hpe_firmware_type . . . 856
<opendmi/entity/intel-rsd/cabled-pcie.h> . . . 861
dmi_intel_rsd_cabled_pcie . . . 861
dmi_intel_rsd_cabled_pcie_port . . . 862
<opendmi/entity/intel-rsd/fpga.h> . . . 864
dmi_intel_rsd_fpga . . . 864
dmi_intel_rsd_fpga_hps_isa . . . 867
dmi_intel_rsd_fpga_hssi_config . . . 868
dmi_intel_rsd_fpga_memory_tech . . . 869
dmi_intel_rsd_fpga_status . . . 870
dmi_intel_rsd_fpga_type . . . 871
<opendmi/entity/intel-rsd/memory-device.h> . . . 873
dmi_intel_rsd_memory_device . . . 873

dmi_intel_rsd_memory_media 874
dmi_intel_rsd_memory_type 876
<opendmi/entity/intel-rsd/network-card.h> 878
dmi_intel_rsd_network_card 878
<opendmi/entity/intel-rsd/pcie.h> 881
dmi_intel_rsd_pcie 881
<opendmi/entity/intel-rsd/phys-device-mapping.h> 883
dmi_intel_rsd_phys_device 883
dmi_intel_rsd_phys_device_mapping 884
dmi_intel_rsd_phys_device_type 885
<opendmi/entity/intel-rsd/processor-cpuid.h> 887
dmi_intel_rsd_cpuid_leaf 887
dmi_intel_rsd_cpuid_subtype 888
dmi_intel_rsd_processor_cpuid 889
<opendmi/entity/intel-rsd/storage-device.h> 892
dmi_intel_rsd_storage_connector 892
dmi_intel_rsd_storage_device 893
dmi_intel_rsd_storage_device_type 895
dmi_intel_rsd_storage_proto 897
<opendmi/entity/intel-rsd/tpm.h> 899
dmi_intel_rsd_tpm 899
dmi_intel_rsd_tpm_status 900
<opendmi/entity/intel-rsd/txt.h> 902
dmi_intel_rsd_txt 902
dmi_intel_rsd_txt_status 903
<opendmi/entity/sun/memory-array-ex.h> 905
dmi_sun_memory_array_ex 905
<opendmi/entity/sun/memory-device-ex.h> 907
dmi_sun_memory_device_ex 907
<opendmi/entity/sun/pcie-root-complex.h> 909
dmi_sun_pcie_root_complex 909
<opendmi/entity/sun/port-ex.h> 911
dmi_sun_port_ex 911
<opendmi/entity/sun/processor-ex.h> 913
dmi_sun_processor_ex 913
Part V: Behind the scenes 915
10. From bytes to structures 916
10.1. The context 916
10.2. Opening the data 916
10.3. Where the pieces live 917
11. Buffers, readers and writers 919
11.1. Buffers 919
11.2. Readers 919
11.3. Writers 920

12. Decoders and encoders . . . 922
12.1. Decoders . . . 922
12.2. Encoders . . . 923
13. Describing a structure . . . 924
13.1. The specification . . . 924
13.2. Structure types . . . 925
13.3. Fields . . . 925
13.4. Attributes . . . 929
13.5. Handlers of the structure . . . 931
14. Name sets . . . 933
14.1. Ranges . . . 933
14.2. Looking the names up . . . 934
14.3. Translations . . . 935
15. The registry . . . 936
15.1. Entities . . . 936
15.2. Looking structures up . . . 936
15.3. Linking . . . 937
15.4. Additional information . . . 938
16. Extension modules . . . 939
16.1. Modules . . . 939
16.2. Enabling a module . . . 939
16.3. Platform . . . 940
16.4. Structures of any platform . . . 941
16.5. Relocations . . . 942
16.6. Signatures . . . 942
16.7. Groups . . . 943
16.8. Modules of a program . . . 943
17. Checking the data . . . 945
17.1. Rules . . . 945
17.2. Checks . . . 945
17.3. Checking the data of a program . . . 946
Part VI: File formats . . . 948
18. Exported documents . . . 949
18.1. Layout of the document . . . 949
18.2. Entry point . . . 949
18.3. Structures . . . 949
18.4. Schemas . . . 951
19. Values . . . 952
19.1. Kinds of values . . . 952
19.2. Units . . . 953
19.3. Values for a person . . . 953
20. YAML . . . 954
20.1. Scalars . . . 954
20.2. Structures . . . 955

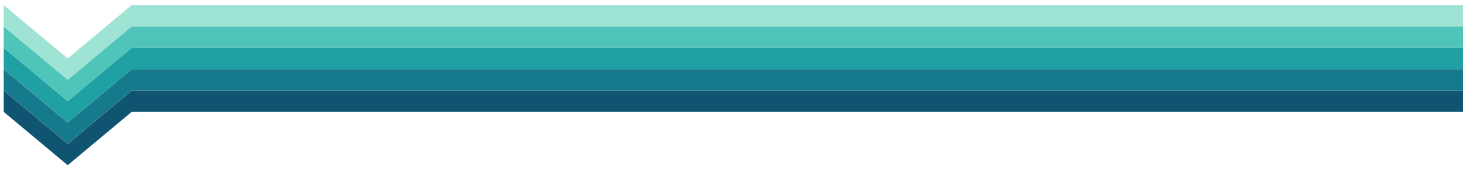
20.3. Schema . . . 955
21. JSON . . . 956
21.1. Values . . . 957
21.2. Structures . . . 957
21.3. Schema . . . 957
22. XML . . . 958
22.1. Document and entry point . . . 958
22.2. Structures . . . 959
22.3. Values . . . 959
22.4. Properties and overlays . . . 959
22.5. Raw data . . . 960
22.6. Schema . . . 960
Part VII: Specifications . . . 961
23. Reading the layouts . . . 962
24. Common structures . . . 963
25. Acer-specific structures . . . 964
25.1. Acer Hotkey Functions (Type 170) . . . 964
25.2. Acer Device List (Type 171) . . . 965
26. AMI-specific structures . . . 967
26.1. AMI FireWire GUID (Type 139) . . . 967
27. Apple-specific structures . . . 968
27.1. Apple Firmware Volume Information (Type 128) . . . 968
27.2. Apple Memory SPD Data (Type 130) . . . 970
27.3. Apple Processor Type Information (Type 131) . . . 970
27.4. Apple Processor Bus Speed Information (Type 132) . . . 971
27.5. Apple Platform Feature Information (Type 133) . . . 971
27.6. Apple SMC Version Information (Type 134) . . . 972
28. Cisco-specific structures . . . 973
28.1. Cisco PCI Slot Buses (Type 201) . . . 973
28.2. Cisco PCI Adapter Information (Type 202) . . . 973
29. Dell-specific structures . . . 975
29.1. Dell BIOS Flags (Type 177) . . . 975
29.2. Dell Hotkeys (Type 178) . . . 975
29.3. Dell Revisions and IDs (Type 208) . . . 976
29.4. Dell Parallel Port (Type 209) . . . 976
29.5. Dell Serial Port (Type 210) . . . 979
29.6. Dell Infrared Port (Type 211) . . . 980
29.7. Dell Indexed IO (Type 212) . . . 982
29.8. Dell Protected Area Type 1 (Type 213) . . . 983
29.9. Dell Protected Area Type 2 (Type 214) . . . 983
29.10. Dell Video BIOS Information (Type 216) . . . 985
29.11. Dell Calling Interface (Type 218) . . . 985
29.12. Dell Device Bay (Type 219) . . . 986
29.13. Dell Token References, Type 1 (Type 220) . . . 987

29.14. Dell Token References, Type 2 (Type 221) . . . 987
29.15. Dell Memory Module Identifiers (Type 223) . . . 988
29.16. Dell Device Names (Type 225) . . . 989
29.17. Dell System ID Record (Type 255) . . . 989
30. HP/HPE-specific structures . . . 991
30.1. HP/HPE Other ROM Information (Type 193) . . . 991
30.2. HP/HPE Super I/O Enable/Disable Indicator (Type 194) . . . 992
30.3. HP/HPE Server System ID (Type 195) . . . 992
30.4. HP/HPE Processor Specific Information (Type 197) . . . 992
30.5. HP/HPE CPU Microcode Patch Support Information (Type 199) . . . 993
30.6. HP/HPE DIMM Location Record (Type 202) . . . 994
30.7. HP/HPE Device Correlation Record (Type 203) . . . 996
30.8. HP/HPE System/Rack Locator (Type 204) . . . 999
30.9. HP/HPE BIOS NIC PCI and MAC Information (Type 209, 221) . . . 999
30.10. HP/HPE Processor TControl Information (Type 211) . . . 1000
30.11. HP/HPE 64-bit CRU Information (Type 212) . . . 1000
30.12. HP/HPE Version Indicator Record (Type 216) . . . 1001
30.13. HP/HPE ProLiant Information (Type 219) . . . 1004
30.14. HP/HPE Trusted Module Status (Type 224) . . . 1004
30.15. HP/HPE Physical Attribute Information (Type 226) . . . 1006
30.16. HP/HPE Reserved Memory Location (Type 229) . . . 1007
30.17. HP/HPE Power Supply Information (Type 230) . . . 1007
30.18. HP/HPE DIMM Attributes Record (Type 232) . . . 1008
30.19. HP/HPE NIC PCI and MAC Information (Type 233) . . . 1009
30.20. HP/HPE HDD Backplane FRU Information (Type 236) . . . 1009
30.21. HP/HPE DIMM Vendor Information (Type 237) . . . 1010
30.22. HP/HPE USB Port Connector Correlation Record (Type 238) . . . 1011
30.23. HP/HPE USB Device Correlation Record (Type 239) . . . 1012
30.24. HP/HPE Firmware Inventory Record (Type 240) . . . 1013
30.25. HP/HPE Hard Drive Inventory Record (Type 242) . . . 1014
30.26. HP/HPE DIMM Current Configuration Record (Type 244) . . . 1016
30.27. HP/HPE Extension Board Inventory Record (Type 245) . . . 1017
31. Intel reference code structures . . . 1019
31.1. Intel ASF Information (Type 129) . . . 1019
31.2. Intel Active Management Technology Information (Type 130) . . . 1019
31.3. Intel vPro Information (Type 131) . . . 1021
31.4. Intel Platform Information (Type 148) . . . 1025
31.5. Intel Management Engine Interface Information (Type 219) . . . 1026
31.6. Intel Firmware Version Information (Type 221) . . . 1029
31.7. Intel Silicon View Technology Milestones (Type 222) . . . 1030
32. Intel RSD structures . . . 1032
32.1. Intel RSD Network Interface Card Information (Type 190) . . . 1032
32.2. Intel RSD PCIe Information (Type 192) . . . 1033
32.3. Intel RSD Processor CPUID Information (Type 193) . . . 1034

32.4. Intel RSD Storage Device Information (Type 194) . . . 1035
32.5. Intel RSD TPM Information (Type 195) . . . 1037
32.6. Intel RSD TXT Information (Type 196) . . . 1038
32.7. Intel RSD Memory Device Extended Information (Type 197) . . . 1039
32.8. Intel RSD FPGA Information (Type 198) . . . 1040
32.9. Intel RSD Cabled PCIe Port Information (Type 199) . . . 1043
32.10. Intel RSD Physical Device Mapping Information (Type 200) . . . 1044
33. Lenovo-specific structures . . . 1046
33.1. Lenovo ThinkVantage Technologies Enablement (Type 131) . . . 1046
33.2. Lenovo Date Record (Type 134) . . . 1046
33.3. Lenovo TPM Information (Type 134) . . . 1047
33.4. Lenovo Mobile PC OEM Data (Type 135) . . . 1047
33.5. Lenovo OEM Data (Type 140) . . . 1048
33.6. Lenovo Machine Type Model (Type 200) . . . 1049
34. Sun-specific structures . . . 1050
34.1. Sun Processor Extended Information (Type 132) . . . 1050
34.2. Sun Port Extended Information (Type 136) . . . 1050
34.3. Sun PCI-Express Root Complex Information (Type 138) . . . 1051
34.4. Sun Memory Array Extended Information (Type 144) . . . 1051
34.5. Sun Memory Device Extended Information (Type 145) . . . 1052
35. Other structures . . . 1053
35.1. Undocumented structures . . . 1053

PART I

Introduction



Chapter 1. About OpenDMI

OpenDMI is a cross-platform framework for reading, decoding and checking DMI/SMBIOS data: the description of a computer its firmware provides, from the vendor and the serial number of the system to its processors, memory modules, slots and ports.

The data is binary, and the specification describing it has grown for three decades, along with the extensions every vendor has added of its own. OpenDMI decodes it into structured data instead of text to parse: every value has a stable code, a type and a unit, so that it reads the same way in the terminal, in JSON and through the API, and a program handling it depends on neither the wording nor the layout of what a person is shown.

1.1. Components

OpenDMI is made of the components listed below, which are built from the same sources and share the decoding done by the library.

Component	Description	Status
`libopendmi`	C library providing access to DMI/SMBIOS data, and the C++ API built on it	Usable, API not frozen until 1.0
`opendmi`	Command line tool to query and check DMI/SMBIOS data	Usable, `import` command is planned
`libopendmi-python`	Python bindings for `libopendmi`	In progress
`opendmi-dbus`	D-Bus service providing access to DMI/SMBIOS data	In progress
`opendmi-sysfs`	Linux kernel module providing DMI/SMBIOS data through SysFS	Planned
`libopendmi-go`	Go bindings for `libopendmi`	Planned
`libopendmi-rust`	Rust bindings for `libopendmi`	Planned

Everything which is planned is scheduled for the 1.0 release.

1.2. License

OpenDMI is free software, distributed under the terms of the BSD 3-Clause License. So is this manual.

Chapter 2. About this manual

This manual covers OpenDMI as a whole, from the command line tool and the interfaces for the programming languages to the layouts of the structures it decodes.

2.1. Organization

The manual is divided into parts:

Introduction

What OpenDMI is, what the data it reads is made of, and how to read this manual.

Getting started

How to build OpenDMI from sources, the commands of the tool most often used, and how a program reads the data through the library.

Command line tool

The `opendmi` tool and its commands.

C library

The functions and the types of the `libopendmi` library, grouped by the headers declaring them. The interfaces for the other programming languages are given parts of their own as they are completed.

Behind the scenes

How the library works inside: how the data is read and decoded, how a structure type is described to the library, and how it is extended.

File formats

The documents `opendmi export` writes in YAML, JSON and XML: how they are structured and how the values of the attributes are written in them.

Specifications

The layouts of the structures OpenDMI decodes, field by field, including the ones the vendors define.

The pages of the command line tool and of the library are the manual pages installed along with them, and are read with `man`(1) as well. The manual itself is available as a PDF and on the website of the project.

2.2. Conventions

Code, commands, file names and literal values are set in `monospace`. Manual pages of this manual are referred to by their names, which link to the pages, e.g. `opendmi` or `dmi_open`. Other manual pages, e.g. the ones of the system, are referred to by their names and sections, e.g. `man`(1).

Numbers written with the `0x` prefix, e.g. `0x1F`, are hexadecimal, and the rest are decimal.

Features which are partly implemented are marked as *(in progress)*, and the ones scheduled for a later release as *(planned)*.

The way the layouts of the structures are given is explained at the beginning of the specifications, see Reading the layouts.

Chapter 3. DMI and SMBIOS

The System Management BIOS (SMBIOS) specification, published by the DMTF as DSP0134, defines the data structures the firmware of a computer describes the computer with. The specification has grown out of the Desktop Management Interface (DMI), and the data is still commonly called DMI data, which is why both names are in use: operating systems and tools often say DMI where the specification says SMBIOS.

The data is laid out by the firmware in memory, where the operating system finds it and usually provides it to the applications, e.g. through SysFS on Linux or the firmware tables on Windows. It is made of an entry point and a table of structures.

3.1. Entry point

The entry point is a small structure, which tells where the table is, how long it is, and which version of the specification the data follows. It comes in several formats:

- The 64-bit entry point of SMBIOS 3.0 and later, anchored by the `_SM3_` string, gives the address of the table and the maximum size of the area it occupies. The end of the table is found by reading it.
- The 32-bit entry point of SMBIOS 2.1 and later, anchored by the `_SM_` string, gives the address and the length of the table, and the number of the structures it holds.
- The legacy entry point, anchored by the `_DMI_` string, is the one the 32-bit entry point embeds, and is found on its own on the oldest systems.

3.2. Structures

The table is a sequence of structures, each of which describes a component of the system: the firmware, the system itself, a baseboard, a processor, a memory device, a slot, a port and so on. Every structure is made of:

- A header of four bytes, which holds the type of the structure, the length of its formatted area, header included, and its handle.
- The rest of the formatted area, which holds the fields the type defines.
- A set of strings, which follows the formatted area. Every string ends with a null byte, and the set ends with another one, so that a structure with no strings ends with two null bytes. Fields refer to the strings by their numbers, counted from one, and zero stands for no string.

The table ends with the end-of-table structure of type 127.

3.3. Types

Types 0 to 127 are defined by the specification. Type 126 marks a structure as inactive, which is left in place but describes nothing, and type 127 ends the table.

Types 128 to 255 are left to the vendors, who define structures of their own. Different vendors use the same numbers for different structures, so the structures of these types are decoded by the extension modules of their vendors, which are enabled on demand.

3.4. Handles

Every structure has a handle, a number unique within the table, which the other structures refer to it by. Memory devices refer to the memory array holding them, cooling devices to the temperature probes watching them, and so on. References are resolved into the structures they point at when the table is linked, and the ones pointing at no structure are reported.

3.5. Versions

Every version of the specification may add fields to the end of the structures defined earlier, so the length of a structure tells which version it has been written for. A structure written for an older version is shorter, and carries the fields of that version only. The fields a later version has added are absent rather than zero, and OpenDMI keeps them apart: such a structure is marked as partial. A structure whose length matches no version, and which ends in the middle of the fields of one, is marked as incomplete, and the fields it carries in full are decoded all the same.

Chapter 4. Features

Features marked as *(in progress)* are partly implemented, and the ones marked as *(planned)* are scheduled for the 1.0 release.

4.1. SMBIOS support

- All standard structure types of the SMBIOS specification up to version 3.10 are decoded field by field.
- Entry points of all kinds are read: the legacy DMI one (`_DMI_`), the 32-bit SMBIOS 2.1 one (`_SM_`) and the 64-bit SMBIOS 3.0 one (`_SM3_`).
- Values are decoded to their meaning rather than left as numbers: enumerations, sets of flags, sizes and other measures with their units, versions, dates and UUIDs, with the format of a value chosen by the fields that define it.
- Structures are linked together: handles are resolved into references to the structures they point at, string properties (type 46) are attached to the structures they describe, and additional information (type 40) can be applied to the structures it amends.
- Structures written for older versions of the specification are decoded as far as they go, and every structure carries its state, e.g. `partial` for a structure of an older version, or `incomplete` for one ending in the middle of its fields.
- Broken firmware does not stop decoding: malformed structures are reported and skipped, errors are collected rather than ending the table, and a strict mode is available when correctness matters more than completeness.
- Known defects of the firmware, such as swapped words of the processor ID or reversed TPM vendor identifiers, are handled.

4.2. Vendor-specific structures

- Structures the vendors define are decoded by extension modules for Acer, AMI, Apple, Cisco, Dell, HP/HPE, Intel, IBM/Lenovo and Sun, and their layouts are described in the specifications part of this manual.
- vPro, AMT, ASF, MEI, FVI, SVT and RSD structures, which the Intel reference code adds on almost every Intel platform, are decoded field by field, even where vendors move them, while `dmidecode` shows them as raw data.
- Modules are enabled on demand, so that a type number a vendor reuses is decoded the way that vendor defines it.
- Structure types are described by their layout on the wire rather than decoded by hand, which is how new ones are added.
- Decoders of more vendor-specific structures *(planned)*.

4.3. Platforms

- Linux, through the SMBIOS tables the kernel exposes in SysFS.
- FreeBSD, through the entry point address the kernel environment holds, and NetBSD, through the one `sysctl` gives.
- macOS, through the `AppleSMBIOS` service of the I/O Kit.

- Windows, through the firmware tables the system provides.
- Scan of the legacy BIOS area for the entry point on FreeBSD and NetBSD for x86, when the kernel gives no address of it.
- Dump files, which are interchangeable with `dmidecode --dump-bin` and `dmidecode --from-dump` in both directions.
- Reading the memory device on Linux, where SysFS gives no access to the tables *(planned)*.
- Solaris and illumos *(planned)*.

4.4. Command line tool

- Structures are shown, listed, explained and exported with the `opendmi` tool, and the table can be dumped to a file.
- Dumps and exported documents can be anonymized to be shared, e.g. in a bug report: serial numbers, asset tags, UUIDs and MAC addresses are replaced wherever the table holds them with values of the same length and format, under a random key which is never saved, and the dump keeps the layout of the table.
- Structures are selected by handle, by type, by the module providing them and by state.
- Output is text for a person, with colors and a pager, or JSON, XML and YAML for a program, where every value is addressed by a stable code.
- Messages and the names of the structures and their values are translated to English and Russian, when built with ICU4C.
- Import of the data from JSON, XML and YAML *(planned)*.

4.5. Validation

- Data is checked against the rules of the specification, from the checksum of the entry point to the values of individual fields, with the `lint` command.
- Findings are reported the way a compiler reports diagnostics: with the offset within the data, the structure and the attribute they belong to, and the rule that has found them.
- Rules are grouped and can be enabled or disabled one by one or by group.
- Data can be checked as read from a platform, where the defects of the firmware are nothing the reader can fix, or as produced, where the same defects are errors.

4.6. Library

- The `libopendmi` library gives access to the decoded structures as plain C types, and to every value through its attribute, which carries its code, type and unit, for the code which handles the structures generically.
- Structures can be encoded back into their bytes, either keeping the bytes the structure has been decoded from, or writing them the way the specification does.
- Values identifying the system are marked in the specifications of the structures, and an anonymized copy of the table is made from them.
- Errors are collected in a queue of the context, and logging is directed to a handler of the application.
- The library depends on nothing but the system, and on ICU4C for the translations when built with it.

Libraries of the output formats are the dependencies of the command line tool alone.

- Writing of the whole SMBIOS table *(planned)*.
- Unified API for the serialization of the data *(planned)*.
- Stable API *(planned)*.

4.7. Other languages and services

- C++ API *(in progress)*.
- Python bindings *(in progress)*.
- Go bindings *(planned)*.
- Rust bindings *(planned)*.
- D-Bus service giving access to the data *(in progress)*.
- Linux kernel module exposing the data through SysFS *(in progress)*.
- Packages for the major Linux distributions *(planned)*.

4.8. Quality

- Decoders are tested against a collection of dumps taken from real machines, which every structure is also encoded back from and compared.
- Every change is built and tested on Linux, FreeBSD, NetBSD, macOS and Windows, and checked by static analysis.
- Debug builds use AddressSanitizer wherever the toolchain supports it.

Chapter 5. Terminology

Abbreviations used throughout this manual and the specifications it follows.

AC	Alternating Current.
ACPI	Advanced Configuration and Power Interface.
AGP	Accelerated Graphics Port.
APM	Advanced Power Management.
ASCII	American Standard Code for Information Interchange.
ATA	Advanced Technology Attachment.
ATAPI	ATA Packet Interface.
BCD	Binary-Coded Decimal.
BIOS	Basic Input/Output System.
BMC	Baseboard Management Controller.
CGA	Color Graphics Array.
CMOS	Complementary Metal-Oxide Semiconductor. "CMOS" is commonly used as a shorthand for "CMOS RAM", the non-volatile RAM used on industry-standard PCs.
CPU	Central Processing Unit.
CPUID	CPU Identification.
CRC	Cyclic Redundancy Check.
DDC	Display Data Channel.
DDR	Double Data Rate SDRAM.
DIMM	Dual In-line Memory Module.
DMA	Direct Memory Access.
DMI	Desktop Management Interface.
DRAM	Dynamic RAM.
DSP	Digital Signal Processing.
ECC	Error Checking and Correction.
EDD	Enhanced Disk Drive.
EDO	Extended Data Out.

EEPROM	Electrically Erasable PROM.
EISA	Extended Industry-Standard Architecture.
EPS	Entry Point Structure.
ESCD	Extended System Configuration Data.
FDC	Floppy Drive Controller.
FIFO	First In, First Out.
FPGA	Field-Programmable Gate Array.
GPNV	General-Purpose NVRAM.
I2O	Intelligent Input/Output.
IEPS	Intermediate Entry Point Structure.
IO	Input/Output
IPMI	Intelligent Platform Management Interface.
IRQ	Interrupt Request.
ISA	Industry Standard Architecture.
LIF	Low Insertion Force.
LSB	Least-Significant Bit.
MCA	Micro Channel Architecture.
MOF	Managed Object Format.
MSB	Most Significant Bit.
NIC	Network Interface Card.
NMI	Non-Maskable Interrupt.
NV	Non-Volatile.
NVRAM	Non-Volatile RAM.
OEM	Original Equipment Manufacturer.
OS	Operating System.
PATA	Parallel ATA.
PCI	Peripheral Component Interconnect.
PCIe	Peripheral Component Interconnect Express (PCI Express).

PCMCIA	Personal Computer Memory Card International Association.
PME	Power Management Event.
PNP	Plug-And-Play.
POST	Power-On Self-Test.
PROM	Programmable ROM.
PXE	Pre-boot Execution Environment.
RAID	Redundant Array of Inexpensive Disks.
RAM	Random-Access Memory.
ROM	Read-Only Memory.
RPM	Revolutions per Minute.
RTC	Real-Time Clock.
SAS	Serial-Attached SCSI.
SATA	Serial ATA.
SCSI	Small Computer System Interface.
SDRAM	Synchronous DRAM.
SIMM	Single In-line Memory Module.
SKU	Stock-Keeping Unit.
SMBus	System Management Bus.
SMM	System Management Module.
SRAM	Static RAM.
TPM	Trusted Platform Module.
UEFI	Unified Extensible Firmware Interface.
UPS	Uninterruptible Power Supply.
USB	Universal Serial Bus.
UUID	Universally Unique Identifier.
VESA	Video Electronics Standards Association.
VL-VESA	VESA Video Local Bus.
ZIF	Zero Insertion Force.

Chapter 6. References

6.1. Specifications

DMTF DSP0134

System Management BIOS (SMBIOS) Reference Specification. Defines the entry points, the structures of the standard types and their fields, which OpenDMI decodes up to version 3.10.

DMTF DSP0270

Redfish Host Interface Specification. Defines the protocol records of the management controller host interface structures.

DMTF DSP0266

Redfish Specification. Defines the format of the dates and times firmware inventory structures carry.

Intelligent Platform Management Interface (IPMI) Specification

Defines the interfaces of the baseboard management controllers IPMI device structures describe.

Intel® RSD Firmware Extension Specification Software v2.3.2

Document 337201-003US. Defines the structures of Intel Rack Scale Design platforms.

6.2. Other sources

Structures some vendors define have no public specification, and their layouts come from the software of the vendors which reads them:

Dell `libsmbios`

Structures of Dell platforms.

illumos

Structures of Sun platforms, as defined in `usr/src/uts/common/sys/smbios_impl.h`.

Where it is known, the source of the layout of a vendor-specific structure is given along with it in the specifications part of this manual.

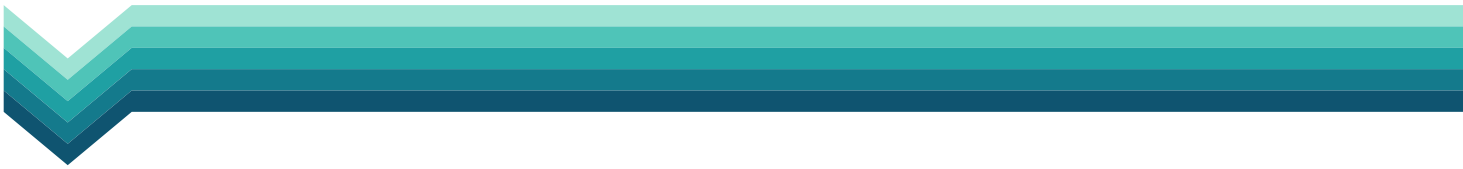
6.3. Related software

`dmidecode`

Reference tool for reading DMI/SMBIOS data on Linux and other systems. Dump files are interchangeable between `dmidecode` and OpenDMI.

PART II

Getting started



Chapter 7. Building from sources

OpenDMI is built from sources with CMake, either through the `build.sh` script in the root of the sources, which keeps the options of the build in a file of its own, or with CMake directly.

7.1. Prerequisites

- A C compiler supporting C23: GCC, Clang or MSVC.
- CMake 3.25 or newer.

The library depends on nothing but the system. The rest of the dependencies are optional, and enable the features listed below when they are found:

Dependency	Feature
`libxml2`	XML output of the command line tool
`libyaml`	YAML output of the command line tool
`libyajl`	JSON output of the command line tool
Curses	Colors in the output of the command line tool
ICU4C	Translations of the messages and the names of the values
AsciiDoctor	Manual pages
AsciiDoctor PDF	This manual as a PDF
CppCheck	Static analysis while building

CMocka, which the tests are written with, is fetched while configuring, and is not installed along with OpenDMI.

7.2. Configuring

The build is configured with:

```
$ ./build.sh configure
```

This configures a release build to be installed into `/opt/opendmi`. The most common options are:

`--prefix <path>`
: Installation prefix.

`--debug`
: Debug build, with optimization disabled and assertions enabled. Debug builds use AddressSanitizer when the toolchain supports it, which `--without-asan` turns off.

`--with-json`, **`--with-xml`**, **`--with-yaml`**
: Require the output formats, and fail to configure instead of building without them.

`--with-icu`
: Build with the translations.

`--enable-dbus`, `--enable-python`, `--enable-golang`, `--enable-rust`
: Build the optional components.

`--enable-website`
: Build the website of the project.

All of the options are listed by `./build.sh --help`. The defaults are kept in `build.conf.dist`, and are overridden by a `build.conf` file in the root of the sources, which is not a part of the sources.

7.3. Building and testing

The configured build is built and tested with:

```
$ ./build.sh build
$ ./build.sh test
```

The number of the parallel jobs is set with `-j`, and the build directory with `-b`, e.g. `./build.sh -j 4 -b /tmp/opendmi build`.

7.4. Installing

The build is installed into the configured prefix with:

```
$ ./build.sh install
```

The library, its headers, the command line tool, the manual pages, and the files which find the library with CMake and `pkg-config` are installed.

7.5. Building with CMake directly

`build.sh` passes the options to CMake as the variables of the same names, so the same build is configured by CMake itself as well:

```
$ cmake -B build -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr/local
$ cmake --build build
$ ctest --test-dir build
$ cmake --install build
```

Optional components are enabled by `-DENABLE_<COMPONENT>=ON`, e.g. `-DENABLE_DBUS=ON`, and features are required by `-DENABLE_<FEATURE>=ON` or turned off by `-DENABLE_<FEATURE>=OFF`, e.g. `-DENABLE_JSON=ON`.

Unlike `build.sh`, CMake builds without AddressSanitizer and does not treat warnings as errors unless

told to by `-DENABLE_ASAN=ON` (or `AUTO`, to use it where it runs) and `-DENABLE_WERROR=ON`. Optimization and debug information come from the flags of the build type, e.g. `CMAKE_C_FLAGS_RELEASE`, which is `-O2 -DNDEBUG` unless set otherwise.

Chapter 8. First steps

This chapter goes through the commands of the `opendmi` tool most often used. Every command is described in full in the part of this manual about the command line tool, and is explained by `opendmi <command> --help` too.

8.1. Reading the data of the system

The structures of the system the tool runs on are shown with:

```
$ opendmi show
```

Reading the data of a platform usually requires privileges, because the table is taken from physical memory or from an interface the kernel restricts, so the tool is often run as the superuser.

Structures are selected by type, given by number or by code, or by handle:

```
$ opendmi show -t processor
$ opendmi show -t 17
$ opendmi show -H 0x0002
```

The structures the table holds are listed, with their handles and types, with:

```
$ opendmi list
```

What a structure type is for, as the specification describes it, is told with:

```
$ opendmi explain memory-device
```

8.2. Exporting the data

The data is exported for a program to read, with every value addressed by a stable code, in JSON, XML or YAML:

```
$ opendmi export -f json -o smbios.json
```

The values are written by their codes, e.g. `4294967296` for a size, and are written for a person to read instead, e.g. `4 GiB`, with `--pretty`.

8.3. Working with dumps

The table is dumped into a file, which is read in place of the system with the `-i` option, on any system and without privileges:

```
$ opendmi dump -o smbios.bin
$ opendmi -i smbios.bin show
```

Dump files are interchangeable with the ones `dmidecode --dump-bin` writes and `dmidecode --from-dump` reads.

8.4. Checking the data

The data is checked against the rules of the specification with:

```
$ opendmi -i smbios.bin lint
```

Every issue found is reported with its offset within the data, the structure and the attribute it belongs to, and the rule which has found it:

```
0x00F3: note: baseboard@0x0002 (Baseboard or module information): string 4
holds whitespace only [string.blank]
```

8.5. Structures the vendors define

Structures of the types the vendors define are decoded once the module of the vendor is enabled. The modules available are listed with `opendmi modules`, and one is enabled with the `-m` option:

```
$ opendmi -m dell show
```

Chapter 9. Using the library

This chapter shows how a program reads the data through the `libopendmi` library. Every function is described in full in the part of this manual about the C library.

9.1. Finding the library

The installed library is found with CMake, by the package it installs:

```
find_package(OpenDMI 0.7 REQUIRED)
```

Binary compatibility is not kept between the minor versions before 1.0, so the package is compatible with the same minor version only.

The library is found with `pkg-config` too, by the `opendmi` package.

A library installed outside the directories CMake and `pkg-config` look in, such as the default `/opt/opendmi` prefix of `build.sh`, is found through the prefix given to CMake, and the directory of the package given to `pkg-config`:

```
$ cmake -B build -DCMAKE_PREFIX_PATH=/opt/opendmi
$ export PKG_CONFIG_PATH=/opt/opendmi/lib/pkgconfig
```

The directory of the libraries is `lib` in the examples of this chapter, and is `lib64` on some systems.

9.2. Linking

The library is built both as a shared library and as a static one, and a program is linked with either.

9.2.1. Linking with the shared library

The shared library is what a program is linked with by default:

```
target_link_libraries(example PRIVATE OpenDMI::opendmi)
```

`OpenDMI::opendmi` and `OpenDMI::shared` are the same library. Without CMake, the flags come from `pkg-config`:

```
$ cc example.c $(pkg-config --cflags --libs opendmi)
```

The library is looked up again when the program starts. A library installed outside the directories the system looks in is found by the path the program is linked with:

```
$ cc example.c $(pkg-config --cflags --libs opendmi) \
```

```
    -Wl,-rpath,/opt/opendmi/lib
```

CMake gives that path to the programs in its build tree by itself, and to the installed ones through the `INSTALL_RPATH` property. The shared library is named after the minor version before 1.0, e.g. `libopendmi.so.0.7` on Linux and `libopendmi.0.7.dylib` on macOS, so that a program is not started with a library it is incompatible with.

9.2.2. Linking with the static library

The static library makes a program which needs no OpenDMI installed to run:

```
target_link_libraries(example PRIVATE OpenDMI::static)
```

The target brings along everything the static library needs, which the other ways of linking give by hand.

The static library is `libopendmi.a`, and `opendmi-static.lib` with MSVC. It is linked by its path, because `-lopendmi` is resolved to the shared library whenever one is installed next to it. The libraries it depends on are linked along with it: none on Linux and the BSDs, the CoreFoundation and IOKit frameworks on macOS, and `ntdll` on Windows, along with ICU4C when the library is built with the translations. `pkg-config --static --libs opendmi` lists them, after the `-lopendmi` the path stands for:

```
$ cc $(pkg-config --cflags opendmi) example.c /opt/opendmi/lib/libopendmi.a
$ cc $(pkg-config --cflags opendmi) example.c /opt/opendmi/lib/libopendmi.a \
    -framework CoreFoundation -framework IOKit
```

The first command links the program on Linux, and the second on macOS.

On Windows, the program linked with the static library defines `DMI_STATIC`, so that the functions of the library are not taken for the ones a DLL imports. `OpenDMI::static` defines it by itself, and `pkg-config --static --cflags opendmi` gives it there too.

9.3. Reading the data

A program creates a context, which holds the data and everything decoded from it, opens the data of the system, and looks up the structures it needs in the registry of the context. Decoded structures are plain C types, one for each structure type:

```
#include <stdio.h>

#include <opendmi/context.h>
#include <opendmi/registry.h>
#include <opendmi/entity.h>
#include <opendmi/entity/system.h>

int main(void)
{
```

```
    dmi_context_t *context = dmi_create(DMI_CONTEXT_FLAG_LINK);

    if (context == nullptr)
        return 1;

    if (dmi_open(context, nullptr)) {
        dmi_registry_t *registry = dmi_get_registry(context);
        dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                             DMI_TYPE(system),
                                                             false);
        dmi_system_t   *system   = dmi_entity_info(entity, DMI_TYPE(system));

        if (system != nullptr)
            printf("%s %s\n", system->vendor, system->product);
    }

    dmi_destroy(context);

    return 0;
}
```

`dmi_open` reads the data of the system, and requires the same privileges the command line tool does. `dmi_load` reads a dump file instead, which is how a program is tried out on the data of another machine:

```
if (dmi_load(context, "smbios.bin")) {
    ...
}
```

9.4. Handling errors

Functions which fail return `false` or `nullptr`, and leave the reason in the error queue of the context, where `dmi_error_get_last` takes it from:

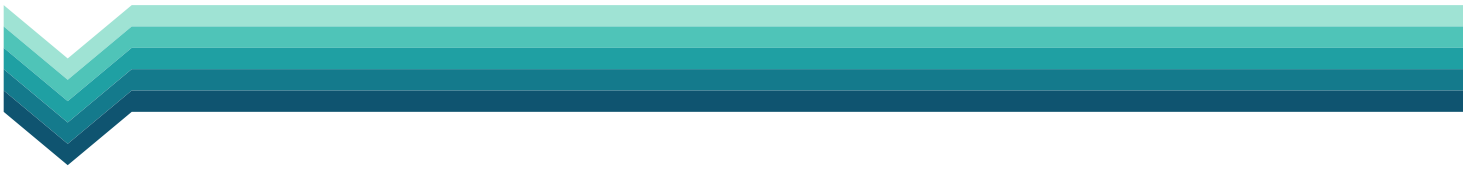
```
if (!dmi_open(context, nullptr)) {
    const dmi_error_t *error = dmi_error_get_last(context);

    if (error != nullptr)
        fprintf(stderr, "%s\n", dmi_error_message(error->reason));
}
```

The last error is the one the failed call has raised itself. The errors raised on the way to it, such as the file which could not be opened, precede it in the queue, and are taken from its beginning with `dmi_error_get_first`.

PART III

Command line tool



Tool

opendmi

Synopsis

opendmi [*global options*] *command* [*command options*] [--] [*command args*]

opendmi -v|--version

opendmi -h|--help

Description

The **opendmi** utility reads the DMI/SMBIOS data of the platform it runs on, or of a dump file, and gives it back as decoded structures rather than as text to parse. Every field is addressed by a stable code, carries a type and a unit, and reads the same way in the terminal, in JSON, XML and YAML, and through the C API of `libopendmi`.

What the utility does is chosen by a *command*, which follows the global options and takes options of its own. Global options apply to every command, and decide where the data comes from and what is decoded; command options decide what is done with it. Values of options are given either as a separate argument (**-i** *path*, **--file** *path*) or attached to the option (**-i*****path*****, *--file**=*path*).

Reading the data of a platform usually requires privileges, because the table is taken from physical memory or from an interface the kernel restricts. Dump files are read with the privileges of the file itself. The commands which read no data, **explain**, **modules**, **types** and **lint** listing its rules, need no privileges at all.

Commands

dump

Dump the entire SMBIOS table to a binary file, see `opendmi-dump`.

entry

Show the data of the SMBIOS entry point, see `opendmi-entry`.

list

List the structures with their handles and types, see `opendmi-list`.

show

Show the data of the structures, see `opendmi-show`.

explain

Explain a structure type, see `opendmi-explain`.

lint

Check the data against the rules of the specification, see `opendmi-lint`.

export

Export the data to an external format, see `opendmi-export`.

import

Import the data from an external format, see `opendmi-import`.

types

List the structure types the utility knows, see `opendmi-types`.

modules

List the extension modules available, see `opendmi-modules`.

Use **opendmi** *command* **--help** for the options of a particular command.

Options

-v, --version

Print version information and exit.

-?, -h, --help

Print help and exit.

-i *path*, --file=*path*

Read the data from a binary dump file instead of the platform. Dump files are interchangeable with the ones of `dmidecode --dump-bin`.

-d *path*, --device=*path*

Set the path to the memory device the data is read through, `/dev/mem` by default. Only used by the backends which read physical memory.

-m *module*, --module=*module*

Enable the named extension module, so that the structures it describes are decoded instead of being left unknown. The option may be given several times. Modules enabled this way take precedence over the ones enabled for the platform. See `opendmi-modules` for the modules available.

-M, --no-auto-modules

Do not enable the modules of the platform automatically. By default, the modules of the platform the data comes from are enabled as the data is read, e.g. the **dell** module for a Dell platform, or the **hpe** module for an HP or HPE server, unless the structures they describe conflict with the ones of the modules enabled with **--module**.

-O, --overlay

Apply additional information entries (type 40) to the structures they refer to, so that the values they carry are shown as part of those structures.

-l, --log

Enable logging of what the library does while it reads and decodes the data.

--log-file=*path*

Enable logging to the named file, which is appended to rather than replaced.

-L *level*, --log-level=*level*

Set the level logging is limited to: `error`, `warning`, `notice`, `info` or `debug`. The default is `notice`.

-P, --no-pager

Disable pager.

Exit status

0

Command has succeeded.

1

Command has failed, e.g. the data could not be read, or the checks of `opendmi-lint` have found an error.

2

Command line is invalid.

Environment

PAGER

Pager the output of the text commands (`show`, `list`, `entry`, `lint`, `types`, `explain`, `modules`) is shown through, `less` if it is not set. Set it to an empty value to disable the pager. Output which is not a terminal is never paged. Quitting the pager before the end of the output is not a failure: the rest of the output is discarded and the command exits with status 0.

LESS

Options of `less`, when it is used as the pager. If it is not set, the utility lets `less` print output which fits on one screen directly.

Files

/dev/mem

Memory device the table is read through on FreeBSD and NetBSD, see **--device**.

/sys/firmware/dmi/tables

Tables exposed by the Linux kernel, which the Linux backend reads.

Examples

Show the data of the local platform:

```
$ opendmi show
```

Show the processors only:

```
$ opendmi show -t processor
```

Save the table of the local platform and read it back later:

```
$ opendmi dump -o smbios.bin
$ opendmi -i smbios.bin show
```

Decode the structures of a vendor as well as the standard ones:

```
$ opendmi -m dell show
```

Check a dump against the rules of the specification:

```
$ opendmi -i smbios.bin lint
```

See also

`opendmi-dump`, `opendmi-entry`, `opendmi-explain`, `opendmi-export`, `opendmi-import`, `opendmi-lint`, `opendmi-list`, `opendmi-modules`, `opendmi-show`, `opendmi-types`

Commands

dump

Synopsis

opendmi [*global options*] **dump** [*command options*]

Description

The **dump** command writes the entry point and the whole table to a binary file, byte for byte, without decoding anything, unless the dump is to be anonymized. The result is the data as the firmware wrote it, which is what to keep when the data has to be looked at later, or on another machine, or reported to someone else.

Dump files are interchangeable with the ones of `dmidecode --dump-bin`, in both directions: a dump written here is read by `dmidecode --from-dump`, and a dump written by `dmidecode` is read by the global **--file** option of every command of this utility.

Only the structures the platform provides are written. Filter options do not apply, since the point of a dump is to keep everything; use `opendmi-export` to write a selected part of the data instead.

Options

-?, -h, --help

Print help and exit.

-o *path*, --output=*path*

Write the dump to the named file. The default is `smbios.bin` in the current directory.

-F, --force

Overwrite the output file if it exists. Without this option an existing file is left alone and the command fails.

-A, --anonymize

Replace the values identifying the system, e.g. serial numbers, asset tags, UUIDs and MAC addresses, so that the dump can be shared. The values are replaced with ones of the same length and format wherever the table holds them, under a random key which is never saved, and the dump keeps the layout of the table. Values of the structures no enabled module decodes are replaced only where they repeat a value found elsewhere, see `dmi_anonymize`. The command fails with the global **--overlay** option.

Exit status

0

Dump has been written.

1

Data could not be read or written, e.g. the output file exists and **--force** was not given.

2

Command line is invalid.

Examples

Dump the table of the local platform:

```
$ opendmi dump -o smbios.bin
```

Read the dump back:

```
$ opendmi -i smbios.bin show
```

Refresh a dump which is already there:

```
$ opendmi dump -F -o smbios.bin
```

Dump the table to attach it to a bug report:

```
$ opendmi dump --anonymize -o smbios.bin
```

See also

`opendmi`, `opendmi-export`, `opendmi-lint`, `dmidecode`(8)

entry

Synopsis

opendmi [*global options*] **entry** [*command options*]

Description

The **entry** command shows the entry point of the SMBIOS data: the anchor which identifies it, the version of the specification the firmware follows, and where the table is and how large it is.

The entry point comes in several formats, and the command shows the fields of the one it has found. A 32-bit entry point (`_SM_` or `_DMI_`) declares the number of the structures and the size of the largest one; a 64-bit entry point (`_SM3_`) declares the maximum size of the table area instead, and leaves the end of the table to be found by reading it.

The version shown here is the one the rest of the utility decodes the data against, which is why a table that disagrees with its own entry point is worth checking with `opendmi-lint`.

The command has no options of its own.

Options

-?, -h, --help
Print help and exit.

Exit status

0
Entry point has been shown.

1
Data could not be read.

2
Command line is invalid.

Examples

Show the entry point of the local platform:

```
$ opendmi entry
```

Show the entry point a dump was saved with:

```
$ opendmi -i smbios.bin entry
```

See also

`opendmi`, `opendmi-lint`, `opendmi-list`, `opendmi-show`

explain

Synopsis

opendmi [*global options*] **explain** [*command options*] [--] *type*

Description

The **explain** command prints what a structure type is for, as the specification describes it, along with its number and its code name.

The *type* argument is either a number or a code name, e.g. `4` or `processor`. Use `opendmi-types` to see the types and their codes, and `--` before the argument if it could be read as an option.

Types described by extension modules are explained as well, once the module is enabled with the global **--module** option. The command reads no SMBIOS data, so it works without access to the platform.

The command has no options of its own.

Options

-?, -h, --help
Print help and exit.

Exit status

0
Type has been explained.

1
Type is unknown.

2
Command line is invalid.

Examples

Explain a type by its code name:

```
$ opendmi explain processor
```

Explain a type by its number:

```
$ opendmi explain 17
```

Explain a type an extension module describes:

```
$ opendmi -m dell explain 212
```

See also

`opendmi`, `opendmi-show`, `opendmi-types`

export

Synopsis

opendmi [*global options*] **export** [*command options*] [*filter options*]

Description

The **export** command writes the decoded SMBIOS structures in a machine-readable format, so that the data can be consumed by something other than a reader of the terminal.

Fields are named by their codes rather than by their labels, e.g. `socket-designation` instead of `Socket Designation`, and enumerated values are written as codes as well. Codes are stable across versions and

locales, so the output can be matched against without parsing human-readable text. **--pretty** writes the values for a person instead, leaving the field names alone.

The formats available depend on how the utility was built, since each of them needs a library of its own. The formats of a particular build are listed at the end of its **--help** output.

Which structures are exported is decided by the filter options, which are common to `opendmi-show` and `opendmi-list`. By default the standard and the OEM-specific structures are exported, and the inactive and the unknown ones are not.

Options

-?, -h, --help

Print help and exit.

-o *path*, --output=*path*

Write the output to the named file instead of the standard output.

-f *format*, --format=*format*

Set the output format: `text`, `json`, `xml` or `yaml`. The default is `yaml`.

-D, --dump

Export the raw bytes of the structures instead of decoding them.

-p, --pretty

Write the values the way a person reads them, as `opendmi-show` prints them, instead of by their codes: `DDR4` rather than `ddr4`, `4 GiB` rather than `4294967296`, and the measurement unit of a field along with its number. Field names stay codes either way, so that the document can still be addressed by them, and the text format is unaffected, since it is already what the option asks the other formats for.

-F, --force

Overwrite the output file if it exists. Without this option an existing file is left alone and the command fails.

-A, --anonymize

Replace the values identifying the system, e.g. serial numbers, asset tags, UUIDs and MAC addresses, so that the document can be shared. The values are replaced the way **opendmi dump --anonymize** replaces them, see `opendmi-dump`, and the structures are exported with the replaced values. The command fails with the global **--overlay** option.

Filter options

-H *handle*, --handle=*handle*

Only export the structures with the given handle, which is read as a hexadecimal number, with or without the `0x` prefix. The option may be given several times.

-t *type*, --type=*type*

Only export the structures of the given type, which is either a number or a code name, e.g. `4` or `processor`. The option may be given several times. See `opendmi-types` for the types and their codes.

-s, --standard

Export the structures of the types the specification defines.

-S, --no-standard

Do not export the structures of the types the specification defines.

-e, --oem

Export the OEM-specific structures, which are the ones of types 128 to 255.

-E, --no-oem

Do not export the OEM-specific structures.

-i, --inactive

Export the structures which the firmware has marked as inactive (type 126).

-I, --no-inactive

Do not export the inactive structures.

-u, --unknown

Export the structures of the types no enabled module describes, which are left as raw data.

-U, --no-unknown

Do not export the structures of the unknown types.

-m ***module*****, --module=***module*

Only export the structures described by the named module, whatever type numbers the platform places them at. The module has to be enabled, either for the platform or with the global **--module** option, or else nothing it describes is decoded. The option may be given several times.

-M, --all-modules

Only export the structures described by the enabled modules.

-a, --all

Export every structure, including the inactive and the unknown ones.

Exit status

0

Data has been exported.

1

Data could not be read or written, e.g. the output file exists and **--force** was not given.

2

Command line is invalid.

Examples

Export everything to YAML, which is the default format:

```
$ opendmi export -o smbios.yaml
```

Export to YAML with serial numbers, asset tags, UUIDs and MAC addresses replaced, to be attached to a bug report:

```
$ opendmi export --anonymize -o smbios.yaml
```

Export to JSON with the values written for a person rather than for a program:

```
$ opendmi export -f json --pretty -o smbios.json
```

Export the processors alone, and read a field out of the result:

```
$ opendmi export -f json -t processor |
    jq '.table[].attributes."socket-designation"'
```

See also

`opendmi`, `opendmi-dump`, `opendmi-import`, `opendmi-show`

import

Synopsis

opendmi [*global options*] **import** [*command options*]

Description

The **import** command is meant to read the data back from the formats `opendmi-export` writes, and to assemble a table out of it, so that SMBIOS data can be authored rather than only observed.

The command is not implemented yet and fails with a message saying so. It is described here because its shape is settled; see the roadmap of the project for when it is due.

Until then, a table is produced by the firmware alone, and is read from a platform or from a binary dump written by `opendmi-dump`. Data which is being authored is checked with `opendmi-lint`, the **--producer** option of which applies the rules the way a producer of the data has to follow them.

Options

-?, -h, --help
: Print help and exit.

Exit status

1

Command is not implemented.

2

Command line is invalid.

See also

`opendmi`, `opendmi-dump`, `opendmi-export`, `opendmi-lint`

lint

Synopsis

opendmi [*global options*] **lint** [*command options*]

Description

The **lint** command checks the SMBIOS data against the rules of the specification and reports what it finds, from the checksum of the entry point down to the values of individual fields.

The checks cover the entry point, the table as a whole, and every structure in it. Rules which apply to any structure look at its length, its strings, the values of its fields, the references it makes and the additional information entries pointing at it. Rules which belong to a particular type look at what that type means: that the enabled cores of a processor fit the ones it has, that the mapped address ranges of the memory arrays do not overlap, that a cooling device points at a temperature probe rather than at a voltage one.

Since the data is decoded before it is checked, the rules see fields rather than bytes, and a report names the structure and the attribute the issue belongs to. Some rules read the raw bytes as well, to catch what the decoder discards.

Output

Every issue is reported the way a compiler reports a diagnostic, so that the output is readable by the tools which are used to that format:

```
0x0009: warning: firmware@0x0000.rom-size (Platform firmware information): ROM
size refers to the extended one, which the structure does not carry
[firmware.rom-size]
0x00F3: note: baseboard@0x0002 (Baseboard or module information): string 4
holds whitespace only [string.blank]
0x103C: note: <table>: table has 290 bytes past the end-of-table structure
[table.trailing-data]
9 issues: 0 errors, 1 warning, 8 notes
```

The line starts with the offset of the issue within the data, followed by its severity. The location which comes next is the code of the structure type and the handle of the structure, and the attribute the issue belongs to if it has one; the translated name of the type follows in parentheses. Issues of the entry point

and of the table itself have no offset and are located at `<entry>` and `<table>` instead. The rule which has found the issue is named in brackets at the end of the line.

Issues have three severities. An **error** means a requirement of the specification is violated, and the data cannot be trusted. A **warning** means a recommendation is not followed. A **note** points out a peculiarity of the data which is not a defect in itself.

Selecting the rules

Rules are named *group.rule*, e.g. `entry.checksum` or `processor.cores`. The groups which apply to any structure are `entry`, `table`, `entity`, `string`, `value`, `link`, `overlay` and `quality`; the rest of the groups are named after the structure types the rules belong to. Every rule is described in the Rules section below.

--enable and **--disable** take either a full name or a group name, which stands for every rule in it. Naming rules with **--enable** leaves the rest of them out, and a rule named explicitly is checked even if it is disabled by default. Use **--list-rules** to see the rules along with their severities; the ones which are only checked with **--all-checks** say so.

Rules of the structure types belong to the specifications of those types, which is how an extension module can bring rules of its own along with the structures it describes.

Profiles

Severity depends on the profile the data is checked against. By default the data is assumed to have been read from a platform, where the defects of the firmware are nothing the reader can fix, and where a peculiarity is worth a note rather than a complaint. With **--producer** the same data is checked as data being written, where it is expected to conform to the specification because it is authored rather than observed, and several rules become errors.

Options

-?, -h, --help

Print help and exit.

-A, --all-checks

Check the rules which are disabled by default, either because they are a matter of taste rather than of the specification, or because they report what is unusual rather than what is wrong.

-e *rule*, --enable=*rule*

Check the named rule or group of rules, and nothing else. The option may be given several times, and a rule named here is checked even if it is disabled by default.

-d *rule*, --disable=*rule*

Skip the named rule or group of rules. The option may be given several times, and takes precedence over **--enable**.

-P, --producer

Check the data as if it was produced, not read, which makes the rules as strict as the specification is.

-S, --strict

Treat warnings as errors, so that the command fails on a warning as well.

-q, --quiet

Report the errors only, leaving the warnings and the notes out. They are still counted in the summary.

-R, --list-rules

List the rules with their descriptions and their severities in the profile **--producer** selects, and exit without checking anything. Listing the rules reads no SMBIOS data, so it works without access to the platform; the rules of the structure types are the ones of the modules which are enabled.

Rules

Rules are listed by their groups, in the order **--list-rules** lists them. Every rule is stated as what the data is expected to be, and an issue is reported where the data is not. The severity of a rule is given for the default profile and for **--producer**.

Entry point

`entry.checksum`

Severity: warning, or error with **--producer**.

Bytes of the entry point add up to zero, which is what its checksum is for. The entry point of SMBIOS 2.1 is checked twice, as a whole and for the intermediate anchor it carries, which has a checksum of its own.

`entry.length`

Severity: warning, or error with **--producer**.

Length the entry point declares is at least the length of its format, and is no longer than the data the entry point has been read from.

`entry.table-address`

Severity: warning, or error with **--producer**.

Entry point declares the address of the table, rather than an address of zero.

`entry.table-size`

Severity: warning, or error with **--producer**.

Table is as long as the entry point declares it. The entry point of SMBIOS 3.0 declares the maximum size of the table rather than its size, and the table is then no longer than that.

`entry.entity-count`

Severity: warning, or error with **--producer**.

Table holds as many structures as the entry point declares. Only the entry points of SMBIOS 2.1 and earlier declare the number.

`entry.entity-max-size`

Severity: warning, or error with **--producer**.

No structure is longer than the size of the largest structure the entry point declares.

Table

`table.truncated`

Severity: error.

Table ends after a whole structure, rather than in the middle of one.

`table.invalid-header`

Severity: error.

Headers of the structures have valid lengths, so that the table is read to its end. Structures are located by the lengths of the ones before them, so a header of an invalid length stops the reading: a header of no length, which stands for the end of the table, of another type than the end-of-table structure while data follows it, or a header of a length shorter than itself. The issue tells the offset of the header and how many bytes of the table are left unread.

`table.terminator`

Severity: warning, or error with **--producer**.

Table ends with an end-of-table structure.

`table.trailing-data`

Severity: note, or error with **--producer**.

No data follows the end-of-table structure within the table. The issue tells how many bytes follow it.

`table.missing-required`

Severity: warning, or error with **--producer**.

Table has a structure of every type the specification requires for the version the entry point declares. Since SMBIOS 2.3, these are the platform firmware information, the system information, the system enclosure, the processor, the cache, the physical memory array, the memory device, the memory array mapped address and the system boot information.

`table.missing-recommended`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Table has a structure of every type the specification recommends for the version the entry point declares. Since SMBIOS 2.3, these are the port connector information and the system slots.

`table.duplicate-singleton`

Severity: warning, or error with **--producer**.

Table has one structure at most of every type the specification allows only one of, which are the platform firmware information and the system information.

`table.reserved-handle`

Severity: warning, or error with **--producer**.

No structure has a handle of the range from 0xFF00 to 0xFFFF, which the specification reserves. The end-of-table structure is left out, since firmware commonly gives it a handle of that range.

Structures

`entity.below-minimum`

Severity: error.

Structure is at least as long as the minimum length of its type. A structure which is shorter is not decoded.

`entity.unknown-length`

Severity: warning, or error with **--producer**.

Structure ends where a version of the specification of its type ends it, rather than in the middle of a set of fields. The fields of such a structure are decoded up to where it ends.

`entity.undecoded`

Severity: warning, or error with **--producer**.

Structure of a type which has fields to decode has been decoded.

`entity.newer-fields`

Severity: note, or error with **--producer**.

Structure has no fields added by a version of the specification newer than the one the entry point declares.

`entity.newer-type`

Severity: warning, or error with **--producer**.

Type of the structure is defined by the version of the specification the entry point declares, or by an earlier one.

`entity.unknown-type`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Type of the structure is described by the specification or by an enabled module. The type of an OEM-specific structure, from 128 to 255, is known to the module of its vendor only, which may be disabled rather than missing.

`entity.obsolete`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Structure is not of a type the specification has made obsolete, which are the memory controller information, the memory module information and the on board devices information.

Strings

`string.too-long`

Severity: warning, or error with **--producer**.

No string is longer than 64 characters.

`string.non-printable`

Severity: warning, or error with **--producer**.

Strings hold printable characters only. A control character is data which has ended up in the strings rather than text. Characters above 0x7F are taken for text in an encoding other than ASCII.

`string.blank`

Severity: note, or warning with **--producer**.

No string holds whitespace only.

`string.padded`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

No string has leading or trailing whitespace.

`string.unreferenced`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Every string of a decoded structure is referenced by one of its fields, rather than taking up space nothing reads.

Values

Rules of the values check the fields of the decoded structures, including the ones of the nested structures and of the arrays. Values which stand for an unknown or an unspecified one are left out.

`value.invalid-enum`

Severity: warning, or error with **--producer**.

Fields of enumerated values hold values defined by a version of the specification. Enumerations which name some of the values only, e.g. the ones established by reverse engineering, are not checked.

`value.reserved`

Severity: note, or error with **--producer**.

Fields of enumerated values hold no values the specification reserves.

`value.reserved-bits`

Severity: note, or error with **--producer**.

Bit fields have no bits set which the specification reserves between the ones it defines. Bits above the highest one it defines are left to the vendors, e.g. the bits 32 to 63 of the firmware characteristics, and are not reported.

`value.bcd`

Severity: warning, or error with **--producer**.

Fields the specification encodes as binary-coded decimals hold a decimal digit in every nibble.

`value.range`

Severity: warning, or error with **--producer**.

Fields hold values within the limits the specification sets for them.

`value.jep106`

Severity: warning, or error with **--producer**.

Manufacturer identification codes of JEDEC lead to one of the nine banks JEP106 defines, and have an odd number of bits set in their identifier byte, whose highest bit is a parity bit. Codes copied from the SPD carry a parity bit in the number of continuation bytes as well, which is not counted.

`value.uuid`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

UUIDs identify something. A UUID of all zeroes stands for one which is not set, and a UUID of all ones for one which is not present: the specification allows both, but they identify nothing.

References

Rules of the references check the fields holding the handles of other structures. Handles 0xFFFE and 0xFFFF stand for no reference, and are left out.

`link.dangling`

Severity: warning, or error with **--producer**.

References point to structures of the table.

`link.wrong-type`

Severity: warning, or error with **--producer**.

References point to structures of the types the fields expect, e.g. the cache handles of a processor to caches.

`link.self`

Severity: warning, or error with **--producer**.

No structure references itself.

`link.orphan`

Severity: note. Checked with **--all-checks** only.

Structures which describe a part of another one are referenced by a structure of the table. These are the caches, the voltage, temperature and current probes, the management devices and the thresholds of the management devices.

Additional information

Rules of the additional information check the entries of the additional information structures, which put values in place of the fields of other structures.

`overlay.dangling`

Severity: warning, or error with **--producer**.

Entries refer to structures of the table.

`overlay.out-of-bounds`

Severity: warning, or error with **--producer**.

Entries refer to fields of the structures: the offset they refer to is past the header of the structure, and the value they carry ends within it.

`overlay.empty`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Entries carry a value or a string.

Data quality

`quality.placeholder`

Severity: note, or warning with **--producer**. Checked with **--all-checks** only.

Strings hold data rather than the placeholders firmware vendors leave in place, e.g. `To Be Filled By O.E.M.`, `Default string`, `System Serial Number` or `N/A`, which are compared regardless of case.

Structure types

Rules of the structure types are named after the codes of the types they belong to.

`firmware.rom-size`

Severity: warning, or error with **--producer**.

ROM size of 0xFF, which refers to the extended ROM size, is used only by the structures of SMBIOS 3.1 and later, which carry it.

`firmware.release-date`

Severity: note, or warning with **--producer**.

Release date is written as *mm/dd/yyyy*, the way the specification requires since SMBIOS 2.3, rather than with a year of two digits.

`processor.family`

Severity: warning, or error with **--producer**.

Processor family of 0xFE, which refers to the second processor family field, is used only by the structures of SMBIOS 2.6 and later, which carry it.

`processor.id`

Severity: note, or error with **--producer**.

Signature and feature flags of the identifier of an x86 processor are stored in the order of the specification, the signature first. Firmware of some vendors stores them the other way round, which the decoder puts back in place, and which is told by the bits the signature reserves.

`processor.cores`

Severity: warning, or error with **--producer**.

Enabled cores and threads are no more than the ones the processor has, and the processor has at least as many threads as cores.

`processor.speed`

Severity: note, or warning with **--producer**.

Current speed of the processor is no higher than its maximum speed.

`processor.cache`

Severity: warning, or error with **--producer**.

L1, L2 and L3 cache handles of the processor refer to caches of those levels.

`memory-controller.module-size`

Severity: warning, or error with **--producer**.

Installed and enabled sizes of the memory modules of the controller are no larger than the largest module the controller supports.

`cache.size`

Severity: warning, or error with **--producer**.

Installed size of the cache is no larger than its maximum size.

`cache.sram`

Severity: warning, or error with **--producer**.

Current SRAM type of the cache is one of the SRAM types it supports.

`slot.width`

Severity: warning, or error with **--producer**.

Physical width of the slot is at least the width of its bus, since a card as wide as the bus has to fit the slot.

`group-assoc.member`

Severity: note, or warning with **--producer**.

Members of the groups of well-known names are the structures the groups list, e.g. the members of the `Firmware Version Info` group are Intel firmware version information. A member which is not is placed by the vendor at a type number no enabled module knows of, or is listed by mistake. Checked for the groups of the enabled modules.

`system-event-log.area`

Severity: warning, or error with **--producer**.

Header and data of the log start within the area of the log.

`system-event-log.descriptors`

Severity: warning, or error with **--producer**.

Descriptors of the supported log types are 2 bytes long, which is the length the specification defines.

`memory-array.extended-capacity`

Severity: warning, or error with **--producer**.

Maximum capacity of 0x80000000, which refers to the extended maximum capacity, is used only by the structures long enough to carry it.

`memory-array.device-count`

Severity: warning, or error with **--producer**.

Number of the memory devices the array declares matches the number of the memory device structures referring to it, which are counted whether they are populated or not.

`memory-array.capacity`

Severity: warning, or error with **--producer**.

Sizes of the memory devices of the array add up to no more than its maximum capacity.

`memory-device.extended-size`

Severity: warning, or error with **--producer**.

Size of 0x7FFF, which refers to the extended size, is used only by the structures long enough to carry it.

`memory-device.width`

Severity: warning, or error with **--producer**.

Total width of the device, which covers the data bits and the bits of error correction, is at least its data width.

`memory-device.speed`

Severity: note, or warning with **--producer**.

Configured speed of the device is no higher than its maximum speed.

`memory-device.voltage`

Severity: warning, or error with **--producer**.

Configured voltage of the device is within its minimum and maximum voltages.

`memory-device.sizes`

Severity: warning, or error with **--producer**.

Volatile and non-volatile sizes of the device add up to no more than its size.

`memory-device.attributes`

Severity: note, or error with **--producer**.

Attributes of the device have none of the bits the specification reserves set: bits 7-4 up to SMBIOS 3.10, and bit 7 from it, which widens the rank to five bits and defines the flags of a disabled and of an unmapped device.

`memory-device.disabled`

Severity: warning, or error with **--producer**.

Device disabled because of an error, as SMBIOS 3.10 and later flag it, is not said to have no error: neither by the error information handle of 0xFFFF, nor by the memory error information structure it refers to reporting the OK kind of error, which is meant for the devices left out of the memory map.

`memory-array-address.range`

Severity: error.

Mapped address range starts no later than it ends.

`memory-array-address.overlap`

Severity: warning, or error with **--producer**.

Mapped address ranges of the memory arrays do not overlap, since they describe the physical address space of the system. Every pair of the ranges is reported once, by the structure of the higher handle.

`memory-device-address.range`

Severity: error.

Mapped address range starts no later than it ends.

`memory-device-address.bounds`

Severity: warning, or error with **--producer**.

Mapped address range of the device lies within the mapped address range of the memory array it belongs to.

`portable-battery.sbds`

Severity: note, or warning with **--producer**.

Battery carries either its own serial number and chemistry or the SBDS ones, rather than both, since the specification puts the SBDS values in place of the others.

`system-reset.limit`

Severity: warning, or error with **--producer**.

Number of the resets is no higher than the limit of them.

`power-controls.bcd`

Severity: warning, or error with **--producer**.

Month, day, hour, minute and second of the next scheduled power-on are binary-coded decimals. Fields which are not set hold 0xFF, and are left out.

`voltage-probe.range`

Severity: warning, or error with **--producer**.

Minimum value of the probe is no higher than its maximum value, and its nominal value is within them.

`cooling-device.probe`

Severity: warning, or error with **--producer**.

Temperature probe handle of the cooling device refers to a temperature probe.

`temperature-probe.range`

Severity: warning, or error with **--producer**.

Minimum value of the probe is no higher than its maximum value, and its nominal value is within them.

`current-probe.range`

Severity: warning, or error with **--producer**.

Minimum value of the probe is no higher than its maximum value, and its nominal value is within them.

`mgmt-device-threshold.order`

Severity: warning, or error with **--producer**.

Lower thresholds grow from the non-recoverable one to the critical and the non-critical ones, and the upper ones grow from the non-critical one to the critical and the non-recoverable ones. Thresholds left as a template are reported by `mgmt-device-threshold.template` instead.

`mgmt-device-threshold.template`

Severity: note, or error with **--producer**.

Thresholds hold values rather than the ordinals of the fields holding them, which firmware leaves in place from the template of the structure.

`memory-channel.load`

Severity: warning, or error with **--producer**.

Loads of the memory devices of the channel add up to no more than the maximum load it supports.

`ipmi-device.revision`

Severity: warning, or error with **--producer**.

Revision of the IPMI specification is a binary-coded decimal, e.g. 0x20 for IPMI 2.0.

`power-supply.probes`

Severity: warning, or error with **--producer**.

Voltage probe, cooling device and current probe handles of the power supply refer to structures of those types.

`onboard-device-ex.instance`

Severity: warning, or error with **--producer**.

No two onboard devices of the same type share an instance number, which is what tells them apart. Every pair is reported once, by the structure of the higher handle.

`mgmt-controller.records`

Severity: warning, or error with **--producer**.

Interface-specific data leaves room for the protocol records, and every protocol record fits the structure.

`tpm-device.vendor`

Severity: note, or error with **--producer**.

Vendor identifier is stored in the order of the specification, rather than as a little-endian double word, which starts with the terminating zero, e.g. `\0XFI` for `IFX`. The decoder puts the identifier back in place.

`tpm-device.version`

Severity: warning, or error with **--producer**.

Specification version of the device is a version of the TPM specifications the TCG has published, 1.x or 2.x.

`firmware-inventory.version`

Severity: warning, or error with **--producer**.

Version of the firmware is no older than the lowest supported version, when both are written in the same format other than a free-form one.

`intel-rsd-cabled-pcie.start-lane`

Severity: warning, or error with **--producer**.

Cable indices of an Intel RSD cabled PCIe port start at lane 0, 4, 8 or 12, since every cable carries a group of four lanes. Checked with the `intel-rsd` module enabled.

`intel-rsd-cabled-pcie.cable-count`

Severity: warning, or error with **--producer**.

Intel RSD cabled PCIe port declares no more than the four cable indices the specification lays out. Checked with the `intel-rsd` module enabled.

Exit status

0

Data has been checked and no error has been found.

1

An error has been found, or a warning in strict mode, or the data could not be read.

2

Command line is invalid.

Examples

Check the data of the local platform:

```
$ opendmi lint
```

Check a dump, which is how the data of another machine is checked:

```
$ opendmi -i smbios.bin lint
```

Check everything, reporting the peculiarities of the data along with its defects:

```
$ opendmi lint -A
```

Check the references between the structures only:

```
$ opendmi lint -e link -e overlay
```

Check everything except the complaints about the strings:

```
$ opendmi lint -A -d string
```

Check the data as if it was being written, and fail on anything the specification does not allow:

```
$ opendmi -i smbios.bin lint -P -S
```

List the rules and what they check:

```
$ opendmi lint -R
```

List the rules with the severities they have for the data being written:

```
$ opendmi lint -R -P
```

See also

`opendmi`, `opendmi-explain`, `opendmi-show`

list

Synopsis

opendmi [*global options*] **list** [*command options*] [*filter options*]

Description

The **list** command prints one line per structure, giving its handle, its type and the name of that type, in the order of the table. It is the way to find out what a table holds before asking `opendmi-show` for the details of a particular structure.

Which structures are listed is decided by the filter options, which are common to `opendmi-show` and `opendmi-export`. By default the standard and the OEM-specific structures are listed, and the inactive and the unknown ones are not.

Options

-?, -h, --help

Print help and exit.

-r, --raw

Print the fields separated by tabs rather than aligned in columns, so that the output is easy to read with `cut`(1) or `awk`(1). This is the default when the standard output is not a terminal.

Filter options

-H *handle*, --handle=*handle*

Only list the structures with the given handle, which is read as a hexadecimal number, with or without the `0x` prefix. The option may be given several times.

-t *type*, --type=*type*

Only list the structures of the given type, which is either a number or a code name, e.g. `4` or `processor`. The option may be given several times. See `opendmi-types` for the types and their codes.

-s, --standard

List the structures of the types the specification defines.

-S, --no-standard

Do not list the structures of the types the specification defines.

-e, --oem

List the OEM-specific structures, which are the ones of types 128 to 255.

-E, --no-oem

Do not list the OEM-specific structures.

-i, --inactive

List the structures which the firmware has marked as inactive (type 126).

-I, --no-inactive

Do not list the inactive structures.

-u, --unknown

List the structures of the types no enabled module describes, which are left as raw data.

-U, --no-unknown

Do not list the structures of the unknown types.

-m *module*, --module=*module*

Only list the structures described by the named module, whatever type numbers the platform places them at. The module has to be enabled, either for the platform or with the global **--module** option, or else nothing it describes is decoded. The option may be given several times.

-M, --all-modules

Only list the structures described by the enabled modules.

-a, --all

List every structure, including the inactive and the unknown ones.

Exit status

0

Structures have been listed.

1

Data could not be read.

2

Command line is invalid.

Examples

List the structures of the local platform:

```
$ opendmi list
```

List everything a dump holds, including the structures no module describes:

```
$ opendmi -i smbios.bin list -a
```

Count the memory devices:

```
$ opendmi list -t memory-device -r | wc -l
```

See also

`opendmi`, `opendmi-show`, `opendmi-types`

modules

Synopsis

opendmi [*global options*] **modules** [*command options*]

Description

The **modules** command lists the extension modules the utility was built with, giving the name of each module and the vendor it covers.

A module describes the OEM-specific structures of a vendor, which are the ones of types 128 to 255. Those types mean different things to different vendors, so a module is only enabled for the platforms of its vendor: automatically, as the data is read, or with the global **--module** option. The structures no enabled module describes are shown as raw data. The global **--no-auto-modules** option leaves the modules of the platform disabled.

Some modules depend on the product line and the generation of the platform too, which are told from the

system information: the **hpe** module is enabled for HP and HPE servers only, and decodes the structures of each generation by their own layout.

The **intel** module describes the structures of the Intel reference code, which the firmware of any vendor may carry, and is enabled for any platform with Intel processors. The modules of the vendors take precedence over it where they give the same types to structures of their own. The **intel-rsd** module describes the structures of Intel Rack Scale Design, found on the platforms built to it only, and is enabled with **--module**.

The name printed here is what **--module** takes, both as a global option which enables a module and as a filter option which limits the output to the structures that module describes. Use `opendmi-types` to see the types a module adds.

The command reads no SMBIOS data, so it works without access to the platform.

Options

-?, -h, --help
: Print help and exit.

-r, --raw
: Print the fields separated by tabs rather than aligned in columns, so that the output is easy to read with `cut`(1) or `awk`(1). This is the default when the standard output is not a terminal.

Exit status

0
: Modules have been listed.

2
: Command line is invalid.

Examples

List the modules available:

```
$ opendmi modules
```

Decode the structures of a vendor with the module for it, when the data does not tell the vendor:

```
$ opendmi -m lenovo show
```

Show the structures that module describes, and nothing else:

```
$ opendmi -m lenovo show -m lenovo
```

See also

`opendmi`, `opendmi-show`, `opendmi-types`

show

Synopsis

opendmi [*global options*] **show** [*command options*] [*filter options*]

Description

The **show** command decodes the SMBIOS structures and prints their fields in human-readable form, one structure after another, in the order of the table.

Every field is decoded to its meaning rather than to the bytes holding it: enumerations are named, bit sets are listed bit by bit, sizes and speeds carry their units, and versions are formatted according to the fields which define them. Handles are resolved into references to the structures they point to, so that a cache of a processor or an array of a memory device is named rather than numbered.

Structures which cannot be decoded completely are still shown, along with whatever has been read from them; **--verbose** tells such structures apart.

Strings come from the firmware as they are, so the control characters they hold, which a terminal would take for commands, are shown as escapes: `\xNN` for the C0 control characters (tabs and line breaks included) and DEL, `\u00NN` for the C1 control characters, and `\xNN` for every byte which is not valid UTF-8. Backslashes are shown as they are. `opendmi-export` writes the strings as the output format represents them.

Which structures are shown is decided by the filter options, which are common to `opendmi-list` and `opendmi-export`. By default the standard and the OEM-specific structures are shown, and the inactive and the unknown ones are not.

Options

-?, -h, --help

Print help and exit.

-q, --quiet

Hide the meta-data of the structures and the references between them, leaving the decoded fields alone.

-V, --verbose

Show the handle, the type, the length and the version of every structure, along with the state of its decoding: `decoded` for a structure which has been read completely, `incomplete` for one which matches no version of its specification, and `partial` for one which has been decoded only in part.

-D, --dump

Show the raw bytes of the structures instead of decoding them.

Filter options

-H *handle*, --handle=*handle*

Only show the structures with the given handle, which is read as a hexadecimal number, with or without the `0x` prefix. The option may be given several times.

-t *type*, --type=*type*

Only show the structures of the given type, which is either a number or a code name, e.g. `4` or `processor`. The option may be given several times. See `opendmi-types` for the types and their codes.

-s, --standard

Show the structures of the types the specification defines.

-S, --no-standard

Do not show the structures of the types the specification defines.

-e, --oem

Show the OEM-specific structures, which are the ones of types 128 to 255.

-E, --no-oem

Do not show the OEM-specific structures.

-i, --inactive

Show the structures which the firmware has marked as inactive (type 126).

-I, --no-inactive

Do not show the inactive structures.

-u, --unknown

Show the structures of the types no enabled module describes, which are left as raw data.

-U, --no-unknown

Do not show the structures of the unknown types.

-m *module*, --module=*module*

Only show the structures described by the named module, whatever type numbers the platform places them at. The module has to be enabled, either for the platform or with the global **--module** option, or else nothing it describes is decoded. The option may be given several times.

-M, --all-modules

Only show the structures described by the enabled modules.

-a, --all

Show every structure, including the inactive and the unknown ones.

Exit status

0

Data has been shown.

1

Data could not be read, or the output could not be written.

2

Command line is invalid.

Examples

Show every structure of the local platform:

```
$ opendmi show
```

Show the memory devices only:

```
$ opendmi show -t memory-device
```

Show a single structure by its handle:

```
$ opendmi show -H 0x0010
```

Show the versions and the decoding states of the structures, to find the ones the firmware writes differently from the specification:

```
$ opendmi show -V
```

Decode the structures of a vendor, and show those alone:

```
$ opendmi -m dell show -m dell
```

See also

`opendmi`, `opendmi-export`, `opendmi-explain`, `opendmi-list`

types

Synopsis

opendmi [*global options*] **types** [*command options*]

Description

The **types** command lists the structure types the utility knows how to decode, giving the number of each type, the module which describes it, its code name and what it holds.

The code name is what the **--type** option of `opendmi-show`, `opendmi-list` and `opendmi-export` accepts in place of the number, and what `opendmi-explain` takes as its argument. It is also the name the type is known by in the output of `opendmi-export` and in the reports of `opendmi-lint`.

Only the types the specification defines are listed by default. **--module** and **--all-modules** list the types of the extension modules in place of those, and **--all** lists everything at once. The command reads no SMBIOS data, so it works without access to the platform.

Options

-?, -h, --help

Print help and exit.

-m ***module*****, --module=***module*

List the types described by the named module instead of the standard ones. The option may be given several times. See `opendmi-modules` for the modules available.

-M, --all-modules

List the types described by every module instead of the standard ones.

-a, --all

List every type the utility knows, the standard ones and the ones of every module alike.

-r, --raw

Print the fields separated by tabs rather than aligned in columns, so that the output is easy to read with `cut`(1) or `awk`(1). This is the default when the standard output is not a terminal.

Exit status

0

Types have been listed.

2

Command line is invalid.

Examples

List the types the specification defines:

```
$ opendmi types
```

List every type, including the OEM-specific ones:

```
$ opendmi types -a
```

List the types a particular vendor adds:

```
$ opendmi types -m dell
```

Find the number of a type by its code:

```
$ opendmi types -a -r | grep memory-device
```

See also

`opendmi`, `opendmi-explain`, `opendmi-modules`, `opendmi-show`

Modules

Extension modules

Types 128 to 255 are left by the specification to the vendors, which give them to structures of their own, so the same type means different things on the platforms of different vendors. The structures of these types are described by the extension modules, one for each vendor, which are enabled for the platforms of their vendors as the data is read, or with the global `--module` option. Structures no enabled module describes are shown as raw data. The global `--no-auto-modules` option leaves the modules of the platform disabled, and the modules the tool is built with are listed by `opendmi modules`.

The platform is told from the vendor of the firmware, of the system, of the baseboard and of the processors, which are given by the structures of types 0, 1, 2 and 4, and from the name of the product of the system.

Module	Vendor	Enabled for
`acer`	Acer	Firmware or systems of Acer
`ami`	AMI	Firmware of AMI
`apple`	Apple	Firmware of Apple
`dell`	Dell	Firmware of Dell
`hpe`	HP and HPE	Servers of HP and HPE: ProLiant, Apollo, Synergy and Edgeline
`intel`	Intel reference code	Platforms with Intel processors
`intel-rsd`	Intel Rack Scale Design	Enabled with `--module` only
`lenovo`	IBM and Lenovo	Firmware or system of IBM or Lenovo
`sun`	Sun	Enabled with `--module` only

Structures a module describes are listed by `opendmi types --module`, and their layouts are described in the part of this manual about the specifications. The name of a module is what the `--module` option takes, both as a global option, which enables the module, and as a filter option of `show`, `list` and `export`, which limits the output to the structures the module describes.

Modules of several vendors

Two modules may describe structures of the same type, e.g. the module of a vendor and the one of the Intel reference code, which the firmware of any vendor may carry. The modules enabled with `--module` take precedence over the ones enabled for the platform, and a module enabled for the platform is left disabled if it describes a type another enabled module describes too. The `intel` module gives way to the other modules instead, and describes only the types they leave free.

Some vendors place the structures of the Intel reference code at types other than the original ones, since those are taken by structures of their own. The modules of these vendors move the structures of the `intel` module to the types the vendor places them at, or leave them out if the platforms of the vendor never carry them.

Some types are given by a vendor to several structures, which are told from each other by the data they hold, e.g. by a signature at a known offset, and a structure which matches none of them is left as raw data.

acer

Structures of the Acer laptops, whose firmware is often made by other vendors, e.g. Insyde, so the module is enabled by the vendor of either the firmware or the system.

Type	Code	Name
170	`acer-hotkeys`	Acer hotkey functions
171	`acer-devices`	Acer device list

ami

Module of the firmware of AMI. It describes no structures yet, and is reserved for the ones the firmware of AMI adds.

apple

Structures of the firmware of Macs, laid out as the `AppleSmBios.h` header of OpenCore describes them. The module is enabled by the vendor of the firmware, which OpenCore sets to Apple on the other systems too. Apple gives types 130 and 131 to structures of its own, which the Intel structures of the same types are told apart from by their signatures.

Type	Code	Name
128	`apple-firmware-volume`	Apple firmware volume information
130	`apple-memory-spd-data`	Apple memory SPD data
131	`apple-processor-type`	Apple processor type information
132	`apple-processor-bus-speed`	Apple processor bus speed information
133	`apple-platform-feature`	Apple platform feature information
134	`apple-smc-version`	Apple SMC version information

dell

Structures of the Dell platforms, which the module is enabled for by the vendor of either the firmware or the system, and by the firmware of Unisys, which carries the firmware of Dell on the servers Dell makes for Unisys. Dell gives types 221 and 222 to structures of its own, and places the structures of the Intel reference code 16 types lower: the management engine interface at type 203, the firmware version information at type 205 and the milestones of the Silicon View Technology at type 206. Some platforms, e.g. Precision Tower 3620, place them 16 types higher instead: the management engine interface at type 235 and the firmware version information at type 237.

Type	Code	Name
177	`dell-bios-flags`	Dell BIOS flags

Type	Code	Name
178	`dell-hotkeys`	Dell hotkeys
208	`dell-revisions`	Dell revisions and IDs
209	`dell-parallel-port`	Dell parallel port
210	`dell-serial-port`	Dell serial port
211	`dell-infrared-port`	Dell infrared port
212	`dell-indexed-io`	Dell indexed IO
213	`dell-protected-area-1`	Dell protected area type 1
214	`dell-protected-area-2`	Dell protected area type 2
216	`dell-video-rom`	Dell video BIOS information
218	`dell-calling-iface`	Dell calling interface
219	`dell-device-bay`	Dell device bay
220	`dell-token-refs-1`	Dell token references, type 1
221	`dell-token-refs-2`	Dell token references, type 2
223	`dell-memory-ids`	Dell memory module identifiers
225	`dell-device-names`	Dell device names
255	`dell-system-id`	Dell system ID record

hpe

Structures of the HP and HPE servers, which the module is enabled for when the name of the product of the system names a server line: ProLiant, Apollo, Synergy or Edgeline. Workstations and laptops of HP have structures of their own, which the module does not describe.

Some structures have changed their layouts over the generations of the servers, which are told from the name of the product: `G1` to `G7` up to G7, and `Gen8` onwards, with `Plus` for the refreshed ones, e.g. `ProLiant DL360 Gen10 Plus`. Structures of these types, e.g. type 226, are decoded by the layout of the generation of the server. Type 245 has two layouts too, which are told by the type of the board the structure describes, a riser of a server or one of an MHS platform. The servers give types 219 and 221 to structures of their own, and never carry the structures of the Intel reference code, which are left out.

Type	Code	Name
193	`hpe-rom-info`	HP/HPE other ROM information
194	`hpe-super-io`	HP/HPE Super I/O enable/disable indicator
195	`hpe-system-id`	HP/HPE server system ID
197	`hpe-processor`	HP/HPE processor specific information
199	`hpe-microcode`	HP/HPE CPU microcode patch support information

Type	Code	Name
202	`hpe-dimm-location`	HP/HPE DIMM location record
203	`hpe-device-correlation`	HP/HPE device correlation record
204	`hpe-rack-locator`	HP/HPE system/rack locator
209	`hpe-pxe-nic`	HP/HPE BIOS PXE NIC PCI and MAC information
211	`hpe-tcontrol`	HP/HPE processor TControl information
212	`hpe-cru`	HP/HPE 64-bit CRU information
216	`hpe-version`	HP/HPE version indicator record
219	`hpe-proliant-info`	HP/HPE ProLiant information
221	`hpe-iscsi-nic`	HP/HPE BIOS iSCSI NIC PCI and MAC information
224	`hpe-trusted-module`	HP/HPE trusted module status
226	`hpe-physical-attrs-legacy`	HP/HPE physical attribute information
226	`hpe-physical-attrs`	HP/HPE physical attribute information
229	`hpe-reserved-memory`	HP/HPE reserved memory location
230	`hpe-power-supply`	HP/HPE power supply information
232	`hpe-dimm-attrs`	HP/HPE DIMM attributes record
233	`hpe-nic-mac`	HP/HPE NIC PCI and MAC information
236	`hpe-backplane`	HP/HPE HDD backplane FRU information
237	`hpe-dimm-vendor`	HP/HPE DIMM vendor information
238	`hpe-usb-port`	HP/HPE USB port connector correlation record
239	`hpe-usb-device`	HP/HPE USB device correlation record
240	`hpe-inventory`	HP/HPE firmware inventory record
242	`hpe-drive`	HP/HPE hard drive inventory record
244	`hpe-dimm-config`	HP/HPE DIMM current configuration record
245	`hpe-riser`	HP/HPE extension board inventory record
245	`hpe-mhs-riser`	HP/HPE extension board inventory record

intel

Structures of the Intel reference code, which the firmware of the platforms with Intel processors carries whatever its vendor is. The module gives way to the modules of the vendors, which give the same types to structures of their own, and describes only the types they leave free.

The firmware lists some of these structures in group associations (type 14) of well-known names: `$MEI` for the management engine interface, `Firmware Version Info` for the firmware version information and `Intel® Silicon View Technology` for the milestones of the Silicon View Technology. `opendmi lint`

checks that the members of these groups are the structures the names stand for, wherever the vendor places them.

Type	Code	Name
130	`intel-amt`	Intel Active Management Technology information
131	`intel-vpro`	Intel vPro information
219	`intel-mei`	Intel Management Engine interface information
221	`intel-fvi`	Intel firmware version information
222	`intel-svt`	Intel Silicon View Technology milestones
222	`intel-svt-aligned`	Intel Silicon View Technology milestones, aligned layout

intel-rsd

Structures of Intel Rack Scale Design, which describe the devices of a system to the management software of the rack. They are found only on the platforms built to the specification, and take the types the vendors give to structures of their own, so the module is enabled with `--module` only.

Type	Code	Name
190	`intel-rsd-network-card`	Intel RSD network card information
192	`intel-rsd-pcie`	Intel RSD PCIe information
193	`intel-rsd-processor-cpuid`	Intel RSD processor CPUID information
194	`intel-rsd-storage-device`	Intel RSD storage device information
195	`intel-rsd-tpm`	Intel RSD Trusted Platform Module (TPM) information
196	`intel-rsd-txt`	Intel RSD TXT information
197	`intel-rsd-memory-device-ex`	Intel RSD memory device extended information
198	`intel-rsd-fpga`	Intel RSD FPGA information
199	`intel-rsd-cabled-pcie`	Intel RSD cabled PCIe port information
200	`intel-rsd-phys-device-mapping`	Intel RSD physical device mapping information

lenovo

Structures of the IBM and Lenovo platforms. Types 135 and 140 are given to several structures, which are told from each other by the signatures they begin with: the OEM structures of known layouts by the signature along with their number and revision, and the rest of them, the OEM data structures, by the signature alone.

Type	Code	Name
131	`lenovo-tvt`	Lenovo ThinkVantage Technologies enablement

Type	Code	Name
135	`lenovo-device-presence`	Lenovo ThinkPad device presence detection
135	`lenovo-bay-io`	Lenovo bay I/O
135	`lenovo-mobile-oem`	Lenovo mobile PC OEM data
140	`lenovo-ecp`	Lenovo ThinkPad embedded controller program
140	`lenovo-oem`	Lenovo OEM data

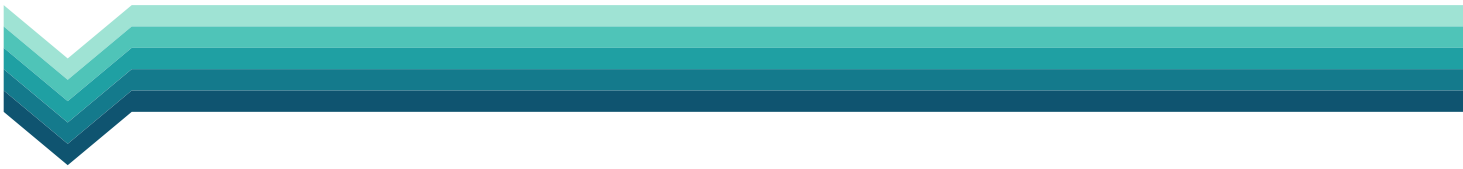
sun

Structures of the Sun servers, which extend the standard structures of the processors, the memory and the ports, and describe the PCI Express root complexes. The module is enabled with `--module` only.

Type	Code	Name
132	`sun-processor-ex`	Sun processor extended information
136	`sun-port-ex`	Sun port extended information
138	`sun-pcie-root-complex`	Sun PCI-express root complex information
144	`sun-memory-array-ex`	Sun memory array extended information
145	`sun-memory-device-ex`	Sun memory device extended information

PART IV

C library



<opendmi/defs.h>

dmi_member

Synopsis

```
#include <opendmi/defs.h>

typedef struct dmi_member_ref {
    size_t offset;
    size_t size;
} dmi_member_ref_t;

dmi_member_ref_t dmi_member(structure, member);
dmi_member_ref_t dmi_member_array(structure, member);

#define DMI_MEMBER_NULL ((dmi_member_ref_t){ 0, 0 })

bool dmi_member_is_present(dmi_member_ref_t ref);

size_t dmi_member_size(structure, member);
size_t dmi_member_offset(structure, member);
size_t dmi_element_size(structure, member);
size_t dmi_member_span(structure, first, last);
```

Description

Declarations of fields and attributes refer to the members of the decoded structure by `dmi_member_ref_t`, the offset and the size of the member, so that the library reads and writes them whatever the structure is.

`dmi_member`
: Reference to `member` of `structure`, of its own size.

`dmi_member_array`
: Reference to an array member, whose size is the size of an element: the member the pointer to the elements of an array is stored in, or the first element of a vector.

`DMI_MEMBER_NULL`
: Reference to no member, which the parameters left unset are, and the fields carrying no value have.

`dmi_member_is_present`
: Tells whether a reference refers to a member, i.e. it is not `DMI_MEMBER_NULL`.

`dmi_member_size`, **`dmi_member_offset`**, **`dmi_element_size`**
: Size and offset of a member, and size of an element of an array member or of the elements a pointer member points at.

`dmi_member_span`

Size of the consecutive members from `first` to `last`, inclusive.

The macros are evaluated at compile time, and are given types and member names rather than values.

Example

```
DMI_FIELD_GROUP(.present = dmi_member(dmi_hpe_trusted_module_t, has_chip)),
```

See also

`DMI_FIELD`, `DMI_ATTRIBUTE`

<opendmi/context.h>

dmi_add_extension

Synopsis

```
#include <opendmi/context.h>

bool dmi_add_extension(dmi_context_t *context, const dmi_module_t *module);
bool dmi_has_extension(const dmi_context_t *context, const dmi_module_t
*module);
```

Description

The `dmi_add_extension` function enables an extension module in a context, so that the structures of the types the module defines are decoded by its specifications. The modules of the vendors are declared in the headers under `<opendmi/module/>`, e.g. `dmi_hpe_module` in `<opendmi/module/hpe.h>`.

The specifications of the module are mapped to the type numbers they give, or to the ones the relocations of the enabled modules place them at, and the ones bound to a range of generations of a platform are mapped only if the generation of the platform lies within the range, see `dmi_get_platform`. Specifications told apart by their signatures share a type number, and the modules flagged with `DMI_MODULE_FLAG_YIELD` give way to the other enabled modules instead of conflicting with them.

The module is checked against the enabled ones before it is enabled, and is not enabled if a type number it maps a specification to is taken by another module. The structures decoded before are not decoded again, and keep the specifications they have been decoded by.

A module enabled this way stays enabled as the context is closed and opened again, until the context is destroyed. Its specifications are chosen again as the context is opened, once the platform is known, see `dmi_open` and `dmi_load`. The modules of the platform, which the context enables as it is opened when it has `DMI_CONTEXT_FLAG_AUTO_MODULES`, see `dmi_create`, stay enabled only until it is closed.

The `dmi_has_extension` function checks whether a module is enabled in a context, either with `dmi_add_extension` or for the platform as the context was opened. Modules are compared by address, so `module` is the very variable the module is declared as, e.g. `&dmi_hpe_module`.

Parameters

`context`

Context to enable the module in or to check, which may be open or closed.

`module`

Extension module to enable or to check.

Return value

The `dmi_add_extension` function returns `true` if the module has been enabled, and `false` otherwise. On failure, the enabled modules and the specifications chosen are left as they were.

The `dmi_has_extension` function returns `true` if the module is enabled, and `false` if it is not, or if `context` or `module` is `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised by `dmi_add_extension` if `module` is `nullptr`.

`DMI_ERROR_MODULE_CONFLICT`
: Module is enabled already, either with `dmi_add_extension` or for the platform as the context was opened, or its specifications have the same type numbers as the specifications of the enabled modules.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

The `dmi_has_extension` function raises no errors.

Examples

```
#include <opendmi/module/hpe.h>

if (!dmi_add_extension(context, &dmi_hpe_module) ||
    !dmi_load(context, "dump.bin")) {
    const dmi_error_t *error;

    while ((error = dmi_error_get_first(context)) != nullptr)
        fprintf(stderr, "%s\n", dmi_error_message(error->reason));
}
```

See also

`dmi_create`, `dmi_open`, `dmi_load`, `dmi_set_platform`, `dmi_get_platform`

dmi_close

Synopsis

```
#include <opendmi/context.h>

bool dmi_close(dmi_context_t *context);
```

Description

The `dmi_close` function releases the SMBIOS data of the context along with everything decoded from it, and leaves the context closed, so that it may be opened again with `dmi_open` or `dmi_load`.

Everything taken from the data refers to freed memory afterwards: the registry and the structures it holds,

and the strings and the binary data they refer to. The error queue, the logging handler and the extension modules of the context are kept.

Closing a context which is closed already does nothing.

Parameters

`context`
: Context to close.

Return value

Returns `true`, or `false` if `context` is `nullptr`.

Errors

None.

See also

`dmi_open`, `dmi_load`, `dmi_destroy`

dmi_create

Synopsis

```
#include <opendmi/context.h>

dmi_context_t *dmi_create(unsigned int flags);
```

Description

The `dmi_create` function creates a context, which holds the SMBIOS data it is given, read from the system with `dmi_open` or from a dump file with `dmi_load`, along with everything decoded from it, the errors raised while doing so, and the logging handler.

A context is created closed: it holds no data until it is opened, and may be opened again once it has been closed with `dmi_close`. The caller destroys it with `dmi_destroy`.

The flags choose how the data is decoded, and are combined with the bitwise OR:

`DMI_CONTEXT_FLAG_RELAXED`, which is zero, makes as much of the table available as possible: malformed structures are left undecoded, broken references and invalid additional information entries are skipped, and the errors are left in the error queue. A structure of an invalid length ends the table, and missing firmware information leaves the vendor unknown.

`DMI_CONTEXT_FLAG_STRICT` makes opening fail if the table has any error. Decoding and linking still go on after the first error, so that all of the errors are reported to the error queue.

`DMI_CONTEXT_FLAG_LINK` resolves the references between the structures once they are decoded, and

attaches string properties (type 46) to the structures they describe.

`DMI_CONTEXT_FLAG_OVERLAY` decodes the structures with the values of additional information entries (type 40) applied. Entries usually carry annotations with placeholder values rather than the values of fields, so this is not done by default.

`DMI_CONTEXT_FLAG_AUTO_MODULES` enables the extension modules of the platform as the context is opened, in addition to the ones enabled with `dmi_add_extension`, unless their types conflict with the types of the modules enabled before. The platform is told from the firmware and system information, see `dmi_get_platform`, and the modules enabled this way stay enabled until the context is closed.

A new context logs nothing, since it has no logging handler until `dmi_set_logger` gives it one. Its logging level is `DMI_LOG_DEBUG`.

Parameters

`flags`
: Flags of the context, `DMI_CONTEXT_FLAG_RELAXED` or a combination of the other `DMI_CONTEXT_FLAG_*` values.

Return value

On success, returns the context. On failure, returns `nullptr`, which happens when memory is exhausted only, and is not reported to any error queue, since there is no context to hold it.

Errors

None.

Examples

```
dmi_context_t *context = dmi_create(DMI_CONTEXT_FLAG_LINK);

if (context == nullptr)
    return false;

if (dmi_open(context, nullptr)) {
    ...
}

dmi_destroy(context);
```

See also

`dmi_destroy`, `dmi_open`, `dmi_load`, `dmi_close`, `dmi_set_logger`, `dmi_set_log_level`

dmi_destroy

Synopsis

```
#include <opendmi/context.h>

void dmi_destroy(dmi_context_t *context);
```

Description

The `dmi_destroy` function closes the context if it is open, see `dmi_close`, clears its error queue, and frees it along with everything it holds.

Everything taken from the context refers to freed memory afterwards: its registry and the structures it holds, the strings and the binary data they refer to, and the errors taken from its queue. The logging handler given with `dmi_set_logger` belongs to the caller, and is left as it is.

Parameters

`context`
: Context to destroy, or `nullptr`, which is ignored.

Errors

None.

See also

`dmi_create`, `dmi_close`

dmi_get_flags

Synopsis

```
#include <opendmi/context.h>

typedef enum dmi_context_flags {
    DMI_CONTEXT_FLAG_RELAXED      = 0,
    DMI_CONTEXT_FLAG_STRICT       = (1 << 0),
    DMI_CONTEXT_FLAG_LINK         = (1 << 1),
    DMI_CONTEXT_FLAG_OVERLAY      = (1 << 2),
    DMI_CONTEXT_FLAG_AUTO_MODULES = (1 << 3)
} dmi_context_flags_t;

unsigned dmi_get_flags(const dmi_context_t *context);
void dmi_set_flags(dmi_context_t *context, unsigned flags);
```

Description

The flags of a context choose how the data it is given is decoded. They are first given to `dmi_create`, and are combined with the bitwise OR.

The `dmi_get_flags` function returns the flags of the context.

The `dmi_set_flags` function replaces the flags of the context with `flags`, as they are: the flags are not checked, and the ones the library does not know are kept and returned by `dmi_get_flags`. To set or clear a single flag, the caller combines it with the flags `dmi_get_flags` returns.

The flags are read as the context is opened with `dmi_open` or `dmi_load`, and as its structures are read anew by `dmi_anonymize_context`. The structures of an open context are not decoded again when the flags change, so new flags take effect the next time the context is opened.

Flags

`DMI_CONTEXT_FLAG_RELAXED`

Zero, the default mode, in which as much of the table as possible is made available: malformed structures are left undecoded, broken references and invalid additional information entries are skipped, and the errors are left in the error queue. A structure of an invalid length ends the table, and missing firmware information leaves the vendor unknown.

`DMI_CONTEXT_FLAG_STRICT`

Makes opening fail if the table has any error: a structure of an invalid length, missing firmware information, a malformed structure, a broken reference or an invalid additional information entry. Decoding and linking still go on after the first error, so that all of the errors are reported to the error queue.

`DMI_CONTEXT_FLAG_LINK`

Resolves the references between the structures once they are decoded, and attaches string properties (type 46) to the structures they describe.

`DMI_CONTEXT_FLAG_OVERLAY`

Decodes the structures with the values of additional information entries (type 40) applied. Entries usually carry annotations with placeholder values rather than the values of fields, so this is not done by default.

`DMI_CONTEXT_FLAG_AUTO_MODULES`

Enables the extension modules of the platform as the context is opened, in addition to the ones enabled with `dmi_add_extension`, unless their types conflict with the types of the modules enabled before. The modules enabled this way stay enabled until the context is closed.

Parameters

`context`

Context to get or set the flags of.

`flags`

New flags of the context, `DMI_CONTEXT_FLAG_RELAXED` or a combination of the other `DMI_CONTEXT_FLAG_*` values.

Return value

The `dmi_get_flags` function returns the flags of the context, or 0 if `context` is `nullptr`, which cannot be told apart from `DMI_CONTEXT_FLAG_RELAXED`.

The `dmi_set_flags` function returns no value, and does nothing if `context` is `nullptr`.

Errors

None.

Examples

```
dmi_set_flags(context, dmi_get_flags(context) | DMI_CONTEXT_FLAG_LINK);
```

See also

`dmi_create`, `dmi_open`, `dmi_load`, `dmi_add_extension`

dmi_get_log_level

Synopsis

```
#include <opendmi/context.h>

dmi_log_level_t dmi_get_log_level(const dmi_context_t *context);
```

Description

The `dmi_get_log_level` function returns the least severe level of the messages the context writes to its handler, which has been set with `dmi_set_log_level`.

Parameters

`context`
: Context to query.

Return value

Returns the logging level, or `DMI_LOG_INVALID` if `context` is `nullptr`.

Errors

None.

See also

`dmi_set_log_level`, `dmi_log`

dmi_get_logger

Synopsis

```
#include <opendmi/context.h>

dmi_log_t *dmi_get_logger(const dmi_context_t *context);
```

Description

The `dmi_get_logger` function returns the handler the messages of the context are written to, which has been set with `dmi_set_logger`.

Parameters

`context`
: Context to query.

Return value

Returns the logging handler, or `nullptr` if the context has none or `context` is `nullptr`.

Errors

None.

See also

`dmi_set_logger`, `dmi_log`

dmi_get_platform

Synopsis

```
#include <opendmi/context.h>

const dmi_platform_t *dmi_get_platform(const dmi_context_t *context);
```

Description

The `dmi_get_platform` function returns the platform the structures of the vendor-specific types of a context are decoded for: the one set with `dmi_set_platform`, if any, or else the one the data of the context comes from.

The platform is told as the context is opened, from the firmware information (type 0), the system information (type 1), the baseboard information (type 2) and the processor information (type 4):

`firmware_vendor`
: Vendor of the platform firmware, `DMI_VENDOR_OTHER` if unknown.

`system_vendor`
: Vendor of the system, `DMI_VENDOR_OTHER` if unknown.

`baseboard_vendor`
: Vendor of the baseboard, `DMI_VENDOR_OTHER` if unknown.

`processor_vendor`
: Vendor of the processors, told by the first processor of a known vendor, since sockets may be empty. `DMI_VENDOR_OTHER` if unknown.

`product`
: Product name, `nullptr` if the table has no system information.

`family`
: Code of the product family, as the vendor defines it, e.g. `"server"` for HP and HPE servers, `nullptr` if the vendor tells no family.

`generation`
: Generation of the platform within its family, as the vendor numbers it, e.g. `DMI_HPE_GEN10`. Generations are ordered, so that a later one has a greater number. Zero means that the generation is unknown.

The vendors are told apart, since each of them adds structures of its own: e.g. the firmware of AMI is found on the boards of many vendors, and the structures of the Intel reference code on Intel platforms of any vendor. The vendor of the firmware tells the family and the generation from the product name, since the layouts of its structures depend on them. The extension modules of the platform are enabled for it when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`, and the specifications of the modules are chosen for its generation.

A closed context has no platform, unless one has been set with `dmi_set_platform`, which is kept as the context is opened and closed.

The platform belongs to the context, and is not destroyed by the caller. It is valid until the context is closed or the platform is set again.

Parameters

`context`
: Context to query.

Return value

Returns the platform, or `nullptr` if `context` is `nullptr`, or if the context is closed and no platform has been set.

Errors

None.

Examples

```
const dmi_platform_t *platform = dmi_get_platform(context);

if ((platform != nullptr) &&
    (platform->firmware_vendor == DMI_VENDOR_HPE) &&
    (platform->generation >= DMI_HPE_GEN10))
    ...
```

See also

`dmi_set_platform`, `dmi_platform_create`, `dmi_open`, `dmi_create`, `dmi_platform_match`, `dmi_platform_in_generations`

dmi_get_registry

Synopsis

```
#include <opendmi/context.h>

dmi_registry_t *dmi_get_registry(dmi_context_t *context);
```

Description

The `dmi_get_registry` function returns the registry of an open context, which holds the structures of its table, looked up by handle or by type.

The registry belongs to the context, and is valid until the context is closed.

Parameters

`context`
: Context to query.

Return value

Returns the registry, or `nullptr` if the context is not open or `context` is `nullptr`.

Errors

None.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                    DMI_TYPE(system), false);
dmi_system_t   *system   = dmi_entity_info(entity, DMI_TYPE(system));
```

See also

`dmi_open`, `dmi_load`, `dmi_close`, `dmi_entity_info`

dmi_is_open

Synopsis

```
#include <opendmi/context.h>

bool dmi_is_open(const dmi_context_t *context);
```

Description

The `dmi_is_open` function tells whether a context holds SMBIOS data: a structure table and the registry of its structures, which `dmi_open` and `dmi_load` fill the context with, and `dmi_close` frees.

The function is defined inline in the header, and reads the state of the context only.

Parameters

`context`
: Context in question, which must not be `nullptr`.

Return value

Returns `true` if the context is open, and `false` otherwise.

Errors

None.

See also

`dmi_open`, `dmi_load`, `dmi_close`

dmi_load

Synopsis

```
#include <opendmi/context.h>

bool dmi_load(dmi_context_t *context, const char *path);
```

Description

The `dmi_load` function reads a dump of SMBIOS data into a closed context, and decodes it the way `dmi_open` decodes the data of the system, so that the data of another machine is read on any system and without privileges.

Dump files are the ones `dmi_save` and `opendmi dump` write, and are interchangeable with the ones `dmidecode --dump-bin` writes and `dmidecode --from-dump` reads: the entry point, padded with zeros, followed by the table.

A context which fails to load the dump is left closed, with the errors telling why in its error queue. A context which is open already is left as it is.

Parameters

`context`

Context to open, which is required to be closed.

`path`

Path to the dump file.

Return value

Returns `true` if the dump has been loaded, and `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Path is `nullptr`.

`DMI_ERROR_STATE_INVALID`

Context is open already.

`DMI_ERROR_FILE_STAT_FAILED`

Dump file does not exist, or cannot be examined.

`DMI_ERROR_FILE_OPEN_FAILED`

Dump file cannot be opened.

`DMI_ERROR_DUMP_INVALID`

File is not a dump of SMBIOS data: it is not a regular file, or it is larger than an entry point followed by a table of `DMI_TABLE_MAX_SIZE` bytes, or it does not start with an entry point.

`DMI_ERROR_BACKEND_OPEN_FAILED`

Dump file cannot be read.

`DMI_ERROR_ENTITY_TRUNCATED`

Table area is empty, or a structure runs past its end.

`DMI_ERROR_FIRMWARE_INFO_NOT_FOUND`

Table holds no platform firmware information structure, in the strict mode only.

`DMI_ERROR_MODULE_CONFLICT`

Specifications of the modules enabled with `dmi_add_extension`, which apply to the platform, have the same type.

`DMI_ERROR_OUT_OF_MEMORY`

Memory is exhausted.

`DMI_ERROR_CONTEXT_OPEN_FAILED`

Context cannot be opened. Raised last on every failure but the one of a context which is open already, after the errors telling why.

Examples

```
dmi_context_t *context = dmi_create(DMI_CONTEXT_FLAG_LINK);

if (dmi_load(context, "smbios.bin")) {
    ...
}

dmi_destroy(context);
```

See also

`dmi_open`, `dmi_save`, `dmi_close`, `dmi_create`, `dmi_get_platform`, `dmi_error_get_first`

dmi_log

Synopsis

```
#include <opendmi/context.h>

bool dmi_log(dmi_context_t *context, dmi_log_level_t level,
             const char *format, ...);
```

Description

The `dmi_log` function writes a message, formatted the way `printf`(3) formats its arguments, to the handler of the context, see `dmi_set_logger`.

The message is dropped if its level is less severe than the level of the context, see `dmi_set_log_level`, or if the context has no handler. The handler may drop it by a level of its own.

Levels are ordered from the most severe one to the least: `DMI_LOG_ERROR`, `DMI_LOG_WARNING`, `DMI_LOG_NOTICE`, `DMI_LOG_INFO` and `DMI_LOG_DEBUG`.

The functions of the library write their messages this way, and so may the extensions and the applications using the context.

The macros `dmi_log_error`, `dmi_log_warning`, `dmi_log_notice`, `dmi_log_info` and `dmi_log_debug` write a message of the level they are named after, and take the same arguments but the level.

Parameters

`context`
Context to write the message to.

`level`
Level of the message.

`format`
Format of the message, followed by the values it refers to.

Return value

Returns `true` if the message has been given to the handler, and `false` otherwise.

Errors

None.

Examples

```
dmi_log_warning(context, "Unexpected value 0x%02x at offset 0x%02zx",
                value, offset);
```

See also

`dmi_set_logger`, `dmi_set_log_level`, `dmi_log_message`

dmi_open

Synopsis

```
#include <opendmi/context.h>

bool dmi_open(dmi_context_t *context, const char *device);
```

Description

The `dmi_open` function reads the SMBIOS data of the system the program runs on into a closed context, through the backend of the platform, and decodes it the way the flags of the context say, see `dmi_create`:

The entry point is read and decoded, on the platforms which give access to it. The structures of the table are found, and the ones of the types the context knows are decoded, with additional information applied when the context has `DMI_CONTEXT_FLAG_OVERLAY`. The references between the structures are resolved when it has `DMI_CONTEXT_FLAG_LINK`.

Reading the data of a platform usually requires privileges, because the table is taken from physical memory or from an interface the kernel restricts. A dump of the data is read with `dmi_load` instead, with the privileges of the file itself.

The structures of the types the vendors define are decoded by the extension modules of the vendors. Before the rest of the table is decoded, the platform is told from the firmware information (type 0) and the system information (type 1), see `dmi_get_platform`. The modules of the platform are then enabled when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`, and the specifications of the enabled modules, including the ones added with `dmi_add_extension` beforehand, are chosen for the platform and its generation.

The context stays open until it is closed with `dmi_close`, and the structures are reached through its registry, see `dmi_get_registry`.

A context which fails to open is left closed, with the errors telling why in its error queue. A context which is open already is left as it is.

Parameters

`context`

Context to open, which is required to be closed.

`device`

Path to the memory device the table is read from on the platforms which read physical memory. It is not used at the moment, since every backend reads the data from where its platform provides it, and is passed as `nullptr`.

Return value

Returns `true` if the context has been opened, and `false` otherwise.

Errors

`DMI_ERROR_STATE_INVALID`

Context is open already.

`DMI_ERROR_BACKEND_OPEN_FAILED`

Data of the platform cannot be accessed, e.g. for the lack of privileges.

`DMI_ERROR_ENTITY_TRUNCATED`

Table area is empty, or a structure runs past its end.

`DMI_ERROR_FIRMWARE_INFO_NOT_FOUND`
: Table holds no platform firmware information structure, in the strict mode only.

`DMI_ERROR_MODULE_CONFLICT`
: Specifications of the modules enabled with `dmi_add_extension`, which apply to the platform, have the same type.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

`DMI_ERROR_CONTEXT_OPEN_FAILED`
: Context cannot be opened. Raised last on every failure but the one of a context which is open already, after the errors telling why.

Examples

```
if (!dmi_open(context, nullptr)) {
    const dmi_error_t *error;

    while ((error = dmi_error_get_first(context)) != nullptr)
        fprintf(stderr, "%s\n", dmi_error_message(error->reason));
}
```

See also

`dmi_load`, `dmi_close`, `dmi_create`, `dmi_get_registry`, `dmi_get_platform`, `dmi_error_get_first`

dmi_save

Synopsis

```
#include <opendmi/context.h>

typedef enum dmi_save_flags {
    DMI_SAVE_FLAG_OVERWRITE = (1 << 0),
    DMI_SAVE_FLAG_ANONYMIZE = (1 << 1)
} dmi_save_flags_t;

bool dmi_save(dmi_context_t *context, const char *path, unsigned flags);
```

Description

The `dmi_save` function writes the SMBIOS data of an open context into a dump file, which `dmi_load` reads back.

The dump is interchangeable with the ones `dmidecode --dump-bin` writes: the entry point, padded with zeros to the largest size of an entry point, followed by the table. The table is written the way it has been

read, and the address of the table the entry point holds is set to its offset within the file. A context read on a platform which gives no access to the entry point has none, and a 64-bit entry point is made up for it.

An existing file is replaced only with `DMI_SAVE_FLAG_OVERWRITE`. A dump into a regular file, existing or not, is written to a temporary file in the same directory, which is renamed over `path` only once it is complete: a failure neither leaves an incomplete dump behind nor destroys the file being replaced, and the replaced file keeps its permissions. Since `path` is replaced rather than written to, the directory must be writable, and other hard links to the file keep its former contents. Devices, pipes and symbolic links, e.g. `/dev/stdout`, are written directly instead. Devices and pipes, and the symbolic links to them, are written whether `DMI_SAVE_FLAG_OVERWRITE` is set or not, since there is no file to keep, while a symbolic link to a file, or to nothing, is taken as an existing file.

With `DMI_SAVE_FLAG_ANONYMIZE`, the table written is the copy `dmi_anonymize` makes of it, in which the values identifying the system, e.g. serial numbers, asset tags, UUIDs and MAC addresses, are replaced. The copy has the length and the layout of the table, so the entry point holds for it as it is, and the context itself is left as it is.

Parameters

`context`
Open context to save.

`path`
Path to the dump file.

`flags`
Flags of `dmi_save_flags_t`, zero for none:

`DMI_SAVE_FLAG_OVERWRITE`
Replace the file if it exists, rather than failing.

`DMI_SAVE_FLAG_ANONYMIZE`
Write the anonymized copy of the table, see `dmi_anonymize`.

Return value

Returns `true` if the dump has been written, and `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
Path is `nullptr`.

`DMI_ERROR_STATE_INVALID`
Context is not open, or the table is to be anonymized and its structures carry additional information entries applied to them.

`DMI_ERROR_ENTRY_LENGTH_INVALID`
Entry point is longer than an entry point can be.

`DMI_ERROR_FILE_OPEN_FAILED`
File or its temporary file cannot be created, or the file exists and `DMI_SAVE_FLAG_OVERWRITE` is not

set.

`DMI_ERROR_FILE_WRITE_FAILED`
: File cannot be written, or the temporary file cannot be renamed over it.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory for the anonymized copy cannot be allocated.

`DMI_ERROR_INTERNAL`
: Table cannot be anonymized, see `dmi_anonymize`.

See also

`dmi_load`, `dmi_anonymize`, `dmi_open`, `dmi_close`

dmi_set_log_level

Synopsis

```
#include <opendmi/context.h>

bool dmi_set_log_level(dmi_context_t *context, dmi_log_level_t level);
```

Description

The `dmi_set_log_level` function sets the least severe level of the messages the context writes to its handler. Messages of the levels less severe than `level` are dropped before they reach the handler, which may drop more of them by a level of its own.

Levels are ordered from the most severe one to the least: `DMI_LOG_ERROR`, `DMI_LOG_WARNING`, `DMI_LOG_NOTICE`, `DMI_LOG_INFO` and `DMI_LOG_DEBUG`.

The level of a new context is `DMI_LOG_DEBUG`, so that every message reaches the handler.

Parameters

`context`
: Context to set the level of.

`level`
: Least severe level of the messages written.

Return value

Returns `true`, or `false` if `context` is `nullptr` or `level` is `DMI_LOG_INVALID`.

Errors

None.

See also

`dmi_get_log_level`, `dmi_log`, `dmi_set_logger`

dmi_set_logger

Synopsis

```
#include <opendmi/context.h>

bool dmi_set_logger(dmi_context_t *context, dmi_log_t *logger);
```

Description

The `dmi_set_logger` function sets the handler the messages of the context are written to, see `dmi_log`. A context has no handler until one is set, and logs nothing.

The handler is a structure whose `handler` member is the function the messages are given to:

```
typedef void dmi_log_handler_fn(
        dmi_log_t       *target,
        dmi_log_level_t  level,
        const char      *format,
        va_list          args);

struct dmi_log
{
    dmi_log_handler_fn *handler;
};
```

The function receives the handler it belongs to, so that a handler needing data of its own, such as a level of its own or a file to write to, embeds `dmi_log_t` as the first member of a structure of its own, and takes the rest of it from `target`.

The handler belongs to the caller, which keeps it valid for as long as the context uses it, and is not freed along with the context.

Parameters

`context`
: Context to set the handler of.

`logger`
: Logging handler, or `nullptr` to stop logging.

Return value

Returns `true`, or `false` if `context` is `nullptr`.

Errors

None.

Examples

```
static void write_message(dmi_log_t *target, dmi_log_level_t level,
                          const char *format, va_list args)
{
    (void)target;

    fprintf(stderr, "%s: ", dmi_log_level_name(level));
    vfprintf(stderr, format, args);
    fputc('\n', stderr);
}

static dmi_log_t logger = { write_message };

dmi_set_logger(context, &logger);
```

See also

`dmi_get_logger`, `dmi_log`, `dmi_set_log_level`

dmi_set_platform

Synopsis

```
#include <opendmi/context.h>

bool dmi_set_platform(dmi_context_t *context, const dmi_platform_t *platform);
```

Description

The `dmi_set_platform` function sets the platform the structures of a context are decoded for, instead of the one told from its data as it is opened, see `dmi_get_platform`. This allows decoding the data of a platform whose firmware or system information is missing or wrong, and decoding structures of a known generation without opening the context.

The context keeps a copy of the platform, which it destroys when the platform is set again or the context is destroyed, so the caller still owns `platform` and destroys it with `dmi_platform_destroy`. The platform set is kept as the context is opened and closed. Passing `nullptr` makes the context use the platform told from its data again, at once if it is open.

The specifications of the enabled modules are chosen for the platform again at once. The structures decoded before are not decoded again, and keep the specifications they have been decoded by.

Parameters

`context`
: Context to update.

`platform`
: Platform to set, or `nullptr`.

Return value

Returns `true` on success, and `false` otherwise. On failure, the platform and the specifications chosen are left as they were.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

`DMI_ERROR_MODULE_CONFLICT`
: Specifications of the enabled modules, which apply to the platform, have the same type.

Examples

```
dmi_platform_t *platform = dmi_platform_create(context);
bool success = false;

if (platform != nullptr) {
    platform->firmware_vendor = DMI_VENDOR_HPE;
    platform->generation      = DMI_HPE_GEN10;

    success = dmi_platform_set_family(platform, DMI_HPE_FAMILY_SERVER) &&
              dmi_set_platform(context, platform) &&
              dmi_add_extension(context, &dmi_hpe_module);

    dmi_platform_destroy(platform);
}
```

See also

`dmi_get_platform`, `dmi_platform_create`, `dmi_open`, `dmi_load`

<opendmi/anonymize.h>

dmi_anonymize

Synopsis

```
#include <opendmi/anonymize.h>

bool dmi_anonymize(dmi_context_t *context, dmi_buffer_t *table);
bool dmi_anonymize_context(dmi_context_t *context);
```

Description

The `dmi_anonymize` function writes into `table` a copy of the structure table of an open context, in which the values identifying the system rather than describing it are replaced, so that the data can be shared, e.g. in a bug report or in a collection of test data. `dmi_save` writes such a copy into a dump file with `DMI_SAVE_FLAG_ANONYMIZE`, and so does the **--anonymize** option of `opendmi-dump`.

The values replaced are the ones of the attributes marked with `DMI_ATTRIBUTE_FLAG_PRIVATE` in the specifications of the structures: serial numbers, asset tags, UUIDs, MAC addresses and the like, of the standard structures and of the vendor-specific ones the enabled modules decode.

The copy has the very layout of the table, so that it decodes the way the table does, and everything but the replaced values stays as it is:

- Every value is replaced with one of the same length. Strings keep their letters, digits and punctuation in place, e.g. a serial number of an upper case letter followed by digits is replaced with another one of that kind.
- MAC addresses keep their first three bytes, which tell the manufacturer of the device rather than the device.
- UUIDs keep their version and variant.
- Values which identify nothing are kept: empty and blank strings, the placeholders firmware vendors leave in place, e.g. `To Be Filled By O.E.M.`, values of one repeated byte, e.g. all zeros, and the values standing for "unspecified".

A value is replaced the same way wherever the table holds it, e.g. a service tag serving as the serial number of both the system and the chassis. A value which looks like an identifier, a single word holding digits, is also looked for in the rest of the table and replaced there, e.g. in the OEM strings, in the options of the system configuration, or in the bytes of a vendor-specific structure holding it in place. A value which does not, e.g. `Battery 0` put in place of the serial number of a battery, is replaced only where the attribute holds it, so that the names it happens to be part of are kept.

The replacements are made from the values by a keyed hash, SipHash-2-4, whose key is taken from the random number generator of the system each time a table is anonymized and is never saved. The same value is thus replaced differently in every copy, and the original values cannot be recovered by trying the ones a serial number may take.

The context itself is left as it is: its structures keep the values they have been decoded with.

The `dmi_anonymize_context` function makes the copy the same way, puts it in place of the table of the context, and reads the structures anew from it, so that everything decoded from the context afterwards, e.g. an export, holds the replaced values, as the **--anonymize** option of `opendmi-export` does. The copy has the very layout of the table, so the platform and the extension modules told from the table stay the same. The structures read before are destroyed, so the pointers to them and to the information decoded from them are no longer valid. If the copy cannot be made or read, the context is left as it is.

Parameters

`context`

Open context, whose structures are decoded.

`table`

Buffer to write the copy into. The data it holds is replaced, and dropped if the copy fails to be made. A call refused for its arguments or for the state of the context leaves the buffer as it is.

Return value

`dmi_anonymize` returns `true` if the copy has been made, and `dmi_anonymize_context` if the context has been anonymized; both return `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Table is `nullptr` (`dmi_anonymize` only).

`DMI_ERROR_STATE_INVALID`

Context is not open, or it has been opened with `DMI_CONTEXT_FLAG_OVERLAY`, whose structures are decoded from copies carrying additional information entries rather than from the table.

`DMI_ERROR_OUT_OF_MEMORY`

Memory cannot be allocated.

`DMI_ERROR_SYSTEM`

Random key cannot be generated.

`DMI_ERROR_INTERNAL`

Structure is not encoded back into its own length.

Notes

Values of the structures no enabled module decodes are replaced only where they repeat a value found elsewhere, so a table carrying unknown OEM structures may still identify its system. The same holds for free-form strings, e.g. OEM strings (type 11), which are not replaced unless they hold such a value. The tables to be shared are worth a look before they are.

See also

`dmi_save`, `dmi_load`, `opendmi-dump`, `opendmi-export`

<opendmi/types.h>

dmi_type

Synopsis

```
#include <opendmi/types.h>

typedef struct dmi_type {
    dmi_type_id_t id;
} dmi_type_t;

#define DMI_TYPE(x)  (&dmi_type_ ## x)
#define DMI_TYPE_ANY ((const dmi_type_t *)nullptr)

extern const dmi_type_t dmi_type_firmware;
extern const dmi_type_t dmi_type_system;
...
extern const dmi_type_t dmi_type_end_of_table;
```

Description

A `dmi_type_t` structure type tells what the information a structure is decoded into is, which is one C type, e.g. `dmi_system_t` for `DMI_TYPE(system)`. Types are told by their objects rather than by their numbers, and are the constants the library declares in `<opendmi/types.h>` for the types of the SMBIOS specification, and the extension modules declare in their headers for the types of the vendors, e.g. `dmi_type_hpe_processor` in `<opendmi/module/hpe.h>`. `DMI_TYPE()` names them: `DMI_TYPE(system)` is `&dmi_type_system`.

A type is not a type number, see `dmi_type_id`:

- Several specifications may decode the structures of one type, e.g. the layouts of the generations of a vendor, and the structures are of the type whatever layout the table holds.
- Several types may share a number, e.g. Intel vPro information, `DMI_TYPE(intel_vpro)`, and Lenovo ThinkVantage Technologies enablement, `DMI_TYPE(lenovo_tvt)`, which are both found at type 131.
- A structure of a type may be found at another number, where a vendor places it, e.g. Intel firmware version information at type 205 on Dell platforms, and is still of its type.

So `dmi_entity_info`, `dmi_entity_data`, the lookups of the registry and the types a handle may refer to take types, and give the structures of the type asked for only. `DMI_TYPE_ANY` stands for any type where a type is optional.

Members

`id`

Type number the specification of the type gives the structures, see `dmi_type_id`.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
DMI_TYPE(intel_fvi), true);

if (entity != nullptr)
    printf("found at type %d\n", dmi_entity_type_id(entity));
```

See also

`dmi_type_id`, `dmi_entity_type`, `dmi_entity_info`, `dmi_entity_data`

dmi_type_id

Synopsis

```
#include <opendmi/types.h>

typedef enum dmi_type_id {
    DMI_TYPE_ID_INVALID      = -1,
    DMI_TYPE_ID_FIRMWARE     = 0,
    DMI_TYPE_ID_SYSTEM       = 1,
    ...
    DMI_TYPE_ID_INACTIVE     = 126,
    DMI_TYPE_ID_END_OF_TABLE = 127
} dmi_type_id_t;

#define DMI_TYPE_ID(x) ((dmi_type_id_t)(DMI_TYPE_ID_ ## x))
#define DMI_TYPE_ID_MAX UINT8_MAX
```

Description

The `dmi_type_id_t` enumeration lists the numbers the SMBIOS specification gives its structure types: 0 through 127 are defined by the specification, and 128 through 255 are available for OEM- and system-specific information. The extension modules list the numbers of the vendor types in their headers, e.g. `DMI_TYPE_ID_HPE_PROCESSOR` in `<opendmi/module/hpe.h>`. `DMI_TYPE_ID()` names them: `DMI_TYPE_ID(SYSTEM)` is `DMI_TYPE_ID_SYSTEM`.

A number tells where a structure is found in the table, which `dmi_entity_type_id` gives, rather than what the structure is: vendors give the same OEM numbers to different structures, and place the structures of others at numbers of their own. What a structure is decoded into is told by its structure type, see `dmi_type`, whose `id` member is the number its specification gives it.

Numbers are for what the table itself holds: the number a structure is found at, the structure types of the table, which `dmi_type_spec` and `dmi_type_name` look up in the context, the types the options of the tool select, and the structures no specification describes, which `dmi_registry_lookup_first_id` finds.

Values

`DMI_TYPE_ID_INVALID`

No type number.

`DMI_TYPE_ID_FIRMWARE` to `DMI_TYPE_ID_END_OF_TABLE`

Numbers of the structure types of the SMBIOS specification.

See also

`dmi_type`, `dmi_type_name`, `dmi_type_spec`, `dmi_type_find`, `dmi_entity_type_id`

`<opendmi/platform.h>`

dmi_platform_clone

Synopsis

```
#include <opendmi/platform.h>

dmi_platform_t *dmi_platform_clone(const dmi_platform_t *platform);
```

Description

The `dmi_platform_clone` function creates a copy of a platform, which belongs to the same context, has the same vendor and generation, and holds copies of the product name and the family code. The copy is independent of the original, and is destroyed with `dmi_platform_destroy`.

Parameters

`platform`
: Platform to copy.

Return value

Returns the copy on success, or `nullptr` if `platform` is `nullptr` or memory is exhausted.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted, reported to the context of `platform`, if it has one.

See also

`dmi_platform_create`, `dmi_platform_destroy`

dmi_platform_create

Synopsis

```
#include <opendmi/platform.h>

dmi_platform_t *dmi_platform_create(dmi_context_t *context);
```

Description

The `dmi_platform_create` function creates a platform descriptor, which tells the vendor, the product,

the family and the generation of the platform the SMBIOS data comes from:

```
struct dmi_platform
{
    dmi_context_t *context;
    dmi_vendor_t   firmware_vendor;
    dmi_vendor_t   system_vendor;
    dmi_vendor_t   baseboard_vendor;
    dmi_vendor_t   processor_vendor;
    char          *product;
    char          *family;
    unsigned       generation;
};
```

The new platform has the vendors `DMI_VENDOR_OTHER`, no product name, no family and an unknown generation, which is zero. The vendors and the generation are plain values, which the caller sets directly, see `dmi_get_platform` for their meaning. The product name and the family code are owned by the platform, and are set with `dmi_platform_set_product` and `dmi_platform_set_family`, which copy them.

A context tells the platform of its data itself as it is opened, see `dmi_get_platform`. A platform created by the caller is given to a context with `dmi_set_platform`, which keeps a copy of it, so the caller destroys the platform with `dmi_platform_destroy` whenever it is done with it.

Parameters

`context`
: Context the platform belongs to, whose error queue its errors are raised against, or `nullptr`, in which case errors are not reported.

Return value

Returns the platform on success, or `nullptr` on failure.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

Examples

```
dmi_platform_t *platform = dmi_platform_create(context);

if (platform == nullptr)
    return false;

platform->firmware_vendor = DMI_VENDOR_HPE;
platform->generation      = DMI_HPE_GEN10;
```

```
bool success = dmi_platform_set_family(platform, DMI_HPE_FAMILY_SERVER) &&
               dmi_set_platform(context, platform);

dmi_platform_destroy(platform);
```

See also

`dmi_platform_clone`, `dmi_platform_destroy`, `dmi_platform_set_product`, `dmi_set_platform`, `dmi_get_platform`

dmi_platform_destroy

Synopsis

```
#include <opendmi/platform.h>

void dmi_platform_destroy(dmi_platform_t *platform);
```

Description

The `dmi_platform_destroy` function destroys a platform along with the product name and the family code it holds. It does nothing if `platform` is `nullptr`.

The platform of a context, returned by `dmi_get_platform`, belongs to the context, and is not destroyed by the caller.

Parameters

`platform`
: Platform to destroy, or `nullptr`.

Return value

None.

Errors

None.

See also

`dmi_platform_create`, `dmi_platform_clone`

dmi_platform_in_generations

Synopsis

```
#include <opendmi/platform.h>

bool dmi_platform_in_generations(const dmi_platform_t *platform,
                                 const dmi_generations_t *generations);
```

Description

The `dmi_platform_in_generations` function checks whether the generation of a platform lies within a range, which a structure specification applies to by its `params.generations` member:

```
struct dmi_generations
{
    unsigned minimum;
    unsigned maximum;
};
```

Both bounds are included in the range, and a zero bound leaves the range unbounded on its side. A range with both bounds zero applies to every platform. A bounded range never applies to a platform whose generation is unknown, since the layouts of different generations are told apart by their ranges only.

Parameters

`platform`
: Platform to check.

`generations`
: Range of generations, `nullptr` for an unbounded one.

Return value

Returns `true` if the range is unbounded, or if the generation of the platform is known and lies within the range, and `false` otherwise.

Errors

None.

Examples

```
const dmi_generations_t gen9_onwards = { .minimum = DMI_HPE_GEN9 };

if (dmi_platform_in_generations(dmi_get_platform(context), &gen9_onwards))
    ...
```

See also

`dmi_get_platform`, `dmi_platform_match`

dmi_platform_match

Synopsis

```
#include <opendmi/platform.h>

bool dmi_platform_match(const dmi_platform_t *platform,
                        const dmi_platform_match_t *match);
```

Description

The `dmi_platform_match` function checks a platform against a condition of an extension module, which lists the platforms it is enabled for by its `platforms` member:

```
struct dmi_platform_match
{
    dmi_vendor_t firmware_vendor;
    dmi_vendor_t system_vendor;
    dmi_vendor_t baseboard_vendor;
    dmi_vendor_t processor_vendor;
    const char  *family;
};
```

A platform satisfies a condition if it satisfies all of its members: each vendor of the condition is either `DMI_VENDOR_ANY` or the corresponding vendor of the platform, and the condition either names no family, or the family of the platform. `DMI_VENDOR_ANY` is zero, so the members a condition leaves out are satisfied by any platform, including one whose vendors are unknown. A list of conditions ends with `DMI_PLATFORM_NULL`, whose firmware vendor is `DMI_VENDOR_INVALID`.

Parameters

`platform`
: Platform to check.

`match`
: Condition to check against.

Return value

Returns `true` if the platform satisfies the condition, and `false` otherwise or if either argument is `nullptr`.

Errors

None.

Examples

```
const dmi_platform_match_t servers = {
    .firmware_vendor = DMI_VENDOR_HPE,
    .family          = DMI_HPE_FAMILY_SERVER
};

if (dmi_platform_match(dmi_get_platform(context), &servers))
    ...
```

See also

`dmi_get_platform`, `dmi_platform_in_generations`

dmi_platform_set_product

Synopsis

```
#include <opendmi/platform.h>

bool dmi_platform_set_product(dmi_platform_t *platform, const char *product);
bool dmi_platform_set_family(dmi_platform_t *platform, const char *family);
```

Description

The `dmi_platform_set_product` function sets the product name of a platform, and the `dmi_platform_set_family` function sets the code of its product family, as the vendor defines it, e.g. `DMI_HPE_FAMILY_SERVER`.

The platform holds copies of the strings, which it frees when they are set again or when it is destroyed, so the caller keeps its own. Passing `nullptr` clears the string.

The product name is told from the system information (type 1) as a context is opened, and the family by the vendor, from the product name. They are set by the caller for a platform given to a context with `dmi_set_platform`.

Parameters

`platform`
: Platform to update.

`product`
: Product name, or `nullptr`.

`family`
: Family code, or `nullptr`.

Return value

Returns `true` on success, and `false` if `platform` is `nullptr` or memory is exhausted, in which case the string is left as it was.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted, reported to the context of `platform`, if it has one.

See also

`dmi_platform_create`, `dmi_set_platform`

<opendmi/vendor.h>

dmi_vendor

Synopsis

```
#include <opendmi/vendor.h>

typedef enum dmi_vendor {
    DMI_VENDOR_INVALID = -1,
    DMI_VENDOR_ANY     = 0,
    DMI_VENDOR_OTHER,
    DMI_VENDOR_ACER,
    DMI_VENDOR_AMD,
    DMI_VENDOR_AMI,
    DMI_VENDOR_APPLE,
    DMI_VENDOR_CISCO,
    DMI_VENDOR_DELL,
    DMI_VENDOR_HONOR,
    DMI_VENDOR_HP,
    DMI_VENDOR_HPE,
    DMI_VENDOR_HUAWEI,
    DMI_VENDOR_IBM,
    DMI_VENDOR_INTEL,
    DMI_VENDOR_LENOVO,
    DMI_VENDOR_UNISYS
} dmi_vendor_t;

typedef bool dmi_vendor_detect_fn(dmi_platform_t *platform);

typedef struct dmi_vendor_spec {
    dmi_vendor_t          id;
    const char           *code;
    const char          **names;
    dmi_vendor_detect_fn *detect;
} dmi_vendor_spec_t;

const char *dmi_vendor_name(dmi_vendor_t vendor);
const dmi_vendor_spec_t *dmi_vendor_detect(const char *name);
```

Description

The `dmi_vendor_t` enumeration tells the vendors the library knows of, which the platform of a context is described by, see `dmi_get_platform`, and the extension modules are enabled for, see `dmi_platform_match`.

The `dmi_vendor_name` function returns the name of a vendor, e.g. `HPE`. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

The `dmi_vendor_detect` function tells the vendor from a vendor name the SMBIOS structures give, e.g. the vendor of the firmware or of the system. Each known vendor has a list of the names it is given by, e.g. `Hewlett Packard Enterprise` and `HPE`; `name` matches a vendor if it is equal to one of them, case and leading or trailing whitespace ignored, e.g. `LENOVO ` matches `Lenovo`. Any other spelling matches no vendor. `DMI_VENDOR_OTHER` has no names, and is never matched.

The `dmi_vendor_spec_t` structure describes a known vendor:

`id`
: Vendor identifier.

`code`
: Machine-readable code of the vendor, e.g. `hpe`.

`names`
: Names the vendor is given by, terminated by `nullptr`.

`detect`
: Handler which tells the family and the generation of a platform of the vendor from its product name, or `nullptr` if the vendor has no families. The handler is called as the context is opened for the vendor of the firmware, with the vendors and the product name of the platform set, and returns `false` if memory is exhausted, and `true` otherwise.

Values

`DMI_VENDOR_INVALID`
: No vendor, which terminates a list of platform conditions.

`DMI_VENDOR_ANY`
: Any vendor, in platform conditions only, see `dmi_platform_match`.

`DMI_VENDOR_OTHER`
: Vendor which the library does not know of, or which is not given.

`DMI_VENDOR_ACER`
: Acer.

`DMI_VENDOR_AMD`
: AMD.

`DMI_VENDOR_AMI`
: AMI (American Megatrends).

`DMI_VENDOR_APPLE`
: Apple.

`DMI_VENDOR_CISCO`
: Cisco.

`DMI_VENDOR_DELL`
: Dell.

`DMI_VENDOR_HONOR`
: Honor.

`DMI_VENDOR_HP`
: HP.

`DMI_VENDOR_HPE`
: HPE (Hewlett Packard Enterprise).

`DMI_VENDOR_HUAWEI`
: Huawei.

`DMI_VENDOR_IBM`
: IBM.

`DMI_VENDOR_INTEL`
: Intel.

`DMI_VENDOR_LENOVO`
: Lenovo.

`DMI_VENDOR_UNISYS`
: Unisys.

Parameters

`vendor`
: Vendor to name.

`name`
: Vendor name to tell the vendor from, or `nullptr` if the structure gives none, in which case no vendor is told.

Return value

The `dmi_vendor_name` function returns the name of the vendor, or `nullptr` if the value has no name: `DMI_VENDOR_INVALID`, `DMI_VENDOR_ANY`, or a value which is not a vendor. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

The `dmi_vendor_detect` function returns the specification of the vendor, which belongs to the library and is static, or `nullptr` if `name` is `nullptr` or matches no vendor.

Errors

None: the functions take no context to raise errors in.

Examples

```
const dmi_vendor_spec_t *spec = dmi_vendor_detect("Dell Inc.");
dmi_vendor_t vendor = (spec != nullptr) ? spec->id : DMI_VENDOR_OTHER;

printf("%s\n", dmi_vendor_name(vendor));
```

See also

`dmi_get_platform`, `dmi_platform_match`, `dmi_set_locale`

<opendmi/module.h>

dmi_module_find

Synopsis

```
#include <opendmi/module.h>

typedef struct dmi_module {
    const char                 *code;
    const char                 *name;
    const dmi_entity_spec_t   **entities;
    unsigned                    flags;
    const dmi_platform_match_t *platforms;
    const dmi_relocation_t     *relocations;
    const dmi_module_group_t   *groups;
    dmi_module_t               *next;
} dmi_module_t;

extern const dmi_module_t *const dmi_builtin_modules[];

const dmi_module_t *dmi_module_find(const char *code);
const dmi_module_t *dmi_module_next(const dmi_module_t *module);
```

Description

An extension module brings the specifications of the vendor-specific structure types, see `dmi_add_extension`. The modules available are the built-in ones, listed by the `dmi_builtin_modules` array, which is terminated by `nullptr`, followed by the ones the program registers with `dmi_module_register`, in the order of registration.

The `dmi_module_find` function finds the available module whose code is `code`, e.g. `hpe`. Codes are compared exactly, case included.

The `dmi_module_next` function lists the available modules: it returns the first one for `nullptr`, and the one which follows `module` otherwise.

The members of a module are the following:

`code`

Code of the module, e.g. `hpe`, which is never translated.

`name`

Printable name of the module.

`entities`

Specifications of the structure types the module brings, terminated by `nullptr`.

`flags`

Flags of the module: `DMI_MODULE_FLAG_YIELD` makes its specifications give way to the ones of the other enabled modules of the same type numbers, instead of conflicting with them.

`platforms`

Platforms the module is enabled for as a context is opened, if the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`, terminated by `DMI_PLATFORM_NULL`, see `dmi_platform_match`, or `nullptr` for a module which is only enabled explicitly.

`relocations`

Structures of other modules, which the platforms of the module place at type numbers of their own, terminated by an entry with no specification.

`groups`

Group associations (type 14) whose members are structures of the module, terminated by an entry with no name.

`next`

Next registered module, which the library sets as the module is registered.

The modules belong to the library, or to the program which has registered them, and must not be modified or freed.

Parameters

`code`

Code of the module to find.

`module`

Module returned by the previous call, or `nullptr` to get the first one. It must be an available module.

Return value

`dmi_module_find` returns the module, or `nullptr` if no available module has the code, or if `code` is `nullptr`.

`dmi_module_next` returns the next module, or `nullptr` if `module` is the last one.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised by `dmi_module_find` if `code` is `nullptr`, in which case it is not reported, since there is no context to hold it.

`dmi_module_next` raises no errors.

Example

List the available modules:

```
for (const dmi_module_t *module = dmi_module_next(nullptr);
```

```
     module != nullptr;
     module = dmi_module_next(module))
    printf("%-12s %s\n", module->code, module->name);
```

See also

`dmi_add_extension`, `dmi_filter_add_module`, `dmi_module_register`, `dmi_platform_match`

dmi_module_register

Synopsis

```
#include <opendmi/module.h>

bool dmi_module_register(dmi_module_t *module);
```

Description

The `dmi_module_register` function makes an extension module of a program available along with the built-in ones, so that a program brings structure types of its own. A registered module is found by `dmi_module_find`, listed by `dmi_module_next` after the built-in ones, in the order of registration, and enabled the same way as they are: with `dmi_add_extension`, or for the platforms it lists as a context is opened, see `dmi_module_find` for the members of a module.

A module is registered once, and stays registered for as long as the program runs: there is no way to unregister it. The module is not copied, so it must stay valid for as long as the program runs. The library links the registered modules through their `next` member, which it sets, so a module is not constant.

Registration is not thread-safe, and belongs to the start of the program, before the contexts which use the module are created.

Parameters

`module`
: Module to register, which must have a `code`.

Return value

Returns `true` if the module has been registered, and `false` if `module` is `nullptr` or has no `code`, or if an available module, either a built-in or a registered one, has the same code, in which case nothing is registered.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `module` is `nullptr`, in which case it is not reported, since there is no context to hold it.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if the `code` of `module` is `nullptr`, and not reported either.

A module whose code is taken is refused silently.

Example

```
static dmi_module_t example_module = {
    .code     = "example",
    .name     = "Example extensions",
    .entities = (const dmi_entity_spec_t *[]){
        &example_spec,
        nullptr
    }
};

if (!dmi_module_register(&example_module))
    return 1;
```

See also

`dmi_add_extension`, `dmi_module_find`, `dmi_module_next`

<opendmi/module/hpe.h>

dmi_hpe_platform_detect

Synopsis

```
#include <opendmi/module/hpe.h>

#define DMI_HPE_FAMILY_SERVER "server"

typedef enum dmi_hpe_generation {
    DMI_HPE_GEN1       = 10,
    DMI_HPE_GEN2       = 20,
    DMI_HPE_GEN3       = 30,
    DMI_HPE_GEN4       = 40,
    DMI_HPE_GEN5       = 50,
    DMI_HPE_GEN6       = 60,
    DMI_HPE_GEN7       = 70,
    DMI_HPE_GEN8       = 80,
    DMI_HPE_GEN9       = 90,
    DMI_HPE_GEN10      = 100,
    DMI_HPE_GEN10_PLUS = 105,
    DMI_HPE_GEN11      = 110,
    DMI_HPE_GEN12      = 120
} dmi_hpe_generation_t;

bool dmi_hpe_platform_detect(dmi_platform_t *platform);
```

Description

The `dmi_hpe_platform_detect` function tells the family and the generation of an HP or HPE platform from its product name, since the layouts of the vendor-specific structures of HP and HPE servers depend on them. A context calls it as it is opened, if the firmware is of HP or HPE and the table has a product name, see `dmi_get_platform`; a program calls it for a platform it gives to a context with `dmi_set_platform`.

A platform is a server of the family `DMI_HPE_FAMILY_SERVER` if its product name holds the name of a product line of HP and HPE servers: `ProLiant`, `Apollo`, `Synergy` or `Edgeline`. Names are compared exactly, case included. The workstations and the laptops of HP have structures of their own, and their platforms are left as they are.

The generation of a server is told by the first word of the product name which is `G` or `Gen` followed by a number, e.g. `G7` or `Gen10`: HP names the generations up to the seventh `G1` to `G7`, and the ones from the eighth onwards `Gen8` and so on. The generation is numbered ten times its number, and five more if the word is followed by the word `Plus`, e.g. `DMI_HPE_GEN10_PLUS` for `ProLiant DL360 Gen10 Plus`.

A server whose product name tells no generation is taken for the newest generation the firmware of its vendor may be of, and for the oldest one HP named without generation, the way dmidecode takes them:

Gen10 Plus if the firmware vendor of the platform is `DMI_VENDOR_HPE`, since the firmware of earlier generations names HP as its vendor, and Gen6 otherwise.

The family and the generation the platform had are replaced.

Parameters

`platform`
: Platform with the firmware vendor and the product name set, see `dmi_platform_create`.

Return value

Returns `false` if memory is exhausted, in which case the platform is left as it was, and `true` otherwise, including the cases when `platform` is `nullptr`, when it has no product name, or when it is not a server, in which cases it is left as it is.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted, reported to the context of `platform`, if it has one.

Example

```
dmi_platform_t *platform = dmi_platform_create(context);

if ((platform != nullptr) &&
    dmi_platform_set_product(platform, "ProLiant DL380 Gen11")) {
    platform->firmware_vendor = DMI_VENDOR_HPE;

    if (dmi_hpe_platform_detect(platform))
        dmi_set_platform(context, platform);
}

dmi_platform_destroy(platform);
```

See also

`dmi_get_platform`, `dmi_platform_create`, `dmi_platform_in_generations`, `dmi_platform_set_family`, `dmi_set_platform`

<opendmi/error.h>

dmi_error_clear

Synopsis

```
#include <opendmi/error.h>

void dmi_error_clear(dmi_context_t *context);
```

Description

The `dmi_error_clear` function removes all error descriptors from the error queue of `context`, freeing any associated message strings. After the call the queue is empty. If `context` is `nullptr`, the function does nothing.

Parameters

`context`
: DMI context.

Errors

None.

See also

`dmi_error_get_first`, `dmi_error_get_last`, `dmi_error_raise`

dmi_error_get_first

Synopsis

```
#include <opendmi/error.h>

dmi_error_t *dmi_error_get_first(dmi_context_t *context);
```

Description

The `dmi_error_get_first` function removes the oldest error descriptor from the error queue of `context` and returns a pointer to it. It can be called repeatedly until there are no more entries.

The returned pointer remains valid until the error queue is next modified by `dmi_error_raise`, `dmi_error_raise_ex`, `dmi_error_get_first`, `dmi_error_get_last`, or `dmi_error_clear`. The descriptor is no longer part of the queue, so its slot, including `message`, is reused by the next error raised for `context`, and any library function called with `context` may raise one. Copy whatever is needed before

calling into the library again.

Parameters

`context`
: DMI context.

Return value

Returns a pointer to the oldest `dmi_error_t` removed from the queue, or `nullptr` if `context` is `nullptr` or the error queue is empty.

Errors

None.

Examples

```
dmi_error_t *error;

while ((error = dmi_error_get_first(context)) != nullptr) {
    fprintf(stderr, "%s:%d: %s\n",
            error->function,
            error->line,
            dmi_error_message(error->reason));
}
```

See also

`dmi_error_peek_first`, `dmi_error_get_last`, `dmi_error_clear`, `dmi_error_raise`, `dmi_error_message`

dmi_error_get_last

Synopsis

```
#include <opendmi/error.h>

dmi_error_t *dmi_error_get_last(dmi_context_t *context);
```

Description

The `dmi_error_get_last` function removes the newest error descriptor from the error queue of `context` and returns a pointer to it. It can be called repeatedly until there are no more entries.

The returned pointer remains valid until the error queue is next modified by `dmi_error_raise`, `dmi_error_raise_ex`, `dmi_error_get_first`, `dmi_error_get_last`, or `dmi_error_clear`. The

descriptor is no longer part of the queue, so its slot, including `message`, is reused by the next error raised for `context`, and any library function called with `context` may raise one. Copy whatever is needed before calling into the library again.

Parameters

`context`
: DMI context.

Return value

Returns a pointer to the newest `dmi_error_t` removed from the queue, or `nullptr` if `context` is `nullptr` or the error queue is empty.

Errors

None.

Examples

```
dmi_error_t *error;

while ((error = dmi_error_get_last(context)) != nullptr) {
    fprintf(stderr, "%s:%d: %s\n",
            error->function,
            error->line,
            dmi_error_message(error->reason));
}
```

See also

`dmi_error_peek_last`, `dmi_error_get_first`, `dmi_error_clear`, `dmi_error_raise`, `dmi_error_message`

dmi_error_message

Synopsis

```
#include <opendmi/error.h>

const char *dmi_error_message(dmi_error_code_t reason);
```

Description

The `dmi_error_message` function returns a human-readable string describing the error identified by `reason`. The returned pointer refers to a static string; the caller must not free or modify it.

If `reason` is out of range or has no associated message, the function returns `"Unknown error"`.

Parameters

`reason`
: Error reason code.

Return value

Returns a pointer to a static null-terminated string. Never returns `nullptr`.

Errors

None.

Examples

```
dmi_error_t *error = dmi_error_get_last(context);
if (error != nullptr) {
    fprintf(stderr, "error: %s\n", dmi_error_message(error->reason));
    dmi_error_clear(context);
}
```

See also

`dmi_error_raise`, `dmi_error_get_first`, `dmi_error_get_last`

dmi_error_peek_first

Synopsis

```
#include <opendmi/error.h>

dmi_error_t *dmi_error_peek_first(dmi_context_t *context);
```

Description

The `dmi_error_peek_first` function returns a pointer to the oldest error descriptor in the error queue of `context` without removing it. The queue is not modified.

Parameters

`context`
: DMI context.

Return value

Returns a pointer to the oldest `dmi_error_t` in the queue, or `nullptr` if `context` is `nullptr` or the error queue is empty.

Errors

None.

See also

`dmi_error_peek_last`, `dmi_error_get_first`, `dmi_error_get_last`, `dmi_error_clear`, `dmi_error_raise`

dmi_error_peek_last

Synopsis

```
#include <opendmi/error.h>

dmi_error_t *dmi_error_peek_last(dmi_context_t *context);
```

Description

The `dmi_error_peek_last` function returns a pointer to the newest error descriptor in the error queue of `context` without removing it. The queue is not modified.

Parameters

`context`
: DMI context.

Return value

Returns a pointer to the newest `dmi_error_t` in the queue, or `nullptr` if `context` is `nullptr` or the error queue is empty.

Errors

None.

See also

`dmi_error_peek_first`, `dmi_error_get_first`, `dmi_error_get_last`, `dmi_error_clear`, `dmi_error_raise`

dmi_error_raise

Synopsis

```
#include <opendmi/error.h>

bool dmi_error_raise(dmi_context_t *context,
                     dmi_error_code_t reason);  /* macro */
```

Description

The `dmi_error_raise` macro adds an error entry with reason code `reason` to the error queue of `context`. The source file name, function name, and line number are captured automatically from the call site.

To attach a formatted message to the error, use `dmi_error_raise_ex`.

The error queue is a ring buffer with a maximum depth of `DMI_ERROR_MAX_DEPTH` entries. When the queue is full, the oldest entry is overwritten silently.

Parameters

`context`
: DMI context.

`reason`
: Error reason code.

Return value

Returns `true` on success. Returns `false` if `context` is `nullptr`.

Errors

None.

Examples

```
if (ptr == nullptr) {
    dmi_error_raise(context, DMI_ERROR_ARGUMENT_NULL);
    return false;
}
```

See also

`dmi_error_raise_ex`, `dmi_error_get_first`, `dmi_error_get_last`, `dmi_error_message`

dmi_error_raise_ex

Synopsis

```
#include <opendmi/error.h>

bool dmi_error_raise_ex(dmi_context_t *context, dmi_error_code_t reason,
                        const char *message, ...);  /* macro */
```

Description

The `dmi_error_raise_ex` macro adds an error entry with reason code `reason` and an additional printf-style formatted message to the error queue of `context`. The source file name, function name, and line number are captured automatically from the call site.

The formatted message is stored in the `message` field of the resulting `dmi_error_t` entry and is freed automatically when the entry is evicted from the queue or the queue is cleared.

To raise an error without an additional message, use `dmi_error_raise`.

The error queue is a ring buffer with a maximum depth of `DMI_ERROR_MAX_DEPTH` entries. When the queue is full, the oldest entry is overwritten silently.

Parameters

`context`
: DMI context.

`reason`
: Error reason code.

`message`
: printf-style format string for the additional message.

`...`
: Format arguments.

Return value

Returns `true` on success. Returns `false` if `context` is `nullptr` or if memory allocation for the formatted message fails. In the latter case the error is still added to the queue, left without a message.

Errors

None.

Examples

```
if (fd < 0) {
```

```
    dmi_error_raise_ex(context, DMI_ERROR_FILE_OPEN_FAILED,
                       "cannot open '%s': %s", path, strerror(errno));
    return false;
}
```

See also

`dmi_error_raise`, `dmi_error_get_first`, `dmi_error_get_last`, `dmi_error_message`

<opendmi/trace.h>

dmi_trace_argument_null

Synopsis

```
#include <opendmi/trace.h>

#define dmi_trace_argument_null(context, arg, ...)
#define dmi_trace_argument_invalid(context, arg, ...)
#define dmi_trace_state_invalid(context, message, ...)
#define dmi_trace_out_of_memory(context, ...)
```

Description

The macros report the standard errors, of the arguments of a function and of the state it works on, the way the functions of the library report them, and are meant for the code which extends the library, e.g. the handlers of a module built out of its tree.

The `dmi_trace_argument_null` macro raises `DMI_ERROR_ARGUMENT_NULL`, and the `dmi_trace_argument_invalid` macro raises `DMI_ERROR_ARGUMENT_INVALID`, on `context`, see `dmi_error_raise`. The message of the error is the expression of `arg`, e.g. `path`, so that the argument is named the way the code names it.

The `dmi_trace_state_invalid` macro raises `DMI_ERROR_STATE_INVALID` on `context` with `message`, which tells what keeps the function from working, e.g. `Context is not open`. The message is taken as it is, rather than as a format string; an error which needs the details formatted is raised with `dmi_error_raise_ex`.

The `dmi_trace_out_of_memory` macro raises `DMI_ERROR_OUT_OF_MEMORY` on `context`, once an allocation has failed. The allocation functions of the library raise it themselves, so the macro is for the allocations made otherwise, e.g. a vector which cannot grow.

Each macro is an expression, which evaluates to the value given last, after `arg`, `message` or `context`, or to `false` if none is given, so that the check reads as a condition and the failure it leads to:

```
bool foo_load(dmi_context_t *context, const char *path, size_t length)
{
    if (path == nullptr)
        return dmi_trace_argument_null(context, path);
    if (length == 0)
        return dmi_trace_argument_invalid(context, length);
    if (!dmi_is_open(context))
        return dmi_trace_state_invalid(context, "Context is not open");
    ...
}

const foo_t *foo_find(dmi_context_t *context, const char *code)
```

```
{
    if (code == nullptr)
        return dmi_trace_argument_null(context, code, nullptr);
    ...
}
```

A function returning nothing casts the value of the macro to `void`, and returns on its own:

```
if (reader == nullptr) {
    (void)dmi_trace_argument_null(nullptr, reader);
    return;
}
```

The value is evaluated after the error is raised, and is returned whether or not the error has been recorded, e.g. if `context` is `nullptr`.

The expression of `arg` is checked by the compiler, but is not evaluated, so that a name which does not exist any longer is not reported.

Parameters

`context`
: Context to raise the error on, which may be `nullptr`.

`arg`
: Argument, which is `nullptr` or the value of which is invalid.

`message`
: Message telling what is wrong with the state.

`…`
: Value to evaluate to, `false` if none is given. A function returning a pointer passes `nullptr`, since the value of the expression is not a null pointer constant.

Return value

The macros evaluate to the value given, or to `false`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised by `dmi_trace_argument_null`.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised by `dmi_trace_argument_invalid`.

`DMI_ERROR_STATE_INVALID`
: Raised by `dmi_trace_state_invalid`.

`DMI_ERROR_OUT_OF_MEMORY`

Raised by `dmi_trace_out_of_memory`.

Notes

The optional value relies on `__VA_OPT__`, which is a part of C23 and of C++20, and is supported by GCC and Clang in the earlier modes as well.

See also

`dmi_error_raise`, `dmi_error_raise_ex`, `dmi_error_get_last`, `dmi_is_open`

<opendmi/log.h>

dmi_log_level

Synopsis

```
#include <opendmi/log.h>

typedef enum dmi_log_level {
    DMI_LOG_INVALID = -1,
    DMI_LOG_ERROR,
    DMI_LOG_WARNING,
    DMI_LOG_NOTICE,
    DMI_LOG_INFO,
    DMI_LOG_DEBUG
} dmi_log_level_t;

const char *dmi_log_level_name(dmi_log_level_t value);
dmi_log_level_t dmi_log_level_find(const char *code);
```

Description

The `dmi_log_level_t` enumeration tells the severity of a log message, see `dmi_log`, and the least severe level of the messages a context writes to its handler, see `dmi_set_log_level`. Levels are ordered from the most severe one to the least, so that a level is less severe than another one if its value is greater.

The `dmi_log_level_name` function returns the name of a level, as the command line tool prints it before a message, e.g. `WARNING`. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

The `dmi_log_level_find` function returns the level whose code is `code`, e.g. to parse a level given on the command line. The codes are the names of the levels in lower case: `error`, `warning`, `notice`, `info` and `debug`. Codes are compared exactly, case included, and are never translated.

Values

`DMI_LOG_INVALID`

No level, which the functions return on failure.

`DMI_LOG_ERROR`

Error, code `error`.

`DMI_LOG_WARNING`

Warning, code `warning`.

`DMI_LOG_NOTICE`

Notice, code `notice`.

`DMI_LOG_INFO`
: Informational message, code `info`.

`DMI_LOG_DEBUG`
: Debugging message, code `debug`.

Parameters

`value`
: Level to name.

`code`
: Code of the level to find.

Return value

The `dmi_log_level_name` function returns the name of the level, or `UNKNOWN` if the value is not a level, e.g. `DMI_LOG_INVALID`, so that a log record can always be prefixed with it, unlike the other `*_name` functions, which return `nullptr` then. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

The `dmi_log_level_find` function returns the level, or `DMI_LOG_INVALID` if `code` is `nullptr` or no level has the code.

Errors

None: the functions take no context to raise errors in.

See also

`dmi_get_log_level`, `dmi_log`, `dmi_log_message`, `dmi_set_locale`, `dmi_set_log_level`

dmi_log_message

Synopsis

```
#include <opendmi/log.h>

bool dmi_log_message(
        dmi_log_t       *target,
        dmi_log_level_t  level,
        const char      *format,
        ...);
```

Description

The `dmi_log_message` function writes a message, formatted the way `printf`(3) formats its arguments, to the given handler.

Unlike `dmi_log`, which writes to the handler of a context, it takes the handler itself, and checks no logging level: every message is given to the handler, which drops the ones it has no use for.

Parameters

`target`
: Logging handler to write the message to.

`level`
: Level of the message.

`format`
: Format of the message, followed by the values it refers to.

Return value

Returns `true` if the message has been given to the handler, and `false` if `target` or its handler function is `nullptr`.

Errors

None.

See also

`dmi_log_message_va`, `dmi_log`, `dmi_set_logger`

dmi_log_message_va

Synopsis

```
#include <opendmi/log.h>

bool dmi_log_message_va(
        dmi_log_t       *target,
        dmi_log_level_t  level,
        const char      *format,
        va_list          args);
```

Description

The `dmi_log_message_va` function is the equivalent of `dmi_log_message` which takes the arguments of the format as a `va_list`, so that a function taking a variable number of arguments passes its own on, the way `vprintf`(3) does for `printf`(3).

It checks no logging level: every message is given to the handler, which drops the ones it has no use for.

Parameters

`target`
: Logging handler to write the message to.

`level`
: Level of the message.

`format`
: Format of the message.

`args`
: Values the format refers to.

Return value

Returns `true` if the message has been given to the handler, and `false` if `target` or its handler function is `nullptr`.

Errors

None.

See also

`dmi_log_message`, `dmi_log`, `dmi_set_logger`

`<opendmi/locale.h>`

dmi_get_locale

Synopsis

```
#include <opendmi/locale.h>

const char *dmi_get_locale(void);
```

Description

The `dmi_get_locale` function tells the locale in which the library looks up printable strings.

While a resource package is open, the name is the locale the most recently opened package has actually been opened with, which may be less specific than the one requested by `dmi_set_locale` when there is no translation for the latter, e.g. `ru` for `ru_RU`, or `root` when there is no translation at all. The resources of the library itself are opened on the first lookup of a printable name.

While no package is open, the name is the one last passed to `dmi_set_locale`, which is `nullptr` for the default locale of the environment.

The name belongs to the library, and must not be freed. It is valid until the locale is changed by `dmi_set_locale`, or the package it is taken from is closed by `dmi_resource_close`.

Return value

Returns the name of the locale, or `nullptr` if the default locale of the environment is requested and no package is open, if the locale of the package cannot be told, or if the library is built without ICU4C support.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_set_locale`, `dmi_locale_setup`, `dmi_resource_open`

dmi_locale_setup

Synopsis

```
#include <opendmi/locale.h>

void dmi_locale_setup(void);
```

Description

The `dmi_locale_setup` function sets up the locale of a program from the environment in one call:

- the `LC_CTYPE` and `LC_MESSAGES` categories of the C library are taken from the environment by `setlocale`(3), so that characters are classified and messages of the system are written in the language of the user;
- the locale of the library is set by `dmi_set_locale` to the default locale of the environment, so that printable names are translated as well, and left in English when there is no translation.

Other categories, `LC_NUMERIC` in particular, are deliberately left in the C locale, so that the decimal point of formatted values does not depend on the environment, and machine-readable output is the same everywhere.

Programs which handle the locale themselves do not need the function, and call `setlocale`(3) and `dmi_set_locale` on their own.

In the kernel, which has no locale environment, only the locale of the library is set.

Thread safety

Like `dmi_set_locale`, the function is not thread-safe. It is meant to be called once, at the start of the program, before the library is used.

Return value

None. A locale which cannot be set leaves printable strings in English.

Errors

None.

See also

`dmi_set_locale`, `dmi_get_locale`, `setlocale`(3)

dmi_message_format

Synopsis

```
#include <opendmi/locale.h>

typedef enum dmi_message_arg_type {
    DMI_MESSAGE_ARG_TEXT,
    DMI_MESSAGE_ARG_NUMBER
} dmi_message_arg_type_t;

typedef struct dmi_message_arg {
    dmi_message_arg_type_t type;
```

```
    union {
        const char *text;
        intmax_t    number;
    };
} dmi_message_arg_t;

#define DMI_MESSAGE_TEXT(value)   ...
#define DMI_MESSAGE_NUMBER(value) ...

char *dmi_message_format(
        dmi_resource_t          *resource,
        const char              *pattern,
        const dmi_message_arg_t *args,
        size_t                   count);
```

Description

The `dmi_message_format` function formats a message by substituting the placeholders of `pattern` with the arguments `args`. Unlike `printf`(3), the placeholders refer to the arguments by their indexes, so that a translation can put them in another order, and select the plural forms of words by numbers, so that it can agree words with numbers the way its language does.

Placeholders of the pattern are:

`{N}`
: Argument number `N`, counted from zero: a text as it is, or a number in decimal.

`{N:word}`
: Plural form of `word` selected by the number of the argument `N`, which must be a number. The forms are the strings of the `plural/word` table of `resource`, keyed by the plural categories of the locale the resource is opened with, e.g. `one`, `few`, `many` and `other` in Russian. A category the table leaves out is given the `other` form.

`{{`
: Literal opening brace. A closing brace outside of a placeholder is taken as it is.

`N` is a single digit, so a message has ten arguments at most.

Plural categories follow the rules of the locale of the bundle the forms come from, e.g. `{0} {0:byte}` gives `1 byte` and `2 bytes` in English, while a Russian translation gives three different forms of the word for 1, 2 and 5 bytes, of the `one`, `few` and `many` categories. The `root` bundle holds English forms, so the English rules are used for it. Patterns with plural forms need `resource` and ICU4C support; other patterns are formatted in any case, with or without them.

Arguments are made with the `DMI_MESSAGE_TEXT` and `DMI_MESSAGE_NUMBER` macros, which give compound literals of `dmi_message_arg_t`:

```
dmi_message_arg_t args[] = {
    DMI_MESSAGE_NUMBER(count),
    DMI_MESSAGE_TEXT(path)
```

```
};

char *text = dmi_message_format(resource, "{0} {0:structure} read from {1}",
                                args, 2);
```

The caller is responsible for freeing the returned string.

Parameters

`resource`
: Resource handle holding the plural forms, or `nullptr` for patterns with no plural forms.

`pattern`
: Message pattern.

`args`
: Arguments of the message. A text argument which is `nullptr` is substituted as an empty text.

`count`
: Number of the arguments.

Return value

Returns the formatted message, or `nullptr` if:

- `pattern` is `nullptr`;
- the pattern has an opening brace with no closing one, or a placeholder which is not of the forms above;
- a placeholder refers to an argument past `count`;
- a plural form is selected by a text argument, or no form of the word is found, which is always the case when `resource` is `nullptr` or the library is built without ICU4C support;
- memory is exhausted.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_resource_message`, `dmi_resource_open`, `dmi_resource_string`

dmi_resource_close

Synopsis

```
#include <opendmi/locale.h>

void dmi_resource_close(dmi_resource_t *resource);
```

Description

The `dmi_resource_close` function closes a resource package opened by `dmi_resource_open`, and frees the handle.

Strings looked up in the package are released with it, so every pointer returned for it by `dmi_resource_string` becomes invalid, and so does the name returned by `dmi_get_locale` when it is taken from this package.

Thread safety

Open packages are kept in a process-wide list, which the function changes, so it must not be called while other threads may use the library.

Parameters

`resource`
: Resource handle to close, or `nullptr`, in which case nothing is done.

Return value

None.

Errors

None.

See also

`dmi_resource_open`, `dmi_resource_string`

dmi_resource_message

Synopsis

```
#include <opendmi/locale.h>

char *dmi_resource_message(
        dmi_resource_t          *resource,
        const char              *table,
        const char              *key,
        const dmi_message_arg_t *args,
        size_t                   count);
```

Description

The `dmi_resource_message` function formats a translated message: it looks up the pattern `key` of the table `table` of a resource package by `dmi_resource_string`, and formats it with the arguments `args` by `dmi_message_format`, taking the plural forms of words from the same package.

When the package has no translation of the message, the function returns `nullptr` rather than formatting anything, so that the caller can format the untranslated pattern instead:

```
char *text = dmi_resource_message(resource, "message", key, args, count);

if (text == nullptr)
    text = dmi_message_format(resource, fallback, args, count);
```

The caller is responsible for freeing the returned string.

Parameters

`resource`
: Resource handle, or `nullptr`, in which case no pattern is found.

`table`
: Path of the table holding the pattern, see `dmi_resource_string`.

`key`
: Key of the pattern within the table.

`args`
: Arguments of the message, see `dmi_message_format`.

`count`
: Number of the arguments.

Return value

Returns the formatted message, or `nullptr` if there is no such pattern, or if the pattern cannot be formatted, see `dmi_message_format`.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_message_format`, `dmi_resource_string`, `dmi_resource_open`

dmi_resource_open

Synopsis

```
#include <opendmi/locale.h>

typedef struct dmi_resource dmi_resource_t;
```

```
dmi_resource_t *dmi_resource_open(const char *package, const void *data);
```

Description

The `dmi_resource_open` function opens a resource package, a set of ICU4C resource bundles holding the strings of a program in every language it is translated to, so that the program can look its strings up the way the library looks up printable names.

The package is embedded into the program rather than installed as a file. Bundles of the package are built from ICU4C resource sources, one `<locale>.txt` per locale and `root.txt` with the default strings, and are packaged by `pkgdata` into a data object which is linked into the program, e.g. by the `add_resources()` function of the OpenDMI build files. The data is then reachable as the `<package>_dat` symbol:

```
extern const char opendmi_dat[];

dmi_resource_t *resource = dmi_resource_open("opendmi", opendmi_dat);
```

The package is opened with the locale set by `dmi_set_locale`, and is reopened with the new one whenever the locale is changed. Locales with no bundle of their own fall back to a less specific locale, and to the `root` bundle at last, but never to the bundle of the locale of the environment. A package with no `root` bundle is opened only with the locales it has bundles for.

The package stays open until it is closed by `dmi_resource_close`. Strings are looked up in it by `dmi_resource_string`, and messages are formatted by `dmi_resource_message`.

Thread safety

Open packages are kept in a process-wide list, which the function changes, so it must not be called while other threads may use the library.

Parameters

`package`

Package name, the one passed to `pkgdata --name` when the package is built, e.g. `opendmi` for the strings of the command line tool. The data is registered with ICU4C under this name, which therefore has to be unique.

`data`

Package data, the `<package>_dat` symbol of the data object.

Return value

Returns a resource handle, or `nullptr` if an argument is `nullptr`, if ICU4C cannot be initialized or the data cannot be registered or opened, if memory is exhausted, or if the library is built without ICU4C support.

Programs are expected to leave their strings untranslated, rather than fail, when no handle is returned: the lookup functions accept a `nullptr` handle, and find no strings in it.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_resource_close`, `dmi_resource_string`, `dmi_resource_message`, `dmi_set_locale`

dmi_resource_string

Synopsis

```
#include <opendmi/locale.h>

const char *dmi_resource_string(dmi_resource_t *resource, const char *table,
const char *key);
```

Description

The `dmi_resource_string` function looks up the string `key` of the table `table` of a resource package, in the locale the package is opened with.

Strings of a table are converted to UTF-8 on the first lookup in it, and are kept until the locale is changed or the package is closed. Tables which the package does not have are remembered as empty ones, so a lookup of a missing string is cheap as well.

Strings of a table are taken from a single bundle: a string which the table of the locale leaves out is not taken from the same table of a less specific locale, nor from the `root` bundle. Callers are expected to fall back to a text of their own, usually the untranslated one, when no string is found:

```
const char *text = dmi_resource_string(resource, "text", "Handle");

puts(text != nullptr ? text : "Handle");
```

The string belongs to the package, and must not be freed. It is valid until the locale is changed by `dmi_set_locale`, or the package is closed by `dmi_resource_close`.

Parameters

`resource`
: Resource handle, or `nullptr`, in which case no string is found.

`table`
: Path of the table holding the string: the name of a top-level table, e.g. a name set code, or names of nested tables separated by slashes, e.g. `processor/attributes`. A path with a name of 64 bytes or longer finds no table.

`key`
: Key of the string within the table.

Return value

Returns the string in UTF-8, or `nullptr` if there is no such string, if an argument is `nullptr`, if memory is exhausted, or if the library is built without ICU4C support.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `table` or `key` is `nullptr`. The function takes no context, so the error reaches no error queue. A `resource` of `nullptr` is not an error.

See also

`dmi_resource_open`, `dmi_resource_message`, `dmi_set_locale`

dmi_set_locale

Synopsis

```
#include <opendmi/locale.h>

bool dmi_set_locale(const char *locale);
```

Description

The `dmi_set_locale` function sets the locale in which the library looks up printable strings: names of values, structure types and lint rules, error messages, and the other strings of the resource packages.

The locale is process-wide rather than a property of a context, since printable names are looked up by functions which take no context, e.g. `dmi_name_lookup`. Resource packages opened before the call are reopened with the new locale, and the ones opened after it use it as well.

A locale with no translation of its own is not a failure: ICU4C falls back to a less specific locale, e.g. `ru` for `ru_RU`, and strings with no translation at all are left in English. The locale of the environment is not fallen back to, so that the strings are never of a locale which has not been asked for.

The name is copied, so the variable `locale` points to need not outlive the call.

Translations are available only if the library is built with ICU4C support. Otherwise the function does nothing, and printable strings are always in English.

Lifetime of strings

Strings looked up for the previous locale are released by the call, so every pointer returned before it by the functions which look up printable strings, e.g. `dmi_name_lookup`, `dmi_spec_name`, `dmi_error_message`, `dmi_lint_rule_name`, `dmi_resource_string` and `dmi_get_locale`, becomes invalid. Strings which are to be kept across the call must be copied.

Thread safety

The function is not thread-safe, and must not be called while other threads may use the library. The locale is meant to be set once, at the start of the program, before the library is used, e.g. by `dmi_locale_setup`, which takes it from the environment.

Parameters

`locale`

: Locale name, e.g. `ru_RU`, or `nullptr` for the default locale of the environment, which ICU4C takes from the `LC_ALL`, `LC_MESSAGES` and `LANG` environment variables.

Return value

Returns `true` if the locale has been set, and `false` if the name cannot be copied, if a resource package opened before the call cannot be reopened, or if the library is built without ICU4C support.

A package which cannot be reopened gives no translated strings until the locale is set again.

The resources of the library itself are the exception: the library has translations for some locales only and leaves its strings in English for the others, so their absence is not a failure. They are opened by the first lookup, and are looked for again once the locale has changed.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_get_locale`, `dmi_locale_setup`, `dmi_name_lookup`, `dmi_resource_open`, `dmi_resource_string`, `dmi_error_message`, `setlocale`(3)

`<opendmi/entity.h>`

dmi_entity_buffer

Synopsis

```
#include <opendmi/entity.h>

const dmi_buffer_t *dmi_entity_buffer(const dmi_entity_t *entity);
```

Description

The `dmi_entity_buffer` function returns the buffer the data of the structure is read out of.

The data read is the copy with additional information applied whenever there is one, and the data of the structure itself otherwise, so the buffer is paired with `dmi_entity_offset` rather than taken on its own.

Parameters

`entity`
: Entity descriptor.

Return value

Buffer holding the data, or `nullptr` if `entity` is `nullptr`.

Errors

None.

See also

`dmi_entity_offset`, `dmi_entity_data`, `dmi_entity_decode`, `dmi_buffer_at`, `dmi_reader_initialize`

dmi_entity_context

Synopsis

```
#include <opendmi/entity.h>

dmi_context_t *dmi_entity_context(const dmi_entity_t *entity);
```

Description

The `dmi_entity_context` function returns the context a structure belongs to, which holds the data the structure has been decoded from.

Code which is given a structure alone, such as a handler of an extension module, reaches the context through it to raise errors, see `dmi_error_raise`, and to write messages, see `dmi_log`.

Parameters

`entity`
: Structure to query.

Return value

Returns the context, or `nullptr` if `entity` is `nullptr`.

Errors

None.

See also

`dmi_entity_create`, `dmi_error_raise`, `dmi_log`

dmi_entity_create

Synopsis

```
#include <opendmi/entity.h>

dmi_entity_t *dmi_entity_create(
        dmi_context_t      *context,
        const dmi_buffer_t *buffer,
        size_t              offset);
```

Description

The `dmi_entity_create` function allocates a new entity descriptor and initializes it from the data of the structure, which begins at `offset` within `buffer` and runs to the end of the data the buffer holds. It parses the structure header, computes the total length (formatted area plus string set), and decodes the string table.

The structure refers to the data of the buffer in place rather than holding a copy of it, so the buffer is required to hold that data for as long as the entity lives. The structures read from a table all refer to the buffer of the table, which the context holds until it is closed.

The caller is responsible for destroying the returned entity descriptor with `dmi_entity_destroy` when it is no longer needed.

Parameters

`context`
: DMI context handle.

`buffer`
: Buffer holding the data of the structure.

`offset`
: Offset the structure begins at within the buffer.

Return value

On success, returns a pointer to a newly allocated entity descriptor. On failure, returns `nullptr` (e.g. if `context` or `buffer` is `nullptr`, the structure length is invalid, or a memory allocation fails).

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Buffer is `nullptr`.

`DMI_ERROR_ENTITY_LENGTH_INVALID`
: Length of the structure is shorter than its header.

`DMI_ERROR_ENTITY_TRUNCATED`
: Structure, or the string set which follows it, runs past the end of the buffer.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

See also

`dmi_entity_decode`, `dmi_entity_link`, `dmi_entity_destroy`, `dmi_entity_buffer`, `dmi_buffer_create`

dmi_entity_data

Synopsis

```
#include <opendmi/entity.h>

const void *dmi_entity_data(const dmi_entity_t *entity, const dmi_type_t
*type);
```

Description

The `dmi_entity_data` function returns a pointer to the raw SMBIOS structure data, including the structure header. The entity type is verified against the expected `type` parameter before returning the data.

The type is the structure type of the specification the structure is decoded by, see `dmi_type`, rather than the number the structure is found at. Some vendors place structures defined by others at type numbers of their own, e.g. Dell moves Intel reference code structures 16 types down, and such a structure is still of the type of its original specification, since its data is laid out as that type expects, while `dmi_entity_type_id` returns the type number found in the table. Structures of different types sharing a type number, e.g. Intel vPro information and Lenovo ThinkVantage Technologies enablement, are told apart, so the pointer returned is always of the C type of `type`.

Parameters

`entity`
: Entity descriptor.

`type`
: Expected structure type, e.g. `DMI_TYPE(cooling_device)`. If the entity is not of this type, the function returns `nullptr` and raises an error. Pass `DMI_TYPE_ANY` to skip type checking.

Return value

Returns a pointer to the raw SMBIOS structure data. The returned pointer must not be modified or freed by the caller. Returns `nullptr` if `entity` is `nullptr` or the entity type does not match `type`.

Errors

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Structure is not of type `type`.

See also

`dmi_entity_info`, `dmi_entity_handle`, `dmi_entity_type`, `dmi_entity_type_id`, `dmi_type`

dmi_entity_decode

Synopsis

```
#include <opendmi/entity.h>

bool dmi_entity_decode(dmi_entity_t *entity);
```

Description

The `dmi_entity_decode` function decodes an SMBIOS entity that was previously created with `dmi_entity_create`.

It looks up the entity specification by type, validates the minimum length constraint, allocates the decoded structure descriptor, and invokes the type-specific decode handler. After successful decoding, the decoded data can be retrieved with `dmi_entity_info`.

This function is idempotent: if decoding has already been performed, it returns `true` immediately without

repeating the operation.

Parameters

`entity`
: Entity descriptor.

Return value

Returns `true` on success, including when the entity has already been decoded or no decoder is registered for its type. Returns `false` on failure (e.g., if `entity` is `nullptr`, the minimum length constraint is violated, or the type-specific decoder fails).

Errors

`DMI_ERROR_ENTITY_LENGTH_INVALID`
: Structure is shorter than its type requires.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

`DMI_ERROR_ENTITY_DECODE_FAILED`
: Structure cannot be decoded. Raised last on every failure of the decoder but the lack of memory, after the errors the decoder has raised.

See also

`dmi_entity_create`, `dmi_entity_link`, `dmi_entity_info`, `dmi_entity_destroy`

dmi_entity_destroy

Synopsis

```
#include <opendmi/entity.h>

void dmi_entity_destroy(dmi_entity_t *entity);
```

Description

The `dmi_entity_destroy` function releases all resources associated with the given entity descriptor, including the decoded structure data, the string table, and the entity descriptor itself.

If `entity` is `nullptr`, this function does nothing.

If decoded data is present, what it holds is freed before the data itself: first the arrays of the structures the references of the attributes are resolved into, see `dmi_attributes_unlink`, then whatever the type-specific cleanup handler frees, if one is registered, and last the arrays the fields of the specification are decoded into, see `dmi_fields_release`.

Parameters

`entity`
: Entity descriptor to destroy, or `nullptr`.

Return value

None.

Errors

None.

See also

`dmi_entity_create`, `dmi_entity_decode`

dmi_entity_encode

Synopsis

```
#include <opendmi/entity.h>

bool dmi_entity_encode(dmi_encoder_t *encoder);
```

Description

The `dmi_entity_encode` function writes a structure back into the bytes it is made of, undoing what `dmi_entity_decode` has done.

Structures whose specification describes their layout are encoded by its fields, see `dmi_fields_encode`, and the rest by the encoding handler of the specification. Structures which carry no data the model holds, such as the end-of-table one, and the ones of the types no specification describes, are kept as they are in the preserve mode: their bytes are all the model the structure has.

The encoder is set up with `dmi_encoder_initialize`, which writes the header, and the formatted area is completed with `dmi_encoder_finish`.

Parameters

`encoder`
: Encoder of the structure, see `dmi_encoder_initialize`.

Return value

`true` if the structure has been encoded, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Encoder is `nullptr`.

`DMI_ERROR_STATE_INVALID`
: Structure cannot be encoded: its type has no specification and the mode is the canonical one, or its specification decodes it by a handler with no encoding one.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffers of the encoder cannot grow.

See also

`dmi_entity_decode`, `dmi_fields_encode`, `dmi_encoder_initialize`, `dmi_encoder_finish`, `dmi_save`

dmi_entity_handle

Synopsis

```
#include <opendmi/entity.h>

dmi_handle_t dmi_entity_handle(const dmi_entity_t *entity);
```

Description

The `dmi_entity_handle` function returns the SMBIOS handle associated with the given entity. Each SMBIOS structure is assigned a unique handle by the platform firmware, which can be used to reference the structure from other structures.

Parameters

`entity`
: Entity descriptor.

Return value

Returns the handle associated with the entity. If `entity` is `nullptr`, returns `DMI_HANDLE_INVALID` (`0xFFFF`).

Errors

None.

See also

`dmi_entity_type`, `dmi_entity_name`, `dmi_entity_data`, `dmi_entity_info`

dmi_entity_info

Synopsis

```
#include <opendmi/entity.h>

void *dmi_entity_info(const dmi_entity_t *entity, const dmi_type_t *type);
```

Description

The `dmi_entity_info` function returns a pointer to the decoded structure descriptor for the given entity. The entity type is verified against the expected `type` parameter before returning the data.

The returned pointer should be cast to the appropriate type-specific structure (e.g., `dmi_cooling_device_t *` for cooling device entities). The entity must have been previously decoded with `dmi_entity_decode`.

The type is the structure type of the specification the structure is decoded by, see `dmi_type`, rather than the number the structure is found at. Some vendors place structures defined by others at type numbers of their own, e.g. Dell moves Intel reference code structures 16 types down, and such a structure is still of the type of its original specification, since its data is laid out as that type expects, while `dmi_entity_type_id` returns the type number found in the table. Structures of different types sharing a type number, e.g. Intel vPro information and Lenovo ThinkVantage Technologies enablement, are told apart, so the pointer returned is always of the C type of `type`.

Parameters

`entity`
: Entity descriptor.

`type`
: Expected structure type, e.g. `DMI_TYPE(cooling_device)`. If the entity is not of this type, the function returns `nullptr` and raises an error. Pass `DMI_TYPE_ANY` to skip type checking.

Return value

Returns a pointer to the decoded structure descriptor. Returns `nullptr` if `entity` is `nullptr`, the entity has not been decoded, or the entity type does not match `type`.

Errors

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Structure is not of type `type`.

Examples

```
dmi_entity_t *entity;
dmi_cooling_device_t *info;
```

```
info = dmi_entity_info(entity, DMI_TYPE(cooling_device));
if (info != nullptr) {
    printf("type: %s\n", dmi_cooling_device_type_name(info->type));
    printf("status: %s\n", dmi_status_name(info->status));
}
```

See also

`dmi_entity_data`, `dmi_entity_decode`, `dmi_entity_handle`, `dmi_entity_type`, `dmi_entity_type_id`, `dmi_type`

dmi_entity_is_decoded

Synopsis

```
#include <opendmi/entity.h>

bool dmi_entity_is_decoded(const dmi_entity_t *entity);
bool dmi_entity_is_linked(const dmi_entity_t *entity);
bool dmi_entity_is_valid(const dmi_entity_t *entity);
bool dmi_entity_is_incomplete(const dmi_entity_t *entity);
bool dmi_entity_is_partial(const dmi_entity_t *entity);
```

Description

The functions tell how far a structure has got, by the flags of its state, the `state` member of the entity:

`dmi_entity_is_decoded`
: The structure has been decoded, and `dmi_entity_info` gives the decoded structure: `DMI_ENTITY_STATE_DECODED`.

`dmi_entity_is_linked`
: The references of the structure have been resolved: `DMI_ENTITY_STATE_LINKED`.

`dmi_entity_is_valid`
: The structure has been validated: `DMI_ENTITY_STATE_VALID`.

`dmi_entity_is_incomplete`
: The data of the structure ends in the middle of the fields of a version of the specification, and only the fields before that point have been decoded: `DMI_ENTITY_STATE_INCOMPLETE`.

`dmi_entity_is_partial`
: The structure ends where an older version of the specification ends it, and holds fewer fields than the decoder knows of: `DMI_ENTITY_STATE_PARTIAL`.

The functions are defined inline in the header, and read the flags only.

Parameters

`entity`
: Structure in question, which must not be `nullptr`.

Return value

Returns `true` if the flag is set, and `false` otherwise.

Errors

None.

See also

`dmi_entity_decode`, `dmi_entity_link`, `dmi_entity_info`

dmi_entity_link

Synopsis

```
#include <opendmi/entity.h>

bool dmi_entity_link(dmi_entity_t *entity);
```

Description

The `dmi_entity_link` function resolves cross-references between SMBIOS entities. It invokes the type-specific link handler, which typically resolves handle-based references to other structures in the SMBIOS table.

This function should be called after `dmi_entity_decode` has been called on all entities, so that referenced entities are available for linking.

This function is idempotent: if linking has already been performed, it returns `true` immediately without repeating the operation.

Parameters

`entity`
: Entity descriptor.

Return value

Returns `true` on success. Returns `false` if `entity` is `nullptr`, the entity has no specification, no link handler is registered, or the type-specific link handler fails.

Errors

`DMI_ERROR_ENTITY_NOT_FOUND`
: Structure refers to a handle no structure has.

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Structure refers to a structure of a type it cannot refer to.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory is exhausted.

`DMI_ERROR_ENTITY_LINK_FAILED`
: References of the structure cannot be resolved. Raised last on every failure but the one of a structure which has no references, after the errors telling why.

See also

`dmi_entity_create`, `dmi_entity_decode`, `dmi_entity_destroy`

dmi_entity_name

Synopsis

```
#include <opendmi/entity.h>

const char *dmi_entity_name(const dmi_entity_t *entity);
```

Description

The `dmi_entity_name` function returns the human-readable name of the entity's SMBIOS structure type (e.g., "BIOS Information", "System Information", "Cooling Device"), as the specification describing the structure names it. Structures which no specification describes, e.g. the ones no signature of their type matches, are named after the range of their type: "OEM-specific" from 128 onwards, and "Unknown" below.

Parameters

`entity`
: Entity descriptor.

Return value

Returns a pointer to a string containing the type name. The returned string must not be modified or freed by the caller. If `entity` is `nullptr`, returns `nullptr`.

Errors

None.

See also

`dmi_entity_handle`, `dmi_entity_type`, `dmi_entity_data`, `dmi_entity_info`

dmi_entity_offset

Synopsis

```
#include <opendmi/entity.h>

size_t dmi_entity_offset(const dmi_entity_t *entity);
```

Description

The `dmi_entity_offset` function returns the offset the data of the structure begins at within the buffer `dmi_entity_buffer` gives, which is where a reader of the structure starts.

Parameters

`entity`
: Entity descriptor.

Return value

Offset of the data, or zero if `entity` is `nullptr`.

Errors

None.

See also

`dmi_entity_buffer`, `dmi_entity_data`, `dmi_reader_initialize`

dmi_entity_property

Synopsis

```
#include <opendmi/entity.h>

const char *dmi_entity_property(const dmi_entity_t *entity, dmi_property_t
property);
```

Description

String properties, such as the UEFI device path of a device, are given to a structure by the string property structures (type 46) which refer to it, see `dmi_string_property`. The structures are attached to the

structures they refer to as the context links them, which it does if it has `DMI_CONTEXT_FLAG_LINK`, see `dmi_create`.

The `dmi_entity_property` function returns the value of the property `property` of `entity`. If several properties of the structure have the identifier, the value of the first one in the order of the table is returned.

Parameters

`entity`

Entity to get the property of.

`property`

Identifier of the property, e.g. `DMI_PROPERTY_ID_UEFI_DEVICE_PATH`, or an identifier of the range of a vendor or of an OEM.

Return value

Returns the value of the property, which belongs to the entity and must not be modified or freed. Returns `nullptr` if `entity` is `nullptr`, if the structure has no such property or the property has no value, or if the structures have not been linked.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised if `entity` is `nullptr`. The function takes no context, so the error reaches no error queue.

A property the structure does not have is not an error.

Example

```
const char *path = dmi_entity_property(entity,
DMI_PROPERTY_ID_UEFI_DEVICE_PATH);

if (path != nullptr)
    printf("UEFI device path: %s\n", path);
```

See also

`dmi_create`, `dmi_entity_string_ex`, `dmi_string_property`

dmi_entity_string_ex

Synopsis

```
#include <opendmi/entity.h>

const char *dmi_entity_string_ex(
```

```
        const dmi_entity_t *entity,
        size_t              num,
        bool                raw);

static inline const char *dmi_entity_string(
        const dmi_entity_t *entity,
        dmi_string_t        num);
```

Description

The `dmi_entity_string_ex` function retrieves a string from the entity's string table by its 1-based index. SMBIOS structures store variable-length text strings in an unformatted section following the formatted data; each string is referenced by a 1-based index.

When `raw` is `false`, the returned string is trimmed of leading and trailing whitespace. When `raw` is `true`, the original untrimmed string is returned.

The `dmi_entity_string` function is a convenience wrapper that calls `dmi_entity_string_ex` with `raw` set to `false`.

Parameters

`entity`
: Entity descriptor.

`num`
: 1-based string index. The string section may contain more strings than can be referenced by `dmi_string_t` fields of the structure, so the index of `dmi_entity_string_ex` is not limited to 255.

`raw`
: If `true`, return the raw (untrimmed) string. If `false`, return the trimmed version.

Return value

Returns a pointer to the requested string. The returned string must not be modified or freed by the caller. Returns `nullptr` if `entity` is `nullptr`, `num` is zero, or `num` exceeds the number of strings in the entity.

Errors

`DMI_ERROR_STRING_NOT_FOUND`
: Structure has fewer strings than `num`.

See also

`dmi_entity_create`, `dmi_entity_decode`, `dmi_entity_info`

dmi_entity_type

Synopsis

```
#include <opendmi/entity.h>

const dmi_type_t *dmi_entity_type(const dmi_entity_t *entity);
```

Description

The `dmi_entity_type` function returns the structure type of the given entity, see `dmi_type`: the type of the specification the entity is decoded by, or would be decoded by if it has not been decoded yet. The type tells the C type `dmi_entity_info` returns, and is compared to the constants the library and the extension modules declare, e.g. `DMI_TYPE(system)`.

The type number the entity is found at is given by `dmi_entity_type_id`, which may differ from the number of its type, e.g. for the structures a vendor places at type numbers of its own.

Parameters

`entity`
: Entity descriptor.

Return value

Returns the structure type of the entity. Returns `nullptr` if `entity` is `nullptr`, or if no specification of the enabled modules describes the structures found at its type number.

Errors

None.

Examples

```
if (dmi_entity_type(entity) == DMI_TYPE(intel_vpro))
    printf("Intel vPro information at handle 0x%04X\n",
dmi_entity_handle(entity));
```

See also

`dmi_entity_type_id`, `dmi_entity_handle`, `dmi_entity_name`, `dmi_entity_info`, `dmi_type`

dmi_entity_type_id

Synopsis

```
#include <opendmi/entity.h>
```

```
dmi_type_id_t dmi_entity_type_id(const dmi_entity_t *entity);
```

Description

The `dmi_entity_type_id` function returns the type number the given entity is found at in the table, see `dmi_type_id`. Types 0 through 127 are defined by the SMBIOS specification, and types 128 through 255 are available for OEM- and system-specific information.

The number tells where the structure is, rather than what it is: vendors give the same OEM type numbers to different structures, and place the structures of others at numbers of their own. What the structure is decoded into is told by its structure type, which `dmi_entity_type` returns.

Parameters

`entity`
: Entity descriptor.

Return value

Returns the type number of the entity. If `entity` is `nullptr`, returns `DMI_TYPE_ID_INVALID`.

Errors

None.

See also

`dmi_entity_type`, `dmi_type_id`, `dmi_entity_handle`

dmi_spec_name

Synopsis

```
#include <opendmi/entity.h>

const char *dmi_spec_name(const dmi_entity_spec_t *spec);
```

Description

The `dmi_spec_name` function returns the printable name of the structures a specification decodes, e.g. `System information`.

The name is translated to the locale set by `dmi_set_locale`, if the library has a translation for the specification: the translation is the `name` string of the table named by the code of the specification, e.g. `system`. Otherwise the name is the `name` member of the specification.

The specification is named whether the module providing it is enabled in a context or not, since the caller holds the specification itself. The specification a structure has been decoded by is the `spec` member of the entity, and the specification a type number is mapped to in a context is returned by `dmi_type_spec`;

`dmi_type_name` names a type number the same way, and also the types no enabled specification is mapped to.

The name belongs to the specification or to the translation, and must not be freed. An untranslated name is static; a translated one is valid until the locale is changed by `dmi_set_locale`.

Parameters

`spec`
: Specification to name.

Return value

Returns the printable name, or `nullptr` if `spec` is `nullptr`.

Errors

None.

See also

`dmi_type_name`, `dmi_type_spec`, `dmi_set_locale`, `dmi_type`

dmi_type_find

Synopsis

```
#include <opendmi/entity.h>

dmi_type_id_t dmi_type_find(dmi_context_t *context, const char *code);
```

Description

The `dmi_type_find` function returns the type number at which the structures of the specification whose code is `code`, e.g. `system` or `hpe-proliant-info`, are found in a context. It is the reverse of `dmi_type_spec`, e.g. to select structures by the code of their type given on the command line.

The number is the one the specification is mapped to in the context, which relocations of the vendor modules may make different from the number of the `type` member of the specification. Only the specifications mapped in the context are found, so the result depends on the modules enabled in it, see `dmi_type_spec`: the code of a module which is not enabled is not found, and the code of a module enabled for the platform only after the data has been read by `dmi_open`.

Every specification mapped to a number is searched, including the ones told apart by signatures. If the specification is mapped to several numbers, the lowest one is returned. Codes are compared exactly, case included.

Parameters

`context`
: Context the specification is looked up in.

`code`
: Code of the specification.

Return value

Returns the type number, or `DMI_TYPE_ID_INVALID` if `context` or `code` is `nullptr`, or if no specification with the code is mapped in the context.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `code` is `nullptr`.

A code which is not found is not an error.

See also

`dmi_type_spec`, `dmi_type_name`, `dmi_type_id`

dmi_type_name

Synopsis

```
#include <opendmi/entity.h>

const char *dmi_type_name(dmi_context_t *context, dmi_type_id_t type);
```

Description

The `dmi_type_name` function returns the printable name of the structures found at the type number `type` in a context, e.g. `System information` for type 1.

The name is the one `dmi_spec_name` gives the specification the number is mapped to in the context, see `dmi_type_spec`, which depends on the modules enabled in it: vendor modules map their specifications to OEM numbers, and may relocate them to other numbers than the ones they are declared with. A number several specifications told apart by signatures are mapped to is named by the one without a signature, or by the first one if they all have signatures, so the name of the structure itself is better taken from the specification it has been decoded by, the `spec` member of the entity.

A number no specification is mapped to is named by its range: `OEM-specific` for the numbers 128 through 255, which are left to OEMs, and `Unknown` for the others, including the numbers of the specification which are not assigned yet, the numbers of the modules which are not enabled, and the numbers out of the range of type numbers. Without a context no specification is mapped, so every number is named by its range.

Names are translated to the locale set by `dmi_set_locale`, if the library has a translation for them. The name belongs to the library, and must not be freed. An untranslated name is static; a translated one is valid until the locale is changed by `dmi_set_locale`.

Parameters

`context`

Context the type number is looked up in, or `nullptr`.

`type`

Type number to name, see `dmi_type_id`.

Return value

Returns the printable name; the function never returns `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_INVALID`

Type number is out of the range of 0 through 255, which is still named `Unknown`.

See also

`dmi_spec_name`, `dmi_type_spec`, `dmi_type_id`, `dmi_type`, `dmi_set_locale`

dmi_type_spec

Synopsis

```
#include <opendmi/entity.h>

const dmi_entity_spec_t *dmi_type_spec(dmi_context_t *context, dmi_type_id_t
type);
```

Description

The `dmi_type_spec` function returns the specification which decodes the structures found at the type number `type` in a context.

Every context maps the type numbers to specifications: the standard ones are mapped when the context is created, and the ones of the vendor modules when the modules are enabled, explicitly by `dmi_add_extension`, or for the platform the data comes from by `dmi_open` and `dmi_set_platform`. Vendor modules map their specifications to OEM numbers, and may relocate them to other numbers than the ones they are declared with, so the `type` member of the specification may tell another number than `type`. The modules enabled for a platform are disabled again by `dmi_close`, so the specification a number is mapped to may be different before the data is read and after it.

A number may be mapped to several specifications at once, which are told apart by the signatures of the structures, see `dmi_signature_t`, and are chosen for each structure as it is decoded. Such a number is

represented by the specification without a signature, or by the first one with a signature if all of them have one. The specification a structure has actually been decoded by is the `spec` member of the entity.

Specifications are constant data of the library and of its modules, so the one returned stays valid as long as the library is loaded, whatever the mapping becomes later.

Parameters

`context`
: Context the type number is looked up in.

`type`
: Type number to look up, see `dmi_type_id`.

Return value

Returns the specification, or `nullptr` if `context` is `nullptr`, if no specification is mapped to the number, e.g. a number of a module which is not enabled or a number the specification does not assign yet, or if the number is out of range.

Errors

`DMI_ERROR_ARGUMENT_INVALID`
: Type number is out of the range of 0 through 255.

A number no specification is mapped to is not an error.

See also

`dmi_type_name`, `dmi_type_find`, `dmi_spec_name`, `dmi_type_id`, `dmi_type`, `dmi_add_extension`, `dmi_set_platform`

<opendmi/registry.h>

dmi_registry_create

Synopsis

```
#include <opendmi/registry.h>

#define DMI_REGISTRY_CAPACITY 64

dmi_registry_t *dmi_registry_create(dmi_context_t *context, size_t capacity);
void dmi_registry_destroy(dmi_registry_t *registry);
```

Description

A registry holds the structures of a table as entities, in the order the table holds them, and indexes them by their handles in a hash table.

The `dmi_registry_create` function creates an empty registry bound to `context`, with a hash table of `capacity` cells. The capacity only tells how long the chains of the cells grow, and does not limit the number of structures the registry holds.

A context creates a registry of its own as it is opened, fills it with the structures of its table, and destroys it as it is closed; the program gets it with `dmi_get_registry`, and neither creates nor destroys it. The functions which fill a registry are internal to the library, so a registry created by the program stays empty.

The `dmi_registry_destroy` function destroys a registry along with the entities it holds, so the entities obtained from it must not be used after that. Passing `nullptr` does nothing.

Parameters

`context`
: Context the registry belongs to, whose error queue its errors are raised against, and whose flags the lookups read.

`capacity`
: Number of cells of the hash table, or zero for `DMI_REGISTRY_CAPACITY`.

`registry`
: Registry to destroy, or `nullptr`.

Return value

On success, `dmi_registry_create` returns the registry. On failure, which happens if `context` is `nullptr` or if memory is exhausted, it returns `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised by `dmi_registry_create` if `context` is `nullptr`, in which case it is not reported, since there is no context to hold it.

`DMI_ERROR_OUT_OF_MEMORY`

Raised by `dmi_registry_create` if memory is exhausted.

See also

`dmi_get_registry`, `dmi_registry_iter_initialize`, `dmi_registry_lookup`, `dmi_registry_status`

dmi_registry_iter_initialize

Synopsis

```
#include <opendmi/registry.h>

typedef struct dmi_registry_iter {
    dmi_registry_t       *registry;
    dmi_filter_t         *filter;
    dmi_registry_entry_t *position;
    bool                  is_done;
} dmi_registry_iter_t;

bool dmi_registry_iter_initialize(
        dmi_registry_iter_t *iter,
        dmi_registry_t      *registry,
        dmi_filter_t        *filter);

bool dmi_registry_iter_has_next(dmi_registry_iter_t *iter);

dmi_entity_t *dmi_registry_iter_next(dmi_registry_iter_t *iter);
```

Description

An iterator walks the structures of a registry in the order of the table, and yields the ones a filter matches, see `dmi_filter_match`, or all of them if there is no filter.

The `dmi_registry_iter_initialize` function sets up an iterator to walk `registry`, from its first structure. An iterator which has been used already is set up again the same way.

The `dmi_registry_iter_next` function moves the iterator to the next structure the filter matches, and returns its entity. Once the iterator has run past the last structure, it stays there, and the function returns `nullptr` for every call that follows.

The `dmi_registry_iter_has_next` function tells whether the next call to `dmi_registry_iter_next`

would return a structure, without moving the iterator.

An iterator is a plain variable, which is not allocated and needs no destroying, and several iterators may walk a registry at once. The members of the iterator are private to the library. The registry and the filter are not copied, and must stay valid while the iterator is used. The entities belong to the registry, see `dmi_registry_lookup`.

Parameters

`iter`
: Iterator to set up, to move or to query.

`registry`
: Registry to walk, see `dmi_get_registry`.

`filter`
: Filter which tells the structures to yield, or `nullptr` to yield all of them.

Return value

`dmi_registry_iter_initialize` returns `true` on success, and `false` if `iter` or `registry` is `nullptr`.

`dmi_registry_iter_has_next` returns `true` if a structure is left to yield, and `false` otherwise, or if `iter` is `nullptr`.

`dmi_registry_iter_next` returns the entity of the next structure, or `nullptr` if no structure is left to yield, or if `iter` is `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised by `dmi_registry_iter_initialize` if `iter` or `registry` is `nullptr`, in which case it is not reported, since there is no context to hold it.

`dmi_registry_iter_has_next` and `dmi_registry_iter_next` raise no errors, and take `nullptr` for an iterator with nothing left to yield.

Example

Print the locators of the memory devices:

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_registry_iter_t iter;
dmi_entity_t       *entity;

if (dmi_registry_iter_initialize(&iter, registry, nullptr)) {
    while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
        if (dmi_entity_type(entity) != DMI_TYPE(memory_device))
            continue;

        const dmi_memory_device_t *device =
```

```
                dmi_entity_info(entity, DMI_TYPE(memory_device));

        if ((device != nullptr) && (device->device_locator != nullptr))
            printf("%s\n", device->device_locator);
    }
}
```

The structure type is checked before the information is asked for, since `dmi_entity_info` raises an error for a structure of another type. See `dmi_filter_create` for an iterator which yields the structures found at the given type numbers only.

See also

`dmi_entity_info`,`dmi_entity_type`,`dmi_filter_create`,`dmi_filter_match`,`dmi_get_registry`, `dmi_registry_lookup`

dmi_registry_lookup

Synopsis

```
#include <opendmi/registry.h>

#define DMI_TYPE_ANY ((const dmi_type_t *)nullptr)

dmi_entity_t *dmi_registry_lookup(
        dmi_registry_t   *registry,
        dmi_handle_t      handle,
        const dmi_type_t *type,
        bool              optional);

dmi_entity_t *dmi_registry_lookup_any(
        dmi_registry_t          *registry,
        dmi_handle_t             handle,
        const dmi_type_t *const *types,
        bool                     optional);

dmi_entity_t *dmi_registry_lookup_first(
        dmi_registry_t   *registry,
        const dmi_type_t *type,
        bool              optional);

dmi_entity_t *dmi_registry_lookup_first_id(
        dmi_registry_t *registry,
        dmi_type_id_t   type_id,
        bool            optional);
```

Description

The `dmi_registry_lookup` function finds the structure of a registry which has the handle `handle`. Unless `type` is `DMI_TYPE_ANY`, the structure must be of the structure type `type`, which is the type of the specification the structure is decoded by, see `dmi_entity_type`, whatever type number the structure is found at. A structure of no known type has no structure type, and only matches `DMI_TYPE_ANY`.

The `dmi_registry_lookup_any` function does the same, but the structure may be of any of the structure types listed by `types`.

The handles `DMI_HANDLE_INVALID` (0xFFFF) and `DMI_HANDLE_UNSUPPORTED` (0xFFFE) stand for no structure: looking them up gives `nullptr`, and raises no error. Some firmware writes the handle 0x0000 for a reference which is not set as well, while 0x0000 is a handle a structure may have, so a structure of handle 0x0000 of another type than the expected one is taken for no structure, and no error is raised, unless the context of the registry has `DMI_CONTEXT_FLAG_STRICT`.

The `dmi_registry_lookup_first` function finds the first structure of the structure type `type`, in the order of the table, and the `dmi_registry_lookup_first_id` function the first structure found at the type number `type_id`, whatever specification it is decoded by, which is how the structures of no known type are looked up.

A structure which is not found is reported as an error, unless `optional` is `true`. A structure which is found, but is of another type than the expected one, is always reported.

The entities belong to the registry, and must not be destroyed by the caller. They are valid until the registry is destroyed, which for the registry of a context happens when the context is closed.

Parameters

`registry`

Registry to look the structure up in, see `dmi_get_registry`.

`handle`

Handle of the structure.

`type`

Expected structure type, e.g. `DMI_TYPE(system)`. For `dmi_registry_lookup` it may be `DMI_TYPE_ANY` to accept a structure of any type, while `dmi_registry_lookup_first` requires a type.

`types`

Array of the expected structure types, terminated by `nullptr`, or `nullptr` to accept a structure of any type.

`type_id`

Type number the structure is found at, e.g. `DMI_TYPE_ID(SYSTEM)`.

`optional`

Whether a missing structure is not an error.

Return value

The functions return the entity of the structure, or `nullptr` if `registry` is `nullptr`, if `handle` stands for no structure, if no structure matches, or if the structure is of another type than the expected one.

`dmi_registry_lookup_first` returns `nullptr` if `type` is `nullptr`, and `dmi_registry_lookup_first_id` if `type_id` is `DMI_TYPE_ID_INVALID`.

Errors

The errors are raised against the context of the registry.

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `registry` is `nullptr`, in which case it is not reported, since there is no context to hold it, and by `dmi_registry_lookup_first` if `type` is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised by `dmi_registry_lookup_first_id` if `type_id` is `DMI_TYPE_ID_INVALID`.

`DMI_ERROR_ENTITY_NOT_FOUND`
: Raised if no structure matches, unless `optional` is `true`.

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Raised by `dmi_registry_lookup` and `dmi_registry_lookup_any` if the structure of the handle is of another type than the expected one, whatever `optional` is.

Example

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                     DMI_TYPE(system), true);

if (entity != nullptr) {
    const dmi_system_t *system = dmi_entity_info(entity, DMI_TYPE(system));

    if ((system != nullptr) && (system->product != nullptr))
        printf("%s\n", system->product);
}
```

See also

`dmi_entity_info`, `dmi_entity_type`, `dmi_entity_type_id`, `dmi_get_registry`, `dmi_registry_iter_initialize`, `dmi_registry_resolve`

dmi_registry_resolve

Synopsis

```
#include <opendmi/registry.h>

bool dmi_registry_resolve(
        dmi_registry_t   *registry,
```

```
        dmi_handle_t      handle,
        const dmi_type_t *type,
        dmi_entity_t    **pentity);

bool dmi_registry_resolve_any(
        dmi_registry_t          *registry,
        dmi_handle_t             handle,
        const dmi_type_t *const *types,
        dmi_entity_t           **pentity);

bool dmi_registry_resolve_id(
        dmi_registry_t *registry,
        dmi_handle_t    handle,
        dmi_type_id_t   type_id,
        dmi_entity_t  **pentity);
```

Description

The functions resolve the handle a structure refers to another one by into the entity of that structure, and are meant for the link handlers of the structure types, including the ones of a module built out of the tree of the library, see `dmi_entity_link`.

Unlike `dmi_registry_lookup`, they tell a reference which is not set, which is not an error, from a broken one, which is: the reference is not set if the handle is `DMI_HANDLE_INVALID` (0xFFFF) or `DMI_HANDLE_UNSUPPORTED` (0xFFFE), and broken if there is no structure of the handle, or if the structure is of another type than the expected one. Some firmware writes the handle 0x0000 for a reference which is not set as well, while 0x0000 is a handle a structure may have, so a reference of 0x0000 to a structure of another type than the expected one is taken as not set, unless the context of the registry has `DMI_CONTEXT_FLAG_STRICT`.

The `dmi_registry_resolve` function expects a structure of the structure type `type`, which is the type of the specification the structure is decoded by, see `dmi_entity_type`, and the `dmi_registry_resolve_any` function a structure of any of the structure types listed by `types`.

The `dmi_registry_resolve_id` function expects a structure found at the type number `type_id`, whatever specification it is decoded by, for the references whose type the data gives itself, e.g. the items of a group association.

Parameters

`registry`
: Registry to look the structure up in.

`handle`
: Handle the reference holds.

`type`
: Expected structure type, or `DMI_TYPE_ANY` to accept a structure of any type.

`types`
: Array of the expected structure types, terminated by `nullptr`, or `nullptr` to accept a structure of any type. The array must list at least one type.

`type_id`
: Expected type number, or `DMI_TYPE_ID_INVALID` to accept a structure of any type.

`pentity`
: Variable to store the entity of the structure in. It is set to `nullptr` if the reference is not set or is broken.

Return value

The functions return `true` if the reference has been resolved or is not set, and `false` if it is broken, if `registry` or `pentity` is `nullptr`, or if `types` lists no type.

Errors

The errors are raised against the context of the registry.

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `registry` or `pentity` is `nullptr`. Without a registry it is not reported, since there is no context to hold it.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised by `dmi_registry_resolve_any` if `types` lists no type.

`DMI_ERROR_ENTITY_NOT_FOUND`
: Raised if there is no structure of the handle.

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Raised if the structure of the handle is of another type than the expected one.

Example

```
static bool cooling_device_link(dmi_entity_t *entity)
{
    dmi_context_t        *context  = dmi_entity_context(entity);
    dmi_registry_t       *registry = dmi_get_registry(context);
    dmi_cooling_device_t *info     = dmi_entity_info(entity,
                                                     DMI_TYPE(cooling_device));

    return dmi_registry_resolve(registry, info->probe_handle,
                                DMI_TYPE(temperature_probe), &info->probe);
}
```

See also

`dmi_entity_link`, `dmi_entity_type`, `dmi_entity_type_id`, `dmi_get_registry`,

dmi_registry_status

Synopsis

```
#include <opendmi/registry.h>

enum dmi_registry_status {
    DMI_REGISTRY_STATUS_SCANNED   = (1 << 0),
    DMI_REGISTRY_STATUS_TRUNCATED = (1 << 1),
    DMI_REGISTRY_STATUS_DECODED   = (1 << 2),
    DMI_REGISTRY_STATUS_LINKED    = (1 << 3),
    DMI_REGISTRY_STATUS_OVERLAID  = (1 << 4)
};

unsigned dmi_registry_status(const dmi_registry_t *registry);
```

Description

The `dmi_registry_status` function returns the status flags of a registry, which tell how far the structures of the table have got as the context is opened:

`DMI_REGISTRY_STATUS_SCANNED`

The table has been scanned, and an entity has been created for every structure found.

`DMI_REGISTRY_STATUS_TRUNCATED`

The table area ends before the end-of-table structure, or a structure runs past its end, so the structures which follow are missing. Unless the context has `DMI_CONTEXT_FLAG_STRICT`, a structure whose length is shorter than its header is taken the same way, since the structures which follow it cannot be found.

`DMI_REGISTRY_STATUS_DECODED`

The structures have been decoded. Unless the context has `DMI_CONTEXT_FLAG_STRICT`, the structures which cannot be decoded are left undecoded, and the flag is set all the same.

`DMI_REGISTRY_STATUS_LINKED`

The references between the structures have been resolved, which happens if the context has `DMI_CONTEXT_FLAG_LINK`, see `dmi_create`. Unless the context has `DMI_CONTEXT_FLAG_STRICT`, the broken references are left unresolved, and the flag is set all the same.

`DMI_REGISTRY_STATUS_OVERLAID`

At least one structure carries additional information entries, which happens if the context has `DMI_CONTEXT_FLAG_OVERLAY` and the table has entries referring to structures of it. Such structures are decoded from copies with the entries applied, which the table does not hold.

Parameters

`registry`
: Registry to query, see `dmi_get_registry`.

Return value

Returns the status flags of the registry, a combination of the values above, or zero if `registry` is `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `registry` is `nullptr`, in which case it is not reported, since there is no context to hold it.

Example

```
if (dmi_registry_status(dmi_get_registry(context)) &
DMI_REGISTRY_STATUS_TRUNCATED)
    fprintf(stderr, "SMBIOS table is truncated\n");
```

See also

`dmi_create`, `dmi_get_registry`, `dmi_lint`, `dmi_open`

<opendmi/filter.h>

dmi_filter_add_type

Synopsis

```
#include <opendmi/filter.h>

bool dmi_filter_add_type(dmi_filter_t *filter, dmi_type_id_t type);
bool dmi_filter_add_handle(dmi_filter_t *filter, dmi_handle_t handle);
bool dmi_filter_add_module(dmi_filter_t *filter, const dmi_module_t *module);
```

Description

The functions add an entry to one of the lists of a filter, see `dmi_filter_create`. Once any of the lists has an entry, the filter matches only the structures which match an entry of any of the lists, and meet its mask:

- The `dmi_filter_add_type` function adds a type number, which matches the structures found at it, whatever specification they are decoded by, see `dmi_entity_type_id`.
- The `dmi_filter_add_handle` function adds a handle, which matches the structure of the handle.
- The `dmi_filter_add_module` function adds an extension module, which matches the structures decoded by its specifications, whatever type numbers the platform places them at. The structures of a module which is not enabled in the context are not decoded by its specifications, and are not matched, see `dmi_add_extension`.

The entries are not checked, and an entry added twice is kept twice. The module is not copied, and must stay valid while the filter is used; the modules of the library and the registered ones stay valid for as long as the program runs.

Parameters

`filter`
: Filter to add the entry to.

`type`
: Type number, e.g. `DMI_TYPE_ID(MEMORY_DEVICE)`.

`handle`
: Handle of a structure.

`module`
: Extension module, e.g. `&dmi_hpe_module` or the one found by `dmi_module_find`.

Return value

The functions return `true` on success, and `false` if `filter` or `module` is `nullptr`, or if memory is exhausted, in which case the filter is left as it was.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised if `filter` is `nullptr`, in which case it is not reported, since there is no context to hold it, and by `dmi_filter_add_module` if `module` is `nullptr`.

`DMI_ERROR_OUT_OF_MEMORY`

Raised if memory is exhausted.

The errors are raised against the context the filter has been created with.

Example

Match the structures of type 17 along with the ones of the HPE module:

```
#include <opendmi/module/hpe.h>

dmi_filter_t *filter = dmi_filter_create(context);

if ((filter == nullptr) ||
    !dmi_filter_add_type(filter, DMI_TYPE_ID(MEMORY_DEVICE)) ||
    !dmi_filter_add_module(filter, &dmi_hpe_module)) {
    dmi_filter_destroy(filter);
    return false;
}
```

See also

`dmi_add_extension`, `dmi_entity_type_id`, `dmi_filter_create`, `dmi_filter_match`, `dmi_module_find`, `dmi_registry_iter_initialize`

dmi_filter_create

Synopsis

```
#include <opendmi/filter.h>

typedef enum dmi_filter_mask {
    DMI_FILTER_MASK_COMMON   = 1 << 0,
    DMI_FILTER_MASK_OEM      = 1 << 1,
    DMI_FILTER_MASK_INACTIVE = 1 << 2,
    DMI_FILTER_MASK_UNKNOWN  = 1 << 3,
    DMI_FILTER_MASK_DEFAULT  = DMI_FILTER_MASK_COMMON | DMI_FILTER_MASK_OEM,
    DMI_FILTER_MASK_ALL      = DMI_FILTER_MASK_COMMON | DMI_FILTER_MASK_OEM |
                               DMI_FILTER_MASK_INACTIVE |
DMI_FILTER_MASK_UNKNOWN
} dmi_filter_mask_t;
```

```
typedef struct dmi_filter {
    dmi_context_t *context;
    unsigned       mask;
    dmi_vector_t   handles;
    dmi_vector_t   types;
    dmi_vector_t   modules;
} dmi_filter_t;

dmi_filter_t *dmi_filter_create(dmi_context_t *context);
void dmi_filter_destroy(dmi_filter_t *filter);
```

Description

A filter tells the structures of a registry an iterator yields, see `dmi_registry_iter_initialize`, by two conditions, both of which a structure must meet, see `dmi_filter_match`:

- The `mask` member tells the categories of the structures to match: the ones of the types the SMBIOS specification defines (`DMI_FILTER_MASK_COMMON`), the OEM-specific ones (`DMI_FILTER_MASK_OEM`), the inactive ones (`DMI_FILTER_MASK_INACTIVE`) and the ones of no known type (`DMI_FILTER_MASK_UNKNOWN`). The mask is set by the caller directly, e.g. to `DMI_FILTER_MASK_DEFAULT`, which leaves the inactive structures and the ones of no known type out.
- The lists of handles, type numbers and modules, which are filled with `dmi_filter_add_handle`, `dmi_filter_add_type` and `dmi_filter_add_module`, tell the structures to match. Empty lists match every structure.

The `dmi_filter_create` function creates a filter with the mask `DMI_FILTER_MASK_ALL` and empty lists, which matches every structure.

The `dmi_filter_destroy` function destroys a filter along with its lists. Passing `nullptr` does nothing. A filter must not be destroyed while an iterator uses it.

Parameters

`context`

Context whose error queue the errors of allocation are raised against, or `nullptr`, in which case they are not reported.

`filter`

Filter to destroy, or `nullptr`.

Return value

On success, `dmi_filter_create` returns the filter. On failure, which happens when memory is exhausted only, it returns `nullptr`.

Errors

`DMI_ERROR_OUT_OF_MEMORY`

Raised by `dmi_filter_create` if memory is exhausted.

Example

Print the handles of the structures found at the type numbers of memory devices:

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_filter_t       *filter   = dmi_filter_create(context);
dmi_registry_iter_t iter;
dmi_entity_t       *entity;

if ((filter != nullptr) &&
    dmi_filter_add_type(filter, DMI_TYPE_ID(MEMORY_DEVICE)) &&
    dmi_registry_iter_initialize(&iter, registry, filter)) {
    while ((entity = dmi_registry_iter_next(&iter)) != nullptr)
        printf("0x%04x\n", dmi_entity_handle(entity));
}

dmi_filter_destroy(filter);
```

See also

`dmi_filter_add_type`, `dmi_filter_match`, `dmi_registry_iter_initialize`

dmi_filter_match

Synopsis

```
#include <opendmi/filter.h>

bool dmi_filter_match(const dmi_filter_t *filter, const dmi_entity_t *entity);
bool dmi_filter_is_empty(const dmi_filter_t *filter);
```

Description

The `dmi_filter_match` function tells whether a structure meets a filter, see `dmi_filter_create`, which is the case if it meets both conditions:

1. Every category of the structure is in the mask of the filter. A structure found at a type number below 128 is a common one, which requires `DMI_FILTER_MASK_COMMON`, and one found at 128 or above an OEM-specific one, which requires `DMI_FILTER_MASK_OEM`. An inactive structure (type 126) requires `DMI_FILTER_MASK_INACTIVE` as well, and a structure of no known type, which no specification decodes, requires `DMI_FILTER_MASK_UNKNOWN`.
2. The structure matches an entry of any of the lists of the filter: its handle is in the list of handles, its type number in the list of type numbers, or its specification belongs to a module of the list of modules, see `dmi_filter_add_type`. A filter whose lists are empty matches every structure which meets its mask.

The `dmi_filter_is_empty` function tells whether the lists of handles, type numbers and modules of a

filter are all empty. The mask is not taken into account.

An iterator checks the structures with `dmi_filter_match`, see `dmi_registry_iter_initialize`, and takes no filter for one which matches every structure. `dmi_filter_match` itself matches nothing with no filter.

Parameters

`filter`
: Filter to check against, or to query.

`entity`
: Structure to check.

Return value

`dmi_filter_match` returns `true` if the structure meets the filter, and `false` otherwise, or if `filter` or `entity` is `nullptr`.

`dmi_filter_is_empty` returns `true` if the lists of the filter are empty, or if `filter` is `nullptr`, and `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised by `dmi_filter_match` against the context of the filter if `entity` is `nullptr`.

Both functions take a `filter` of `nullptr` silently.

See also

`dmi_entity_handle`, `dmi_entity_type_id`, `dmi_filter_add_type`, `dmi_filter_create`, `dmi_registry_iter_initialize`

<opendmi/attribute.h>

DMI_ATTRIBUTE

Synopsis

```
#include <opendmi/attribute.h>

DMI_ATTRIBUTE(structure, member, TYPE, { ... })
DMI_ATTRIBUTE_ARRAY(structure, member, counter, TYPE, { ... })
DMI_ATTRIBUTE_VECTOR(structure, member, TYPE, { ... })

DMI_ATTRIBUTES({ attribute, ..., {} })
```

Description

The macros declare the attributes of a specification, the `attributes` member of `dmi_entity_spec_t`, which describe the values of the decoded structure: what they are called, how they are written, what they mean, and what they refer to. The output formats of the command line tool, linking, the lint rules and anonymization all go by them, see `dmi_attributes_walk`.

Attributes describe the values, while the fields declared with `DMI_FIELD` describe the bytes they are read from; the two are joined by the member they name.

`DMI_ATTRIBUTE`

Value of `member`, whose type `TYPE` names, e.g. `ENUM`.

`DMI_ATTRIBUTE_ARRAY`

Elements of an array the structure points at, as `DMI_FIELD_ARRAY` decodes them, whose number `counter` holds.

`DMI_ATTRIBUTE_VECTOR`

Elements of the array member `member`, which the structure holds in place, as `DMI_FIELD_VECTOR` decodes them.

`DMI_ATTRIBUTES`

List of the attributes, a compound literal of `dmi_attribute_t`, terminated by an empty attribute, `{}`, which the list itself ends with.

Types

`TYPE` is the suffix of `dmi_attribute_type_t`:

`STRUCT`

Nested structure, whose attributes `.attrs` lists.

`HANDLE`

Handle of another structure.

`STRING`
: String of the structure.

`BOOL`
: Boolean, which `.values` may name.

`INTEGER`
: Integer, signed with `DMI_ATTRIBUTE_FLAG_SIGNED`.

`DECIMAL`
: Integer with `.scale` digits after the decimal point.

`SIZE`
: Size in bytes, written in the largest unit it is a whole number of.

`ADDRESS`
: Address.

`ENUM`
: Enumeration, whose values `.values` names.

`SET`
: Set of flags, whose bits `.values` names.

`VERSION`
: Version number.

`DATE`
: Date.

`UUID`
: UUID.

`BINARY`
: Bytes taken as they are.

Variant attributes, whose description depends on another member, have a page of their own, `DMI_ATTRIBUTE_VARIANT`.

Parameters

`structure`, `member`
: Structure and member holding the value.

`counter`
: Member holding the number of the elements of an array.

`{ … }`
: Parameters, given as the designated initializers of `dmi_attribute_params_t`:

`.code`

Code name, which the value is serialized by, e.g. `bus-width`.

`.name`

Printable name, e.g. `Data bus width`, which is translated, see `dmi_attribute_name`.

`.unit`, `.scale`

Unit of an integer or a decimal, and the number of its decimal digits.

`.unspec`, `.unknown`

Pointers to the values standing for "unspecified" and "unknown", which are written as such rather than as values.

`.flags`

Flags of `dmi_attribute_flag_t`, e.g. `DMI_ATTRIBUTE_FLAG_HEX`, or `DMI_ATTRIBUTE_FLAG_PRIVATE` for the values identifying the system, which anonymization replaces, or `DMI_ATTRIBUTE_FLAG_OWNED` for a string the structure owns rather than the table: a handler allocates it, e.g. with `dmi_string_set`, into a member of `char *`, and the library frees it as the structure is released.

`.values`

Names of the values of an enumeration, a set or a boolean, see `DMI_NAMES`.

`.targets`

Types a handle may refer to, terminated by `nullptr`.

`.link`

Member the structure a handle refers to is put into once the table is linked, see `dmi_member`.

`.minimum`, `.maximum`

Pointers to the limits the specification puts on the value, which the lint rules check.

`.attrs`

Attributes of a nested structure, or of an element of an array of them.

`.level`

Version of the specification the value appears in, which the output leaves out for the structures of an earlier level.

Example

```
DMI_ATTRIBUTE(dmi_slot_t, type, ENUM, {
    .code    = "type",
    .name    = "Type",
    .unspec  = &(const dmi_slot_type_t){ DMI_SLOT_TYPE_UNSPEC },
    .unknown = &(const dmi_slot_type_t){ DMI_SLOT_TYPE_UNKNOWN },
    .values  = &dmi_slot_type_names
}),
```

See also

`DMI_ATTRIBUTE_VARIANT`, `DMI_NAMES`, `DMI_FIELD`, `dmi_attributes_walk`, `dmi_member`

DMI_ATTRIBUTE_VARIANT

Synopsis

```
#include <opendmi/attribute.h>

DMI_ATTRIBUTE_VARIANT(structure, selector, { ... })
DMI_VARIANT(value, structure, member, TYPE, { ... })
DMI_VARIANT_DEFAULT(structure, member, TYPE, { ... })

DMI_VARIANTS({ variant, ..., {} })
```

Description

A variant attribute describes a value differently depending on the member `selector`, e.g. a field which means one thing for one kind of device and another for another, or which the structure holds only when a flag says so. The attribute itself carries the code, the name and the level of the value, and each of its variants, the description of the value for one value of the selector, see `DMI_ATTRIBUTE`.

`DMI_ATTRIBUTE_VARIANT`

Variant attribute, whose `.variants` lists the variants.

`DMI_VARIANT`

Variant used when the selector equals `value`.

`DMI_VARIANT_DEFAULT`

Variant used when no other one matches the selector.

`DMI_VARIANTS`

List of the variants, terminated by an empty variant, `{}`, which the list itself ends with.

The variant matching the selector describes the value, see `dmi_attribute_resolve`, and the code, name and level of the variant itself are not used. If no variant matches and there is no default one, the value does not exist, and is left out.

Parameters

`structure`

Type of the structure holding the selector and the value.

`selector`

Member the variant is chosen by, e.g. a type, or a `bool` member telling whether the structure holds the value, see `DMI_FIELD_GROUP`.

`value`
: Value of the selector the variant is used for.

`member`, `TYPE`, `{ … }`
: Member, type and parameters of the attribute the variant describes the value as, see `DMI_ATTRIBUTE`.

Example

Disable reason of a trusted module, which is there only when the module is disabled:

```
DMI_ATTRIBUTE_VARIANT(dmi_hpe_trusted_module_t, presence, {
    .code     = "disable-reason",
    .name     = "Disable reason",
    .variants = DMI_VARIANTS({
        DMI_VARIANT(DMI_HPE_TM_PRESENCE_DISABLED, dmi_hpe_trusted_module_t,
disable_reason, ENUM, {
            .values = &dmi_hpe_tm_disable_reason_names
        }),
        {}
    })
}),
```

See also

`DMI_ATTRIBUTE`, `DMI_FIELD_GROUP`, `dmi_attributes_walk`

dmi_attribute_format

Synopsis

```
#include <opendmi/attribute.h>

char *dmi_attribute_format(
        dmi_context_t         *context,
        const dmi_attribute_t *attribute,
        const void            *value,
        bool                   pretty);
```

Description

The `dmi_attribute_format` function formats the member `value` points to as a newly allocated string, the way the type and the parameters of the attribute describe it, see `DMI_ATTRIBUTE`. The `pretty` flag chooses between two forms of the value: the one meant to be read, e.g. by the text output of the command line tool, and the one meant to be parsed, e.g. by its JSON, YAML and XML outputs.

The value is formatted according to the type of the attribute:

`DMI_ATTRIBUTE_TYPE_HANDLE`

Hexadecimal number of four digits, e.g. `0x0010`.

`DMI_ATTRIBUTE_TYPE_STRING`

Copy of the string.

`DMI_ATTRIBUTE_TYPE_BOOL`

Name of the value by `.values`, if the attribute has one for it, and `yes` or `no` otherwise; the code of the value, e.g. `true` or `false`, if `pretty` is `false`.

`DMI_ATTRIBUTE_TYPE_INTEGER`

Decimal number, signed with `DMI_ATTRIBUTE_FLAG_SIGNED`, or hexadecimal one with `DMI_ATTRIBUTE_FLAG_HEX`, e.g. `0x1F`.

`DMI_ATTRIBUTE_TYPE_DECIMAL`

Decimal number with `.scale` digits after the decimal point, of which the trailing zeros are dropped, keeping at least one digit, e.g. `1.5` or `2.0`. A scale of zero formats the value as an integer.

`DMI_ATTRIBUTE_TYPE_SIZE`

Number of the bytes, or, if `pretty` is `true`, the number of the largest unit the size is a whole number of, along with the unit, e.g. `16 GiB`.

`DMI_ATTRIBUTE_TYPE_ADDRESS`

Hexadecimal number of 8 digits if the table has been read through a 32-bit entry point, and of 16 digits otherwise.

`DMI_ATTRIBUTE_TYPE_ENUM`

Name of the value by `.values`, or its code if `pretty` is `false`. A value the names do not cover is formatted as a hexadecimal number, e.g. `0x7f`, and marked as invalid if `pretty` is `true`.

`DMI_ATTRIBUTE_TYPE_SET`

Hexadecimal number of as many digits as the member holds.

`DMI_ATTRIBUTE_TYPE_VERSION`

Version number, e.g. `3.2.0`, of as many parts as `.scale` says, which are three if it is neither 1 nor 2.

`DMI_ATTRIBUTE_TYPE_DATE`

Date, see `dmi_date_format`.

`DMI_ATTRIBUTE_TYPE_UUID`

UUID in the form of RFC 4122, upper-case.

`DMI_ATTRIBUTE_TYPE_BINARY`

Bytes in hexadecimal, separated by spaces and upper-case if `pretty` is `true`. With `DMI_ATTRIBUTE_FLAG_MAC`, the bytes are separated by colons instead, and the trailing zero bytes beyond a MAC-48 address are dropped; with `DMI_ATTRIBUTE_FLAG_IP`, 4 bytes are formatted as an IPv4 address and 16 bytes as an IPv6 one, in the text form of RFC 5952.

A structure has no value of its own, and is not formatted. Neither is a variant attribute, whose member holds the selector: the value is formatted with the attribute `dmi_attribute_resolve` returns for it. The values standing for "unspecified" and "unknown" are formatted as any other, so they are told apart beforehand, see `dmi_attribute_is_unspecified`.

Names of the values are translated to the locale set by `dmi_set_locale`, while codes never are.

The caller is responsible for freeing the string with `free()` or `dmi_free`.

Parameters

`context`
: DMI context the errors are raised against, whose entry point also tells the width of the addresses.

`attribute`
: Attribute describing the member.

`value`
: Member to format: the member itself, or an element of an array.

`pretty`
: `true` to format the value to be read, `false` to format it to be parsed.

Return value

On success, returns the string. Returns `nullptr` on failure, and also if the attribute is a structure, a variant attribute or of a type which has no value, or if a string member is `nullptr`, in which case no error is raised.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `attribute` or `value` is `nullptr`. If `context` is `nullptr`, the function returns `nullptr` and the error reaches no error queue.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if an integer member is not 1, 2, 4 or 8 bytes wide, or if an enumeration attribute has no `.values`.

`DMI_ERROR_OUT_OF_MEMORY`
: Raised if the string cannot be allocated.

Example

```
static bool print_value(void *context, const dmi_attribute_node_t *node)
{
    char *text = dmi_attribute_format(context, node->attr, node->value, true);

    if (text == nullptr)
        return true;

    printf("%s: %s\n", dmi_attribute_name(node->member, nullptr), text);
    dmi_free(text);

    return true;
```

```
}

static const dmi_attribute_visitor_t visitor = {
    .value = print_value
};

dmi_attributes_walk(entity->spec->attributes, entity->info, &visitor, context);
```

See also

`DMI_ATTRIBUTE`,`dmi_attribute_is_unspecified`,`dmi_attribute_name`,`dmi_attribute_resolve`, `dmi_attributes_walk`, `dmi_free`

dmi_attribute_get_int

Synopsis

```
#include <opendmi/attribute.h>

intmax_t dmi_attribute_get_int(const dmi_attribute_t *attr, const void *value);
uintmax_t dmi_attribute_get_uint(const dmi_attribute_t *attr, const void
*value);
bool dmi_attribute_get_bool(const dmi_attribute_t *attr, const void *value);

bool dmi_attribute_is_array(const dmi_attribute_t *attr);
size_t dmi_attribute_get_count(const dmi_attribute_t *attr, const void *info);
const void *dmi_attribute_get_elements(const dmi_attribute_t *attr, const void
*value);
```

Description

The functions read the members of a decoded structure the way its attributes describe them, see `DMI_ATTRIBUTE`, so that a value is read without its C type being known, e.g. by the callbacks of `dmi_attributes_walk`.

The `dmi_attribute_get_int` function reads the integer `value` points to as a signed one, and the `dmi_attribute_get_uint` function as an unsigned one. The width of the integer is the size of the member the attribute names, `attr→value.size`, which is one of 1, 2, 4 or 8 bytes; for an element of an array, the size is the one of an element. The value is widened to `intmax_t` or `uintmax_t`, with the sign extended by `dmi_attribute_get_int`.

The `dmi_attribute_get_bool` function reads the `bool` `value` points to. The attribute is not used, and may be `nullptr`.

The `dmi_attribute_is_array` function tells whether the attribute describes several elements rather than one value: an array, declared with `DMI_ATTRIBUTE_ARRAY`, whose elements are kept apart from the structure, or a vector, declared with `DMI_ATTRIBUTE_VECTOR`, whose elements the structure holds in place. The elements are then read one by one, each as a member of its own.

The `dmi_attribute_get_count` function returns the number of the elements of an array or of a vector attribute: the fixed number of a vector, or the value of the counter member of an array, read from the structure `info` as an unsigned integer of the width of the counter. The counter of an array tells the number of the elements which have been decoded; the number is not checked against the elements, so an array whose elements have not been allocated is told by `dmi_attribute_get_elements` returning `nullptr`.

The `dmi_attribute_get_elements` function returns the first element of an array or of a vector attribute, given the member of the attribute, `value`. The member of an array points to its elements, which are kept apart from the structure, while the elements of a vector are the member itself. Elements follow each other `attr→value.size` bytes apart.

Parameters

`attr`
Attribute describing the member.

`value`
Member to read, e.g. the `value` of the node a walk of the attributes has reached.

`info`
Decoded structure holding the counter of an array, e.g. the `info` of that node.

Return value

The `dmi_attribute_get_int` and `dmi_attribute_get_uint` functions return the value of the member. They return zero if `attr` or `value` is `nullptr`, and `INTMAX_MAX` or `UINTMAX_MAX` if the size of the member is not one of the supported widths.

The `dmi_attribute_get_bool` function returns the value of the member, or `false` if `value` is `nullptr`.

The `dmi_attribute_is_array` function returns `true` for an array or a vector, and `false` otherwise or if `attr` is `nullptr`.

The `dmi_attribute_get_count` function returns the number of the elements. It returns zero if `attr` or `info` is `nullptr`, if the attribute is neither an array nor a vector, if the width of the counter is not supported, or if the counter holds a number which does not fit into `size_t`.

The `dmi_attribute_get_elements` function returns the first element, or `nullptr` if `attr` or `value` is `nullptr`, or if the elements of an array have not been allocated.

Errors

`DMI_ERROR_ARGUMENT_NULL`
Raised if `attr`, `value` or `info` is `nullptr`, except the `attr` of `dmi_attribute_get_bool`. The functions take no context, so the error reaches no error queue.

Example

The following callback of `dmi_attributes_walk` sums the unsigned integers of the arrays of a structure:

```
static dmi_attribute_walk_t sum_array(void *context, const dmi_attribute_node_t
```

```
*node)
{
    const dmi_attribute_t *attr = node->attr;
    const dmi_data_t *element = dmi_attribute_get_elements(attr, node->value);

    if (attr->type != DMI_ATTRIBUTE_TYPE_INTEGER || element == nullptr)
        return DMI_ATTRIBUTE_WALK_SKIP;

    for (size_t i = 0; i < node->count; i++, element += attr->value.size)
        *(uintmax_t *)context += dmi_attribute_get_uint(attr, element);

    return DMI_ATTRIBUTE_WALK_SKIP;
}

static const dmi_attribute_visitor_t visitor = {
    .array_start = sum_array
};
```

See also

`DMI_ATTRIBUTE`, `DMI_ATTRIBUTE_ARRAY`, `DMI_ATTRIBUTE_VECTOR`, `dmi_attribute_format`, `dmi_attribute_is_unspecified`, `dmi_attributes_walk`

dmi_attribute_is_unspecified

Synopsis

```
#include <opendmi/attribute.h>

bool dmi_attribute_is_unspecified(const dmi_attribute_t *attr, const void
*value);
bool dmi_attribute_is_unknown(const dmi_attribute_t *attr, const void *value);
```

Description

The specification reserves values of many fields which stand for no value at all: "unspecified", for the ones the firmware gives no value, and "unknown", for the ones it does not know. An attribute names them by its `.unspec` and `.unknown` parameters, see `DMI_ATTRIBUTE`, so that they are written as such rather than as values, e.g. by the output formats of the command line tool.

The `dmi_attribute_is_unspecified` function tells whether the member `value` points to is unspecified:

- if the attribute has an `.unspec` value, the member is compared with it byte by byte, over the size of the member, `attr→value.size`;
- otherwise, a string is unspecified if its pointer is `nullptr`, a handle if it is `DMI_HANDLE_INVALID`, and binary data if its `length` is zero;

- a value of any other type is never unspecified without an `.unspec` value.

The `dmi_attribute_is_unknown` function tells whether the member is unknown, by comparing it with the `.unknown` value of the attribute byte by byte, over the size of the member. A value is never unknown if the attribute has no `.unknown` value.

For an element of an array, the size of the member is the one of an element, so the element is compared the same way.

Parameters

`attr`
: Attribute describing the member.

`value`
: Member to check.

Return value

Returns `true` if the value stands for "unspecified" or "unknown", and `false` otherwise or if `attr` or `value` is `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `attr` or `value` is `nullptr`. The functions take no context, so the error reaches no error queue.

Example

```
static bool print_value(void *context, const dmi_attribute_node_t *node)
{
    dmi_context_t *dmi = context;

    if (dmi_attribute_is_unspecified(node->attr, node->value)) {
        puts("Not specified");
    } else if (dmi_attribute_is_unknown(node->attr, node->value)) {
        puts("Unknown");
    } else {
        char *text = dmi_attribute_format(dmi, node->attr, node->value, true);

        puts(text != nullptr ? text : "?");
        dmi_free(text);
    }

    return true;
}
```

See also

`DMI_ATTRIBUTE`, `dmi_attribute_format`, `dmi_attribute_get_int`, `dmi_attributes_walk`

dmi_attribute_name

Synopsis

```
#include <opendmi/attribute.h>

const char *dmi_attribute_name(const dmi_attribute_t *attr, const char *owner);
```

Description

The `dmi_attribute_name` function returns the printable name of the attribute `attr`, e.g. `Data bus width`, translated to the locale set by `dmi_set_locale` if the library has a translation for it, and the `.name` the attribute is declared with otherwise, see `DMI_ATTRIBUTE`.

Translations are keyed by the code of the attribute, `.code`, and are looked up in the `attribute` table of the structure the attribute belongs to, named by `owner`, and then in the shared `attribute` table, which holds the names common to all structures, e.g. `manufacturer`. The name of the structure takes precedence, so that a common name may be overridden where it does not fit. An attribute with no code is never translated.

The string belongs to the attribute or to the translation, and must not be freed. A translated name is valid until the locale is changed by `dmi_set_locale`.

Parameters

`attr`
: Attribute to name.

`owner`
: Code of the structure the attribute belongs to, e.g. the `code` of its specification, `dmi_entity_spec_t`, or `nullptr` to look the name up in the shared table only.

Return value

Returns the printable name of the attribute, or `nullptr` if `attr` is `nullptr`, or if the attribute has neither a translation nor a `.name`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `attr` is `nullptr`. The function takes no context, so the error reaches no error queue.

Example

```
static dmi_attribute_walk_t print_member(void *context, const
dmi_attribute_node_t *node)
{
    const dmi_entity_t *entity = context;

    puts(dmi_attribute_name(node->member, entity->spec->code));

    return DMI_ATTRIBUTE_WALK_SKIP;
}
```

See also

`DMI_ATTRIBUTE`, `dmi_attribute_format`, `dmi_attributes_walk`, `dmi_set_locale`

dmi_attribute_resolve

Synopsis

```
#include <opendmi/attribute.h>

const dmi_attribute_t *dmi_attribute_resolve(const dmi_attribute_t *attr, const
void *info);
```

Description

The `dmi_attribute_resolve` function returns the attribute which describes the value of the member `attr` declares in the decoded structure `info`.

An attribute of any type but a variant one describes its value itself, and is returned as it is. The member of a variant attribute holds the selector rather than the value, see `DMI_ATTRIBUTE_VARIANT`: the selector is read from `info` as a signed integer of the width of the member, and the variant whose selector it equals is returned, or the default variant if no other one matches. The value is then read, checked and formatted with the attribute returned, e.g. by `dmi_attribute_format`, while the variant attribute keeps naming the member, since the code, the name and the level of a variant are not used.

`dmi_attributes_walk` resolves the attributes on its own, and gives the callbacks both of them.

The attribute returned belongs to the specification, and must not be freed.

Parameters

`attr`
: Attribute to resolve.

`info`
: Decoded structure holding the member, e.g. the `info` member of an entity.

Return value

Returns `attr` if it is not a variant attribute, and the variant matching the selector or the default variant otherwise. Returns `nullptr` if no variant matches and there is no default one, which means the structure holds no value for the attribute, or if `attr` or `info` is `nullptr`, or if `attr` is a variant attribute without `.variants`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `attr` or `info` is `nullptr`. The function takes no context, so the error reaches no error queue.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if `attr` is a variant attribute whose `.variants` is `nullptr`, and reaches no error queue either.

Example

```
const dmi_data_t *info = entity->info;
const dmi_attribute_t *value_attr = dmi_attribute_resolve(attr, info);

if (value_attr != nullptr) {
    char *text = dmi_attribute_format(context, value_attr,
                                      info + value_attr->value.offset, true);

    if (text != nullptr) {
        printf("%s: %s\n", dmi_attribute_name(attr, entity->spec->code), text);
        dmi_free(text);
    }
}
```

See also

`DMI_ATTRIBUTE_VARIANT`, `dmi_attribute_format`, `dmi_attributes_walk`

dmi_attributes_link

Synopsis

```
#include <opendmi/attribute.h>

bool dmi_attributes_link(dmi_entity_t *entity);
void dmi_attributes_unlink(dmi_entity_t *entity);
```

Description

A handle names another structure of the table by its number. An attribute of a handle may name the member the structure the handle refers to is put into, its `.link` parameter, along with the types the

structure may be of, its `.targets`, see `DMI_ATTRIBUTE`. This is the part of linking the specifications describe rather than perform.

The `dmi_attributes_link` function resolves every handle of the decoded structure of `entity` whose attribute has a `.link`, including the handles of nested structures and of the elements of arrays, which are reached by `dmi_attributes_walk`:

- A single handle is resolved into a `dmi_entity_t *` member, which is set to the structure the handle refers to, and to `nullptr` if it is not resolved.
- An array of handles is resolved into a `dmi_entity_t **` member, which is set to an array of its own, one pointer per handle, allocated by the function. An array left over by an earlier call is freed first, since linking starts anew each time.
- A handle which is not set, `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, is resolved into `nullptr` without an error. So is a handle of zero which refers to a structure of an unexpected type, since some firmware writes it for a handle which is not set, unless the context is strict, see `DMI_CONTEXT_FLAG_STRICT`.

Resolving goes on after a handle fails to resolve, so that all the broken references of the structure are reported at once.

`dmi_entity_link` calls the function before the link handler of the specification, which is left for what the attributes cannot say, so it rarely has to be called directly; unlike `dmi_entity_link`, it does not mark the entity as linked.

The `dmi_attributes_unlink` function frees the arrays `dmi_attributes_link` has allocated, which is the only thing it allocates, and resets the members holding them to `nullptr`, so that it may be called more than once. Members of single handles are left as they are. `dmi_entity_decode` and `dmi_entity_destroy` call it before the cleanup handler of the specification and `dmi_fields_release`, since the arrays of handles may be held by the elements of the arrays either of them frees.

Structures with no specification, no attributes or no decoded structure have nothing to link or unlink.

Parameters

`entity`

Structure to link or unlink, whose structures it refers to have been registered in its context. `dmi_attributes_unlink` also takes `nullptr`, which does nothing.

Return value

The `dmi_attributes_link` function returns `true` if every handle is resolved or not set, and `false` if `entity` is `nullptr`, or if a handle cannot be resolved.

The `dmi_attributes_unlink` function returns no value.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised by `dmi_attributes_link` if `entity` is `nullptr`. The error reaches no error queue, since there is no context to raise it against.

`DMI_ERROR_ENTITY_NOT_FOUND`
: Raised if a handle belongs to no structure of the table.

`DMI_ERROR_ENTITY_TYPE_INVALID`
: Raised if a handle refers to a structure of a type its attribute does not list in `.targets`.

`DMI_ERROR_OUT_OF_MEMORY`
: Raised if the array an array of handles is resolved into cannot be allocated.

The `dmi_attributes_unlink` function raises no errors.

Example

A baseboard refers to the chassis holding it, and to the structures it holds, which are put into the `chassis` and `objects` members once the table is linked:

```
DMI_ATTRIBUTE(dmi_baseboard_t, chassis_handle, HANDLE, {
    .code    = "chassis-handle",
    .name    = "Chassis handle",
    .targets = (const dmi_type_t *const[]){ DMI_TYPE(chassis), nullptr },
    .link    = dmi_member(dmi_baseboard_t, chassis)
}),
DMI_ATTRIBUTE_ARRAY(dmi_baseboard_t, object_handles, object_count, HANDLE, {
    .code    = "contained-objects",
    .name    = "Contained objects",
    .link    = dmi_member(dmi_baseboard_t, objects)
}),
```

See also

`DMI_ATTRIBUTE`, `dmi_attributes_walk`, `dmi_entity_decode`, `dmi_entity_destroy`, `dmi_entity_link`, `dmi_fields_release`, `dmi_member`

dmi_attributes_walk

Synopsis

```
#include <opendmi/attribute.h>

typedef enum dmi_attribute_walk {
    DMI_ATTRIBUTE_WALK_STOP,
    DMI_ATTRIBUTE_WALK_CONTINUE,
    DMI_ATTRIBUTE_WALK_SKIP
} dmi_attribute_walk_t;

typedef struct dmi_attribute_node {
    const dmi_attribute_t *member;
```

```
    const dmi_attribute_t *attr;
    dmi_data_t            *info;
    dmi_data_t            *value;
    size_t                 index;
    size_t                 count;
} dmi_attribute_node_t;

typedef dmi_attribute_walk_t dmi_attribute_enter_fn(void *context, const
dmi_attribute_node_t *node);
typedef bool dmi_attribute_leave_fn(void *context, const dmi_attribute_node_t
*node);
typedef bool dmi_attribute_visit_fn(void *context, const dmi_attribute_node_t
*node);

typedef struct dmi_attribute_visitor {
    dmi_attribute_enter_fn *member_start;
    dmi_attribute_leave_fn *member_end;
    dmi_attribute_enter_fn *struct_start;
    dmi_attribute_leave_fn *struct_end;
    dmi_attribute_enter_fn *array_start;
    dmi_attribute_leave_fn *array_end;
    dmi_attribute_enter_fn *item_start;
    dmi_attribute_leave_fn *item_end;
    dmi_attribute_visit_fn *value;
} dmi_attribute_visitor_t;

bool dmi_attributes_walk(
        const dmi_attribute_t         *attrs,
        dmi_data_t                    *info,
        const dmi_attribute_visitor_t *visitor,
        void                          *context);

bool dmi_attribute_walk(
        const dmi_attribute_t         *attr,
        dmi_data_t                    *info,
        const dmi_attribute_visitor_t *visitor,
        void                          *context);
```

Description

The `dmi_attributes_walk` function walks the members of a decoded structure the list of attributes `attrs` describes, in the order the list declares them, along with the members of the nested structures and the elements of arrays, and calls the callbacks of `visitor` for each place it reaches. The `dmi_attribute_walk` function walks a single member, the one `attr` describes.

This is how the library links the references the attributes declare, see `dmi_attributes_link`, and how the command line tool writes structures in each of its output formats: the walk decides what is reached and in which order, and the callbacks what is done there.

Walk

A member is walked as follows:

1. The attribute describing the value is resolved first: a variant attribute is described by the variant matching its selector, see `dmi_attribute_resolve`, and a member no variant matches has no value, and is left out without any callback.
2. `member_start` enters the member.
3. An array or a vector is entered with `array_start`, and each of its elements with `item_start`, and left with `item_end` and `array_end`. An array whose elements have not been allocated holds none, whatever its counter says.
4. A structure, either the member itself or an element of an array, is entered with `struct_start`, its members are walked the same way in turn, and it is left with `struct_end`.
5. Any other value, of a member or of an element, is given to `value`.
6. `member_end` leaves the member.

An entering callback returns `DMI_ATTRIBUTE_WALK_CONTINUE` to go into what it enters, `DMI_ATTRIBUTE_WALK_SKIP` to leave it out, along with the matching leaving callback, and `DMI_ATTRIBUTE_WALK_STOP` to stop the walk. A leaving callback, and `value`, return `false` to stop the walk. Any callback may be `nullptr`, which goes on as if it had returned `DMI_ATTRIBUTE_WALK_CONTINUE` or `true`.

Nodes

Each callback is given the place the walk has reached as a `dmi_attribute_node_t`:

`member`
: Attribute naming the member, which is the declared one rather than its variant, e.g. for the label the member is written with.

`attr`
: Attribute describing the value, which is the variant of a variant attribute, and the array attribute for its elements.

`info`
: Structure holding the member, along with the counter of an array.

`value`
: Value: the member itself, or the element of an array.

`index`
: Index of the element of an array, or `SIZE_MAX` for a member.

`count`
: Number of the elements of an array, and zero otherwise.

The node is valid during the callback only.

Parameters

`attrs`

Attributes of the structure, terminated with an empty attribute, or `nullptr`, in which case there is nothing to walk.

`attr`

Attribute of the member to walk.

`info`

Decoded structure, e.g. the `info` member of an entity, or `nullptr` for `dmi_attributes_walk`, in which case there is nothing to walk.

`visitor`

Callbacks of the walk.

`context`

Context passed to the callbacks as it is.

Return value

Returns `false` if a callback has stopped the walk or on error, and `true` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised if `visitor` is `nullptr`, and by `dmi_attribute_walk` if `attr` or `info` is. It is not reported, since the functions take no context of the library to hold it.

Errors of the walk itself are left to the callbacks.

Example

The following callbacks print the codes of the members of a structure, one per line, indented by the depth of nesting:

```
static dmi_attribute_walk_t print_member(void *context, const
dmi_attribute_node_t *node)
{
    printf("%*s%s\n", 2 * *(int *)context, "", node->member->params.code);
    return DMI_ATTRIBUTE_WALK_CONTINUE;
}

static dmi_attribute_walk_t indent(void *context, const dmi_attribute_node_t
*node)
{
    (*(int *)context)++;
    return DMI_ATTRIBUTE_WALK_CONTINUE;
}

static bool outdent(void *context, const dmi_attribute_node_t *node)
{
```

```
    (*(int *)context)--;
    return true;
}

static const dmi_attribute_visitor_t visitor = {
    .member_start = print_member,
    .struct_start = indent,
    .struct_end   = outdent
};

int depth = 0;

dmi_attributes_walk(entity->spec->attributes, entity->info, &visitor, &depth);
```

See also

`dmi_entity_info`

<opendmi/field.h>

DMI_FIELD

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD(structure, member, wire_type, ...)
DMI_FIELD_STRING(structure, member, ...)
DMI_FIELD_UUID(structure, member, ...)
DMI_FIELD_BCD(structure, member, wire_type, ...)
DMI_FIELD_BINARY(structure, member, length, ...)
DMI_FIELD_SKIP(length)
DMI_FIELD_PRESET(structure, member, value)

DMI_FIELDS({ field, ..., {} })

#define DMI_FIELD_LENGTH_REST   SIZE_MAX
#define DMI_FIELD_LENGTH_MEMBER (SIZE_MAX - 1)
```

Description

The macros declare the fields of a specification, the `fields` member of `dmi_entity_spec_t`, which `dmi_fields_decode` reads a structure by and `dmi_fields_encode` writes it back by. Fields are listed in the order the structure holds them, starting right after the header, and each one names the member of the decoded structure it is read into.

Fields describe the bytes, while the attributes declared with `DMI_ATTRIBUTE` describe the values they are read into; the two are joined by the member they name.

`DMI_FIELD`

Integer of the width of `wire_type`, e.g. `dmi_word_t`, read little-endian into `member`, which may be wider than the field: the value standing for "unknown" is then read as the largest value of the member, so that it does not collide with a value a device may really have.

`DMI_FIELD_STRING`

String reference, read into a `const char *` member pointing at the string it refers to, or at `nullptr` if it refers to none.

`DMI_FIELD_UUID`

Sixteen bytes holding a UUID, read into a `dmi_uuid_t` member with the fields put into the order RFC 4122 gives them.

`DMI_FIELD_BCD`

Binary-coded decimal of the width of `wire_type`, read into the number its digits spell.

`DMI_FIELD_BINARY`

`length` bytes taken as they are, read into a `dmi_binary_t` member which refers to them in place. `DMI_FIELD_LENGTH_REST` takes the bytes running to the end of the structure, and `DMI_FIELD_LENGTH_MEMBER` as many bytes as the `length` of the member says, which a field before has set.

`DMI_FIELD_SKIP`

`length` bytes the structure holds and nothing reads, which are stepped over on decoding and kept as they are on encoding.

`DMI_FIELD_PRESET`

Member the data does not carry, set to `value` before any field is read, for the members meaning something other than zero until the rest of the table gives them a value. It occupies no bytes.

`DMI_FIELDS`

List of the fields, a compound literal of `dmi_field_t`. The list is terminated by an empty field, `{}`, which the list itself ends with.

Other kinds of fields have pages of their own: ranges of bits, `DMI_FIELD_BITS`; arrays and vectors, `DMI_FIELD_ARRAY`; groups and boundaries, `DMI_FIELD_GROUP`; fields widened by a later version, `DMI_FIELD_EXTENDED`; and fields feeding several members, `DMI_FIELD_SPLIT`.

Parameters

`structure`

Type of the structure the fields belong to: the decoded structure, or the element of an array.

`member`

Member of `structure` the field is read into.

`wire_type`

Type the field is written with in the data, whose size is the width of the field, e.g. `dmi_byte_t`, `dmi_word_t`, `dmi_dword_t` or `dmi_qword_t`.

`…`

Further parameters, given as the designated initializers of `dmi_field_params_t`, of which the fields take:

`.decode`, `.encode`

Pair of handlers turning the data into the member and back, for the values which mean something other than the number the data carries. They are required in pairs for the structure to be encoded.

`.unknown_raw`

Value standing for "unknown" in the data, which the member holds as its largest value. Zero means the field has none.

`.absent`

Pointer to the value the member holds when the data ends before the field, for the members whose zero means something else.

Example

Fields of a system slot, from the beginning of its version 2.0 part:

```
.fields = DMI_FIELDS({
    DMI_FIELD_STRING(dmi_slot_t, designator),
    DMI_FIELD(dmi_slot_t, type,      dmi_byte_t),
    DMI_FIELD(dmi_slot_t, bus_width, dmi_byte_t),
    ...
    {}
})
```

See also

`DMI_FIELD_BITS`, `DMI_FIELD_ARRAY`, `DMI_FIELD_GROUP`, `DMI_FIELD_EXTENDED`, `DMI_FIELD_SPLIT`, `DMI_ATTRIBUTE`, `dmi_member`, `dmi_fields_decode`, `dmi_fields_encode`

DMI_FIELD_ARRAY

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD_ARRAY(structure, member, counter, ...)
DMI_FIELD_VECTOR(structure, member, ...)
DMI_FIELD_ELEMENT(structure, member, wire_type, ...)
```

Description

The macros declare the elements a structure holds in a row, each described by fields of its own, see `DMI_FIELD`.

`DMI_FIELD_ARRAY`

Array whose number of elements the data gives. The elements are decoded into an array of their own, which the library allocates and frees with the decoded structure, see `dmi_fields_release`: `member` is a pointer to the element type, and `counter`, a `size_t` member, holds the number of the elements decoded in full. A structure which ends in the middle of an element is short of the elements it declares rather than broken, and the counter leaves the cut element out.

`DMI_FIELD_VECTOR`

Fixed number of elements, decoded in place into the array member `member`, whose number of elements is the one of the vector.

`DMI_FIELD_ELEMENT`

Element of an array or a vector of plain values, which is the whole element rather than a member of one. It is the single field of `.fields` of such an array, and names the array itself.

Parameters

`structure`
: Type of the structure holding the array.

`member`
: Member holding the array: a pointer to the elements for an array, and the array of the elements for a vector.

`counter`
: `size_t` member the number of the decoded elements is stored into.

`wire_type`
: Type an element of plain values is written with.

`…`
: Further parameters, given as the designated initializers of `dmi_field_params_t`:

  `.fields`
  : Fields of an element, whose `structure` is the element type.

  `.count_length`
  : Width of the number of the elements, which precedes them. Zero means the data carries no number, and the array runs to the end of the structure, in elements of `.stride` bytes.

  `.count_member`
  : Member keeping the number the data declares, which is greater than `counter` when the data ends early.

  `.stride_length`, `.stride_member`
  : Width of the length of an element, which follows the number of them, and the member keeping it. Elements may then be longer than their fields, and the next one is found by that length.

  `.stride_minimum`
  : Smallest length of an element, below which the elements are of no use: they are stepped over rather than decoded, and kept as they are on encoding.

Example

Peer groups of a system slot, preceded by the number of them:

```
DMI_FIELD_ARRAY(dmi_slot_t, peer_groups, peer_group_count,
    .count_length = sizeof(dmi_byte_t),
    .fields       = DMI_FIELDS({
        DMI_FIELD(dmi_slot_peer_group_t, address, dmi_dword_t,
                  .decode = dmi_pci_addr_decode,
                  .encode = dmi_pci_addr_encode),
        DMI_FIELD(dmi_slot_peer_group_t, bus_width, dmi_byte_t),
        {}
    })),
```

Vector of plain values:

```
DMI_FIELD_VECTOR(dmi_hpe_system_id_t, platform_id,
    .fields = DMI_FIELDS({
        DMI_FIELD_ELEMENT(dmi_hpe_system_id_t, platform_id, dmi_byte_t),
        {}
    })),
```

See also

`DMI_FIELD`, `DMI_ATTRIBUTE`, `dmi_fields_release`, `dmi_fields_decode`

DMI_FIELD_BITS

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD_BITS(structure, member, bits, ...)
DMI_FIELD_BITS_SKIP(bits)
DMI_FIELD_PAD(unit_type)
```

Description

The macros declare the fields which share the bits of a byte, a word or a wider unit, e.g. the flags of a structure, see `DMI_FIELD`.

`DMI_FIELD_BITS`

Range of `bits` bits, read into `member`. Ranges are taken from the least significant bit of the unit up, in the order they are declared, and a range is allowed to span the bytes of the unit, the way the bit fields of a little-endian value do.

`DMI_FIELD_BITS_SKIP`

Range of `bits` bits the specification reserves between the ranges, which carries no value. On encoding, it holds what the source data had in the preserve mode, and zeros in the canonical one.

`DMI_FIELD_PAD`

Bits left over at the end of the unit, of the type `unit_type`, e.g. `dmi_word_t`, which close it. Every run of ranges ends with one, and no field of another kind begins in the middle of a unit.

Versions of ranges

A version of the specification may give a meaning to the reserved bits of a unit the structure already holds, which says nothing about its length: the version the entry point declares decides it. Such a range declares the version it is defined from as `.from`, and is left to the padding of the unit in the tables of an earlier version. A range the version widens is declared twice, up to the version with `.before`, and from it with `.from`, both reading into one member.

Parameters

`structure`, `member`
: Structure and member the range is read into.

`bits`
: Width of the range, in bits.

`unit_type`
: Type of the unit, whose size gives its width.

`...`
: Further parameters, given as the designated initializers of `dmi_field_params_t`: `.from` and `.before`, and the `.decode` and `.encode` handlers, see `DMI_FIELD`.

Example

```
DMI_FIELD_BITS(dmi_hpe_trusted_module_t, presence,          2),
DMI_FIELD_BITS(dmi_hpe_trusted_module_t, is_orom_measuring, 1),
DMI_FIELD_BITS(dmi_hpe_trusted_module_t, is_hidden,         1),
DMI_FIELD_PAD(dmi_byte_t),
```

See also

`DMI_FIELD`, `dmi_fields_decode`

DMI_FIELD_EXTENDED

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD_EXTENDED(structure, member, wire_type, ...)
```

Description

The specification widens a field by leaving the plain one in place and adding a wider one to a later version, which the plain one points at by holding a value of its own, e.g. `0xFFFF`. Both fields name the same member: the plain one, declared with `DMI_FIELD`, decodes into it, and the extended one replaces the value when the plain one says that the real value is there.

The extended field is read wherever it is declared even when the plain one does not point at it, since it occupies its bytes either way. On encoding, the plain field takes the value whenever it fits, and points at the extended one otherwise; in the preserve mode, the representation of the source data is kept as long as the value fits it.

Parameters

`structure`, **`member`**
: Structure and member the plain field and the extended one decode into.

`wire_type`
: Type the extended field is written with.

`…`
: Further parameters, given as the designated initializers of `dmi_field_params_t`:

  `.when_raw`
  : Value of the plain field which says that the extended field carries the real value.

  `.when`
  : Member whose plain field says so, for a run of extended fields the specification keys on one plain field, which are widened together. Left unset for the extended fields answering to their own member.

  `.decode`, **`.encode`**
  : Handlers of the extended field, see `DMI_FIELD`.

Example

Start and end addresses of a memory device mapped address, both widened when the start one says so:

```
DMI_FIELD_GROUP(.since = DMI_VERSION(2, 7, 0)),
DMI_FIELD_EXTENDED(dmi_memory_device_addr_t, start_address, dmi_qword_t,
                   .when_raw = 0xFFFFFFFFu),
DMI_FIELD_EXTENDED(dmi_memory_device_addr_t, end_address, dmi_qword_t,
                   .when     = dmi_member(dmi_memory_device_addr_t,
start_address),
                   .when_raw = 0xFFFFFFFFu),
```

See also

`DMI_FIELD`, `DMI_FIELD_GROUP`, `dmi_fields_encode`

DMI_FIELD_GROUP

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD_GROUP(...)
DMI_FIELD_OFFSET(offset)
```

Description

The macros occupy no bytes, and say something about the fields after them, see `DMI_FIELD`.

`DMI_FIELD_GROUP`

Beginning of a group of fields the data may end before. The firmware writes the structure of the version of the specification it follows, so a structure is allowed to end at the beginning of any group: the fields of that group and of the ones after it are left unset, and the structure is marked as partial rather than broken. A structure which ends in the middle of a group is decoded up to where it ends, and is marked as incomplete. The fields before the first group are required.

`DMI_FIELD_OFFSET`

Boundary the fields after it begin at, counted from the beginning of the structure the way the tables of the specification count it. It says nothing about where the data may end, and is checked while the structure is decoded and encoded, so that a field left out of the declaration, or declared twice, is caught where it happens rather than in the values of everything after it.

Parameters

`offset`

Offset of the next field, from the beginning of the structure, header included.

`…`

Parameters of a group, given as the designated initializers of `dmi_field_params_t`:

`.since`

Version of the specification the fields of the group appear in, which becomes the level of the structure once the data reaches the group. Groups of the vendors, whose fields no version brings, leave it unset.

`.present`

`bool` member, see `dmi_member`, set once every field of the group has been read, and left `false` if the data ends before the group or in the middle of it. Attributes tell the fields the structure holds from the ones it is too short for by it, and encoding writes the fields of no group the structure does not hold: the canonical mode ends the structure there, and the preserve mode keeps the source bytes of the group as they are.

Errors

A boundary the fields do not reach exactly fails decoding and encoding with `DMI_ERROR_INTERNAL`, which tells a declaration that does not match the data.

Example

```
DMI_FIELD(dmi_cache_t, associativity, dmi_byte_t),

DMI_FIELD_GROUP(.since = DMI_VERSION(3, 1, 0)),
DMI_FIELD_EXTENDED(dmi_cache_t, maximum_size, dmi_dword_t, ...),
```

```
DMI_FIELD_GROUP(.present = dmi_member(dmi_hpe_trusted_module_t, has_chip)),
DMI_FIELD(dmi_hpe_trusted_module_t, chip, dmi_byte_t),
```

See also

`DMI_FIELD`, `dmi_fields_decode`, `dmi_entity_is_partial`

DMI_FIELD_SPLIT

Synopsis

```
#include <opendmi/field.h>

DMI_FIELD_SPLIT(structure, wire_type, ...)
DMI_FIELD_SPLIT_BINARY(structure, length, ...)
```

Description

Some fields carry the data of several members at once, e.g. a byte whose most significant bit tells whether the rest is a structure type or a baseboard type. The handlers of such a field are given the whole structure rather than one member of it, and decode the data into the members it feeds and encode it back from them.

`DMI_FIELD_SPLIT`
: Integer of the width of `wire_type`.

`DMI_FIELD_SPLIT_BINARY`
: `length` bytes taken as they are.

The members a handler writes are a function of the data alone, so that encoding can tell whether the data it has decodes into the values the structure holds. Whatever combines several fields is derived once all of them have been read, by the `derive` handler of the specification.

Parameters

`structure`
: Type the handlers are given: the decoded structure for the fields of a specification, and the element type for the fields of an array.

`wire_type`
: Type the field is written with.

`length`
: Number of the bytes of a binary field.

`…`
: `.decode` and `.encode` handlers, which are given the structure as the value, see `DMI_FIELD`.

Example

```
DMI_FIELD_SPLIT(dmi_chassis_element_t, dmi_byte_t,
                .decode = dmi_chassis_decode_element_type,
                .encode = dmi_chassis_encode_element_type),
```

See also

`DMI_FIELD`, `dmi_fields_decode`

dmi_field_decode_kilobytes

Synopsis

```
#include <opendmi/field.h>

bool dmi_field_decode_kilobytes(
        const dmi_field_t      *field,
        const dmi_field_data_t *data,
        void                   *value);

bool dmi_field_decode_kilobytes_last(
        const dmi_field_t      *field,
        const dmi_field_data_t *data,
        void                   *value);

bool dmi_field_encode_kilobytes(
        const dmi_field_t *field,
        const void        *value,
        dmi_field_data_t  *data);
```

Description

The functions are decoding and encoding handlers of the fields, given as the `.decode` and `.encode` parameters of `DMI_FIELD`, for the sizes and the addresses the specification writes in kilobytes, which the decoded structure holds in bytes, e.g. the addresses of the memory array mapped addresses structure. The member decoded into is usually wider than the field, so that it also holds the values of the extended fields which carry the bytes themselves, see `DMI_FIELD_EXTENDED`.

The `dmi_field_decode_kilobytes` function stores the number of the bytes the kilobytes the field carries stand for, `data→number` multiplied by 1024, into the member `value` points to.

The `dmi_field_decode_kilobytes_last` function decodes the address of the last kilobyte of a range into the address of the last byte of the range, which is what the extended field of the address carries: ranges are given by their first and last kilobytes, so that the last kilobyte belongs to the range as a whole. The address of the kilobyte is multiplied by 1024, and 1023 is added.

The `dmi_field_encode_kilobytes` function undoes either of them: the member is divided by 1024, rounding down, into `data→number`.

The value standing for "unknown", the `.unknown_raw` parameter of the field, is decoded and encoded by the engine on its own, and never reaches the handlers.

Members are read and written by `dmi_field_get` and `dmi_field_set`, so a number too wide for the member is cut down to its width, and a `nullptr` member is accepted.

Parameters

`field`
: Field being decoded or encoded.

`data`
: Data the field carries, see `dmi_field_data_t`.

`value`
: Member the field decodes into.

Return value

The `dmi_field_decode_kilobytes` and `dmi_field_decode_kilobytes_last` functions return `true` if the member has been set, and `false` if `field` is `nullptr` or if the width of the member is not supported.

The `dmi_field_encode_kilobytes` function always returns `true`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `field` is `nullptr`. The functions take no context, so the error reaches no error queue.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if the width of the member is not supported, which is a mistake of the specification, and reaches no error queue either.

Notes

`data` must not be `nullptr`, nor may the `value` of `dmi_field_encode_kilobytes`, which are given by the engine.

Example

The starting and the ending addresses of a range, which the extended fields carry in bytes once the plain ones are too narrow:

```
DMI_FIELD(dmi_memory_array_addr_t, start_address, dmi_dword_t,
          .decode = dmi_field_decode_kilobytes,
          .encode = dmi_field_encode_kilobytes),
DMI_FIELD(dmi_memory_array_addr_t, end_address, dmi_dword_t,
```

```
          .decode = dmi_field_decode_kilobytes_last,
          .encode = dmi_field_encode_kilobytes),
...
DMI_FIELD_EXTENDED(dmi_memory_array_addr_t, start_address, dmi_qword_t,
                   .when_raw = 0xFFFFFFFFu),
DMI_FIELD_EXTENDED(dmi_memory_array_addr_t, end_address, dmi_qword_t,
                   .when     = dmi_member(dmi_memory_array_addr_t,
start_address),
                   .when_raw = 0xFFFFFFFFu),
```

See also

`DMI_FIELD`, `DMI_FIELD_EXTENDED`, `dmi_field_get`, `dmi_fields_decode`, `dmi_fields_encode`

dmi_field_get

Synopsis

```
#include <opendmi/field.h>

uintmax_t dmi_field_get(const dmi_field_t *field, const void *value);
bool dmi_field_set(const dmi_field_t *field, void *value, uintmax_t number);
```

Description

The functions read and write the member of the decoded structure a field is decoded into, whatever the width of the member, which is the size of the member the field names, `field->member.size`, and is one of 1, 2, 4 or 8 bytes. They are meant for the decoding and encoding handlers of the fields, the `.decode` and `.encode` parameters of `DMI_FIELD`, which are given the member rather than the structure, and are shared by members of different widths.

The `dmi_field_get` function returns the value of the member `value` points to, read as an unsigned integer.

The `dmi_field_set` function stores `number` into the member `value` points to. The member may be wider than the data the field carries, e.g. so that the value standing for "unknown" is stored as the largest value of the member, `UINTMAX_MAX` cut down to its width. A number wider than the member is cut down to its width as well, without an error. A `nullptr` member is accepted, and nothing is stored, since a field decoded to be compared rather than kept has no member to store into.

Parameters

`field`

Field the member belongs to.

`value`

Member of the field, as the handler is given it.

`number`
: Value to store.

Return value

The `dmi_field_get` function returns the value of the member, or zero if `field` or `value` is `nullptr`, or if the width of the member is not supported.

The `dmi_field_set` function returns `true` if the number has been stored, or if `value` is `nullptr`, and `false` if `field` is `nullptr` or if the width of the member is not supported.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `field` is `nullptr`, or if `value` of `dmi_field_get` is `nullptr`. The functions take no context, so the error reaches no error queue.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if the width of the member is not supported, which is a mistake of the specification, and reaches no error queue either.

Example

Pair of handlers decoding a size carried as a power of two of megabytes, and encoding it back:

```
static bool decode_size(const dmi_field_t *field, const dmi_field_data_t *data,
void *value)
{
    if (data->number + 20 >= 63)
        return dmi_field_set(field, value, UINTMAX_MAX);

    return dmi_field_set(field, value, (uintmax_t)1 << (data->number + 20));
}

static bool encode_size(const dmi_field_t *field, const void *value,
dmi_field_data_t *data)
{
    uintmax_t size  = dmi_field_get(field, value);
    unsigned  power = 0;

    while (power + 20 < 63 && ((uintmax_t)1 << (power + 20)) < size)
        power++;

    data->number = power;

    return true;
}
```

See also

`DMI_FIELD`, `dmi_field_decode_kilobytes`, `dmi_fields_decode`, `dmi_fields_encode`

dmi_fields_decode

Synopsis

```
#include <opendmi/field.h>

bool dmi_fields_decode(dmi_decoder_t *decoder);
```

Description

The `dmi_fields_decode` function reads a structure according to the layout its specification declares, instead of a decoding handler written for the type.

A specification describes its layout with the `fields` member of `dmi_entity_spec_t`, a list of `dmi_field_t` written with `DMI_FIELDS` and terminated with an empty field. Fields are read in the order they are declared, starting at the beginning of the structure body. A specification which declares fields and no decoding handler is decoded by this function, which `dmi_entity_decode` calls on its own, so it rarely has to be named. The bytes are read through the decoder the structure is being decoded with, see `dmi_decoder_initialize`.

Fields describe the bytes, while the attributes of `dmi_attribute_t` describe the values they are decoded into. The two are joined by the member of the decoded structure they name, and neither is a complete description of the other: a field may carry no member of its own, and a member may be computed rather than read.

Groups and truncation

The firmware of a platform writes the fields of the version of the specification it follows, and leaves out the ones of the newer versions, so a structure of a type is not as long as the decoder knows it to be.

`DMI_FIELD_GROUP` declares where the data is allowed to end. When the data ends at the beginning of a group, the fields of that group and of the ones after it are left unset, and the structure is marked as stopped rather than as broken; the version of the group the data has been read up to becomes the level of the structure. Ending anywhere else marks the structure as incomplete, except within the fields before the first group, which every structure of the type is required to carry.

A group may also name a `bool` member with `present`, which the function sets once every field of the group has been read, and leaves `false` if the data ends before the group or in the middle of it, e.g. `DMI_FIELD_GROUP(.present = dmi_member(dmi_hpe_trusted_module_t, has_chip))`. This is how the structures of the vendors, whose groups carry no version, tell the fields they hold from the ones they are too short for.

Integers and other values

An integer field names the type it is written with, e.g. `DMI_FIELD(dmi_slot_t, ident, dmi_word_t)`, whose size is the width of the field. The member it is decoded into may be wider than the field, so that the

value standing for "unknown" does not collide with a value a device may really have. String references, UUIDs, binary-coded decimals and bytes taken as they are have macros of their own: `DMI_FIELD_STRING`, `DMI_FIELD_UUID`, `DMI_FIELD_BCD` and `DMI_FIELD_BINARY`.

Arrays

`DMI_FIELD_ARRAY` declares a number of elements the data gives, which are decoded into an array of their own, allocated by the function: the member holds the pointer to the elements, and the counter the field names holds the number of the ones decoded in full. The arrays belong to the library, which frees them with the decoded structure, see `dmi_fields_release`, so the cleanup handler of a specification does not free them.

Values which need decoding

A field whose value means something other than the number the data carries declares a pair of handlers, `decode` and `encode`, which turn the data into the member and back. A field declared with `DMI_FIELD_SPLIT` gives its handlers the whole structure instead of one member, which is how the data feeding several members at once is decoded.

Parameters

`decoder`
: Decoder of the structure, see `dmi_decoder_initialize`. The structure is decoded into the structure of its type.

Return value

Returns `true` if the structure has been decoded, which includes the structures the data ends within, and `false` if it cannot be.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Decoder is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: A field declares a number wider than the widest one the engine reads and writes, which is eight bytes.

`DMI_ERROR_STATE_INVALID`
: Specification declares no fields.

`DMI_ERROR_INTERNAL`
: Fields reach another offset than the one declared.

`DMI_ERROR_OUT_OF_MEMORY`
: Elements of an array cannot be allocated.

Examples

```
const dmi_entity_spec_t dmi_slot_spec =
{
```

```
    .code   = "slot",
    .type   = DMI_TYPE(system_slots),
    ...
    .fields = DMI_FIELDS({
        DMI_FIELD_STRING(dmi_slot_t, designator),
        DMI_FIELD(dmi_slot_t, type,      dmi_byte_t),
        DMI_FIELD(dmi_slot_t, bus_width, dmi_byte_t),

        DMI_FIELD_GROUP(.since = DMI_VERSION(2, 1, 0)),
        DMI_FIELD(dmi_slot_t, features_ex, dmi_byte_t),

        DMI_FIELD_GROUP(.since = DMI_VERSION(3, 2, 0)),
        DMI_FIELD_ARRAY(dmi_slot_t, peer_groups, peer_group_count,
            .count_length = sizeof(dmi_byte_t),
            .fields       = DMI_FIELDS({
                DMI_FIELD(dmi_slot_peer_group_t, bus_width, dmi_byte_t),
                {}
            })),
        {}
    })
};
```

See also

`dmi_fields_encode`, `dmi_fields_decode_into`, `dmi_fields_release`, `dmi_entity_decode`, `dmi_decoder_initialize`

dmi_fields_decode_into

Synopsis

```
#include <opendmi/field.h>

bool dmi_fields_decode_into(
        dmi_decoder_t     *decoder,
        const dmi_field_t *fields,
        size_t             length,
        void              *info);

bool dmi_fields_encode_from(
        dmi_encoder_t     *encoder,
        const dmi_field_t *fields,
        size_t             length,
        const void        *info);
```

Description

The `dmi_fields_decode_into` function reads a part of a structure according to a list of fields, the way `dmi_fields_decode` reads a whole structure, for the decoding handlers which read the rest of the structure themselves: e.g. a record or a descriptor whose layout depends on a type the structure gives, whose length the structure gives too.

The fields, declared with `DMI_FIELD` and its kin, are read from the next `length` bytes of the structure, from the position of `decoder`, into `info`, a structure of the type the fields name. The decoder is then moved past the `length` bytes, whatever the fields have left unread, so that the bytes a part holds beyond the fields it is known to hold are stepped over.

The bytes of the part are the data the groups of the fields may end before, see `DMI_FIELD_GROUP`. A part which ends at the beginning of a group, or in the middle of it, leaves the fields it is too short for unset, along with the `present` members of their groups, and the structure is marked neither as partial nor as incomplete, since it is the part which is short rather than the structure. The fields before the first group are required.

The `dmi_fields_encode_from` function undoes it, for the encoding handlers which write the rest of the structure themselves, the way `dmi_fields_encode` writes a whole structure: the fields are written from `info` into the next `length` bytes of the structure, and the bytes of the part the fields do not describe are written the way the encoder writes the bytes the model does not hold, i.e. the ones of the source data in the preserve mode, and zeros in the canonical one. Writing stops at the first group the part does not hold, and, in the preserve mode, where the source data of the part ends.

Parameters

`decoder`

Decoder of the structure, see `dmi_decoder_initialize`, positioned at the part.

`encoder`

Encoder of the structure, see `dmi_encoder_initialize`, positioned at the part.

`fields`

Fields of the part, terminated by an empty field.

`length`

Number of the bytes of the part.

`info`

Structure the fields are read into, or written from.

Return value

`dmi_fields_decode_into` returns `true` if the part has been decoded, and `false` if the structure holds fewer than `length` bytes from the position of the decoder, if the part is too short for the required fields, or on the errors below.

`dmi_fields_encode_from` returns `true` if the part has been encoded, and `false` on the errors below.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Decoder or encoder, fields or structure is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: A field declares a number wider than the widest one the engine reads and writes, which is eight bytes.

`DMI_ERROR_STATE_INVALID`
: A field cannot be written, or the fields are longer than the part (`dmi_fields_encode_from`).

`DMI_ERROR_OUT_OF_MEMORY`
: Elements of an array cannot be allocated, or buffers of the encoder cannot grow.

Example

The v2 descriptor of a USB network interface, whose length follows the device type and counts it along with itself:

```
static const dmi_field_t usb_v2_fields[] = {
    DMI_FIELD(dmi_mgmt_nhi_usb_v2_t, vendor_id,  dmi_word_t),
    DMI_FIELD(dmi_mgmt_nhi_usb_v2_t, product_id, dmi_word_t),
    DMI_FIELD_STRING(dmi_mgmt_nhi_usb_v2_t, serial_number),
    DMI_FIELD_BINARY(dmi_mgmt_nhi_usb_v2_t, mac_address, 6),

    // Device characteristics are present since DSP0270 1.3
    DMI_FIELD_GROUP(),
    DMI_FIELD(dmi_mgmt_nhi_usb_v2_t, characteristics,   dmi_word_t),
    DMI_FIELD(dmi_mgmt_nhi_usb_v2_t, credential_bootstrapping_handle,
dmi_word_t),
    {}
};

dmi_byte_t length = 0;

if (dmi_decoder_get(decoder, dmi_byte_t, &length) and (length >= 2))
    dmi_fields_decode_into(decoder, usb_v2_fields, length - 2, &nhi->usb_v2);
```

See also

`dmi_fields_decode`, `dmi_fields_encode`, `DMI_FIELD`, `DMI_FIELD_GROUP`, `dmi_decoder_initialize`

dmi_fields_encode

Synopsis

```
#include <opendmi/field.h>

bool dmi_fields_encode(dmi_encoder_t *encoder);
```

Description

The `dmi_fields_encode` function writes a structure according to the layout its specification declares, undoing what `dmi_fields_decode` has done.

Fields are written in the order they are declared, after the header the encoder has written, so that a structure which has not been changed gives back the very bytes it has been decoded from. A specification which declares fields and no encoding handler is encoded by this function, which `dmi_entity_encode` calls on its own, so it rarely has to be named.

The groups the layout declares decide where the structure ends: in the preserve mode it is written as long as it has been read, and in the canonical one as long as the version the encoder is given says, see `dmi_encoder_initialize`.

The bytes the layout does not describe, such as the bits the specification reserves and the tails of the records longer than the fields they are known to hold, are written with `dmi_encoder_copy`: taken from the data the structure has been decoded from in the preserve mode, and written as zeros in the canonical one.

Parameters

`encoder`
: Encoder of the structure, see `dmi_encoder_initialize`.

Return value

`true` if the structure has been encoded, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Encoder is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: A field declares a number wider than the widest one the engine reads and writes, which is eight bytes.

`DMI_ERROR_STATE_INVALID`
: Structure is not decoded, its specification declares no fields, or a field it cannot write back, e.g. a conversion without the one undoing it.

`DMI_ERROR_INTERNAL`
: Fields reach another offset than the one declared.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffers of the encoder cannot grow.

See also

`dmi_fields_decode`, `dmi_entity_encode`, `dmi_encoder_initialize`, `dmi_encoder_copy`, `dmi_encoder_finish`

dmi_fields_release

Synopsis

```
#include <opendmi/field.h>

void dmi_fields_release(dmi_entity_t *entity);
```

Description

The `dmi_fields_release` function frees the arrays `dmi_fields_decode` has allocated for a structure: the elements of every array the fields of its specification declare, including the arrays held by nested structures, by the elements of vectors and by the elements of other arrays, which are freed before the array holding them. The decoded structure itself is left to the caller.

The members holding the arrays are reset to `nullptr`, and the counters of the decoded elements to zero, so the function may be called more than once, and a structure whose decoding has failed half way is released the same way: an element which has failed to decode has been released as it failed.

`dmi_entity_decode`, when decoding fails, and `dmi_entity_destroy` call the function on their own, after the cleanup handler of the specification. The handler therefore frees only what the handlers of the specification allocate themselves, e.g. an array a deriving handler fills in, and may still read the arrays the fields declare. The arrays of the structures the references of the attributes are resolved into are freed before the handler, by `dmi_attributes_unlink`, since the elements of those arrays may hold them.

A structure decoded by a decoding handler of its own is released only as far as its specification declares fields.

Parameters

`entity`

Structure to release, or `nullptr`, in which case nothing is done. A structure which has not been decoded, or whose specification declares no fields, is left as it is.

Return value

None.

Errors

None.

See also

`dmi_fields_decode`, `dmi_entity_decode`, `dmi_entity_destroy`

<opendmi/value.h>

dmi_bool_code

Synopsis

```
#include <opendmi/value.h>

const char *dmi_bool_code(bool value);
const char *dmi_bool_name(bool value);
```

Description

The `dmi_bool_code` function returns the code of a boolean value, a machine-readable identifier which is the same in every language: `true` or `false`.

The `dmi_bool_name` function returns the name of a boolean value, as the command line tool prints a flag: `yes` or `no`. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Boolean attributes whose values are not named by the attribute itself are formatted with these functions, see `DMI_ATTRIBUTE`.

Parameters

`value`
: Value to get the code or the name of.

Return value

The functions return the code or the name of the value, which is never `nullptr`. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None: the functions take no context to raise errors in.

See also

`DMI_ATTRIBUTE`, `dmi_name_lookup`, `dmi_set_locale`, `dmi_unit`

dmi_unit

Synopsis

```
#include <opendmi/value.h>
```

```
typedef enum dmi_unit {
    DMI_UNIT_NONE,
    DMI_UNIT_HOUR,
    DMI_UNIT_MINUTE,
    DMI_UNIT_SECOND,
    DMI_UNIT_MILLISECOND,
    DMI_UNIT_MICROSECOND,
    DMI_UNIT_NANOSECOND,
    DMI_UNIT_AMPERE,
    DMI_UNIT_MILLIAMPERE,
    DMI_UNIT_VOLT,
    DMI_UNIT_MILLIVOLT,
    DMI_UNIT_WATT,
    DMI_UNIT_MILLIWATT,
    DMI_UNIT_WATT_HOUR,
    DMI_UNIT_CELSIUS,
    DMI_UNIT_BIT,
    DMI_UNIT_BYTE,
    DMI_UNIT_KILOBYTE,
    DMI_UNIT_MEGABYTE,
    DMI_UNIT_GIGABYTE,
    DMI_UNIT_KIBIBYTE,
    DMI_UNIT_MEBIBYTE,
    DMI_UNIT_GIBIBYTE,
    DMI_UNIT_TEBIBYTE,
    DMI_UNIT_PEBIBYTE,
    DMI_UNIT_EXBIBYTE,
    DMI_UNIT_ZEBIBYTE,
    DMI_UNIT_YOBIBYTE,
    DMI_UNIT_ROBIBYTE,
    DMI_UNIT_QUEBIBYTE,
    DMI_UNIT_BAUD,
    DMI_UNIT_MEGAXA_SECOND,
    DMI_UNIT_KILOBIT_SECOND,
    DMI_UNIT_MEGABIT_SECOND,
    DMI_UNIT_GIGABIT_SECOND,
    DMI_UNIT_PERCENT,
    DMI_UNIT_REVOLUTION,
    DMI_UNIT_RACK,
    DMI_UNIT_MHZ,
    DMI_UNIT_MILLIMETER
} dmi_unit_t;

const char *dmi_unit_code(dmi_unit_t value);
const char *dmi_unit_name(dmi_unit_t value);
```

Description

The `dmi_unit_t` enumeration tells the measurement units of an integer or a decimal attribute, and is the type of the `.unit` member of an attribute, see `DMI_ATTRIBUTE`.

The `dmi_unit_code` function returns the code of a unit, a machine-readable identifier which is the same in every language, e.g. `millivolt`, as the XML output of the command line tool gives it.

The `dmi_unit_name` function returns the symbol of a unit, as the command line tool prints it after a value, e.g. `mV`. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Decimal units of size are powers of 1000, and binary ones are powers of 1024.

Values

`DMI_UNIT_NONE`
: No measurement units, e.g. of a count. The value has neither a code nor a name.

`DMI_UNIT_HOUR`
: Hours, code `hour`, name `h`.

`DMI_UNIT_MINUTE`
: Minutes, code `minute`, name `m`.

`DMI_UNIT_SECOND`
: Seconds, code `second`, name `s`.

`DMI_UNIT_MILLISECOND`
: Milliseconds, code `millisecond`, name `ms`.

`DMI_UNIT_MICROSECOND`
: Microseconds, code `microsecond`, name `us`.

`DMI_UNIT_NANOSECOND`
: Nanoseconds, code `nanosecond`, name `ns`.

`DMI_UNIT_AMPERE`
: Amperes, code `ampere`, name `A`.

`DMI_UNIT_MILLIAMPERE`
: Milliamperes, code `milliampere`, name `mA`.

`DMI_UNIT_VOLT`
: Volts, code `volt`, name `V`.

`DMI_UNIT_MILLIVOLT`
: Millivolts, code `millivolt`, name `mV`.

`DMI_UNIT_WATT`
: Watts, code `watt`, name `W`.

`DMI_UNIT_MILLIWATT`

Milliwatts, code `milliwatt`, name `mW`.

`DMI_UNIT_WATT_HOUR`

Watt-hours, code `watt-hour`, name `W*h`.

`DMI_UNIT_CELSIUS`

Degrees Celsius, code `celsius`, name `°C`.

`DMI_UNIT_BIT`

Bits, code `bit`, name `bits`.

`DMI_UNIT_BYTE`

Bytes, code `byte`, name `bytes`.

`DMI_UNIT_KILOBYTE`

Kilobytes, 1000 bytes, code `kilobyte`, name `KB`.

`DMI_UNIT_MEGABYTE`

Megabytes, 1000 kilobytes, code `megabyte`, name `MB`.

`DMI_UNIT_GIGABYTE`

Gigabytes, 1000 megabytes, code `gigabyte`, name `GB`.

`DMI_UNIT_KIBIBYTE`

Kibibytes, 1024 bytes, code `kibibyte`, name `KiB`.

`DMI_UNIT_MEBIBYTE`

Mebibytes, 1024 kibibytes, code `mebibyte`, name `MiB`.

`DMI_UNIT_GIBIBYTE`

Gibibytes, 1024 mebibytes, code `gibibyte`, name `GiB`.

`DMI_UNIT_TEBIBYTE`

Tebibytes, 1024 gibibytes, code `tebibyte`, name `TiB`.

`DMI_UNIT_PEBIBYTE`

Pebibytes, 1024 tebibytes, code `pebibyte`, name `PiB`.

`DMI_UNIT_EXBIBYTE`

Exbibytes, 1024 pebibytes, code `exbibyte`, name `EiB`.

`DMI_UNIT_ZEBIBYTE`

Zebibytes, 1024 exbibytes, code `zebibyte`, name `ZiB`.

`DMI_UNIT_YOBIBYTE`

Yobibytes, 1024 zebibytes, code `yobibyte`, name `YiB`.

`DMI_UNIT_ROBIBYTE`

Robibytes, 1024 yobibytes, code `robibyte`, name `RiB`.

`DMI_UNIT_QUEBIBYTE`
: Quebibytes, 1024 robibytes, code `quebibyte`, name `QiB`.

`DMI_UNIT_BAUD`
: Symbols per second, code `baud`, name `baud`.

`DMI_UNIT_MEGAXA_SECOND`
: Megatransfers per second, code `megaxa-second`, name `MT/s`.

`DMI_UNIT_KILOBIT_SECOND`
: Kilobits per second, code `kilobit-second`, name `Kbps`.

`DMI_UNIT_MEGABIT_SECOND`
: Megabits per second, code `megabit-second`, name `Mbps`.

`DMI_UNIT_GIGABIT_SECOND`
: Gigabits per second, code `gigabit-second`, name `Gbps`.

`DMI_UNIT_PERCENT`
: Percents, code `percent`, name `%`.

`DMI_UNIT_REVOLUTION`
: Revolutions per minute, code `revolution`, name `rpm`.

`DMI_UNIT_RACK`
: Rack units, code `rack`, name `U`.

`DMI_UNIT_MHZ`
: Megahertz, code `mhz`, name `MHz`.

`DMI_UNIT_MILLIMETER`
: Millimeters, code `millimeter`, name `mm`.

Parameters

`value`
: Unit to get the code or the name of.

Return value

The functions return the code or the name of the unit, or `nullptr` if the unit has none, e.g. `DMI_UNIT_NONE`, or if the value is not a unit. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None: the functions take no context to raise errors in.

See also

`DMI_ATTRIBUTE`, `dmi_bool_code`, `dmi_name_lookup`, `dmi_set_locale`

`<opendmi/buffer.h>`

dmi_buffer_assign

Synopsis

```
#include <opendmi/buffer.h>

bool dmi_buffer_assign(dmi_buffer_t *buffer, const void *ptr, size_t length);
```

Description

The `dmi_buffer_assign` function replaces whatever the buffer holds with a copy of `length` bytes at `ptr`.

Buffers own their memory, so the data is copied rather than referred to, and the caller keeps whatever it has passed.

Parameters

`buffer`
: Buffer to fill.

`ptr`
: Data to copy, which may be `nullptr` when `length` is zero.

`length`
: Number of the bytes to copy.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Data is `nullptr`, while the length is not zero.

`DMI_ERROR_OUT_OF_MEMORY`
: Memory of the buffer cannot grow.

See also

`dmi_buffer_write`, `dmi_buffer_clear`, `dmi_buffer_at`, `dmi_file_load`

dmi_buffer_at

Synopsis

```
#include <opendmi/buffer.h>

const dmi_data_t *dmi_buffer_at(
        const dmi_buffer_t *buffer,
        size_t              offset,
        size_t              length);
```

Description

The `dmi_buffer_at` function refers to `length` bytes at `offset` from the beginning of the data of the buffer.

The data is referred to in place rather than copied, which is how the strings and the binary members of the decoded structures refer to it, and stays valid for as long as the buffer holds it: writing to the buffer may move the data, and destroying the buffer frees it.

Parameters

`buffer`
: Buffer to refer to.

`offset`
: Byte offset from the beginning of the data.

`length`
: Number of the bytes to refer to, which may be zero.

Return value

Pointer to the data, or `nullptr` if the buffer holds fewer bytes than the requested range.

A buffer which has never held data has no memory to point into, so `nullptr` is returned for the range of no bytes at its beginning as well.

Errors

None.

See also

`dmi_buffer_read`, `dmi_reader_ref_bytes`, `dmi_entity_data`

dmi_buffer_clear

Synopsis

```
#include <opendmi/buffer.h>

void dmi_buffer_clear(dmi_buffer_t *buffer);
```

Description

The `dmi_buffer_clear` function drops the data of the buffer, keeping the memory it holds for the data written next, so that a buffer filled again and again allocates once.

Everything referring to the data of the buffer, such as the binary members of the structures decoded from it, refers to data which has been dropped afterwards.

Parameters

`buffer`
: Buffer to clear.

Errors

None.

See also

`dmi_buffer_resize`, `dmi_buffer_is_empty`, `dmi_buffer_destroy`

dmi_buffer_context

Synopsis

```
#include <opendmi/buffer.h>

dmi_context_t *dmi_buffer_context(const dmi_buffer_t *buffer);
```

Description

The `dmi_buffer_context` function returns the context the buffer belongs to, so that everything holding a buffer can reach the context without being handed it separately.

Parameters

`buffer`
: Buffer to query.

Return value

Context of the buffer, or `nullptr` if `buffer` is `nullptr` or belongs to none.

Errors

None.

See also

`dmi_buffer_create`, `dmi_entity_buffer`, `dmi_entity_context`

dmi_buffer_create

Synopsis

```
#include <opendmi/buffer.h>

dmi_buffer_t *dmi_buffer_create(dmi_context_t *context);
```

Description

The `dmi_buffer_create` function creates an empty buffer belonging to `context`. No memory for the data is allocated until the buffer is written to or grown.

A buffer owns the bytes it holds, whether they have been read from a device, from a dump, or written by the framework itself, so that everything reading them - the decoded structures, the strings they refer to and the binary members referring to the data in place - is valid for as long as the buffer is. Buffers live on their own rather than within the scope which makes them, because the structures decoded from a buffer outlive whatever has read it in.

Data is read and written through `dmi_reader_t` and `dmi_writer_t`, which are cursors over a range of a buffer and hold no memory of their own.

The caller is responsible for destroying the buffer with `dmi_buffer_destroy` once nothing refers to the data any more.

Parameters

`context`
: DMI context the buffer belongs to, whose memory it holds and whose error queue its errors are raised against.

Return value

On success, returns a pointer to the buffer. On failure, returns `nullptr`.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer cannot be allocated.

See also

`dmi_buffer_destroy`, `dmi_buffer_assign`, `dmi_buffer_context`, `dmi_reader_initialize`, `dmi_writer_initialize`

dmi_buffer_destroy

Synopsis

```
#include <opendmi/buffer.h>

void dmi_buffer_destroy(dmi_buffer_t *buffer);
```

Description

The `dmi_buffer_destroy` function frees the buffer and the data it holds.

Everything referring to the data of the buffer, such as the strings and the binary members of the structures decoded from it, refers to freed memory afterwards, so a buffer is destroyed once nothing reads it any more. The buffers of a context are destroyed by `dmi_close`.

Parameters

`buffer`
: Buffer to destroy, or `nullptr`.

Errors

None.

See also

`dmi_buffer_create`, `dmi_buffer_clear`, `dmi_close`

dmi_buffer_is_empty

Synopsis

```
#include <opendmi/buffer.h>

bool dmi_buffer_is_empty(const dmi_buffer_t *buffer);
```

Description

The `dmi_buffer_is_empty` function checks whether the buffer holds any data. A buffer holds no data before anything has been put into it, and after it has been cleared, whether it keeps the memory it has had or not.

Parameters

`buffer`
: Buffer to check.

Return value

`true` if the buffer holds no data, `false` otherwise.

Errors

None.

See also

`dmi_buffer_clear`, `dmi_buffer_resize`, `dmi_buffer_assign`

dmi_buffer_read

Synopsis

```
#include <opendmi/buffer.h>

bool dmi_buffer_read(
        const dmi_buffer_t *buffer,
        void               *ptr,
        size_t              offset,
        size_t              length);
```

Description

The `dmi_buffer_read` function copies `length` bytes at `offset` from the beginning of the data of the buffer into `ptr`.

This is what `dmi_reader_t` reads through; the reader keeps a cursor and a range of its own, and is what the decoding of the data is written against.

Parameters

`buffer`
: Buffer to read from.

`ptr`
: Buffer that receives the data; must be at least `length` bytes.

`offset`
: Byte offset from the beginning of the data.

`length`
: Number of the bytes to read.

Return value

`true` on success, `false` if the buffer holds fewer bytes than the requested range.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Data is `nullptr`, while the length is not zero.

See also

`dmi_buffer_write`, `dmi_buffer_at`, `dmi_reader_get_bytes`

dmi_buffer_resize

Synopsis

```
#include <opendmi/buffer.h>

bool dmi_buffer_resize(dmi_buffer_t *buffer, size_t length);
```

Description

The `dmi_buffer_resize` function sets the number of the bytes the buffer holds. The bytes a longer buffer gains are set to zero, and the ones a shorter buffer loses are dropped without the memory being given back, so that the buffer emptied with a length of zero writes the data next without allocating again.

Parameters

`buffer`
: Buffer to resize.

`length`
: Number of the bytes the buffer is to hold.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Memory of the buffer cannot grow.

See also

`dmi_buffer_clear`, `dmi_buffer_write`, `dmi_buffer_is_empty`

dmi_buffer_write

Synopsis

```
#include <opendmi/buffer.h>

bool dmi_buffer_write(
        dmi_buffer_t *buffer,
        const void   *ptr,
        size_t        offset,
        size_t        length);
```

Description

The `dmi_buffer_write` function copies `length` bytes at `ptr` into the data of the buffer, at `offset` from its beginning.

The data grows if the bytes are written past its end, and the ones between the end and `offset` are set to zero, the way the bytes a longer buffer gains are.

This is what `dmi_writer_t` writes through; the writer keeps a cursor and a range of its own, and is what the encoding of the data is written against.

Parameters

`buffer`
: Buffer to write to.

`ptr`
: Data to write.

`offset`
: Byte offset from the beginning of the data.

`length`
: Number of the bytes to write.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Data is `nullptr`, while the length is not zero.

`DMI_ERROR_ARGUMENT_INVALID`

Data ends past what a size holds, i.e. the sum of the offset and the length overflows.

`DMI_ERROR_OUT_OF_MEMORY`

Memory of the buffer cannot grow.

See also

`dmi_buffer_read`, `dmi_buffer_assign`, `dmi_writer_put_bytes`

`<opendmi/reader.h>`

dmi_reader_get_bytes

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_get_bytes(dmi_reader_t *reader, void *ptr, size_t length);
```

Description

The `dmi_reader_get_bytes` function copies `length` bytes from the reader into the buffer pointed to by `ptr` and advances the reader cursor.

Referring to the bytes of the buffer in place, rather than copying them, is what `dmi_reader_ref_bytes` does.

Parameters

`reader`
: Reader to read from.

`ptr`
: Buffer that receives the data; must be at least `length` bytes.

`length`
: Number of bytes to read.

Return value

`true` on success, `false` if there are fewer than `length` bytes remaining in the reader, or on error.

Errors

The error is raised against the context of the buffer being read.

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

See also

`dmi_reader_get_bytes_at`, `dmi_reader_ref_bytes`, `dmi_reader_skip`, `dmi_reader_has`, `dmi_decoder_get`

dmi_reader_get_bytes_at

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_get_bytes_at(
        const dmi_reader_t *reader,
        void               *ptr,
        size_t              offset,
        size_t              length);
```

Description

The `dmi_reader_get_bytes_at` function copies `length` bytes starting at `offset` from the beginning of the range into the buffer pointed to by `ptr`. The reader cursor is not modified.

Parameters

`reader`
: Reader to read from.

`ptr`
: Buffer that receives the data; must be at least `length` bytes.

`offset`
: Byte offset from the beginning of the range.

`length`
: Number of bytes to read.

Return value

`true` on success, `false` if the requested range exceeds the reader bounds, or on error.

Errors

The error is raised against the context of the buffer being read.

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

See also

`dmi_reader_get_bytes`, `dmi_reader_ref_bytes`, `dmi_decoder_get_at`

dmi_reader_has

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_has(const dmi_reader_t *reader, size_t length);
```

Description

The `dmi_reader_has` function checks whether at least `length` bytes remain from the current position, which is how a decoder tells a field the data carries in full from one it ends in the middle of.

Parameters

`reader`
: Reader to check.

`length`
: Number of bytes.

Return value

`true` if at least `length` bytes remain from the current position, `false` otherwise.

Errors

None.

See also

`dmi_reader_remaining`, `dmi_reader_is_done`, `dmi_decoder_incomplete`, `dmi_decoder_stop`

dmi_reader_initialize

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_initialize(
        dmi_reader_t       *reader,
        const dmi_buffer_t *buffer,
        size_t              offset,
        size_t              length);
```

Description

The `dmi_reader_initialize` function sets the reader up over `length` bytes at `offset` within `buffer`. The cursor is positioned at the beginning of the range, and the reader reads nothing outside it.

A reader reads bytes out of a range of a `dmi_buffer_t`, which holds them, the way `dmi_writer_t` writes them into one. What the bytes stand for is of no concern to the reader: the range it is given is a structure body, an entry point or whatever else the caller reads. Reading the values of a structure is what `dmi_decoder_t` does over a reader, see `dmi_decoder_initialize`.

The reader holds no memory of its own, so there is nothing to destroy: the data belongs to the buffer and outlives the readers of it.

Parameters

`reader`
: Reader to initialize.

`buffer`
: Buffer holding the data to read.

`offset`
: Offset the range begins at within the buffer.

`length`
: Number of the bytes of the range.

Return value

`true` on success, `false` if `reader` or `buffer` is `nullptr`, or if the buffer holds fewer bytes than the range.

Errors

None.

See also

`dmi_reader_reset`, `dmi_reader_seek`, `dmi_reader_get_bytes`, `dmi_decoder_initialize`, `dmi_writer_initialize`, `dmi_buffer_create`

dmi_reader_is_done

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_is_done(const dmi_reader_t *reader);
```

Description

The `dmi_reader_is_done` function checks whether the reader cursor has reached the end of the range.

Parameters

`reader`
: Reader to check.

Return value

`true` if there are no remaining bytes to read, `false` otherwise.

Errors

None.

See also

`dmi_reader_remaining`, `dmi_reader_has`, `dmi_reader_reset`

dmi_reader_mark

Synopsis

```
#include <opendmi/reader.h>

dmi_reader_mark_t dmi_reader_mark(const dmi_reader_t *reader);
```

Description

The `dmi_reader_mark` function takes a mark of the current position of the reader cursor.

Marks name a place to come back to without counting the bytes read in between, which is what records of a declared length need: mark the beginning of the record, read whatever fields it holds, then rewind with `dmi_reader_rewind`, or step over the record by its length with `dmi_reader_skip_ex`.

Marks are values: they are copied rather than allocated, nest without any bookkeeping, and need no release. A mark stays valid for as long as the reader reads the same range of the same buffer.

Parameters

`reader`
: Reader to mark.

Return value

Mark of the current position, which is rejected by `dmi_reader_rewind` if `reader` is `nullptr`.

Errors

None.

See also

`dmi_reader_rewind`, `dmi_reader_skip_ex`, `dmi_reader_seek`, `dmi_writer_mark`

dmi_reader_ref_bytes

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_ref_bytes(
        dmi_reader_t      *reader,
        size_t             length,
        const dmi_data_t **ptr);
```

Description

The `dmi_reader_ref_bytes` function refers to `length` bytes at the current position and advances the cursor, the way reading them does.

The bytes belong to the buffer being read and stay valid for as long as it holds them, which is how the binary members of the decoded structures refer to the data in place rather than holding a copy of it.

Parameters

`reader`
: Reader to read from.

`length`
: Number of bytes to refer to, may be zero.

`ptr`
: Pointer to the bytes, set to `nullptr` if `length` is zero.

Return value

`true` on success, `false` if there are fewer than `length` bytes remaining in the range, or on error.

Errors

The error is raised against the context of the buffer being read.

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`.

See also

`dmi_reader_get_bytes`, `dmi_buffer_at`, `dmi_decoder_get_binary`

dmi_reader_remaining

Synopsis

```
#include <opendmi/reader.h>

size_t dmi_reader_remaining(const dmi_reader_t *reader);
```

Description

The `dmi_reader_remaining` function returns the number of bytes from the current cursor position to the end of the range.

Parameters

`reader`
: Reader to query.

Return value

Number of bytes left to read, or zero if `reader` is `nullptr`.

Errors

None.

See also

`dmi_reader_has`, `dmi_reader_is_done`, `dmi_reader_tell`, `dmi_decoder_remaining`

dmi_reader_reset

Synopsis

```
#include <opendmi/reader.h>

void dmi_reader_reset(dmi_reader_t *reader);
```

Description

The `dmi_reader_reset` function repositions the reader cursor back to the start of the range so that it can be read again from the beginning.

Parameters

`reader`
: Reader to reset.

Errors

None.

See also

`dmi_reader_seek`, `dmi_reader_initialize`, `dmi_writer_reset`

dmi_reader_rewind

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_rewind(dmi_reader_t *reader, dmi_reader_mark_t mark);
```

Description

The `dmi_reader_rewind` function repositions the reader cursor to a mark taken with `dmi_reader_mark`.

Parameters

`reader`
: Reader to reposition.

`mark`
: Mark taken from the same reader with `dmi_reader_mark`.

Return value

`true` on success, `false` if the mark was taken from a reader reading another range.

Errors

None.

See also

`dmi_reader_mark`, `dmi_reader_skip_ex`, `dmi_reader_seek`

dmi_reader_seek

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_seek(dmi_reader_t *reader, size_t position);
```

Description

The `dmi_reader_seek` function repositions the reader cursor to `position` bytes from the beginning of the range.

The end of the range is accepted, since it is where the cursor stands once everything has been read, and a range may be read from there with nothing left in it.

Parameters

`reader`
: Reader to reposition.

`position`
: Byte offset from the beginning of the range.

Return value

`true` on success, `false` if `position` is past the end of the range.

Errors

None.

See also

`dmi_reader_tell`, `dmi_reader_reset`, `dmi_reader_mark`, `dmi_reader_rewind`, `dmi_reader_skip`

dmi_reader_skip

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_skip(dmi_reader_t *reader, size_t length);
```

Description

The `dmi_reader_skip` function moves the reader cursor forward by `length` bytes, effectively discarding that many bytes from the current position.

Stepping over a record by its declared length, rather than by the number of bytes read from it, is what

`dmi_reader_skip_ex` does.

Parameters

`reader`
: Reader to advance.

`length`
: Number of bytes to skip.

Return value

`true` on success, `false` if there are fewer than `length` bytes remaining in the reader.

Errors

None.

See also

`dmi_reader_skip_ex`, `dmi_reader_seek`, `dmi_reader_remaining`, `dmi_writer_skip`

dmi_reader_skip_ex

Synopsis

```
#include <opendmi/reader.h>

bool dmi_reader_skip_ex(
        dmi_reader_t      *reader,
        dmi_reader_mark_t  from,
        size_t             length);
```

Description

The `dmi_reader_skip_ex` function advances the reader cursor to `length` bytes past the mark `from`.

Records of a declared length are stepped over by that length rather than by the number of bytes read from them: mark the beginning of the record with `dmi_reader_mark`, read whatever fields it is known to hold, then advance by the length of the record from the mark.

Counting from the current position is what `dmi_reader_skip` does.

Parameters

`reader`
: Reader to advance.

`from`
: Mark to count `length` from, taken from the same reader with `dmi_reader_mark`.

`length`
: Number of bytes to advance by.

Return value

`true` on success, `false` if the resulting position is beyond the end of the range, or if `from` was taken from a reader reading another range.

Errors

None.

See also

`dmi_reader_skip`, `dmi_reader_mark`, `dmi_reader_rewind`, `dmi_writer_skip_ex`

dmi_reader_tell

Synopsis

```
#include <opendmi/reader.h>

size_t dmi_reader_tell(const dmi_reader_t *reader);
```

Description

The `dmi_reader_tell` function returns the offset of the next byte to be read from the beginning of the range, so reading a field right after the call gives the offset of that field within the range.

Parameters

`reader`
: Reader to query.

Return value

Offset of the cursor from the beginning of the range, or zero if `reader` is `nullptr`.

Errors

None.

See also

`dmi_reader_seek`, `dmi_reader_remaining`, `dmi_reader_mark`, `dmi_writer_tell`

<opendmi/writer.h>

dmi_writer_initialize

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_initialize(
        dmi_writer_t *writer,
        dmi_buffer_t *buffer,
        size_t        offset,
        ssize_t       length);
```

Description

The `dmi_writer_initialize` function sets the writer up over the range beginning at `offset` within `buffer`. The cursor is positioned at the beginning of the range, which the data is written into and past: a range beginning at the end of the buffer appends to it, and one beginning within the data written already writes over it.

A writer writes bytes into a range of a `dmi_buffer_t`, which holds them, the way `dmi_reader_t` reads them out of one. What the bytes stand for is of no concern to the writer: the caller writes the structures, the entry points and whatever else the data is made of. Writing the values of a structure is what `dmi_encoder_t` does over a writer, see `dmi_encoder_initialize`.

A range of a length of its own is one which something else follows, such as a record of a declared length or an area of a fixed size, and the writer refuses to write past it rather than writing over whatever comes next. A negative `length` gives a range bounded by nothing but the data itself.

The writer holds no memory of its own, so there is nothing to destroy: the data belongs to the buffer and outlives the writers of it.

Parameters

`writer`
: Writer to initialize.

`buffer`
: Buffer to write the data into.

`offset`
: Offset the range begins at within the buffer.

`length`
: Number of the bytes the range holds at most, or a negative number for a range of no length of its own.

Return value

`true` on success, `false` if `writer` or `buffer` is `nullptr`, or if the buffer holds fewer bytes than `offset`.

Errors

None.

See also

`dmi_writer_reset`, `dmi_writer_put_bytes`, `dmi_writer_seek`, `dmi_encoder_initialize`, `dmi_reader_initialize`, `dmi_buffer_create`

dmi_writer_mark

Synopsis

```
#include <opendmi/writer.h>

dmi_writer_mark_t dmi_writer_mark(const dmi_writer_t *writer);
```

Description

The `dmi_writer_mark` function takes a mark of the current position of the writer cursor.

Marks name a place to come back to without counting the bytes written in between, which is what the values written last need: mark the place, write whatever the value is counted over, then rewind with `dmi_writer_rewind` and write it, or pad the record out to its declared length with `dmi_writer_skip_ex`.

Marks are values: they are copied rather than allocated, nest without any bookkeeping, and need no release. A mark stays valid for as long as the writer holds the data it was taken over.

Parameters

`writer`
: Writer to mark.

Return value

Mark of the current position, which is rejected by `dmi_writer_rewind` if `writer` is `nullptr`.

Errors

None.

See also

`dmi_writer_rewind`, `dmi_writer_skip_ex`, `dmi_writer_seek`, `dmi_reader_mark`

dmi_writer_put_bytes

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_put_bytes(dmi_writer_t *writer, const void *ptr,
                          size_t length);
```

Description

The `dmi_writer_put_bytes` function writes `length` bytes at `ptr` at the current position and advances the writer cursor.

The buffer grows as far as the range allows, so writing past the data it holds appends to it.

Parameters

`writer`
: Writer to write to.

`ptr`
: Data to write.

`length`
: Number of bytes to write.

Return value

`true` on success, `false` if the bytes do not fit into the range, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_writer_put_bytes_at`, `dmi_writer_skip`, `dmi_buffer_write`, `dmi_encoder_put`

dmi_writer_put_bytes_at

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_put_bytes_at(
        dmi_writer_t *writer,
        const void   *ptr,
        size_t        offset,
        size_t        length);
```

Description

The `dmi_writer_put_bytes_at` function writes `length` bytes at `ptr` at `offset` from the beginning of the range. The writer cursor is not modified.

The data grows if the bytes are written past its end, and the ones between the end and `offset` are made up as zeros, the way the ones which are padded over are.

Parameters

`writer`
: Writer to write to.

`ptr`
: Data to write.

`offset`
: Byte offset from the beginning of the range.

`length`
: Number of bytes to write.

Return value

`true` on success, `false` if the bytes do not fit into the range, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_writer_put_bytes`, `dmi_writer_seek`, `dmi_encoder_put_at`

dmi_writer_reset

Synopsis

```
#include <opendmi/writer.h>

void dmi_writer_reset(dmi_writer_t *writer);
```

Description

The `dmi_writer_reset` function repositions the writer cursor to the beginning of its range, so that the data is written again from there.

The data written already is kept until it is written over, the way the data of a buffer is kept until it is.

Parameters

`writer`
: Writer to reset.

Errors

None.

See also

`dmi_writer_seek`, `dmi_writer_initialize`, `dmi_buffer_clear`, `dmi_reader_reset`

dmi_writer_rewind

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_rewind(dmi_writer_t *writer, dmi_writer_mark_t mark);
```

Description

The `dmi_writer_rewind` function repositions the writer cursor to a mark taken with `dmi_writer_mark`, so that the bytes written from there on are written again.

Parameters

`writer`
: Writer to reposition.

`mark`
: Mark taken from the same writer with `dmi_writer_mark`.

Return value

`true` on success, `false` if the mark was taken of another writer, or over data which has been discarded since.

Errors

None.

See also

`dmi_writer_mark`, `dmi_writer_seek`, `dmi_writer_skip_ex`

dmi_writer_seek

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_seek(dmi_writer_t *writer, size_t position);
```

Description

The `dmi_writer_seek` function repositions the writer cursor to `position` bytes from the beginning of the range.

The bytes at the position are written over, and the ones after them are left as they are, which is how a value the rest of the data decides, such as a length or a checksum, is filled in once it is known.

Parameters

`writer`
: Writer to reposition.

`position`
: Byte offset from the beginning of the data.

Return value

`true` on success, `false` if `position` is past the bytes of the range written so far.

Errors

None.

See also

`dmi_writer_tell`, `dmi_writer_mark`, `dmi_writer_rewind`, `dmi_writer_put_bytes_at`, `dmi_writer_reset`

dmi_writer_skip

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_skip(dmi_writer_t *writer, size_t length);
```

Description

The `dmi_writer_skip` function pads the data up to `length` bytes from the current position and advances the writer cursor.

Only the bytes past the end of the data are made up, as zeros, which is what the bytes a structure holds and nothing reads are left as. The ones written already are stepped over rather than written over, so that padding over a part of the data which is being written again keeps it.

Padding a record out to its declared length is what `dmi_writer_skip_ex` does.

Parameters

`writer`
: Writer to write to.

`length`
: Number of bytes to pad with.

Return value

`true` on success, `false` if the padding does not fit into the range.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_writer_skip_ex`, `dmi_writer_put_bytes`, `dmi_writer_seek`, `dmi_reader_skip`

dmi_writer_skip_ex

Synopsis

```
#include <opendmi/writer.h>

bool dmi_writer_skip_ex(
        dmi_writer_t      *writer,
```

```
        dmi_writer_mark_t  from,
        size_t             length);
```

Description

The `dmi_writer_skip_ex` function pads the data out to `length` bytes past the mark `from`.

Records of a declared length are padded out to that length rather than to the number of bytes written into them: mark the beginning of the record with `dmi_writer_mark`, write whatever fields it holds, then pad it out to the length of the record from the mark. The bytes written already are stepped over, the way `dmi_writer_skip` steps over them.

Counting from the current position is what `dmi_writer_skip` does.

Parameters

`writer`
: Writer to write to.

`from`
: Mark to count `length` from, taken from the same writer with `dmi_writer_mark`.

`length`
: Number of bytes the record takes.

Return value

`true` on success, `false` if the record is written past its own length already, if the padding does not fit into the range, or if `from` was taken of another writer.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_writer_skip`, `dmi_writer_mark`, `dmi_writer_rewind`, `dmi_reader_skip_ex`

dmi_writer_tell

Synopsis

```
#include <opendmi/writer.h>

size_t dmi_writer_tell(const dmi_writer_t *writer);
```

Description

The `dmi_writer_tell` function returns the offset the next byte is written at, counted from the beginning of the range, so marking a field right after the call gives the offset of that field within the range.

Parameters

`writer`
: Writer to query.

Return value

Offset the next byte is written at, counted from the beginning of the range, or zero if `writer` is `nullptr`.

Errors

None.

See also

`dmi_writer_seek`, `dmi_writer_mark`, `dmi_reader_tell`, `dmi_encoder_tell`

<opendmi/decoder.h>

dmi_decoder_context

Synopsis

```
#include <opendmi/decoder.h>

dmi_context_t *dmi_decoder_context(const dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_context` function returns the context of the structure the decoder is reading, which is where its errors are raised and its memory comes from.

Parameters

`decoder`
: Decoder to query.

Return value

Context of the structure, or `nullptr` if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_decoder_entity`, `dmi_entity_context`, `dmi_error_get_last`

dmi_decoder_entity

Synopsis

```
#include <opendmi/decoder.h>

dmi_entity_t *dmi_decoder_entity(const dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_entity` function returns the structure the decoder is reading, so that a decoding handler reaches it without being handed it separately.

Parameters

`decoder`
: Decoder to query.

Return value

Structure being decoded, or `nullptr` if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_decoder_initialize`, `dmi_decoder_context`, `dmi_encoder_entity`

dmi_decoder_get

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get(decoder, type, pvalue)
```

Description

The `dmi_decoder_get` macro reads `sizeof(type)` bytes of the structure, converts the value from SMBIOS wire format (little-endian) into host byte order using `dmi_decode`, and stores the result in the variable pointed to by `pvalue`. The cursor is advanced.

Parameters

`decoder`
: Decoder to read from.

`type`
: Wire-format type to read (e.g. `dmi_word_t`, `dmi_dword_t`).

`pvalue`
: Pointer to the variable that receives the decoded value.

Return value

`true` on success, `false` if there are not enough bytes remaining.

Errors

None.

See also

`dmi_decoder_get_at`, `dmi_decoder_get_bcd`, `dmi_decoder_get_uuid`, `dmi_decoder_get_string`, `dmi_decoder_get_bytes`, `dmi_encoder_put`

dmi_decoder_get_at

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get_at(decoder, offset, type, pvalue)
```

Description

The `dmi_decoder_get_at` macro reads `sizeof(type)` bytes at `offset` from the beginning of the body of the structure, converts the value from SMBIOS wire format (little-endian) into host byte order using `dmi_decode`, and stores the result in the variable pointed to by `pvalue`. The cursor is not moved.

Parameters

`decoder`
: Decoder to read from.

`offset`
: Byte offset from the beginning of the structure.

`type`
: Wire-format type to read (e.g. `dmi_word_t`, `dmi_dword_t`).

`pvalue`
: Pointer to the variable that receives the decoded value.

Return value

`true` on success, `false` if the requested range exceeds the structure bounds.

Errors

None.

See also

`dmi_decoder_get`, `dmi_decoder_get_bcd_at`, `dmi_decoder_get_uuid_at`, `dmi_decoder_get_bytes_at`, `dmi_encoder_put_at`

dmi_decoder_get_bcd

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get_bcd(decoder, type, pvalue)
```

Description

The `dmi_decoder_get_bcd` macro reads `sizeof(type)` bytes of the structure, decodes the value from BCD encoding using `dmi_decode_bcd`, and stores the result in the variable pointed to by `pvalue`. The cursor is advanced.

Parameters

`decoder`
: Decoder to read from.

`type`
: Wire-format type to read (e.g. `dmi_byte_t`).

`pvalue`
: Pointer to the variable that receives the decoded value.

Return value

`true` on success, `false` if there are not enough bytes remaining.

Errors

None.

See also

`dmi_decoder_get_bcd_at`, `dmi_decoder_get`, `dmi_encoder_put_bcd`

dmi_decoder_get_bcd_at

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get_bcd_at(decoder, offset, type, pvalue)
```

Description

The `dmi_decoder_get_bcd_at` macro reads `sizeof(type)` bytes at `offset` from the beginning of the body of the structure, decodes the value from BCD encoding using `dmi_decode_bcd`, and stores the result in the object pointed to by `pvalue`. The cursor is not moved.

Parameters

`decoder`
Decoder to read from.

`offset`
Byte offset from the beginning of the structure.

`type`
Wire-format type to read (e.g. `dmi_byte_t`).

`pvalue`
Pointer to the variable that receives the decoded value.

Return value

`true` on success, `false` if the requested range exceeds the structure bounds.

Errors

None.

See also

`dmi_decoder_get_bcd`, `dmi_decoder_get_at`, `dmi_encoder_put_bcd`

dmi_decoder_get_binary

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_get_binary(
        dmi_decoder_t *decoder,
        size_t         length,
        dmi_binary_t  *value);
```

Description

The `dmi_decoder_get_binary` function refers to `length` bytes of the structure at the current position and advances the cursor.

The bytes belong to the buffer the structure has been decoded from and stay valid for as long as it holds

them, which is how a structure carries whatever data the framework does not decode without holding a copy of it.

Parameters

`decoder`
: Decoder to read from.

`length`
: Number of the bytes to refer to, may be zero.

`value`
: Binary data descriptor.

Return value

`true` on success, `false` if there are not enough bytes remaining, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `value` is `nullptr`.

See also

`dmi_decoder_get_bytes`, `dmi_reader_ref_bytes`, `dmi_buffer_at`

dmi_decoder_get_bytes

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_get_bytes(dmi_decoder_t *decoder, void *ptr, size_t length);
```

Description

The `dmi_decoder_get_bytes` function copies `length` bytes of the structure into `ptr` and advances the cursor.

Referring to the bytes in place, rather than copying them, is what `dmi_decoder_get_binary` does.

Parameters

`decoder`
: Decoder to read from.

`ptr`
: Buffer that receives the bytes; must be at least `length` bytes.

`length`
: Number of the bytes to read.

Return value

`true` on success, `false` if there are not enough bytes remaining, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

See also

`dmi_decoder_get_bytes_at`, `dmi_decoder_get_binary`, `dmi_decoder_get`, `dmi_decoder_skip`

dmi_decoder_get_bytes_at

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_get_bytes_at(
        const dmi_decoder_t *decoder,
        void                *ptr,
        size_t               offset,
        size_t               length);
```

Description

The `dmi_decoder_get_bytes_at` function copies `length` bytes at `offset` from the beginning of the body of the structure into `ptr`. The cursor is not moved.

Parameters

`decoder`
: Decoder to read from.

`ptr`
: Buffer that receives the bytes; must be at least `length` bytes.

`offset`
: Byte offset from the beginning of the structure.

`length`
: Number of the bytes to read.

Return value

`true` on success, `false` if the requested range exceeds the structure bounds, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

See also

`dmi_decoder_get_bytes`, `dmi_decoder_get_at`, `dmi_reader_get_bytes_at`

dmi_decoder_get_string

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_get_string(dmi_decoder_t *decoder, const char **value);
```

Description

The `dmi_decoder_get_string` function reads the number a string is referred to by and looks the string up among the ones of the structure.

A reference to no string gives `nullptr`, and so does a reference to a string the structure does not carry, which raises an error of its own.

Parameters

`decoder`
: Decoder to read from.

`value`
: String the reference stands for.

Return value

`true` if the reference has been read, `false` if there are not enough bytes remaining, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `value` is `nullptr`.

`DMI_ERROR_STRING_NOT_FOUND`
: Structure carries no such string.

See also

`dmi_decoder_get`, `dmi_entity_string`, `dmi_encoder_put_string`

dmi_decoder_get_uuid

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get_uuid(decoder, pvalue)
```

Description

The `dmi_decoder_get_uuid` macro reads 16 bytes of the structure, converts the UUID from SMBIOS byte order into RFC 4122 representation using `dmi_uuid_decode`, and stores the result in the variable pointed to by `pvalue`. The cursor is advanced.

Parameters

`decoder`
: Decoder to read from.

`pvalue`
: Pointer to a `dmi_uuid_t` that receives the decoded UUID.

Return value

`true` on success, `false` if there are not enough bytes remaining.

Errors

None.

See also

`dmi_decoder_get_uuid_at`, `dmi_decoder_get`, `dmi_uuid_decode`, `dmi_encoder_put_uuid`

dmi_decoder_get_uuid_at

Synopsis

```
#include <opendmi/decoder.h>

dmi_decoder_get_uuid_at(decoder, offset, pvalue)
```

Description

The `dmi_decoder_get_uuid_at` macro reads 16 bytes at `offset` from the beginning of the body of the structure, converts the UUID from SMBIOS byte order into RFC 4122 representation using `dmi_uuid_decode`, and stores the result in the variable pointed to by `pvalue`. The cursor is not moved.

Parameters

`decoder`
Decoder to read from.

`offset`
Byte offset from the beginning of the structure.

`pvalue`
Pointer to a `dmi_uuid_t` that receives the decoded UUID.

Return value

`true` on success, `false` if the requested range exceeds the structure bounds.

Errors

None.

See also

`dmi_decoder_get_uuid`, `dmi_decoder_get_at`, `dmi_uuid_decode`

dmi_decoder_incomplete

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_incomplete(dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_incomplete` function marks the structure with `DMI_ENTITY_STATE_INCOMPLETE`, and the data left over, a part of the next field if any, is ignored.

It is intended for the decoders of the structures which newer versions of the specification have extended, and is called once the data ends in the middle of a set of fields, so that the length of the structure matches no version of the specification. The fields which are there in full are decoded before the call.

Parameters

`decoder`
: Decoder of the structure.

Return value

`true`, so that it can be returned by the decoder, or `false` if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_decoder_stop`, `dmi_decoder_remaining`, `dmi_reader_has`, `dmi_entity_decode`

dmi_decoder_initialize

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_initialize(dmi_decoder_t *decoder, dmi_entity_t *entity);
```

Description

A decoder reads the data of one structure into the values the model holds: `dmi_reader_t` gives it the bytes, and the structure being decoded says what they refer to, e.g. which string a number stands for. Neither knows of the other, which is what the decoder brings together.

The `dmi_decoder_initialize` function sets the decoder up over `entity`. The reader is set up over the body of the structure, positioned right after the header, and reads the copy with additional information applied whenever there is one.

The decoder holds no memory of its own, so there is nothing to destroy: the data belongs to the buffer of the structure.

Parameters

`decoder`
: Decoder to initialize.

`entity`
: Structure to decode.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Decoder or entity is `nullptr`.

See also

`dmi_decoder_entity`, `dmi_decoder_reader`, `dmi_decoder_get`, `dmi_entity_decode`, `dmi_fields_decode`, `dmi_encoder_initialize`

dmi_decoder_reader

Synopsis

```
#include <opendmi/decoder.h>

dmi_reader_t *dmi_decoder_reader(dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_reader` function returns the reader the bytes are read through, which the decoder positions and the decoding handlers move as they read.

Values are read through the decoder itself; the reader is for what it does not say, such as marking a record with `dmi_reader_mark` and stepping over it by its declared length with `dmi_reader_skip_ex`.

Parameters

`decoder`
: Decoder to query.

Return value

Reader of the data, or `nullptr` if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_reader_mark`, `dmi_reader_skip_ex`, `dmi_reader_tell`, `dmi_encoder_writer`

dmi_decoder_remaining

Synopsis

```
#include <opendmi/decoder.h>

size_t dmi_decoder_remaining(const dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_remaining` function returns the number of the bytes from the current position to the end of the body of the structure, which is how a decoder tells how many records of a repeating group the data carries.

Parameters

`decoder`
: Decoder to query.

Return value

Number of the bytes, or zero if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_decoder_skip`, `dmi_reader_remaining`, `dmi_reader_has`, `dmi_decoder_incomplete`

dmi_decoder_skip

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_skip(dmi_decoder_t *decoder, size_t length);
```

Description

The `dmi_decoder_skip` function moves the cursor forward by `length` bytes, which is what the fields of a structure the framework does not decode are stepped over with.

Parameters

`decoder`
: Decoder to advance.

`length`
: Number of the bytes to skip.

Return value

`true` on success, `false` if there are not enough bytes remaining.

Errors

None.

See also

`dmi_decoder_remaining`, `dmi_reader_skip`, `dmi_reader_skip_ex`

dmi_decoder_stop

Synopsis

```
#include <opendmi/decoder.h>

bool dmi_decoder_stop(dmi_decoder_t *decoder);
```

Description

The `dmi_decoder_stop` function marks the structure with `DMI_ENTITY_STATE_PARTIAL`.

It is intended for the decoders of the structures which newer versions of the specification have extended, and is called once the data has been read to its end while the decoder knows of more fields.

The data is required to be exhausted. A structure which ends anywhere else is incomplete instead, see `dmi_decoder_incomplete`.

Parameters

`decoder`
: Decoder of the structure.

Return value

`true`, so that it can be returned by the decoder, or `false` if `decoder` is `nullptr`.

Errors

None.

See also

`dmi_decoder_incomplete`, `dmi_decoder_remaining`, `dmi_reader_has`, `dmi_entity_decode`

`<opendmi/encoder.h>`

dmi_encoder_copy

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_copy(dmi_encoder_t *encoder, size_t length);
```

Description

The `dmi_encoder_copy` function writes `length` bytes the model does not hold at the current position: the ones the source data has there in the preserve mode, and zeros in the canonical one, or wherever the source data ends before them.

This is what the bits the specification reserves and the tails of the records longer than the fields they are known to hold are written with.

Parameters

`encoder`
: Encoder to write to.

`length`
: Number of the bytes.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_peek`, `dmi_encoder_remaining`, `dmi_encoder_put_bytes`

dmi_encoder_entity

Synopsis

```
#include <opendmi/encoder.h>
```

```
const dmi_entity_t *dmi_encoder_entity(const dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_entity` function returns the structure the encoder is writing, so that an encoding handler reaches it without being handed it separately.

Parameters

`encoder`
: Encoder to query.

Return value

Structure being encoded, or `nullptr` if `encoder` is `nullptr`.

Errors

None.

See also

`dmi_encoder_initialize`, `dmi_encoder_writer`, `dmi_decoder_entity`

dmi_encoder_finalize

Synopsis

```
#include <opendmi/encoder.h>

void dmi_encoder_finalize(dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_finalize` function frees the strings the encoder holds.

The data written belongs to the buffer it has been written into, and is left as it is, so an encoder is finished with rather than destroyed.

Parameters

`encoder`
: Encoder to finalize, or `nullptr`.

Errors

None.

See also

`dmi_encoder_initialize`, `dmi_encoder_finish`, `dmi_buffer_destroy`

dmi_encoder_finish

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_finish(dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_finish` function completes the formatted area by filling in the length of the header, which is known once every field has been written.

Parameters

`encoder`
: Encoder to complete.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_STATE_INVALID`
: Writer of the encoder has been moved back over the header, so that there is no formatted area to complete.

`DMI_ERROR_ENTITY_LENGTH_INVALID`
: Formatted area is longer than a byte can count.

See also

`dmi_encoder_initialize`, `dmi_encoder_finalize`, `dmi_encoder_put_bytes_at`, `dmi_entity_encode`

dmi_encoder_initialize

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_initialize(
```

```
        dmi_encoder_t      *encoder,
        dmi_buffer_t       *buffer,
        const dmi_entity_t *entity,
        dmi_encode_mode_t   mode,
        dmi_version_t       version);
```

Description

An encoder writes one structure back into the bytes it is made of: `dmi_writer_t` puts the bytes into a buffer, and the structure being encoded says what they are to carry. In the preserve mode a reader of the data the structure has been decoded from goes alongside, in step with what is written, so that the bytes the model does not hold are found at the position they are written at.

The `dmi_encoder_initialize` function sets the encoder up and writes the header of the structure. The structure is written into `buffer`, starting at the end of the data it holds, so that the structures of a table are written one after another. The length of the header is left for `dmi_encoder_finish` to fill in, once the length of the formatted area is known.

In the preserve mode, the strings of the structure are taken over as they are numbered, so that the fields referring to them keep their numbers, and the strings nothing refers to are kept too.

The encoder holds the strings of the structure, which are freed by `dmi_encoder_finalize`.

Parameters

`encoder`

Encoder to initialize.

`buffer`

Buffer to write the structure into.

`entity`

Structure to encode.

`mode`

How the bytes the model does not hold are written: `DMI_ENCODE_MODE_PRESERVE` takes them from the data the structure has been decoded from, so that encoding a structure which has not been changed gives back the very bytes it has been decoded from, and `DMI_ENCODE_MODE_CANONICAL` writes them the way the specification says they are, which is how a structure is repaired or built from scratch.

`version`

Version of the specification the canonical mode writes the structure for, which decides the groups of the fields it carries. Ignored in the preserve mode.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Encoder, buffer or entity is `nullptr`.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffers of the encoder cannot be allocated.

See also

`dmi_encoder_finalize`, `dmi_encoder_finish`, `dmi_encoder_put`, `dmi_entity_encode`, `dmi_fields_encode`, `dmi_decoder_initialize`

dmi_encoder_peek

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_peek(const dmi_encoder_t *encoder, void *ptr, size_t length);
```

Description

The `dmi_encoder_peek` function copies `length` bytes the source data holds at the current position into `ptr`, without advancing.

This is what the bits a field shares with the ones the specification reserves are read with, so that the reserved ones are written back as the structure has been decoded with them.

Parameters

`encoder`
: Encoder to read from.

`ptr`
: Bytes the source holds at the position.

`length`
: Number of the bytes to read.

Return value

`true` if the source holds the bytes, `false` if it has none, e.g. in the canonical mode, or ends before them, or on error.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero and the encoder has a source to look at.

See also

`dmi_encoder_remaining`, `dmi_encoder_copy`, `dmi_encoder_tell`

dmi_encoder_put

Synopsis

```
#include <opendmi/encoder.h>

dmi_encoder_put(encoder, type, value)
```

Description

The `dmi_encoder_put` macro converts `value` into `type` in SMBIOS wire format (little-endian) using `dmi_encode` and writes `sizeof(type)` bytes. The cursor is advanced.

Parameters

`encoder`
: Encoder to write to.

`type`
: Wire-format type to write (e.g. `dmi_word_t`, `dmi_dword_t`).

`value`
: Value to write.

Return value

`true` on success, `false` if the buffer cannot grow.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_put_at`, `dmi_encoder_put_bcd`, `dmi_encoder_put_uuid`, `dmi_encoder_put_string`, `dmi_encoder_put_bytes`, `dmi_decoder_get`

dmi_encoder_put_at

Synopsis

```
#include <opendmi/encoder.h>
```

```
dmi_encoder_put_at(encoder, offset, type, value)
```

Description

The `dmi_encoder_put_at` macro converts `value` into `type` in SMBIOS wire format (little-endian) using `dmi_encode` and writes `sizeof(type)` bytes at `offset` from the beginning of the structure. The cursor is not moved.

Parameters

`encoder`
Encoder to write to.

`offset`
Byte offset from the beginning of the structure.

`type`
Wire-format type to write (e.g. `dmi_word_t`, `dmi_dword_t`).

`value`
Value to write.

Return value

`true` on success, `false` if the bytes do not fit into the range.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
Buffer of the writer cannot grow.

See also

`dmi_encoder_put`, `dmi_encoder_put_bytes_at`, `dmi_decoder_get_at`

dmi_encoder_put_bcd

Synopsis

```
#include <opendmi/encoder.h>

dmi_encoder_put_bcd(encoder, type, value)
```

Description

The `dmi_encoder_put_bcd` macro spells `value` out as the decimal digits `sizeof(type)` bytes hold, two per byte, using `dmi_encode_bcd`, and writes them. The cursor is advanced.

Parameters

`encoder`
: Encoder to write to.

`type`
: Wire-format type to write (e.g. `dmi_byte_t`).

`value`
: Value to write.

Return value

`true` on success, `false` if the buffer cannot grow.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_put`, `dmi_decoder_get_bcd`

dmi_encoder_put_bytes

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_put_bytes(dmi_encoder_t *encoder, const void *ptr,
                           size_t length);
```

Description

The `dmi_encoder_put_bytes` function writes `length` bytes at `ptr` at the current position.

The position in the source data advances by the same number of bytes, so that the two stay in step, as long as there are bytes left in it.

Parameters

`encoder`
: Encoder to write to.

`ptr`
: Bytes to write.

`length`
: Number of the bytes.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_put_bytes_at`, `dmi_encoder_put`, `dmi_encoder_copy`, `dmi_writer_put_bytes`

dmi_encoder_put_bytes_at

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_put_bytes_at(
        dmi_encoder_t *encoder,
        const void    *ptr,
        size_t         offset,
        size_t         length);
```

Description

The `dmi_encoder_put_bytes_at` function writes `length` bytes at `ptr` at `offset` from the beginning of the structure, without moving the position.

The function is meant for the bytes written already, which is how a value the rest of the structure decides, such as a length, is filled in once it is known. The bytes past the position are not refused, but are written over by whatever is written next, since the position does not follow them.

Parameters

`encoder`
: Encoder to write to.

`ptr`
: Bytes to write.

`offset`
: Byte offset from the beginning of the structure.

`length`
: Number of the bytes.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `ptr` is `nullptr`, while `length` is not zero.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_put_bytes`, `dmi_encoder_put_at`, `dmi_encoder_finish`, `dmi_writer_put_bytes_at`

dmi_encoder_put_string

Synopsis

```
#include <opendmi/encoder.h>

bool dmi_encoder_put_string(dmi_encoder_t *encoder, const char *value);
```

Description

The `dmi_encoder_put_string` function writes the number the string is referred to by.

The string is looked up among the ones the structure has been decoded with, so that a field keeps the number of the string it refers to, and is added to the strings of the encoder otherwise. A missing string is written as zero.

Parameters

`encoder`
: Encoder to write to.

`value`
: String to refer to, or `nullptr`.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffers of the encoder cannot grow.

`DMI_ERROR_ARGUMENT_INVALID`
: Structure refers to more strings than a byte can number.

See also

`dmi_encoder_put`, `dmi_encoder_finish`, `dmi_decoder_get_string`

dmi_encoder_put_uuid

Synopsis

```
#include <opendmi/encoder.h>

dmi_encoder_put_uuid(encoder, value)
```

Description

The `dmi_encoder_put_uuid` macro converts `value` from the RFC 4122 representation into SMBIOS byte order using `dmi_uuid_encode` and writes 16 bytes. The cursor is advanced.

Parameters

`encoder`
: Encoder to write to.

`value`
: `dmi_uuid_t` to write.

Return value

`true` on success, `false` if the buffer cannot grow.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer of the writer cannot grow.

See also

`dmi_encoder_put`, `dmi_uuid_encode`, `dmi_decoder_get_uuid`

dmi_encoder_remaining

Synopsis

```
#include <opendmi/encoder.h>

size_t dmi_encoder_remaining(const dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_remaining` function returns the number of the bytes the source data holds from the current position on, which is zero in the canonical mode.

Parameters

`encoder`
: Encoder to query.

Return value

Number of the bytes.

Errors

None.

See also

`dmi_encoder_peek`, `dmi_encoder_copy`, `dmi_decoder_remaining`

dmi_encoder_tell

Synopsis

```
#include <opendmi/encoder.h>

size_t dmi_encoder_tell(const dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_tell` function returns the position of the next byte to write, counted from the beginning of the structure the way the specification counts it, so that it is the offset the field written next takes.

Parameters

`encoder`
: Encoder to query.

Return value

Position of the next byte to write.

Errors

None.

See also

`dmi_encoder_put_bytes_at`, `dmi_writer_tell`, `dmi_reader_tell`

dmi_encoder_writer

Synopsis

```
#include <opendmi/encoder.h>

dmi_writer_t *dmi_encoder_writer(dmi_encoder_t *encoder);
```

Description

The `dmi_encoder_writer` function returns the writer the bytes are written through, which the encoder positions and the encoding handlers move as they write.

Values are written through the encoder itself; the writer is for what it does not say, such as marking a record with `dmi_writer_mark` and padding it out to its declared length with `dmi_writer_skip_ex`.

Parameters

`encoder`
: Encoder to query.

Return value

Writer of the data, or `nullptr` if `encoder` is `nullptr`.

Errors

None.

See also

`dmi_writer_mark`, `dmi_writer_skip_ex`, `dmi_writer_tell`, `dmi_decoder_reader`

<opendmi/lint.h>

dmi_lint

Synopsis

```
#include <opendmi/lint.h>

bool dmi_lint(
        dmi_context_t            *context,
        const dmi_lint_options_t *options,
        dmi_lint_issue_fn        *handler,
        void                     *data);
```

Description

The `dmi_lint` function checks the SMBIOS data of an open context against the rules of the library, and reports every issue it finds to `handler`. The **opendmi lint** command is built on it, see `opendmi-lint`.

Rules come in three scopes, which tell what a rule is checked against:

`DMI_LINT_SCOPE_ENTRY`
: The entry point, which is checked once.

`DMI_LINT_SCOPE_ENTITY`
: A single structure, which is checked for every structure of the table. Rules which apply to any structure come first, and the rules the specification of the structure declares follow them.

`DMI_LINT_SCOPE_TABLE`
: The table as a whole, which is checked once, after every structure.

Issues are reported as they are found: the ones of the entry point first, then the ones of the structures in their order in the table, and the ones of the table as a whole last. Totals of the table, e.g. the number of the structures of every type, are gathered before any rule is checked, so that every rule is free to use them, see `dmi_lint_totals`.

A rule is checked unless one of the following holds:

- The rule is optional, and `all` of the options is not set. Optional rules are the ones which are expensive, or which are a matter of taste rather than of the specification.
- The severity of the rule in the profile of the check is `DMI_LINT_SEVERITY_NONE`, see `dmi_lint_rule_severity`.
- The rule filter of the options rejects it.

The profile tells what the data is expected to be:

`DMI_LINT_PROFILE_READER`
: Data has been read from a platform, and is expected to have the defects of its firmware, which nobody

is going to fix. Issues are reported with a lower severity: requirements of the specification which are not met are mostly warnings, and recommendations which are not followed are notes.

`DMI_LINT_PROFILE_PRODUCER`
: Data is being produced, e.g. by firmware under development or by a generator, and is expected to conform to the specification. Requirements which are not met are errors, and recommendations which are not followed are warnings.

Each rule declares its severity in both profiles, see `dmi_lint_rule_severity`.

The data is checked against the version of the specification given in the options, or against the version of the entry point if the options give none.

Options

The `dmi_lint_options_t` structure holds the following members, all of which may be left zero for the defaults:

`profile`
: Profile the data is checked against, `DMI_LINT_PROFILE_READER` by default.

`all`
: Whether the optional rules are checked as well.

`version`
: Version of the specification the data is checked against, or `DMI_VERSION_NONE` to take the version from the entry point.

`rule_filter`
: Function deciding whether a rule is checked, or `nullptr` to check all the rules enabled by the profile:

```
typedef bool dmi_lint_rule_fn(void *data, const dmi_lint_rule_t *rule);
```

`rule_filter_data`
: Data passed to the rule filter.

Issues

The handler is called for every issue found:

```
typedef void dmi_lint_issue_fn(void *data, const dmi_lint_issue_t *issue);
```

The `dmi_lint_issue_t` structure holds the following members:

`rule`
: Rule which has found the issue, see `dmi_lint_rule_name`.

`severity`
: Severity of the issue, which is the severity of the rule in the profile of the check: `DMI_LINT_SEVERITY_NOTE`, `DMI_LINT_SEVERITY_WARNING` or `DMI_LINT_SEVERITY_ERROR`.

`handle`

Handle of the structure the issue belongs to, or `DMI_HANDLE_INVALID` if it belongs to the entry point or to the table as a whole.

`type`

Type of the structure the issue belongs to.

`attribute`

Code name of the attribute the issue belongs to, or `nullptr` if it belongs to the whole structure.

`offset`

Offset of the issue from the beginning of the table, or `DMI_LINT_NO_OFFSET` if the issue has no offset of its own.

`message`

Details of the issue, e.g. the expected and the actual value.

The issue and the strings it points to are valid until the handler returns, so the handler has to copy whatever it keeps. Issues are not counted by the library: the handler counts them, since it has to tell their severities apart anyway.

Parameters

`context`

Open context, the data of which is checked.

`options`

Options of the check, or `nullptr` for the defaults.

`handler`

Handler called for every issue found, or `nullptr` to check the data without reporting.

`data`

Data passed to the handler.

Return value

Returns `true` if the data has been checked, whatever issues have been found, and `false` if it cannot be checked. Returns `false` without raising an error if `context` is `nullptr`.

Errors

`DMI_ERROR_STATE_INVALID`

Context is not open, see `dmi_is_open`.

`DMI_ERROR_OUT_OF_MEMORY`

State of the check cannot be allocated.

Example

Count the errors in the data of a dump:

```
static void count_errors(void *data, const dmi_lint_issue_t *issue)
{
    if (issue->severity == DMI_LINT_SEVERITY_ERROR)
        (*(size_t *)data)++;
}

size_t errors = 0;

if (dmi_load(context, "smbios.bin") &&
    dmi_lint(context, nullptr, count_errors, &errors))
{
    printf("%zu errors\n", errors);
}
```

See also

`dmi_lint_rules`, `dmi_lint_rule_name`, `dmi_lint_issue`, `dmi_is_open`, `opendmi-lint`

dmi_lint_context

Synopsis

```
#include <opendmi/lint.h>

dmi_context_t *dmi_lint_context(dmi_lint_t *lint);

dmi_version_t dmi_lint_version(const dmi_lint_t *lint);

const dmi_lint_totals_t *dmi_lint_totals(const dmi_lint_t *lint);
```

Description

The functions give the checks of the lint rules what they need to know of the check in progress, see `dmi_lint`.

The `dmi_lint_context` function returns the context the data of which is being checked, e.g. to look up another structure a structure refers to.

The `dmi_lint_version` function returns the version of the specification the data is checked against: the one given in the options of the check, or the version of the entry point if the options give none. Rules of the requirements a version introduces are checked against it, so that the data of an older version is not held to them.

The `dmi_lint_totals` function returns the totals of the table, which are gathered before any rule is checked, so that they are complete for the rules of every scope. The `dmi_lint_totals_t` structure holds the following members:

`entity_count`
: Number of the structures of the table.

`entity_max_size`
: Length of the longest structure of the table, with its strings.

`type_counts`
: Number of the structures of every type, indexed by the type number.

`terminator`
: First end-of-table structure (type 127), or `nullptr` if the table has none.

The totals belong to the check, and are valid while it is in progress.

Parameters

`lint`
: Check in progress, as passed to the check of the rule.

Return value

`dmi_lint_context` returns the context, `dmi_lint_version` the version, and `dmi_lint_totals` the totals of the table. If `lint` is `nullptr`, they return `nullptr`, `DMI_VERSION_NONE` and `nullptr` respectively.

Errors

None.

See also

`dmi_lint`, `dmi_lint_issue`

dmi_lint_issue

Synopsis

```
#include <opendmi/lint.h>

void dmi_lint_issue(
        dmi_lint_t         *lint,
        const dmi_entity_t *entity,
        const char         *attribute,
        size_t              offset,
        const char         *format,
        ...);

size_t dmi_lint_entity_offset(const dmi_lint_t *lint, const dmi_entity_t
*entity);
```

```
size_t dmi_lint_string_offset(
        const dmi_lint_t   *lint,
        const dmi_entity_t *entity,
        size_t              num);
```

Description

The functions are used by the checks of the lint rules, which the library calls while `dmi_lint` is in progress:

```
typedef void dmi_lint_check_fn(dmi_lint_t *lint, const dmi_entity_t *entity);
```

The `dmi_lint_issue` function reports an issue the rule being checked has found. The issue is passed to the handler of the check, see `dmi_lint`, along with the rule, which the check is not to name, and with its severity in the profile of the check. The details are formatted the way `printf`(3) does it, and are cut to 255 bytes. If the check has no handler, nothing is reported.

The `dmi_lint_entity_offset` function returns the offset of a structure from the beginning of the table, which issues of the structure as a whole are reported at. An issue of a field is reported at the offset of the structure plus the offset of the field.

The `dmi_lint_string_offset` function returns the offset of a string of a structure from the beginning of the table, which issues of the string are reported at.

Offsets are the ones of the table the context holds, so a structure decoded from anywhere else, e.g. from a copy carrying additional information entries, has none.

Parameters

`lint`
: Check in progress, as passed to the check of the rule.

`entity`
: Structure the issue belongs to, or `nullptr` if it belongs to the entry point or to the table as a whole.

`attribute`
: Code name of the attribute the issue belongs to, e.g. `serial-number`, or `nullptr` if it belongs to the whole structure.

`offset`
: Offset of the issue from the beginning of the table, or `DMI_LINT_NO_OFFSET` if it has no offset of its own.

`format`
: `printf`(3)-style format string of the details of the issue.

`…`
: Format arguments.

`num`
: One-based number of the string.

Return value

`dmi_lint_issue` returns no value.

`dmi_lint_entity_offset` and `dmi_lint_string_offset` return the offset, or `DMI_LINT_NO_OFFSET` if it cannot be told: if `lint` or `entity` is `nullptr`, if the structure is not one of the table of the context, or if the structure has no string of number `num`.

Errors

The errors are raised against the context being checked.

`DMI_ERROR_ARGUMENT_NULL`
: Raised by `dmi_lint_issue` if `lint` or `format` is `nullptr`, and by `dmi_lint_entity_offset` and `dmi_lint_string_offset` if `lint` or `entity` is `nullptr`. Without a check in progress it is not reported, since there is no context to hold it.

Notes

`dmi_lint_issue` may be called only by the check of a rule, while the rule is being checked.

Example

Check of a rule of a structure type, which reports a value out of range:

```
static void check_speed(dmi_lint_t *lint, const dmi_entity_t *entity)
{
    const dmi_foo_t *info = dmi_entity_info(entity, DMI_TYPE(foo));
    if ((info == nullptr) || (info->speed <= 100))
        return;

    dmi_lint_issue(lint, entity, "speed", dmi_lint_entity_offset(lint, entity)
+ 0x05,
                   "speed of %u is above the limit of 100", info->speed);
}
```

See also

`dmi_lint`, `dmi_lint_context`, `dmi_lint_rules`

dmi_lint_rule_name

Synopsis

```
#include <opendmi/lint.h>
```

```
const char *dmi_lint_rule_name(const dmi_lint_rule_t *rule);
```

Description

The `dmi_lint_rule_name` function returns the printable description of what a lint rule checks, which tools show next to the code of the rule, e.g. `entry.checksum`.

The name is translated to the locale set by `dmi_set_locale`, if the library has a translation for the rule: the translation is the string keyed by the code of the rule in the `lint-rule` table. Otherwise the name is the `name` member of the parameters of the rule, as the rule declares it. Codes of the rules are never translated.

The name belongs to the rule or to the translation, and must not be freed. An untranslated name is static; a translated one is valid until the locale is changed by `dmi_set_locale`.

Parameters

`rule`
: Rule to name, e.g. one of the rules returned by `dmi_lint_rules`.

Return value

Returns the printable name of the rule, or `nullptr` if `rule` is `nullptr`.

Errors

None.

See also

`dmi_set_locale`

dmi_lint_rules

Synopsis

```
#include <opendmi/lint.h>

const dmi_lint_rule_t *const *dmi_lint_rules(void);

const dmi_lint_rule_t *dmi_lint_rule_find(dmi_context_t *context, const char
*code);

dmi_lint_severity_t dmi_lint_rule_severity(
        const dmi_lint_rule_t *rule,
        dmi_lint_profile_t     profile);
```

Description

Rules of the library are of two kinds:

- Rules which apply to any structure, to the entry point or to the table as a whole, e.g. `entry.checksum` or `table.terminator`, which the library holds itself.
- Rules which apply to the structures of a single type, e.g. `memory-device.attributes`, which the specification of the type declares. The rules of the vendor-specific types are available only once the modules decoding them are enabled.

The `dmi_lint_rules` function returns the rules of the first kind.

The `dmi_lint_rule_find` function finds a rule of either kind by its code name. The rules of the library are searched first, and the ones of the types known to `context` then, so that the rules of the enabled modules are found as well.

The `dmi_lint_rule_severity` function returns the severity of the issues a rule reports in a profile, see `dmi_lint`.

The `dmi_lint_rule_t` structure holds the following members:

`code`
: Code name of the rule, e.g. `entry.checksum`, which is never translated.

`scope`
: Part of the data the rule is checked against: `DMI_LINT_SCOPE_ENTRY`, `DMI_LINT_SCOPE_ENTITY` or `DMI_LINT_SCOPE_TABLE`.

`check`
: Function performing the check.

`params`
: Parameters of the rule: its printable `name`, see `dmi_lint_rule_name`, its severities `reader_severity` and `producer_severity`, and whether it is `optional`, that is checked only when all the rules are asked for.

Rules are static, and must not be modified or freed.

Parameters

`context`
: Context, the types of which are searched for the rule, or `nullptr` to search the rules which apply to any structure only.

`code`
: Code name of the rule.

`rule`
: Rule in question.

`profile`
: Profile the data is checked against: `DMI_LINT_PROFILE_READER` or `DMI_LINT_PROFILE_PRODUCER`.

Return value

`dmi_lint_rules` returns an array of the rules, terminated with `nullptr`.

`dmi_lint_rule_find` returns the rule, or `nullptr` if there is no rule with such code name, or if `code` is `nullptr`.

`dmi_lint_rule_severity` returns the severity of the issues the rule reports in the profile, or `DMI_LINT_SEVERITY_NONE` if the rule is not checked in the profile, or if `rule` is `nullptr`.

Errors

None.

Example

List the rules which apply to any structure, along with their severities in the producer profile:

```
for (const dmi_lint_rule_t *const *rule = dmi_lint_rules(); *rule != nullptr;
rule++) {
    dmi_lint_severity_t severity = dmi_lint_rule_severity(*rule,
DMI_LINT_PROFILE_PRODUCER);

    printf("%-32s %-8s %s\n", (*rule)->code,
           dmi_name_lookup(&dmi_lint_severity_names, severity),
           dmi_lint_rule_name(*rule));
}
```

See also

`dmi_lint`, `dmi_lint_rule_name`, `dmi_name_lookup`, `opendmi-lint`

<opendmi/backend/generic.h>

dmi_generic_find_entry_addr

Synopsis

```
#include <opendmi/backend/generic.h>

bool dmi_generic_find_entry_addr(
        dmi_context_t *context,
        const char    *device,
        uint64_t      *paddr);

bool dmi_generic_find_anchor(
        dmi_context_t *context,
        dmi_data_t    *buffer,
        uint64_t       base_addr,
        size_t         area_size,
        const char    *anchor,
        uint64_t      *paddr);
```

Description

The functions find the SMBIOS entry point the way legacy firmware lets it be found, without the help of the system: by scanning the memory between `0xF0000` and `0xFFFFF` for the anchor string of the entry point, on 16-byte boundaries. The backends fall back on them when the system does not tell the address, e.g. on legacy firmware with no EFI configuration tables.

The `dmi_generic_find_entry_addr` function reads the memory area through `device`, see `dmi_memory_load`, and looks for the anchors of the entry points of SMBIOS 3.0 (*`SM3`*), SMBIOS 2.1 (*`SM`*) and the legacy one (*`DMI`*), in this order, so that the newest entry point the firmware provides is found.

The `dmi_generic_find_anchor` function looks for a single anchor in a memory area which has been read already.

Anchor strings occur in firmware code and data as well, so a match of the anchor alone proves nothing. An entry point is taken only if its length is valid and fits in the area, and its checksum is valid, the way dmidecode checks it. The intermediate entry point of SMBIOS 2.1 must be present and valid as well. Entry points which are not are skipped, and the scan goes on past them.

Parameters

`context`

Context to raise errors on, and to log the progress of the scan on.

`device`

Path to the memory device to read the area through, e.g. `/dev/mem`, or `nullptr` in the kernel, where the memory is mapped directly.

`buffer`
: Memory area which has been read.

`base_addr`
: Physical address of the memory area, which is aligned on 16 bytes.

`area_size`
: Size of the memory area, which is a multiple of 16 bytes.

`anchor`
: Anchor string of the entry point to look for, e.g. `DMI_ANCHOR_V30`.

`paddr`
: Variable to store the physical address of the entry point in. It is left as it is if no entry point is found.

Return value

Returns `true` if a valid entry point has been found, and `false` otherwise.

Errors

`dmi_generic_find_entry_addr` raises the errors of `dmi_memory_load` if the memory area cannot be read. Finding no entry point is not an error.

`dmi_generic_find_anchor` raises `DMI_ERROR_ARGUMENT_NULL` if `buffer`, `anchor` or `paddr` is `nullptr`.

Notes

The functions are available on x86 only, since other architectures have no such memory area, and the header is installed only on the systems whose backends use them, which are FreeBSD and NetBSD.

See also

`dmi_generic_parse_entry_addr`, `dmi_memory_load`, `dmi_open`

dmi_generic_parse_entry_addr

Synopsis

```
#include <opendmi/backend/generic.h>

bool dmi_generic_parse_entry_addr(dmi_context_t *context, const char *str,
uint64_t *paddr);
```

Description

The `dmi_generic_parse_entry_addr` function parses the physical address of the SMBIOS entry point, as a system tells it in text, e.g. the `hint.smbios.0.mem` kernel environment variable of FreeBSD. The

backends of such systems read the entry point from that address, see `dmi_memory_load`.

The address is read the way `strtoull`(3) reads it with the base of zero: as hexadecimal with the `0x` prefix, as octal with the `0` prefix, and as decimal otherwise. The whole string must be the address.

The address is 64-bit on every system, so that an address above 4 GiB is passed on as it is, and is rejected by `dmi_memory_load` if the system cannot reach it.

Parameters

`context`
: Context to raise errors on.

`str`
: Address in text.

`paddr`
: Variable to store the address in. It is left as it is on failure.

Return value

Returns `true` if the address has been parsed, and `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `str` or `paddr` is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: String is not an address, or the address does not fit in 64 bits.

Notes

The header is installed only on the systems whose backends use it, which are FreeBSD and NetBSD.

See also

`dmi_generic_find_entry_addr`, `dmi_memory_load`, `dmi_open`

<opendmi/utils.h>

dmi_alloc

Synopsis

```
#include <opendmi/utils.h>

void *dmi_alloc(dmi_context_t *context, size_t size);
void dmi_free(void *ptr);
```

Description

The `dmi_alloc` function allocates `size` bytes of memory, set to zero, the way `calloc(1, size)` does. If the memory cannot be allocated, the failure is reported on `context`.

The `dmi_free` function frees the memory `ptr` points to, the way `free()` does, and does nothing if `ptr` is `nullptr`.

Memory allocated by `dmi_alloc` is freed with `dmi_free` or `free()`, and so is memory the library allocates for the caller, e.g. the strings returned by `dmi_asprintf` or `dmi_date_format`.

Parameters

`context`
: Context to raise the error on, or `nullptr` to report none.

`size`
: Number of bytes to allocate.

`ptr`
: Memory to free, or `nullptr`.

Return value

The `dmi_alloc` function returns a pointer to the allocated memory, or `nullptr` if it cannot be allocated.

The `dmi_free` function returns no value.

Errors

`DMI_ERROR_OUT_OF_MEMORY`
: Raised by `dmi_alloc` on `context` if the memory cannot be allocated, unless `context` is `nullptr`.

See also

`calloc`(3), `free`(3), `dmi_error_get_last`, `dmi_trace_out_of_memory`

dmi_checksum_calc

Synopsis

```
#include <opendmi/utils.h>

bool dmi_checksum_calc(const void *data, size_t length, uint8_t *pchecksum);
bool dmi_checksum_test(const void *data, size_t length);
```

Description

SMBIOS protects its entry points with an 8-bit checksum: the bytes of the data, the checksum byte included, add up to zero modulo 256.

The `dmi_checksum_calc` function computes the checksum of the `length` bytes at `data`, that is the value which brings their sum to zero modulo 256, and stores it in the variable `pchecksum` points to. The checksum byte within the data is counted as any other byte, so it is set to zero before the checksum is computed.

The `dmi_checksum_test` function tells whether the `length` bytes at `data` add up to zero modulo 256, i.e. whether the checksum they hold is valid.

Parameters

`data`
: Data to sum.

`length`
: Number of bytes to sum.

`pchecksum`
: Variable to store the checksum in.

Return value

The `dmi_checksum_calc` function returns `true` if the checksum has been computed, and `false` if `data` or `pchecksum` is `nullptr`, in which case the variable is left as it is. The checksum of zero bytes is zero.

The `dmi_checksum_test` function returns `true` if the bytes add up to zero, which they do when `length` is zero, and `false` otherwise, or if `data` is `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `data` or `pchecksum` is `nullptr`. The functions take no context, so the error reaches no error queue.

Example

```
uint8_t data[16] = { ... };

data[5] = 0;
if (!dmi_checksum_calc(data, sizeof(data), &data[5]))
    return false;

assert(dmi_checksum_test(data, sizeof(data)));
```

See also

`dmi_generic_find_anchor`, `dmi_generic_find_entry_addr`

dmi_file_load

Synopsis

```
#include <opendmi/utils.h>

bool dmi_file_load(
        dmi_buffer_t *buffer,
        const char   *path,
        off_t         offset,
        size_t        length);
```

Description

The `dmi_file_load` function opens the file at `path` and reads its contents into `buffer`, which is made to hold exactly the bytes read: fewer than requested, when the file holds fewer.

The data the buffer held before is dropped, and so is whatever has been read when the reading fails.

This is what the backends reading a table or an entry point out of a file fill the buffer they are given with, see `dmi_backend_t`.

Parameters

`buffer`
: Buffer to read the contents into.

`path`
: Path to the file to read.

`offset`
: Offset to read from, or a negative value to read from the beginning of the file.

`length`
: Maximum number of bytes to read, or zero to read the whole file.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Path is `nullptr`.

`DMI_ERROR_FILE_OPEN_FAILED`
: File cannot be opened.

`DMI_ERROR_FILE_STAT_FAILED`
: Length of the file cannot be told.

`DMI_ERROR_FILE_READ_FAILED`
: File cannot be read.

`DMI_ERROR_OUT_OF_MEMORY`
: Buffer cannot hold the data.

Notes

Errors are raised on the context of the buffer, see `dmi_buffer_context`.

See also

`dmi_memory_load`, `dmi_buffer_create`, `dmi_buffer_assign`, `dmi_file_read`, `dmi_load`

dmi_ipow32

Synopsis

```
#include <opendmi/utils.h>

uint32_t dmi_ipow32(uint32_t value, unsigned int factor);
uint64_t dmi_ipow64(uint64_t value, unsigned int factor);
```

Description

The `dmi_ipow32` and `dmi_ipow64` functions raise `value` to the power `factor` in unsigned integer arithmetic of 32 and 64 bits, by binary exponentiation, e.g. to scale a value by a power of ten given in SMBIOS data. Any value raised to the power of zero is 1, zero included.

A result which does not fit in the type wraps around, as unsigned multiplication does: the functions return it modulo 2^{32} and 2^{64}.

Parameters

`value`
: Base.

`factor`
: Exponent.

Return value

The functions return `value` raised to the power `factor`, modulo 2^{32} or 2^{64}.

Errors

None.

See also

`pow`(3)

dmi_memory_load

Synopsis

```
#include <opendmi/utils.h>

bool dmi_memory_load(
        dmi_buffer_t *buffer,
        const char   *path,
        uint64_t      base,
        size_t        length);
```

Description

The `dmi_memory_load` function opens `path`, maps `length` bytes starting at physical offset `base` into memory using `mmap`(2), copies the data into `buffer`, then unmaps the region. Page alignment is handled internally.

The data the buffer held before is dropped, and so is whatever has been read when the reading fails.

This is what the backends reading a table or an entry point out of physical memory fill the buffer they are given with, see `dmi_backend_t`.

Parameters

`buffer`
: Buffer to read the region into.

`path`

Path to the device or file to read (e.g. `/dev/mem`).

`base`

Physical byte offset to start reading from. It is 64-bit on every system, so that the address of an SMBIOS 3.0 table is passed as is; the one the system cannot address is rejected.

`length`

Number of bytes to read; must be greater than zero.

Return value

`true` on success, `false` otherwise.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Path is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`

Length is zero, or the process runs as root and `path` is not a character device, so that root is not tricked into mapping a file it is given.

`DMI_ERROR_FILE_OPEN_FAILED`

Device cannot be opened.

`DMI_ERROR_FILE_STAT_FAILED`

Device cannot be examined.

`DMI_ERROR_FILE_MAP_FAILED`

Region cannot be mapped, lies beyond the end of `path` if it is a regular file, or lies beyond the range the system can address.

`DMI_ERROR_OUT_OF_MEMORY`

Buffer cannot hold the data.

Notes

Errors are raised on the context of the buffer, see `dmi_buffer_context`.

This function is not available on Windows.

See also

`dmi_file_load`, `dmi_buffer_create`, `dmi_open`, `mmap`(2)

<opendmi/utils/base64.h>

dmi_base64_decode

Synopsis

```
#include <opendmi/utils/base64.h>

dmi_data_t *dmi_base64_decode(
        const char *data,
        size_t      data_length,
        size_t     *poutput_length);
```

Description

The `dmi_base64_decode` function decodes a Base64-encoded string into binary data.

The function allocates and returns a buffer containing the decoded binary data corresponding to the given Base64-encoded string. The caller is responsible for freeing the returned buffer.

The string is required to be what `dmi_base64_encode` gives: whole groups of four characters of the alphabet of RFC 4648, the last one padded with `=` for the bytes it is short of. Anything else is invalid, including white space and the characters of the variant of the alphabet for file names (`-` and `_`).

Parameters

`data`
: Pointer to the Base64-encoded string to decode.

`data_length`
: Length of `data` in bytes, excluding any null terminator.

`poutput_length`
: If non-`nullptr`, receives the number of bytes in the returned buffer.

Return value

On success, returns a newly allocated buffer containing the decoded binary data. On failure, returns `nullptr` (e.g., allocation error, invalid Base64 input, or invalid arguments).

Errors

None.

Examples

```
const char *input = "SGVsbG8=";
```

```
size_t output_length;

dmi_data_t *decoded = dmi_base64_decode(input, strlen(input), &output_length);
if (decoded != nullptr) {
    printf("decoded %zu bytes\n", output_length);
    free(decoded);
}
```

See also

`dmi_base64_encode`

dmi_base64_encode

Synopsis

```
#include <opendmi/utils/base64.h>

char *dmi_base64_encode(
        const dmi_data_t *data,
        size_t            data_length,
        size_t           *poutput_length);
```

Description

The `dmi_base64_encode` function encodes binary data into a Base64 string.

The function allocates and returns a null-terminated Base64-encoded string representing the given binary buffer. The caller is responsible for freeing the returned buffer.

Parameters

`data`
Pointer to the binary data to encode.

`data_length`
Number of bytes in `data`.

`poutput_length`
If non-`nullptr`, receives the length of the returned string, excluding the null terminator.

Return value

On success, returns a newly allocated null-terminated Base64 string. On failure, returns `nullptr` (e.g., allocation error or invalid arguments).

Errors

None.

Examples

```
const dmi_data_t input[] = { 0x48, 0x65, 0x6C, 0x6C, 0x6F };
size_t output_length;

char *encoded = dmi_base64_encode(input, sizeof(input), &output_length);
if (encoded != nullptr) {
    printf("encoded: %s (length: %zu)\n", encoded, output_length);
    free(encoded);
}
```

See also

`dmi_base64_decode`

<opendmi/utils/datetime.h>

dmi_date_parse

Synopsis

```
#include <opendmi/utils/datetime.h>

typedef uint32_t dmi_date_t;

#define DMI_DATE(year, month, day) ...
#define DMI_DATE_NONE DMI_DATE(0, 0, 0)

dmi_date_t dmi_date_parse(const char *str);
char *dmi_date_format(dmi_date_t date);
```

Description

A date, `dmi_date_t`, holds the year in its upper 16 bits, the month in the next 8 bits and the day in the lower 8 bits, so that dates compare as integers. The `DMI_DATE` macro and the `dmi_date` function make a date of its parts, and the `dmi_date_year`, `dmi_date_month` and `dmi_date_day` functions take them apart. `DMI_DATE_NONE`, all parts zero, stands for no date.

The `dmi_date_parse` function parses `str` as a date of the form SMBIOS gives dates in, e.g. the release date of the firmware: `MM/DD/YY` or `MM/DD/YYYY`. The month and the day are numbers of any number of digits, within 1 to 12 and 1 to 31; no further check against the calendar is made, e.g. `02/31/2020` is accepted. A year of two digits is taken as years since 1900, so that `05` is 1905, as the SMBIOS specification has it. A string of more than 10 characters, a part which holds anything but digits, e.g. a sign or a blank, and a year of other than two or four digits are rejected.

The `dmi_date_format` function formats `date` as `YYYY-MM-DD`, the year padded with zeros to four digits, and the month and the day to two, e.g. `2025-03-07`. `DMI_DATE_NONE` is formatted as `0000-00-00`. The caller is responsible for freeing the string with `free()` or `dmi_free`.

Parameters

`str`
: String to parse.

`date`
: Date to format.

Return value

The `dmi_date_parse` function returns the date, or `DMI_DATE_NONE` if `str` is `nullptr` or is not a date of the form.

The `dmi_date_format` function returns a newly allocated string, or `nullptr` if memory is exhausted.

Errors

`DMI_ERROR_ARGUMENT_NULL`

Raised by `dmi_date_parse` if `str` is `nullptr`. The functions take no context, so the error reaches no error queue.

Example

```
dmi_date_t date = dmi_date_parse("03/07/2025");

if (date != DMI_DATE_NONE) {
    char *text = dmi_date_format(date);

    if (text != nullptr) {
        puts(text); // "2025-03-07"
        free(text);
    }
}
```

See also

`dmi_free`, `dmi_battery`, `dmi_firmware`

`<opendmi/utils/endian.h>`

dmi_bswap

Synopsis

```
#include <opendmi/utils/endian.h>

uint16_t dmi_bswap16(uint16_t value);
uint32_t dmi_bswap32(uint32_t value);
uint64_t dmi_bswap64(uint64_t value);
```

Description

These functions return a value in which the order of the bytes in their 16-, 32-, or 64-bit arguments is reversed.

Return value

These functions return the value of their argument with the bytes reversed.

Errors

None.

See also

`bswap`(3)

`<opendmi/utils/file.h>`

dmi_file_close

Synopsis

```
#include <opendmi/utils/file.h>

int dmi_file_close(int fd);
```

Description

The `dmi_file_close` function is a cross-platform wrapper around `close`(2).

Parameters

`fd`
: Open file descriptor to close.

Return value

Returns `0` on success, or `-1` on error with `errno` set accordingly.

Errors

None.

See also

`dmi_file_read`, `dmi_file_stat`, `close`(2)

dmi_file_lock

Synopsis

```
#include <opendmi/utils/file.h>

bool dmi_file_lock(int fd, off_t size);
```

Description

The `dmi_file_lock` function locks `size` bytes of the file starting at the current offset using `lockf(F_LOCK)`. The call blocks until the lock is acquired. It retries automatically if interrupted by a signal (`EINTR`).

Passing `size` of `0` locks from the current offset to the end of the file.

Parameters

`fd`
: Open file descriptor.

`size`
: Number of bytes to lock, or `0` to lock to end of file.

Return value

Returns `true` if the lock was acquired, `false` on error.

Errors

None.

See also

`dmi_file_unlock`, `lockf`(3)

dmi_file_read

Synopsis

```
#include <opendmi/utils/file.h>

ssize_t dmi_file_read(int fd, dmi_data_t *data, off_t offset, size_t size);
```

Description

The `dmi_file_read` function performs a full read by looping until all requested bytes are read or end-of-file is reached. It retries automatically if interrupted by a signal (`EINTR`). Fewer than `size` bytes are returned only if EOF is encountered before the read is complete.

Parameters

`fd`
: Open file descriptor to read from.

`data`
: Buffer that receives the data; must be at least `size` bytes.

`offset`
: Starting offset, or `-1` to read from the current position.

`size`
: Number of bytes to read.

Return value

Returns the total number of bytes read (may be less than `size` at EOF), or `-1` on error with `errno` set accordingly.

A missing argument is not an error of the system: `-1` is returned, and `errno` is left as it was.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `data` is `nullptr`. The error reaches no error queue, since the function takes no context to hold it.

See also

`dmi_file_load`, `dmi_file_seek`, `dmi_file_stat`, `dmi_file_close`, `read`(2)

dmi_file_seek

Synopsis

```
#include <opendmi/utils/file.h>

off_t dmi_file_seek(int fd, off_t offset, int whence);
```

Description

The `dmi_file_seek` function is a wrapper around `lseek`(2) that retries automatically if interrupted by a signal (`EINTR`).

Parameters

`fd`
: Open file descriptor.

`offset`
: Byte offset to apply, relative to `whence`.

`whence`
: Offset directive: `SEEK_SET` (beginning of file), `SEEK_CUR` (current position), or `SEEK_END` (end of file).

Return value

Returns the resulting byte offset from the beginning of the file, or `-1` on error with `errno` set accordingly.

Errors

None.

See also

`dmi_file_tell`, `dmi_file_read`, `lseek`(2)

dmi_file_stat

Synopsis

```
#include <opendmi/utils/file.h>

int dmi_file_stat(int fd, dmi_file_stat_t *st);
```

Description

The `dmi_file_stat` function is a cross-platform wrapper around `fstat`(2) that retries automatically if interrupted by a signal (`EINTR`).

Parameters

`fd`
: Open file descriptor to query.

`st`
: Pointer to a `dmi_file_stat_t` structure that receives the file status. On POSIX systems this is a typedef for `struct stat`; on Windows it is `struct _stat`.

Return value

Returns `0` on success, or `-1` on error with `errno` set accordingly.

A missing argument is not an error of the system: `-1` is returned, and `errno` is left as it was.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `st` is `nullptr`. The error reaches no error queue, since the function takes no context to hold it.

See also

`dmi_file_read`, `dmi_file_seek`, `dmi_file_close`, `fstat`(2)

dmi_file_tell

Synopsis

```
#include <opendmi/utils/file.h>

```

```
off_t dmi_file_tell(int fd);
```

Description

The `dmi_file_tell` function returns the current file offset for an open file descriptor. It is equivalent to calling `dmi_file_seek(fd, 0, SEEK_CUR)`.

Parameters

`fd`
: Open file descriptor.

Return value

Returns the current byte offset from the beginning of the file, or `-1` on error with `errno` set accordingly.

Errors

None.

See also

`dmi_file_seek`, `dmi_file_read`, `lseek`(2)

dmi_file_unlock

Synopsis

```
#include <opendmi/utils/file.h>

bool dmi_file_unlock(int fd, off_t size);
```

Description

The `dmi_file_unlock` function unlocks `size` bytes of the file starting at the current offset using `lockf(F_ULOCK)`. It retries automatically if interrupted by a signal (`EINTR`).

Passing `size` of `0` unlocks from the current offset to the end of the file.

Parameters

`fd`
: Open file descriptor.

`size`
: Number of bytes to unlock, or `0` to unlock to end of file.

Return value

Returns `true` if the lock was released, `false` on error.

Errors

None.

See also

`dmi_file_lock`, `lockf`(3)

dmi_file_write

Synopsis

```
#include <opendmi/utils/file.h>

ssize_t dmi_file_write(int fd, const dmi_data_t *data, size_t size);
```

Description

The `dmi_file_write` function performs a full write by looping until all requested bytes are written or a zero-length write is returned. It retries automatically if interrupted by a signal (`EINTR`). Fewer than `size` bytes are returned only if the underlying `write`(2) call returns zero before all bytes are written.

Parameters

`fd`
: Open file descriptor to write to.

`data`
: Buffer containing the data to write; must be at least `size` bytes.

`size`
: Number of bytes to write.

Return value

Returns the total number of bytes written (may be less than `size` if a zero-length write occurs), or `-1` on error with `errno` set accordingly.

A missing argument is not an error of the system: `-1` is returned, and `errno` is left as it was.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: `data` is `nullptr`. The error reaches no error queue, since the function takes no context to hold it.

See also

`dmi_file_read`, `dmi_file_seek`, `dmi_file_close`, `write`(2)

<opendmi/utils/name.h>

DMI_NAMES

Synopsis

```
#include <opendmi/utils/name.h>

DMI_NAMES({ name, ..., {} })
DMI_NAME_RANGES({ range, ..., {} })

DMI_NAME_UNSPEC(id)
DMI_NAME_UNKNOWN(id)
DMI_NAME_OTHER(id)
DMI_NAME_NONE(id)
DMI_NAME_UNSUPPORTED(id)
DMI_NAME_RESERVED(id)

DMI_NAME_FUNCTION(prefix)
```

Description

A name set, `dmi_name_set_t`, names the values of an enumeration or the bits of a set, see `dmi_name_lookup`, and is what the `.values` of an attribute refers to, see `DMI_ATTRIBUTE`.

`DMI_NAMES`

List of the names of single values, a compound literal of `dmi_name_t`, each of an `.id`, a `.code` and a `.name`, terminated by an empty entry, `{}`, which the list itself ends with.

`DMI_NAME_RANGES`

List of the names of ranges of values, a compound literal of `dmi_name_range_t`, each of a `.start_id` and an `.end_id`, inclusive, a `.code` and a `.name`, terminated the same way. A value named on its own is named so rather than by its range.

`DMI_NAME_UNSPEC`, **`DMI_NAME_UNKNOWN`**, **`DMI_NAME_OTHER`**, **`DMI_NAME_NONE`**, **`DMI_NAME_UNSUPPORTED`**, **`DMI_NAME_RESERVED`**

Entries of `DMI_NAMES` for the values the specification names the same way everywhere, e.g. `unspecified` and `Unspecified`.

`DMI_NAME_FUNCTION`

Definition of the function naming the values of an enumeration, which returns the name of a value of `<prefix>_t` in the name set `<prefix>_names`, see `dmi_name_lookup`. The function, `<prefix>_name`, is declared on its own, and the macro is not followed by a semicolon.

The code of the name set, `.code`, names the table its names are translated in, by the codes of the entries, see `dmi_name_lookup`.

Parameters

`id`
: Value the entry names.

`prefix`
: Prefix of the names of the function, the enumeration type and the name set, e.g. `dmi_chassis_type`.

Example

```
const dmi_name_set_t dmi_chassis_type_names =
{
    .code  = "chassis-type",
    .names = DMI_NAMES({
        DMI_NAME_UNSPEC(DMI_CHASSIS_TYPE_UNSPEC),
        DMI_NAME_OTHER(DMI_CHASSIS_TYPE_OTHER),
        DMI_NAME_UNKNOWN(DMI_CHASSIS_TYPE_UNKNOWN),
        {
            .id   = DMI_CHASSIS_TYPE_DESKTOP,
            .code = "desktop",
            .name = "Desktop"
        },
        ...
        {}
    })
};

DMI_NAME_FUNCTION(dmi_chassis_type)
```

See also

`dmi_name_lookup`, `DMI_ATTRIBUTE`

dmi_name_lookup

Synopsis

```
#include <opendmi/utils/name.h>

typedef enum dmi_name_type {
    DMI_NAME_TYPE_NONE,
    DMI_NAME_TYPE_EXACT,
    DMI_NAME_TYPE_RANGE
} dmi_name_type_t;

typedef struct dmi_name {
    int         id;
```

```
    const char *code;
    const char *name;
} dmi_name_t;

typedef struct dmi_name_range {
    int         start_id;
    int         end_id;
    const char *code;
    const char *name;
} dmi_name_range_t;

typedef struct dmi_name_set {
    const char             *code;
    const char             *name;
    const char            **description;
    const dmi_name_t       *names;
    const dmi_name_range_t *ranges;
} dmi_name_set_t;

const char *dmi_name_lookup(const dmi_name_set_t *dict, int id);
const char *dmi_name_lookup_ex(const dmi_name_set_t *dict, int id,
dmi_name_type_t *ptype);

const char *dmi_code_lookup(const dmi_name_set_t *dict, int id);
const char *dmi_code_lookup_ex(const dmi_name_set_t *dict, int id,
dmi_name_type_t *ptype);

int dmi_code_lookup_rev(const dmi_name_set_t *dict, const char *code);
```

Description

A name set, `dmi_name_set_t`, names the values of an enumeration or the bits of a set: every value has a code, a machine-readable identifier which is the same in every language, e.g. `desktop`, and a printable name, e.g. `Desktop`. The name sets of the library are what the `*_name` functions of the structure types, e.g. `dmi_chassis_type_name`, look the values up in.

Values are named one by one by the `names` list, and in ranges by the `ranges` list, e.g. the values reserved for OEMs. Both lists are terminated by an entry whose `code` is `nullptr`, and either of them may be `nullptr`.

The `dmi_name_lookup` function returns the printable name of the value `id`, and the `dmi_code_lookup` function returns its code. An entry of the value itself takes precedence over the ranges it belongs to; of several matching entries, the first one is taken.

Printable names are translated to the locale set by `dmi_set_locale`, if the library has a translation for the entry: the translation is the string keyed by the code of the entry in the table named by the code of the set. Codes are never translated.

The `_ex` variants also tell the kind of the matching entry, since the code and the name of a range are shared by all values of the range, so the value itself is needed to tell them apart.

The `dmi_code_lookup_rev` function does the reverse: it returns the value whose code is `code`, e.g. to parse a value given by its code on the command line. Only the `names` list is searched, since the code of a range is shared by all values of the range and tells no value of its own; of several entries with the code, the first one is taken. Codes are compared exactly, case included.

The strings belong to the name set or to the translation, and must not be freed. Untranslated strings are static; a translated name is valid until the locale is changed by `dmi_set_locale`.

Parameters

`dict`
: Name set to look the value up in.

`id`
: Value to look up.

`code`
: Code to look up.

`ptype`
: Variable which receives the kind of the matching entry, `DMI_NAME_TYPE_EXACT` for an entry of the value itself, `DMI_NAME_TYPE_RANGE` for a range, and `DMI_NAME_TYPE_NONE` if there is no matching entry, or `nullptr`.

Return value

The `dmi_name_lookup`, `dmi_name_lookup_ex`, `dmi_code_lookup` and `dmi_code_lookup_ex` functions return the printable name or the code of the value, or `nullptr` if `dict` is `nullptr`, if `id` is negative, or if no entry matches the value.

The `dmi_code_lookup_rev` function returns the value, or -1 if `dict` or `code` is `nullptr`, if the set has no `names` list, or if no entry of the list has the code. Since values are never negative, -1 is told apart from any of them.

Errors

None: the functions take no context to raise errors in.

See also

`dmi_set_locale`

<opendmi/utils/string.h>

dmi_asprintf

Synopsis

```
#include <opendmi/utils/string.h>

int dmi_asprintf(char **strp, const char *format, ...);
int dmi_vasprintf(char **strp, const char *format, va_list args);
```

Description

The `dmi_asprintf` function formats its arguments according to `format`, as `sprintf`(3) does, into a string it allocates, and stores the pointer to the string in the variable `strp` points to. The `dmi_vasprintf` function does the same with the arguments given as `args`.

The functions behave the same on every platform: they wrap `asprintf`(3) and `vasprintf`(3) where the C library has them, and implement them on Windows, which has none.

The caller is responsible for freeing the string with `free()` or `dmi_free`.

Parameters

`strp`
: Variable to store the pointer to the string in. On failure, it is set to `nullptr`, unless `strp` itself is `nullptr`.

`format`
: Format string, see `printf`(3).

`…`, **`args`**
: Arguments of the format string.

Return value

On success, the functions return the number of bytes written, excluding the terminating null byte. On failure, they return a negative value, e.g. if `strp` or `format` is `nullptr`, if memory is exhausted, or if the string cannot be formatted.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `strp` or `format` is `nullptr`. The functions take no context, so the error reaches no error queue.

Example

```
char *text = nullptr;

if (dmi_asprintf(&text, "%s, version %u", name, version) < 0)
    return false;

puts(text);
free(text);
```

See also

`asprintf`(3), `printf`(3), `dmi_free`

dmi_string_is_placeholder

Synopsis

```
#include <opendmi/utils/string.h>

bool dmi_string_is_placeholder(const char *text);
```

Description

Firmware often leaves a string such as `To Be Filled By O.E.M.` or `Default string` in place of the data it has none of, e.g. the serial number of a board which has not been given one. Such strings are valid, but tell nothing.

The `dmi_string_is_placeholder` function tells whether `text` is one of the placeholders the library knows of. The whole string is compared, regardless of case; a string which merely contains a placeholder, or has blanks around it, is not one.

The placeholders known are:

- `To Be Filled By O.E.M.`, `Filled By OEM`, `Fill By OEM`, `OEM`;
- `Default string`, `Default`;
- `System manufacturer`, `System Product Name`, `System Version`, `System Serial Number`, `System SKU Number`;
- `Chassis Manufacture`, `Chassis Version`, `Chassis Serial Number`;
- `Not Specified`, `Not Applicable`, `Not Available`, `N/A`, `NA`;
- `None`, `Unknown`, `TBD`, `Empty`, `[Empty]`, `NULL`, `INVALID`;
- `No Asset Tag`, `NO DIMM`;
- `0123456789`, `123456789`, `XXXXXXXX`.

The library uses the function to report placeholders, see `dmi_lint`, and to keep them when anonymizing

the data, see `dmi_anonymize`, since they identify nothing.

Parameters

`text`
: String to check.

Return value

Returns `true` if `text` is a placeholder, and `false` otherwise, or if `text` is `nullptr`.

Errors

None: the function takes no context to raise errors in.

See also

`dmi_anonymize`, `dmi_lint`

dmi_string_set

Synopsis

```
#include <opendmi/utils/string.h>

#define DMI_STRING_BYTES_MAX 64

bool dmi_string_set(dmi_context_t *context, char **pstring, const char *value);
bool dmi_string_set_bytes(
        dmi_context_t *context,
        char         **pstring,
        const uint8_t *data,
        size_t         length,
        bool           trim);

const char *dmi_string_from_bytes(
        const uint8_t *data,
        size_t         length,
        char          *buffer,
        bool           trim);
```

Description

The `dmi_string_set` function copies `value`, frees the string held by the variable `pstring` points to, and stores the copy in the variable. If `value` is `nullptr`, the string is freed and the variable is set to `nullptr`.

The copy is made before the string is freed, so that `value` may be the string itself, and a failure leaves the variable as it was.

The caller is responsible for freeing the string with `free()` or `dmi_free` once it is no longer needed.

The `dmi_string_from_bytes` function copies the text the `length` bytes at `data` spell, e.g. a signature a structure holds, into `buffer`, and terminates it. The text is taken only if all the bytes are printable; with `trim`, the spaces are left out.

The `dmi_string_set_bytes` function does the same into a string of its own, which it stores the way `dmi_string_set` does. It is meant for the strings a structure spells in its own bytes, which the structure owns, see `DMI_ATTRIBUTE_FLAG_OWNED` in `DMI_ATTRIBUTE`: a handler sets them, and the library frees them as the structure is released.

Parameters

`context`
Context to raise the error on, or `nullptr` to report none.

`pstring`
Variable holding the string, which is either `nullptr` or allocated with `malloc()`.

`value`
String to copy, or `nullptr`.

`data`
Bytes the text is taken from.

`length`
Number of the bytes, at most `DMI_STRING_BYTES_MAX` for `dmi_string_set_bytes`.

`buffer`
Buffer at least one byte longer than the bytes, which the text is copied into.

`trim`
Whether the spaces are left out.

Return value

The `dmi_string_set` and `dmi_string_set_bytes` functions return `true` on success, and `false` on failure. If the bytes are not all printable, or spell no text, `dmi_string_set_bytes` sets the string to `nullptr`, which is not a failure.

The `dmi_string_from_bytes` function returns the text in `buffer`, or `nullptr` if the bytes are not all printable or spell no text.

The bytes spell no text if there are none of them, or, with `trim`, if they are all spaces.

Errors

`DMI_ERROR_ARGUMENT_NULL`
Raised if `pstring` is `nullptr`, by `dmi_string_set_bytes` if `data` is `nullptr` while `length` is not zero, and by `dmi_string_from_bytes` if `data` or `buffer` is `nullptr`. The function takes no context, so its error reaches no error queue.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised by `dmi_string_set_bytes` if `length` exceeds `DMI_STRING_BYTES_MAX`.

`DMI_ERROR_OUT_OF_MEMORY`
: Raised if memory is exhausted.

Example

```
char *vendor = nullptr;

if (!dmi_string_set(context, &vendor, "Acme"))
    return false;
...
dmi_string_set(context, &vendor, nullptr);
```

See also

`strdup`(3), `dmi_free`, `DMI_ATTRIBUTE`

dmi_string_tolower

Synopsis

```
#include <opendmi/utils/string.h>

void dmi_string_tolower(char *str);
void dmi_string_toupper(char *str);
```

Description

The `dmi_string_tolower` function converts every character of the null-terminated string `str` to lower case with `tolower`(3), and the `dmi_string_toupper` function converts it to upper case with `toupper`(3). The string is changed in place; no memory is allocated.

The string is converted byte by byte, according to the `LC_CTYPE` category of the current locale, so that characters taking more than one byte, e.g. the non-ASCII characters of UTF-8, are not converted.

If `str` is `nullptr`, the functions do nothing.

Parameters

`str`
: String to convert.

Return value

The functions return no value.

Errors

None: the functions take no context to raise errors in.

See also

`tolower`(3), `toupper`(3)

<opendmi/utils/uuid.h>

dmi_uuid_decode

Synopsis

```
#include <opendmi/utils/uuid.h>

dmi_uuid_t dmi_uuid_decode(const dmi_byte_t value[16]);
```

Description

The `dmi_uuid_decode` function converts a raw 16-byte UUID as stored in an SMBIOS structure into a `dmi_uuid_t` value conforming to RFC 4122.

SMBIOS stores the first three UUID fields (`time_low`, `time_mid`, and `time_hi_and_version`) in little-endian byte order. RFC 4122 requires all multi-byte fields to be in big-endian (network) byte order. This function copies the raw SMBIOS value and byte-swaps those three fields so that the returned `dmi_uuid_t` conforms to RFC 4122. The `clock_seq_hi_and_reserved`, `clock_seq_low`, and `node` fields are stored identically in both formats and are not modified.

Parameters

`value`
: Raw 16-byte UUID as stored in an SMBIOS structure (little-endian byte order for multi-byte fields).

Return value

Returns a `dmi_uuid_t` with fields in RFC 4122 (big-endian) byte order.

Errors

None.

Examples

```
const dmi_byte_t raw[16] = { /* raw SMBIOS UUID bytes */ };
dmi_uuid_t uuid = dmi_uuid_decode(raw);

printf("UUID: %08x-%04x-%04x-%02x%02x-",
       uuid.time_low, uuid.time_mid,
       uuid.time_hi_and_version,
       uuid.clock_seq_hi_and_reserved,
       uuid.clock_seq_low);

for (int i = 0; i < 6; i++)
    printf("%02x", uuid.node[i]);
```

```
printf("\n");
```

See also

`dmi_uuid_encode`

dmi_uuid_encode

Synopsis

```
#include <opendmi/utils/uuid.h>

void dmi_uuid_encode(dmi_uuid_t value, uint8_t out[16]);
```

Description

The `dmi_uuid_encode` function performs the inverse of `dmi_uuid_decode`: it converts a `dmi_uuid_t` value in RFC 4122 (big-endian) byte order back into the raw 16-byte format used by SMBIOS structures.

The `time_low`, `time_mid`, and `time_hi_and_version` fields are byte-swapped from big-endian (RFC 4122) to little-endian (SMBIOS) byte order. The `clock_seq_hi_and_reserved`, `clock_seq_low`, and `node` fields are copied without modification.

Parameters

`value`
: UUID in RFC 4122 (big-endian) byte order.

`out`
: Buffer of exactly 16 bytes that receives the UUID in SMBIOS (little-endian) byte order.

Errors

None.

Examples

```
/* Construct a UUID in RFC 4122 representation */
dmi_uuid_t uuid = {
    .time_low                = 0x1624a9df,
    .time_mid                = 0x5e13,
    .time_hi_and_version     = 0x47fc,
    .clock_seq_hi_and_reserved = 0x87,
    .clock_seq_low             = 0x4a,
    .node                    = { 0xdf, 0x3a, 0xff, 0x14, 0x30, 0x89 }
};
```

```
/* Encode into the 16-byte SMBIOS format */
uint8_t out[16];
dmi_uuid_encode(uuid, out);
```

See also

`dmi_uuid_decode`

<opendmi/utils/vector.h>

dmi_vector_initialize

Synopsis

```
#include <opendmi/utils/vector.h>

typedef bool dmi_vector_match_fn(uintptr_t entry, uintptr_t key);

typedef struct dmi_vector {
    uintptr_t           *data;
    size_t               capacity;
    size_t               length;
    dmi_vector_match_fn *matcher;
    dmi_context_t       *context;
} dmi_vector_t;

bool dmi_vector_initialize(dmi_vector_t *vector, dmi_context_t *context,
dmi_vector_match_fn *matcher);
bool dmi_vector_clear(dmi_vector_t *vector);

bool dmi_vector_push(dmi_vector_t *vector, uintptr_t value);
bool dmi_vector_pop(dmi_vector_t *vector, uintptr_t *value);
bool dmi_vector_get(const dmi_vector_t *vector, size_t index, uintptr_t
*value);

bool dmi_vector_find(const dmi_vector_t *vector, uintptr_t key, uintptr_t
*value);
bool dmi_vector_exists(const dmi_vector_t *vector, uintptr_t key);

size_t dmi_vector_length(const dmi_vector_t *vector);
bool dmi_vector_is_empty(const dmi_vector_t *vector);
```

Description

A vector, `dmi_vector_t`, is an array of `uintptr_t` values, typically pointers cast to `uintptr_t`, which grows as values are appended to it. The vector is a variable of the caller, usually a member of a structure; it owns the memory of the array, but not whatever the values point to.

The `data` member points to the array, `length` is the number of values the array holds, and `capacity` the number it can hold before it is grown. The members are read, but are changed only by the functions.

The `dmi_vector_initialize` function sets the vector up empty, with no memory allocated, with `context` as the context its errors are raised against, and with `matcher` as the function `dmi_vector_find` and `dmi_vector_exists` compare the values with keys by. The `dmi_vector_clear` function removes all values and frees the array, and leaves the vector empty, as `dmi_vector_initialize` does, with the

context and the matcher kept. A vector is cleared once it is no longer needed.

The `dmi_vector_push` function appends `value` to the vector, growing the array by 16 values when it is full. The `dmi_vector_pop` function removes the last value and stores it in the variable `value` points to; the array is freed once the last value is removed.

The `dmi_vector_get` function stores the value at `index`, counted from zero, in the variable `value` points to.

The `dmi_vector_find` function calls the matcher with each value, in order, and `key`, and stores the first value it returns `true` for in the variable `value` points to. The `dmi_vector_exists` function tells whether there is such a value.

The `dmi_vector_length` function returns the number of values in the vector, and the `dmi_vector_is_empty` function tells whether there are none. Both take a `nullptr` vector as an empty one.

Parameters

`vector`
: Vector to work on.

`context`
: Context to raise the errors of the vector against, or `nullptr` if there is none, in which case the errors are not recorded.

`matcher`
: Function to compare the values with keys by, or `nullptr` if the vector is not searched.

`value`
: For `dmi_vector_push`, value to append. For `dmi_vector_get`, variable to store the value in. For `dmi_vector_pop` and `dmi_vector_find`, variable to store the value in, or `nullptr` if the value is not needed.

`index`
: Index of the value, counted from zero.

`key`
: Key to pass to the matcher.

`entry`
: Value of the vector the matcher compares with `key`.

Return value

The `dmi_vector_initialize`, `dmi_vector_clear`, `dmi_vector_push` and `dmi_vector_get` functions return `true` on success, and `false` on failure. If `dmi_vector_push` fails, the vector is left as it was.

The `dmi_vector_pop` function returns `true` if a value has been removed, and `false` if the vector is empty, which is not an error, so that a vector can be emptied by popping its values until there is none, or on failure.

The `dmi_vector_find` and `dmi_vector_exists` functions return `true` if a value matches `key`, and

`false` if none does, which is not an error either, or on failure.

Errors

`DMI_ERROR_ARGUMENT_NULL`
: Raised if `vector` is `nullptr`, or, for `dmi_vector_get`, if `value` is `nullptr`.

`DMI_ERROR_ARGUMENT_INVALID`
: Raised by `dmi_vector_get` if `index` is out of range.

`DMI_ERROR_STATE_INVALID`
: Raised by `dmi_vector_find` and `dmi_vector_exists` if the vector has no matcher.

`DMI_ERROR_OUT_OF_MEMORY`
: Raised by `dmi_vector_push` if the array cannot be grown.

The errors are raised against the context of the vector, and reach no error queue if the vector has none, or if `vector` itself is `nullptr`. The one of `dmi_vector_initialize` is raised against `context`.

Example

```
static bool match_name(uintptr_t entry, uintptr_t key)
{
    return strcmp((const char *)entry, (const char *)key) == 0;
}

dmi_vector_t names;

dmi_vector_initialize(&names, context, match_name);

if (!dmi_vector_push(&names, (uintptr_t)"cpu") ||
    !dmi_vector_push(&names, (uintptr_t)"memory")) {
    dmi_vector_clear(&names);
    return false;
}

if (dmi_vector_exists(&names, (uintptr_t)"memory"))
    puts("found");

for (size_t i = 0; i < dmi_vector_length(&names); i++) {
    uintptr_t value;

    if (dmi_vector_get(&names, i, &value))
        puts((const char *)value);
}

dmi_vector_clear(&names);
```

See also

`realloc`(3)

<opendmi/utils/version.h>

dmi_version

Synopsis

```
#include <opendmi/utils/version.h>

typedef uint32_t dmi_version_t;

#define DMI_VERSION(major, minor, revision) ...
#define DMI_VERSION_NONE DMI_VERSION(0, 0, 0)

dmi_version_t dmi_version(unsigned int major, unsigned int minor,
                          unsigned int revision);

unsigned int dmi_version_major(dmi_version_t version);
unsigned int dmi_version_minor(dmi_version_t version);
unsigned int dmi_version_revision(dmi_version_t version);
```

Description

A `dmi_version_t` value is a version number of the SMBIOS specification, or of another specification a structure refers to, e.g. the IPMI specification a management controller conforms to. The major version is held in bits 16 to 23, the minor version in bits 8 to 15, and the revision in bits 0 to 7, so version numbers compare the way the versions do, e.g. `version >= DMI_VERSION(3, 0, 0)`.

The `dmi_version` function and the `DMI_VERSION` macro construct a version number from its components, each of which is cut to its eight low bits. The macro is a constant expression, and is used where one is required, e.g. in the initializers of static variables.

The `dmi_version_major`, `dmi_version_minor` and `dmi_version_revision` functions give the components of a version number back.

`DMI_VERSION_NONE`, which is zero, stands for a version which is unknown or not given, e.g. by a field present in every version of a structure.

These functions are defined in the header rather than in the library, as inline functions in C, and as `constexpr` functions in C++, where they are used in constant expressions too.

Parameters

`major`

Major version number, from 0 to 255.

`minor`

Minor version number, from 0 to 255.

`revision`
: Revision number, from 0 to 255.

`version`
: Version number.

Return value

The `dmi_version` function returns the version number constructed.

The `dmi_version_major`, `dmi_version_minor` and `dmi_version_revision` functions return the component of the version number, from 0 to 255.

Errors

None.

Examples

```
dmi_version_t version = dmi_version(3, 4, 0);

if (version >= DMI_VERSION(3, 0, 0))
    printf("SMBIOS %u.%u\n", dmi_version_major(version),
dmi_version_minor(version));
```

See also

`dmi_version_format`, `dmi_version_format_ex`

dmi_version_format

Synopsis

```
#include <opendmi/utils/version.h>

char *dmi_version_format(dmi_version_t version);
```

Description

The `dmi_version_format` function formats an SMBIOS version number as a human-readable, dynamically allocated string.

If the revision component of `version` is zero, the result uses the "major.minor" form (e.g., "3.4"). Otherwise, the full "major.minor.revision" form is used (e.g., "3.4.1").

The caller is responsible for freeing the returned string.

Parameters

`version`
: SMBIOS version number constructed with `dmi_version` or the `DMI_VERSION` macro.

Return value

On success, returns a newly allocated null-terminated string containing the formatted version. On failure (e.g., allocation error), returns `nullptr`.

Errors

None.

Examples

```
dmi_version_t ver = dmi_version(3, 4, 0);
char *str = dmi_version_format(ver);
if (str != nullptr) {
    printf("SMBIOS version: %s\n", str); // "3.4"
    free(str);
}
```

See also

`dmi_version_format_ex`

dmi_version_format_ex

Synopsis

```
#include <opendmi/utils/version.h>

char *dmi_version_format_ex(
        dmi_version_t       version,
        dmi_version_level_t level);
```

Description

The `dmi_version_format_ex` function formats an SMBIOS version number as a human-readable, dynamically allocated string. The `level` parameter controls how many components are included in the output:

`DMI_VERSION_LEVEL_MAJOR`
: Format as "major" (e.g., "3").

`DMI_VERSION_LEVEL_MINOR`
: Format as "major.minor" (e.g., "3.4").

`DMI_VERSION_LEVEL_REVISION`
: Format as "major.minor.revision" (e.g., "3.4.1").

The caller is responsible for freeing the returned string.

Parameters

`version`
: SMBIOS version number constructed with `dmi_version` or the `DMI_VERSION` macro.

`level`
: Detail level that determines how many version components are included in the formatted string.

Return value

On success, returns a newly allocated null-terminated string containing the formatted version. On failure (e.g., allocation error or invalid `level`), returns `nullptr`.

Errors

`DMI_ERROR_ARGUMENT_INVALID`
: Raised if `level` is not a detail level. The function takes no context, so the error reaches no error queue.

Examples

```
dmi_version_t ver = dmi_version(3, 4, 1);

char *full = dmi_version_format_ex(ver, DMI_VERSION_LEVEL_REVISION);
if (full != nullptr) {
    printf("Full version: %s\n", full); // "3.4.1"
    free(full);
}

char *short_ver = dmi_version_format_ex(ver, DMI_VERSION_LEVEL_MAJOR);
if (short_ver != nullptr) {
    printf("Major only: %s\n", short_ver); // "3"
    free(short_ver);
}
```

See also

`dmi_version_format`

<opendmi/entity/additional-info.h>

dmi_additional_info

Synopsis

```
#include <opendmi/entity/additional-info.h>

typedef struct dmi_additional_info {
    size_t                       entry_count;
    dmi_additional_info_entry_t *entries;
} dmi_additional_info_t;
```

Description

The `dmi_additional_info_t` structure holds the decoded additional information structure (type 40) of SMBIOS. It carries values for fields of other structures which the specification could not express yet: enumerated values which have not been published, or updated contents of a field. Each entry names a structure by its handle and a field by its offset, and gives the value of the field, along with an optional string describing it.

The library applies the entries to the structures they refer to: before one of these structures is decoded, the bytes of the values replace the bytes of the fields they refer to, later entries overriding earlier ones, so that its decoded information already reflects them. An entry which refers to a structure the table does not have, or which reaches past the end of the structure, is reported as an error and left out; in strict mode, it makes the table fail to decode.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(additional_info)`, and lives as long as the context it has been decoded in.

Fields

`entry_count`

Number of elements of `entries`, as the structure declares it.

`entries`

Array of the additional information entries, see `dmi_additional_info_entry`.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(additional_info), false);
dmi_additional_info_t *info     = dmi_entity_info(entity,
                                      DMI_TYPE(additional_info));

if (info != nullptr) {
```

```
    for (size_t i = 0; i < info->entry_count; i++) {
        const dmi_additional_info_entry_t *entry = &info->entries[i];

        printf("Structure 0x%04x, offset 0x%02x: %zu bytes (%s)\n",
               (unsigned)entry->ref_handle, entry->ref_offset,
               entry->value.length,
               entry->string != nullptr ? entry->string : "no string");
    }
}
```

See also

`dmi_entity_info`, `dmi_additional_info_entry`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.6 or newer.

dmi_additional_info_entry

Synopsis

```
#include <opendmi/entity/additional-info.h>

#define DMI_ADDITIONAL_INFO_ENTRY_HEADER 5

typedef struct dmi_additional_info_entry {
    dmi_handle_t  referenced_handle;
    unsigned      referenced_offset;
    const char   *string;
    dmi_binary_t  value;
} dmi_additional_info_entry_t;
```

Description

The `dmi_additional_info_entry_t` structure describes one entry of the additional information structure, and is the type of the elements of the `entries` array of `dmi_additional_info_t`. It gives the value of one field of another structure.

`DMI_ADDITIONAL_INFO_ENTRY_HEADER` is the length, in bytes, of the part of an entry which precedes the value: the entry length, the referenced handle, the referenced offset and the string number.

Fields

`referenced_handle`
: Handle of the structure the entry gives a value for.

`referenced_offset`

Offset of the field within the referenced structure, counted from the first byte of its header. The decoder warns when it falls within the header.

`string`

Optional string describing the value, such as the name of a new processor, `nullptr` if the entry has none.

`value`

Value of the field, referenced in place in the data of the structure: new contents of the field, or an enumerated value which has not been published yet. Its length is the one of the referenced field, that is the length of the entry less `DMI_ADDITIONAL_INFO_ENTRY_HEADER`, cut to what the structure holds.

See also

`dmi_additional_info`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.6 or newer.

<opendmi/entity/baseboard.h>

dmi_baseboard

Synopsis

```
#include <opendmi/entity/baseboard.h>

typedef struct dmi_baseboard {
    const char *vendor;
    const char *product;
    const char *version;
    const char *serial_number;
    const char *asset_tag;
    dmi_baseboard_features_t features;
    const char *location;
    dmi_handle_t chassis_handle;
    dmi_entity_t *chassis;
    dmi_baseboard_type_t type;
    size_t object_count;
    dmi_handle_t *object_handles;
    dmi_entity_t **objects;
} dmi_baseboard_t;
```

Description

The `dmi_baseboard_t` structure holds the decoded baseboard or module information structure (type 2) of SMBIOS: the identity and the properties of a board of the system, such as the motherboard, a server blade or a module plugged into the system.

A table has one such structure per board it describes. When there are several of them, each one lists the structures of the elements mounted on the board. When there is only one and it lists no elements, or when there is none, every element of the table belongs to a single motherboard.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(baseboard)`, and lives as long as the context it has been decoded in.

Fields

`vendor`

Name of the manufacturer of the board, or `nullptr` if the structure does not name one.

`product`

Product name of the board, or `nullptr`.

`version`

Version of the board, or `nullptr`.

`serial_number`

Serial number of the board, or `nullptr`.

`asset_tag`

Asset tag of the board, or `nullptr` if the structure carries none or is too short to hold it.

`features`

Feature flags of the board, see `dmi_baseboard_features`. All the flags are clear if the structure is too short to hold them.

`location`

Location of the board within the chassis it resides in, or `nullptr`.

`chassis_handle`

Handle of the chassis structure (type 3) the board resides in. `DMI_HANDLE_INVALID` if the structure is too short to hold it.

`chassis`

Chassis the board resides in, resolved from `chassis_handle` once the table has been linked. `nullptr` if the handle is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or does not refer to a chassis structure.

`type`

Type of the board, see `dmi_baseboard_type`. `DMI_BASEBOARD_TYPE_UNSPEC` if the structure is too short to hold it.

`object_count`

Number of the elements of `object_handles` and of `objects`, which is the number of the handles of the contained elements the structure holds in full. Zero if the structure lists no elements, or is too short to hold the list.

`object_handles`

Handles of the structures describing the elements the board contains, such as other boards, processors, ports, system slots or memory devices, or `nullptr` if there are none.

`objects`

Structures `object_handles` refer to, in the same order, resolved once the table has been linked. An element is `nullptr` if its handle does not refer to a structure of the table. The array is `nullptr` if there are no elements.

Examples

```
dmi_registry_t  *registry = dmi_get_registry(context);
dmi_entity_t    *entity   = dmi_registry_lookup_first(registry,
                                                      DMI_TYPE(baseboard),
                                                      false);
dmi_baseboard_t *board    = dmi_entity_info(entity, DMI_TYPE(baseboard));

if ((board != nullptr) && (board->product != nullptr))
    printf("%s, %zu contained elements\n", board->product,
```

```
            board->object_count);
```

See also

`dmi_entity_info`, `dmi_baseboard_features`, `dmi_baseboard_type`, `dmi_chassis`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_baseboard_features

Synopsis

```
#include <opendmi/entity/baseboard.h>

typedef union dmi_baseboard_features {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_hosting_board       : 1;
        dmi_byte_t require_daughter_board : 1;
        dmi_byte_t is_removable           : 1;
        dmi_byte_t is_replaceable         : 1;
        dmi_byte_t is_hot_swappable       : 1;
        dmi_byte_t __reserved             : 3;
    };
} dmi_baseboard_features_t;
```

Description

The `dmi_baseboard_features_t` union holds the feature flags of a board, which tell how the board is built into the system and whether it can be taken out. It is the type of the `features` field of `dmi_baseboard_t`.

The union is packed into a single byte: the flags can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the flags, as the byte the structure carries them in, `is_hosting_board` being bit 0.

`is_hosting_board`
: Set if the board hosts the other components of the system, the way a motherboard does.

`require_daughter_board`
: Set if the board does not work without at least one daughter board or auxiliary card.

`is_removable`
: Set if the board can be taken out of the chassis and put back without harming the function of the chassis.

`is_replaceable`
: Set if the board can be exchanged for a physically different one, as a repair or as an upgrade. A replaceable board is removable too.

`is_hot_swappable`
: Set if the board can be exchanged for a physically different but equivalent one while it is powered. A hot-swappable board is replaceable and removable too.

`__reserved`
: Reserved bits, zero.

See also

`dmi_baseboard`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_baseboard_type

Synopsis

```
#include <opendmi/entity/baseboard.h>

typedef enum dmi_baseboard_type {
    DMI_BASEBOARD_TYPE_UNSPEC                   = 0x00,
    DMI_BASEBOARD_TYPE_UNKNOWN                  = 0x01,
    DMI_BASEBOARD_TYPE_OTHER                    = 0x02,
    DMI_BASEBOARD_TYPE_SERVER_BLADE             = 0x03,
    DMI_BASEBOARD_TYPE_CONNECTIVITY_SWITCH      = 0x04,
    DMI_BASEBOARD_TYPE_SYSTEM_MANAGEMENT_MODULE = 0x05,
    DMI_BASEBOARD_TYPE_PROCESSOR_MODULE         = 0x06,
    DMI_BASEBOARD_TYPE_IO_MODULE                = 0x07,
    DMI_BASEBOARD_TYPE_MEMORY_MODULE            = 0x08,
    DMI_BASEBOARD_TYPE_DAUGHTERBOARD            = 0x09,
    DMI_BASEBOARD_TYPE_MOTHERBOARD              = 0x0A,
    DMI_BASEBOARD_TYPE_PROCESSOR_MEMORY_MODULE  = 0x0B,
    DMI_BASEBOARD_TYPE_PROCESSOR_IO_MODULE      = 0x0C,
    DMI_BASEBOARD_TYPE_INTERCONNECT_BOARD       = 0x0D,
} dmi_baseboard_type_t;

const char *dmi_baseboard_type_name(dmi_baseboard_type_t value);
```

Description

The `dmi_baseboard_type_t` enumeration tells what kind of board a baseboard structure describes, and is the type of the `type` field of `dmi_baseboard_t`. The same values name the boards a chassis contains, in the `board_type` field of `dmi_chassis_element_t`.

The `dmi_baseboard_type_name` function returns the name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Values

`DMI_BASEBOARD_TYPE_UNSPEC`

Unspecified. The specification defines no board type of zero; the value is also what the field holds when the structure is too short to carry it.

`DMI_BASEBOARD_TYPE_UNKNOWN`

Unknown.

`DMI_BASEBOARD_TYPE_OTHER`

Other type, which none of the values below names.

`DMI_BASEBOARD_TYPE_SERVER_BLADE`

Server blade.

`DMI_BASEBOARD_TYPE_CONNECTIVITY_SWITCH`

Connectivity switch.

`DMI_BASEBOARD_TYPE_SYSTEM_MANAGEMENT_MODULE`

System management module.

`DMI_BASEBOARD_TYPE_PROCESSOR_MODULE`

Processor module.

`DMI_BASEBOARD_TYPE_IO_MODULE`

I/O module.

`DMI_BASEBOARD_TYPE_MEMORY_MODULE`

Memory module.

`DMI_BASEBOARD_TYPE_DAUGHTERBOARD`

Daughter board.

`DMI_BASEBOARD_TYPE_MOTHERBOARD`

Motherboard, which holds the processors, the memory and the I/O.

`DMI_BASEBOARD_TYPE_PROCESSOR_MEMORY_MODULE`

Module holding processors and memory.

`DMI_BASEBOARD_TYPE_PROCESSOR_IO_MODULE`

Module holding processors and I/O.

`DMI_BASEBOARD_TYPE_INTERCONNECT_BOARD`
: Interconnect board.

Return value

`dmi_baseboard_type_name` returns the name of the value, or `nullptr` if the value has no name, e.g. a value the specification reserves. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None.

See also

`dmi_baseboard`, `dmi_chassis_element`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/battery.h>

dmi_battery

Synopsis

```
#include <opendmi/entity/battery.h>

typedef struct dmi_battery {
    const char *location;
    const char *vendor;
    dmi_date_t manufacture_date;
    const char *manufacture_date_string;
    const char *serial_number;
    const char *name;
    dmi_battery_chemistry_t chemistry;
    unsigned int capacity;
    unsigned short design_capacity;
    unsigned short voltage;
    const char *sbds_version;
    unsigned short maximum_error;
    uint16_t sbds_serial_number;
    dmi_date_t sbds_manufacture_date;
    const char *sbds_chemistry;
    uint8_t design_capacity_multiplier;
    uint32_t oem_defined;
} dmi_battery_t;
```

Description

The `dmi_battery_t` structure holds a decoded portable battery structure (type 22) of SMBIOS: the static attributes of a battery pack of a portable system, such as its maker, chemistry, capacity and voltage. There is one structure for each battery pack.

Smart batteries, which follow the Smart Battery Data Specification (SBDS), may leave the manufacture date, serial number and chemistry of the structure empty, and give them in the `sbds_` fields instead. The decoder combines both for the manufacture date and the capacity.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(portable_battery)`, and lives as long as the context it has been decoded in.

Fields

`location`

Where the battery is, for example "in the back, on the left-hand side", or `nullptr` if not given.

`vendor`

Name of the manufacturer of the battery, or `nullptr` if not given.

`manufacture_date`

Date the battery was made on: the date `manufacture_date_string` spells, or `sbds_manufacture_date` when the string is missing or cannot be parsed. `DMI_DATE_NONE` if neither gives a date.

`manufacture_date_string`

Manufacture date as the structure spells it, expected in the `MM/DD/YY` or `MM/DD/YYYY` format, or `nullptr` if not given, as smart batteries do.

`serial_number`

Serial number of the battery, or `nullptr` if not given, as smart batteries do (see `sbds_serial_number`).

`name`

Name of the battery device, for example "DR-36", or `nullptr` if not given.

`chemistry`

Chemistry of the battery. Smart batteries set it to `DMI_BATTERY_CHEMISTRY_UNKNOWN`, and give it in `sbds_chemistry`.

`capacity`

Design capacity of the battery, in milliwatt-hours: `design_capacity` times `design_capacity_multiplier`. 0 if unknown.

`design_capacity`

Design capacity as the structure carries it, in units of `design_capacity_multiplier` milliwatt-hours. 0 if unknown.

`voltage`

Design voltage of the battery, in millivolts, or 0 if unknown.

`sbds_version`

Version of the Smart Battery Data Specification the battery supports, or `nullptr` if it supports none.

`maximum_error`

Upper bound of the error of the energy the battery reports, as a percentage from 0 to 100, or `USHRT_MAX` if unknown.

`sbds_serial_number`

Serial number of a smart battery, which identifies it together with the manufacturer, the device name and the manufacture date. It is meaningful only when `serial_number` is `nullptr`. Present since SMBIOS 2.2; 0 in older structures.

`sbds_manufacture_date`

Manufacture date of a smart battery, decoded from the packed SBDS format. It is meaningful only when `manufacture_date_string` is `nullptr`. `DMI_DATE_NONE` if the structure carries none. Present since SMBIOS 2.2.

`sbds_chemistry`

Chemistry of a smart battery, for example "PbAc", or `nullptr` if not given. It is meaningful only when `chemistry` is `DMI_BATTERY_CHEMISTRY_UNKNOWN`. Present since SMBIOS 2.2.

`design_capacity_multiplier`

Factor the design capacity is multiplied by, which smart battery implementations set to 10. Present since SMBIOS 2.2; 1 in older structures.

`oem_defined`

Information specific to the OEM or to the firmware vendor. Present since SMBIOS 2.2; 0 in older structures.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   =
        dmi_registry_lookup_first(registry, DMI_TYPE(portable_battery),
                                  true);
dmi_battery_t  *battery  =
        dmi_entity_info(entity, DMI_TYPE(portable_battery));

if (battery != nullptr) {
    const char *chemistry = battery->sbds_chemistry;

    if (battery->chemistry != DMI_BATTERY_CHEMISTRY_UNKNOWN)
        chemistry = dmi_battery_chemistry_name(battery->chemistry);

    printf("%s: %u mWh, %s\n", battery->name ? battery->name : "Battery",
           battery->capacity, chemistry ? chemistry : "unknown");
}
```

See also

`dmi_entity_info`, `dmi_battery_chemistry`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_battery_chemistry

Synopsis

```
#include <opendmi/entity/battery.h>

typedef enum dmi_battery_chemistry {
    DMI_BATTERY_CHEMISTRY_UNSPEC    = 0x00,
```

```
    DMI_BATTERY_CHEMISTRY_OTHER     = 0x01,
    DMI_BATTERY_CHEMISTRY_UNKNOWN   = 0x02,
    DMI_BATTERY_CHEMISTRY_LEAD_ACID = 0x03,
    DMI_BATTERY_CHEMISTRY_NI_CD     = 0x04,
    DMI_BATTERY_CHEMISTRY_NI_MH     = 0x05,
    DMI_BATTERY_CHEMISTRY_LI_ION    = 0x06,
    DMI_BATTERY_CHEMISTRY_ZN_AIR    = 0x07,
    DMI_BATTERY_CHEMISTRY_LI_PO     = 0x08,
} dmi_battery_chemistry_t;

const char *dmi_battery_chemistry_name(dmi_battery_chemistry_t value);
```

Description

The `dmi_battery_chemistry_t` enumeration tells the chemistry of a battery, and is the type of the `chemistry` field of `dmi_battery_t`. Smart batteries report `DMI_BATTERY_CHEMISTRY_UNKNOWN`, and give their chemistry as a string in the `sbds_chemistry` field instead.

The `dmi_battery_chemistry_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_BATTERY_CHEMISTRY_UNSPEC`
: Unspecified.

`DMI_BATTERY_CHEMISTRY_OTHER`
: Other chemistry.

`DMI_BATTERY_CHEMISTRY_UNKNOWN`
: Unknown chemistry, or chemistry given by the `sbds_chemistry` field.

`DMI_BATTERY_CHEMISTRY_LEAD_ACID`
: Lead acid.

`DMI_BATTERY_CHEMISTRY_NI_CD`
: Nickel cadmium.

`DMI_BATTERY_CHEMISTRY_NI_MH`
: Nickel metal hydride.

`DMI_BATTERY_CHEMISTRY_LI_ION`
: Lithium-ion.

`DMI_BATTERY_CHEMISTRY_ZN_AIR`
: Zinc air.

`DMI_BATTERY_CHEMISTRY_LI_PO`
: Lithium polymer.

Return value

`dmi_battery_chemistry_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_battery`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/bis-entry-point.h>

dmi_bis_entry_point

Synopsis

```
#include <opendmi/entity/bis-entry-point.h>

typedef struct dmi_bis_entry_point {
    uint8_t                    checksum;
    dmi_bis_real_mode_addr_t entry_point_16;
    uint32_t                   entry_point_32;
    bool                       is_valid;
} dmi_bis_entry_point_t;
```

Description

The `dmi_bis_entry_point_t` structure holds the decoded Boot Integrity Services (BIS) entry point structure (type 31) of SMBIOS: the addresses at which the Boot Integrity Services can be called. SMBIOS reserves the structure type for BIS, and the Boot Integrity Services API specification defines its content.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(bis_entry_point)`, and lives as long as the context it has been decoded in.

Fields

`checksum`
: Checksum of the structure, which makes the sum of all its bytes zero.

`entry_point_16`
: Entry point for the 16-bit real mode, see `dmi_bis_real_mode_addr`.

`entry_point_32`
: Entry point for the 32-bit protected mode, a physical address.

`is_valid`
: `true` if the bytes of the structure sum to zero, that is, if the checksum is valid.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(
        registry, DMI_TYPE(bis_entry_point), true);
dmi_bis_entry_point_t *bis      = dmi_entity_info(entity,
                                                  DMI_TYPE(bis_entry_point));

if ((bis != nullptr) && bis->is_valid)
```

```
    printf("BIS entry point: %04X:%04X\n",
           (unsigned)bis->entry_point_16.segment,
           (unsigned)bis->entry_point_16.offset);
```

See also

`dmi_entity_info`, `dmi_bis_real_mode_addr`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_bis_real_mode_addr

Synopsis

```
#include <opendmi/entity/bis-entry-point.h>

typedef struct dmi_bis_real_mode_addr {
    uint16_t segment;
    uint16_t offset;
} dmi_bis_real_mode_addr_t;
```

Description

The `dmi_bis_real_mode_addr_t` structure holds a real mode (16:16) address, made of a segment and an offset: the linear address is `segment` * 16 + `offset`. It is the type of the `entry_point_16` field of `dmi_bis_entry_point_t`.

Fields

`segment`
: Segment of the address.

`offset`
: Offset of the address within the segment.

See also

`dmi_bis_entry_point`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/cache.h>

dmi_cache

Synopsis

```
#include <opendmi/entity/cache.h>

typedef struct dmi_cache {
    const char *socket_designator;
    unsigned short level;
    bool socketed;
    dmi_cache_location_t location;
    bool enabled;
    dmi_cache_mode_t mode;
    dmi_size_t maximum_size;
    dmi_size_t installed_size;
    dmi_cache_sram_type_t supported_sram;
    dmi_cache_sram_type_t current_sram;
    unsigned int speed;
    dmi_error_correct_type_t error_correction;
    dmi_cache_type_t type;
    dmi_cache_assoc_t associativity;
} dmi_cache_t;
```

Description

The `dmi_cache_t` structure holds the decoded cache information structure (type 7) of SMBIOS: the level, location, operational mode, sizes, SRAM types, speed and associativity of one processor cache. A table carries one such structure for each cache, whether the cache is internal or external to the processor module. Processor information structures (type 4) refer to the caches of their processor by handle.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(cache)`, and lives as long as the context it has been decoded in.

Fields

`socket_designator`

Reference designation of the cache socket, for example "CACHE1", or `nullptr` if the structure has none.

`level`

Cache level, 1 to 8 (1 for an L1 cache). The data counts the levels from zero, and the decoder adds one.

`socketed`

`true` if the cache sits in a socket (for example, cache on a stick), `false` otherwise.

`location`

Location of the cache, relative to the processor module, see `dmi_cache_location`.

`enabled`

`true` if the cache is enabled at boot time.

`mode`

Operational mode of the cache, see `dmi_cache_mode`.

`maximum_size`

Largest size of cache the socket can take, in bytes. The data carries it in granules of 1 KiB or 64 KiB. Since SMBIOS 3.1, a cache of 2 GiB or more sets the 16-bit field to 0xFFFF and carries its size in a 32-bit extended field, which the decoder then uses.

`installed_size`

Size of the installed cache, in bytes, or 0 if no cache is installed. Carried the same way as `maximum_size`.

`supported_sram`

Set of the SRAM types the cache supports, see `dmi_cache_sram_type`.

`current_sram`

Set of the SRAM types the cache currently uses, see `dmi_cache_sram_type`.

`speed`

Speed of the cache module, in nanoseconds, or 0 if unknown. Present since SMBIOS 2.1; 0 in older structures.

`error_correction`

Error-correction scheme the cache supports, see `dmi_error_correct_type`. Present since SMBIOS 2.1; `DMI_ERROR_CORRECT_TYPE_UNSPEC` in older structures.

`type`

Logical type of the cache: instruction, data or unified, see `dmi_cache_type`. Present since SMBIOS 2.1; `DMI_CACHE_TYPE_UNSPEC` in older structures.

`associativity`

Associativity of the cache, see `dmi_cache_assoc`. Present since SMBIOS 2.1; `DMI_CACHE_ASSOC_UNSPEC` in older structures.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                     DMI_TYPE(cache), false);
dmi_cache_t    *cache    = dmi_entity_info(entity, DMI_TYPE(cache));

if (cache != nullptr)
    printf("L%u cache: %s, %llu KiB\n", cache->level,
           dmi_cache_type_name(cache->type),
```

```
            (unsigned long long)(cache->installed_size >> 10));
```

See also

`dmi_entity_info`, `dmi_cache_type`, `dmi_cache_mode`, `dmi_cache_assoc`, `dmi_cache_location`, `dmi_cache_sram_type`, `dmi_error_correct_type`, `dmi_processor`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_cache_assoc

Synopsis

```
#include <opendmi/entity/cache.h>

typedef enum dmi_cache_assoc {
    DMI_CACHE_ASSOC_UNSPEC  = 0x00,
    DMI_CACHE_ASSOC_OTHER   = 0x01,
    DMI_CACHE_ASSOC_UNKNOWN = 0x02,
    DMI_CACHE_ASSOC_DIRECT  = 0x03,
    DMI_CACHE_ASSOC_2WAY    = 0x04,
    DMI_CACHE_ASSOC_4WAY    = 0x05,
    DMI_CACHE_ASSOC_FULL    = 0x06,
    DMI_CACHE_ASSOC_8WAY    = 0x07,
    DMI_CACHE_ASSOC_16WAY   = 0x08,
    DMI_CACHE_ASSOC_12WAY   = 0x09,
    DMI_CACHE_ASSOC_24WAY   = 0x0A,
    DMI_CACHE_ASSOC_32WAY   = 0x0B,
    DMI_CACHE_ASSOC_48WAY   = 0x0C,
    DMI_CACHE_ASSOC_64WAY   = 0x0D,
    DMI_CACHE_ASSOC_20WAY   = 0x0E,
} dmi_cache_assoc_t;

const char *dmi_cache_assoc_name(dmi_cache_assoc_t value);
```

Description

The `dmi_cache_assoc_t` enumeration tells how the memory addresses map to the lines of a cache, and is the type of the `associativity` field of `dmi_cache_t`. The values are not in the order of the number of ways: the specification has added them over time.

The `dmi_cache_assoc_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CACHE_ASSOC_UNSPEC`
: Unspecified: the structure predates SMBIOS 2.1, or leaves the field at zero.

`DMI_CACHE_ASSOC_OTHER`
: Other associativity.

`DMI_CACHE_ASSOC_UNKNOWN`
: Unknown associativity.

`DMI_CACHE_ASSOC_DIRECT`
: Direct-mapped: each memory address has one place in the cache.

`DMI_CACHE_ASSOC_2WAY`
: 2-way set-associative.

`DMI_CACHE_ASSOC_4WAY`
: 4-way set-associative.

`DMI_CACHE_ASSOC_FULL`
: Fully associative: a memory address may go anywhere in the cache.

`DMI_CACHE_ASSOC_8WAY`
: 8-way set-associative.

`DMI_CACHE_ASSOC_16WAY`
: 16-way set-associative.

`DMI_CACHE_ASSOC_12WAY`
: 12-way set-associative.

`DMI_CACHE_ASSOC_24WAY`
: 24-way set-associative.

`DMI_CACHE_ASSOC_32WAY`
: 32-way set-associative.

`DMI_CACHE_ASSOC_48WAY`
: 48-way set-associative.

`DMI_CACHE_ASSOC_64WAY`
: 64-way set-associative.

`DMI_CACHE_ASSOC_20WAY`
: 20-way set-associative.

Return value

`dmi_cache_assoc_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cache`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_cache_location

Synopsis

```
#include <opendmi/entity/cache.h>

typedef enum dmi_cache_location {
    DMI_CACHE_LOCATION_INTERNAL = 0x00,
    DMI_CACHE_LOCATION_EXTERNAL = 0x01,
    DMI_CACHE_LOCATION_RESERVED = 0x02,
    DMI_CACHE_LOCATION_UNKNOWN  = 0x03,
} dmi_cache_location_t;

const char *dmi_cache_location_name(dmi_cache_location_t value);
```

Description

The `dmi_cache_location_t` enumeration tells where a cache is, relative to the processor module, and is the type of the `location` field of `dmi_cache_t`.

The `dmi_cache_location_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CACHE_LOCATION_INTERNAL`
: Internal to the processor module.

`DMI_CACHE_LOCATION_EXTERNAL`
: External to the processor module.

`DMI_CACHE_LOCATION_RESERVED`
: Reserved by the specification.

`DMI_CACHE_LOCATION_UNKNOWN`
: Unknown location.

Return value

`dmi_cache_location_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cache`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_cache_mode

Synopsis

```
#include <opendmi/entity/cache.h>

typedef enum dmi_cache_mode {
    DMI_CACHE_MODE_WRITE_THROUGH = 0x00,
    DMI_CACHE_MODE_WRITE_BACK    = 0x01,
    DMI_CACHE_MODE_VARIABLE      = 0x02,
    DMI_CACHE_MODE_UNKNOWN       = 0x03,
} dmi_cache_mode_t;

const char *dmi_cache_mode_name(dmi_cache_mode_t value);
```

Description

The `dmi_cache_mode_t` enumeration tells how a cache handles writes, and is the type of the `mode` field of `dmi_cache_t`.

The `dmi_cache_mode_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CACHE_MODE_WRITE_THROUGH`
: Write-through: writes go to memory at once.

`DMI_CACHE_MODE_WRITE_BACK`
: Write-back: writes go to memory when the cache line is evicted.

`DMI_CACHE_MODE_VARIABLE`
: The mode varies with the memory address.

`DMI_CACHE_MODE_UNKNOWN`
: Unknown mode.

Return value

`dmi_cache_mode_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cache`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_cache_sram_type

Synopsis

```
#include <opendmi/entity/cache.h>

typedef union dmi_cache_sram_type {
    dmi_word_t __value;

    struct {
        dmi_word_t other          : 1;
        dmi_word_t unknown        : 1;
        dmi_word_t non_burst      : 1;
        dmi_word_t burst          : 1;
        dmi_word_t pipeline_burst : 1;
        dmi_word_t synchronous    : 1;
        dmi_word_t asynchronous   : 1;
    };
} dmi_cache_sram_type_t;
```

Description

The `dmi_cache_sram_type_t` union holds a set of the SRAM types of a cache, one bit per type, and is the type of the `supported_sram` and `current_sram` fields of `dmi_cache_t`. A bit is 1 if the type is in the set. The union gives access to the whole word as well as to the single bits.

Fields

`__value`
: Whole 16-bit word, as the data carries it. Bits 7 to 15 are reserved, and are zero.

`other`
: Other SRAM type.

`unknown`
: Unknown SRAM type.

`non_burst`
: Non-burst SRAM.

`burst`
: Burst SRAM.

`pipeline_burst`
: Pipeline burst SRAM.

`synchronous`
: Synchronous SRAM.

`asynchronous`
: Asynchronous SRAM.

See also

`dmi_cache`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_cache_type

Synopsis

```
#include <opendmi/entity/cache.h>

typedef enum dmi_cache_type {
    DMI_CACHE_TYPE_UNSPEC      = 0x00,
    DMI_CACHE_TYPE_OTHER       = 0x01,
    DMI_CACHE_TYPE_UNKNOWN     = 0x02,
    DMI_CACHE_TYPE_INSTRUCTION = 0x03,
    DMI_CACHE_TYPE_DATA        = 0x04,
    DMI_CACHE_TYPE_UNIFIED     = 0x05,
} dmi_cache_type_t;
```

```
const char *dmi_cache_type_name(dmi_cache_type_t value);
```

Description

The `dmi_cache_type_t` enumeration tells what a cache holds: instructions, data, or both, and is the type of the `type` field of `dmi_cache_t`. A level of cache is described as unified when its caches hold a mix of instructions and data.

The `dmi_cache_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CACHE_TYPE_UNSPEC`

Unspecified: the structure predates SMBIOS 2.1, or leaves the field at zero.

`DMI_CACHE_TYPE_OTHER`

Other type.

`DMI_CACHE_TYPE_UNKNOWN`

Unknown type.

`DMI_CACHE_TYPE_INSTRUCTION`

Instruction cache.

`DMI_CACHE_TYPE_DATA`

Data cache.

`DMI_CACHE_TYPE_UNIFIED`

Unified cache, holding both instructions and data.

Return value

`dmi_cache_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cache`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/chassis.h>

dmi_chassis

Synopsis

```
#include <opendmi/entity/chassis.h>

typedef struct dmi_chassis {
    const char *vendor;
    dmi_chassis_type_t type;
    bool is_lock_present;
    const char *version;
    const char *serial_number;
    const char *asset_tag;
    dmi_status_t bootup_state;
    dmi_status_t power_supply_state;
    dmi_status_t thermal_state;
    dmi_chassis_security_status_t security_status;
    uint32_t oem_defined;
    unsigned short height;
    unsigned short power_cord_count;
    size_t element_count;
    size_t element_size;
    dmi_chassis_element_t *elements;
    const char *sku_number;
    dmi_rack_type_t rack_type;
    unsigned short rack_height;
} dmi_chassis_t;
```

Description

The `dmi_chassis_t` structure holds a decoded system enclosure or chassis structure (type 3) of SMBIOS: the mechanical enclosure of the system, its identification, its state at the last boot, its size and the elements it contains. A system with several enclosures, such as a separate enclosure for its peripherals, has one structure for each of them. Tables of SMBIOS 2.3 and newer are required to have at least one.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(chassis)`, and lives as long as the context it has been decoded in.

Fields

`vendor`

Name of the manufacturer of the enclosure, or `nullptr` if not given.

`type`

Type of the enclosure.

`is_lock_present`

Whether the enclosure has a lock. `false` when it has none, or when it is not known whether it has one.

`version`

Version of the enclosure, or `nullptr` if not given.

`serial_number`

Serial number of the enclosure, or `nullptr` if not given. Systems sharing a multi-system chassis report the same serial number.

`asset_tag`

Asset tag of the enclosure, or `nullptr` if not given.

`bootup_state`

State of the enclosure at the last boot. Present since SMBIOS 2.1; `DMI_STATUS_UNSPEC` in older structures.

`power_supply_state`

State of the power supplies of the enclosure at the last boot. Present since SMBIOS 2.1; `DMI_STATUS_UNSPEC` in older structures.

`thermal_state`

Thermal state of the enclosure at the last boot. Present since SMBIOS 2.1; `DMI_STATUS_UNSPEC` in older structures.

`security_status`

Physical security status of the enclosure at the last boot. Present since SMBIOS 2.1; `DMI_CHASSIS_SECURITY_STATUS_UNSPEC` in older structures.

`oem_defined`

Information specific to the OEM or to the firmware vendor. Present since SMBIOS 2.3; 0 in older structures.

`height`

Height of the enclosure, in rack units (U, 1.75 inches or 4.445 cm). 0 if unspecified; 255 if the height is given by `rack_height` instead. Present since SMBIOS 2.3; 0 in older structures.

`power_cord_count`

Number of power cords of the enclosure, or 0 if unspecified. Present since SMBIOS 2.3; 0 in older structures.

`element_count`

Number of items of `elements`. Present since SMBIOS 2.3; 0 in older structures.

`element_size`

Size of each contained element record of the structure, in bytes, which may be larger than the part the library decodes. Present since SMBIOS 2.3; 0 in older structures.

`elements`

Elements contained in the enclosure, such as power supplies or blades, an array of `element_count` items, or `nullptr` if there is none. Present since SMBIOS 2.3.

`sku_number`

SKU number of the enclosure, or `nullptr` if not given. Present since SMBIOS 2.7.

`rack_type`

Type of the rack, which gives the unit of `rack_height`. Present since SMBIOS 3.9; `DMI_RACK_TYPE_UNSPEC` in older structures.

`rack_height`

Height of the enclosure, in the unit of the rack type, for example in OU (48 mm) for Open Rack. Present since SMBIOS 3.9; 0 in older structures.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   =
        dmi_registry_lookup_first(registry, DMI_TYPE(chassis), false);
dmi_chassis_t  *chassis  = dmi_entity_info(entity, DMI_TYPE(chassis));

if (chassis != nullptr) {
    printf("%s (%s)\n", dmi_chassis_type_name(chassis->type),
           chassis->vendor ? chassis->vendor : "unknown vendor");

    for (size_t i = 0; i < chassis->element_count; i++) {
        const dmi_chassis_element_t *element = &chassis->elements[i];

        if (element->type != DMI_TYPE_ID_INVALID)
            printf("  structures of type %d: %zu to %zu\n",
                   (int)element->type, element->minimum_count,
                   element->maximum_count);
    }
}
```

See also

`dmi_entity_info`, `dmi_chassis_type`, `dmi_chassis_security_status`, `dmi_chassis_element`, `dmi_rack_type`, `dmi_status`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_chassis_element

Synopsis

```
#include <opendmi/entity/chassis.h>
```

```
typedef struct dmi_chassis_element {
    dmi_type_id_t type;
    dmi_baseboard_type_t board_type;
    size_t minimum_count;
    size_t maximum_count;
} dmi_chassis_element_t;
```

Description

The `dmi_chassis_element_t` structure describes a kind of element a system enclosure contains, and how many of them it takes. The `elements` field of `dmi_chassis_t` is an array of them.

An element is named either by the type of the SMBIOS structures describing it, for example `DMI_TYPE(power_supply)` for power supplies, or by a baseboard type, for example `DMI_BASEBOARD_TYPE_SERVER_BLADE` for blades. One of `type` and `board_type` is set, and the other one holds the value telling that it is not used.

Fields

`type`

Type number of the SMBIOS structures describing the element, or `DMI_TYPE_ID_INVALID` if the element is named by `board_type`.

`board_type`

Baseboard type of the element, or `DMI_BASEBOARD_TYPE_UNSPEC` if the element is named by `type`.

`minimum_count`

Number of such elements the enclosure needs to work properly, 0 to 254, or `SIZE_MAX` if the data carries 255, a value the specification reserves.

`maximum_count`

Number of such elements the enclosure can hold, 1 to 255, or `SIZE_MAX` if the data carries 0, a value the specification reserves, which the library takes as no maximum.

See also

`dmi_chassis`, `dmi_baseboard_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_chassis_security_status

Synopsis

```
#include <opendmi/entity/chassis.h>

typedef enum dmi_chassis_security_status {
```

```
    DMI_CHASSIS_SECURITY_STATUS_UNSPEC         = 0x00,
    DMI_CHASSIS_SECURITY_STATUS_OTHER          = 0x01,
    DMI_CHASSIS_SECURITY_STATUS_UNKNOWN        = 0x02,
    DMI_CHASSIS_SECURITY_STATUS_NONE           = 0x03,
    DMI_CHASSIS_SECURITY_STATUS_EXT_IF_LOCKED  = 0x04,
    DMI_CHASSIS_SECURITY_STATUS_EXT_IF_ENABLED = 0x05,
} dmi_chassis_security_status_t;

const char *dmi_chassis_security_status_name(
        dmi_chassis_security_status_t value);
```

Description

The `dmi_chassis_security_status_t` enumeration tells the physical security status of a system enclosure at the last boot, and is the type of the `security_status` field of `dmi_chassis_t`.

The `dmi_chassis_security_status_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CHASSIS_SECURITY_STATUS_UNSPEC`
: Unspecified.

`DMI_CHASSIS_SECURITY_STATUS_OTHER`
: Other status.

`DMI_CHASSIS_SECURITY_STATUS_UNKNOWN`
: Unknown status.

`DMI_CHASSIS_SECURITY_STATUS_NONE`
: No physical security.

`DMI_CHASSIS_SECURITY_STATUS_EXT_IF_LOCKED`
: External interfaces are locked out.

`DMI_CHASSIS_SECURITY_STATUS_EXT_IF_ENABLED`
: External interfaces are enabled.

Return value

`dmi_chassis_security_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_chassis`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_chassis_type

Synopsis

```
#include <opendmi/entity/chassis.h>

typedef enum dmi_chassis_type {
    DMI_CHASSIS_TYPE_UNSPEC              = 0x00,
    DMI_CHASSIS_TYPE_OTHER               = 0x01,
    DMI_CHASSIS_TYPE_UNKNOWN             = 0x02,
    DMI_CHASSIS_TYPE_DESKTOP             = 0x03,
    DMI_CHASSIS_TYPE_LOW_PROFILE_DESKTOP = 0x04,
    DMI_CHASSIS_TYPE_PIZZA_BOX           = 0x05,
    DMI_CHASSIS_TYPE_MINI_TOWER          = 0x06,
    DMI_CHASSIS_TYPE_TOWER               = 0x07,
    DMI_CHASSIS_TYPE_PORTABLE            = 0x08,
    DMI_CHASSIS_TYPE_LAPTOP              = 0x09,
    DMI_CHASSIS_TYPE_NOTEBOOK            = 0x0A,
    DMI_CHASSIS_TYPE_HAND_HELD           = 0x0B,
    DMI_CHASSIS_TYPE_DOCKING_STATION     = 0x0C,
    DMI_CHASSIS_TYPE_ALL_IN_ONE          = 0x0D,
    DMI_CHASSIS_TYPE_SUB_NOTEBOOK        = 0x0E,
    DMI_CHASSIS_TYPE_SPACE_SAVING        = 0x0F,
    DMI_CHASSIS_TYPE_LUNCH_BOX           = 0x10,
    DMI_CHASSIS_TYPE_MAIN_SERVER         = 0x11,
    DMI_CHASSIS_TYPE_EXPANSION           = 0x12,
    DMI_CHASSIS_TYPE_SUB_CHASSIS         = 0x13,
    DMI_CHASSIS_TYPE_BUS_EXPANSION       = 0x14,
    DMI_CHASSIS_TYPE_PERIPHERAL          = 0x15,
    DMI_CHASSIS_TYPE_RAID                = 0x16,
    DMI_CHASSIS_TYPE_RACK_MOUNT          = 0x17,
    DMI_CHASSIS_TYPE_SEALED_CASE_PC      = 0x18,
    DMI_CHASSIS_TYPE_MULTI_SYSTEM        = 0x19,
    DMI_CHASSIS_TYPE_COMPACT_PCI         = 0x1A,
    DMI_CHASSIS_TYPE_ADVANCED_TCA        = 0x1B,
    DMI_CHASSIS_TYPE_BLADE               = 0x1C,
    DMI_CHASSIS_TYPE_BLADE_ENCLOSURE     = 0x1D,
    DMI_CHASSIS_TYPE_TABLET              = 0x1E,
    DMI_CHASSIS_TYPE_CONVERTIBLE         = 0x1F,
    DMI_CHASSIS_TYPE_DETACHABLE          = 0x20,
```

```
    DMI_CHASSIS_TYPE_IOT_GATEWAY         = 0x21,
    DMI_CHASSIS_TYPE_EMBEDDED_PC         = 0x22,
    DMI_CHASSIS_TYPE_MINI_PC             = 0x23,
    DMI_CHASSIS_TYPE_STICK_PC            = 0x24,
} dmi_chassis_type_t;

const char *dmi_chassis_type_name(dmi_chassis_type_t value);
```

Description

The `dmi_chassis_type_t` enumeration tells the kind of a system enclosure, and is the type of the `type` field of `dmi_chassis_t`.

The `dmi_chassis_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_CHASSIS_TYPE_UNSPEC`
Unspecified.

`DMI_CHASSIS_TYPE_OTHER`
Other type.

`DMI_CHASSIS_TYPE_UNKNOWN`
Unknown type.

`DMI_CHASSIS_TYPE_DESKTOP`
Desktop.

`DMI_CHASSIS_TYPE_LOW_PROFILE_DESKTOP`
Low-profile desktop.

`DMI_CHASSIS_TYPE_PIZZA_BOX`
Pizza box.

`DMI_CHASSIS_TYPE_MINI_TOWER`
Mini tower.

`DMI_CHASSIS_TYPE_TOWER`
Tower.

`DMI_CHASSIS_TYPE_PORTABLE`
Portable.

`DMI_CHASSIS_TYPE_LAPTOP`
Laptop.

`DMI_CHASSIS_TYPE_NOTEBOOK`
: Notebook.

`DMI_CHASSIS_TYPE_HAND_HELD`
: Hand-held device.

`DMI_CHASSIS_TYPE_DOCKING_STATION`
: Docking station.

`DMI_CHASSIS_TYPE_ALL_IN_ONE`
: All-in-one computer.

`DMI_CHASSIS_TYPE_SUB_NOTEBOOK`
: Sub-notebook.

`DMI_CHASSIS_TYPE_SPACE_SAVING`
: Space-saving computer.

`DMI_CHASSIS_TYPE_LUNCH_BOX`
: Lunch box.

`DMI_CHASSIS_TYPE_MAIN_SERVER`
: Main server chassis.

`DMI_CHASSIS_TYPE_EXPANSION`
: Expansion chassis.

`DMI_CHASSIS_TYPE_SUB_CHASSIS`
: Sub-chassis.

`DMI_CHASSIS_TYPE_BUS_EXPANSION`
: Bus expansion chassis.

`DMI_CHASSIS_TYPE_PERIPHERAL`
: Peripheral chassis.

`DMI_CHASSIS_TYPE_RAID`
: RAID chassis.

`DMI_CHASSIS_TYPE_RACK_MOUNT`
: Rack-mount chassis.

`DMI_CHASSIS_TYPE_SEALED_CASE_PC`
: Sealed-case PC.

`DMI_CHASSIS_TYPE_MULTI_SYSTEM`
: Chassis able to hold several independently reporting systems, whether or not it is configured so at the moment. The systems it holds report the same serial number.

`DMI_CHASSIS_TYPE_COMPACT_PCI`
: CompactPCI chassis.

`DMI_CHASSIS_TYPE_ADVANCED_TCA`

AdvancedTCA chassis.

`DMI_CHASSIS_TYPE_BLADE`

Blade. A blade system has a structure for the blade and another one for the blade enclosure.

`DMI_CHASSIS_TYPE_BLADE_ENCLOSURE`

Blade enclosure, which holds blades and provides them with power, cooling, networking and other shared infrastructure.

`DMI_CHASSIS_TYPE_TABLET`

Tablet.

`DMI_CHASSIS_TYPE_CONVERTIBLE`

Convertible.

`DMI_CHASSIS_TYPE_DETACHABLE`

Detachable.

`DMI_CHASSIS_TYPE_IOT_GATEWAY`

IoT gateway.

`DMI_CHASSIS_TYPE_EMBEDDED_PC`

Embedded PC.

`DMI_CHASSIS_TYPE_MINI_PC`

Mini PC.

`DMI_CHASSIS_TYPE_STICK_PC`

Stick PC.

Return value

`dmi_chassis_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_chassis`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_rack_type

Synopsis

```
#include <opendmi/entity/chassis.h>

typedef enum dmi_rack_type {
    DMI_RACK_TYPE_UNSPEC = 0x00,
    DMI_RACK_TYPE_OPEN   = 0x01,
} dmi_rack_type_t;

const char *dmi_rack_type_name(dmi_rack_type_t value);
```

Description

The `dmi_rack_type_t` enumeration tells the kind of rack a system enclosure is made for, which gives the unit of its rack height. It is the type of the `rack_type` field of `dmi_chassis_t`, whose `rack_height` field holds the height in that unit.

The `dmi_rack_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_RACK_TYPE_UNSPEC`
: Unspecified.

`DMI_RACK_TYPE_OPEN`
: Open Rack, whose height unit is the OU (48 mm).

Return value

`dmi_rack_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_chassis`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.9 or newer.

<opendmi/entity/common.h>

dmi_error_correct_type

Synopsis

```
#include <opendmi/entity/common.h>

typedef enum dmi_error_correct_type {
    DMI_ERROR_CORRECT_TYPE_UNSPEC     = 0x00,
    DMI_ERROR_CORRECT_TYPE_OTHER      = 0x01,
    DMI_ERROR_CORRECT_TYPE_UNKNOWN    = 0x02,
    DMI_ERROR_CORRECT_TYPE_NONE       = 0x03,
    DMI_ERROR_CORRECT_TYPE_PARITY     = 0x04,
    DMI_ERROR_CORRECT_TYPE_SINGLE_BIT = 0x05,
    DMI_ERROR_CORRECT_TYPE_MULTI_BIT  = 0x06,
    DMI_ERROR_CORRECT_TYPE_CRC        = 0x07,
} dmi_error_correct_type_t;

const char *dmi_error_correct_type_name(dmi_error_correct_type_t value);
```

Description

The `dmi_error_correct_type_t` enumeration tells how errors of a memory are detected or corrected. It is shared by several structures: it is the type of the `error_correction` fields of `dmi_cache_t` and `dmi_memory_array_t`.

The `dmi_error_correct_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_ERROR_CORRECT_TYPE_UNSPEC`

Unspecified: the data carries no error correction type.

`DMI_ERROR_CORRECT_TYPE_OTHER`

Other error correction.

`DMI_ERROR_CORRECT_TYPE_UNKNOWN`

Unknown error correction.

`DMI_ERROR_CORRECT_TYPE_NONE`

No error detection or correction.

`DMI_ERROR_CORRECT_TYPE_PARITY`

Parity checking.

`DMI_ERROR_CORRECT_TYPE_SINGLE_BIT`
: Error-correcting code correcting single-bit errors.

`DMI_ERROR_CORRECT_TYPE_MULTI_BIT`
: Error-correcting code correcting multi-bit errors.

`DMI_ERROR_CORRECT_TYPE_CRC`
: Cyclic redundancy check.

Return value

`dmi_error_correct_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cache`, `dmi_memory_array`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_pci_addr

Synopsis

```
#include <opendmi/entity/common.h>

typedef struct dmi_pci_addr {
    uint16_t segment_group;
    uint8_t bus_number;
    uint8_t device_number;
    uint8_t function_number;
} dmi_pci_addr_t;
```

Description

The `dmi_pci_addr_t` structure holds the address of a PCI, AGP, PCI-X or PCI Express function: its segment group, bus, device and function numbers. It is shared by several structures: it is the type of the `base_address` field of `dmi_slot_t`, of the `address` field of the peer groups of slots (`dmi_slot_peer_group_t`), and of the `address` field of `dmi_onboard_device_ex_t`.

SMBIOS packs the device and function numbers into one byte; the structure holds them apart. Devices which have no such address carry the largest value in each field.

Fields

`segment_group`
: Segment group, as the PCI firmware specification defines it. It is 0 for a single-segment topology, and `UINT16_MAX` if not applicable.

`bus_number`
: Bus number, or `UINT8_MAX` if not applicable.

`device_number`
: Device number, 0 to 31, or `UINT8_MAX` if the bus number is not applicable.

`function_number`
: Function number, 0 to 7, or `UINT8_MAX` if the bus number is not applicable.

See also

`dmi_slot`, `dmi_slot_peer_group`, `dmi_onboard_device_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.6 or newer.

dmi_pci_class

Synopsis

```
#include <opendmi/entity/common.h>

typedef uint8_t dmi_pci_class_t;
```

Description

The `dmi_pci_class_t` type holds the class code of a PCI function, which tells the generic function of the device (for example 0x01 for mass storage controllers, 0x02 for network controllers).

The type is shared by several structures: it is the type of the `pci_class` field of `dmi_intel_rsd_pcie_t`, the `pci_class` field of `dmi_intel_rsd_network_card_t`, and the `pci_class` field of `dmi_intel_rsd_storage_device_t`.

See also

`dmi_intel_rsd_pcie`, `dmi_intel_rsd_network_card`, `dmi_intel_rsd_storage_device`, `dmi_pci_addr`

Standards

Not part of the SMBIOS specification. Used by the structures defined by Intel Rack Scale Design firmware.

dmi_pci_device_id

Synopsis

```
#include <opendmi/entity/common.h>

typedef uint16_t dmi_pci_device_id_t;
```

Description

The `dmi_pci_device_id_t` type holds a PCI device identifier, which the vendor assigns to its devices, as in the device and subsystem identifier registers of a function.

The type is shared by several structures: it is the type of the `device_id` and `sub_device_id` fields of `dmi_intel_rsd_pcie_t`, the `device_id` and `sub_device_id` fields of `dmi_intel_rsd_network_card_t`, and the `device_id` and `sub_device_id` fields of `dmi_intel_rsd_storage_device_t`.

See also

`dmi_intel_rsd_pcie`, `dmi_intel_rsd_network_card`, `dmi_intel_rsd_storage_device`, `dmi_pci_addr`

Standards

Not part of the SMBIOS specification. Used by the structures defined by Intel Rack Scale Design firmware.

dmi_pci_slot

Synopsis

```
#include <opendmi/entity/common.h>

typedef uint16_t dmi_pci_slot_t;
```

Description

The `dmi_pci_slot_t` type holds the identifier of a PCI slot, which matches the slot identifier of the system slot structures (type 9) describing it.

The type is shared by several structures: it is the type of the `pci_slot_id` field of `dmi_intel_rsd_pcie_t`, the `pci_slot_id` field of `dmi_intel_rsd_network_card_t`, the `pci_slot_id` field of `dmi_intel_rsd_fpga_t`, and the `pci_slot_id` field of `dmi_intel_rsd_cabled_pcie_t`.

See also

`dmi_intel_rsd_pcie`, `dmi_intel_rsd_network_card`, `dmi_intel_rsd_fpga`, `dmi_intel_rsd_cabled_pcie`, `dmi_pci_addr`

Standards

Not part of the SMBIOS specification. Used by the structures defined by Intel Rack Scale Design firmware.

dmi_pci_vendor_id

Synopsis

```
#include <opendmi/entity/common.h>

typedef uint16_t dmi_pci_vendor_id_t;
```

Description

The `dmi_pci_vendor_id_t` type holds a PCI vendor identifier, which the PCI-SIG assigns to the vendors of PCI devices, as in the vendor and subsystem vendor identifier registers of a function.

The type is shared by several structures: it is the type of the `vendor_id` and `sub_vendor_id` fields of `dmi_intel_rsd_pcie_t`, the `vendor_id` and `sub_vendor_id` fields of `dmi_intel_rsd_network_card_t`, the `vendor_id` and `sub_vendor_id` fields of `dmi_intel_rsd_storage_device_t`, and the `vendor_id` and `subsystem_vendor_id` fields of `dmi_cisco_pci_adapter_t`.

See also

`dmi_cisco_pci_adapter`, `dmi_intel_rsd_pcie`, `dmi_intel_rsd_network_card`, `dmi_intel_rsd_storage_device`, `dmi_pci_addr`

Standards

Not part of the SMBIOS specification. Used by the structures defined by Intel Rack Scale Design firmware and by the firmware of Cisco UCS servers.

dmi_status

Synopsis

```
#include <opendmi/entity/common.h>

typedef enum dmi_status {
    DMI_STATUS_UNSPEC          = 0x00,
    DMI_STATUS_OTHER           = 0x01,
```

```
    DMI_STATUS_UNKNOWN         = 0x02,
    DMI_STATUS_OK              = 0x03,
    DMI_STATUS_NON_CRITICAL    = 0x04,
    DMI_STATUS_CRITICAL        = 0x05,
    DMI_STATUS_NON_RECOVERABLE = 0x06,
} dmi_status_t;

const char *dmi_status_name(dmi_status_t value);
```

Description

The `dmi_status_t` enumeration tells the health of a hardware component, as the firmware saw it. It is shared by several structures: it is the type of the `bootup_state`, `power_supply_state` and `thermal_state` fields of `dmi_chassis_t`, and of the `status` fields of `dmi_probe_t` (voltage, temperature and electrical current probes), `dmi_cooling_device_t` and `dmi_power_supply_t`.

The structures describing a system enclosure call the `OK` and `Non-critical` values `Safe` and `Warning` respectively; the values are the same.

The `dmi_status_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_STATUS_UNSPEC`
: Unspecified: the data carries no status.

`DMI_STATUS_OTHER`
: Other status.

`DMI_STATUS_UNKNOWN`
: Unknown status.

`DMI_STATUS_OK`
: The component works normally (safe).

`DMI_STATUS_NON_CRITICAL`
: The component works, but something needs attention (warning).

`DMI_STATUS_CRITICAL`
: The component is in a critical state.

`DMI_STATUS_NON_RECOVERABLE`
: The component has failed and cannot recover.

Return value

`dmi_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_chassis`, `dmi_probe`, `dmi_cooling_device`, `dmi_power_supply`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/cooling-device.h>

dmi_cooling_device

Synopsis

```
#include <opendmi/entity/cooling-device.h>

typedef struct dmi_cooling_device {
    dmi_handle_t probe_handle;
    dmi_entity_t *probe;
    dmi_cooling_device_type_t type;
    dmi_status_t status;
    unsigned short group;
    uint32_t oem_defined;
    short nominal_speed;
    const char *description;
} dmi_cooling_device_t;
```

Description

The `dmi_cooling_device_t` structure holds the decoded cooling device structure (type 27) of SMBIOS: the type, status, redundancy group and nominal speed of one cooling device of the system, such as a fan or a heat pipe, and the temperature probe monitoring it. A table carries one such structure for each cooling device.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(cooling_device)`, and lives as long as the context it has been decoded in.

Fields

`probe_handle`

Handle of the temperature probe (type 28) monitoring this cooling device. `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED` mean that no probe is set.

`probe`

Temperature probe monitoring this cooling device, or `nullptr` if no probe is set, if the table has no structure with the handle, or if that structure is not a temperature probe. It is set when the structure is linked.

`type`

Type of the cooling device, see `dmi_cooling_device_type`.

`status`

Status of the cooling device, see `dmi_status`.

`group`

Cooling unit group of the device, or 0 if the device belongs to no redundant cooling unit. Devices with

the same non-zero group form a redundant unit, so a table has at least two of them.

`oem_defined`

Information specific to the OEM or to the firmware vendor.

`nominal_speed`

Nominal rotational speed of the device, in revolutions per minute, or `SHRT_MIN` if the speed is unknown, if the device does not rotate, or if the structure ends before this field (its length is 0x0C).

`description`

Additional description of the device or of its location, or `nullptr` if the structure has none. Present since SMBIOS 2.7.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(cooling_device), false);
dmi_cooling_device_t *device   = dmi_entity_info(entity,
                                      DMI_TYPE(cooling_device));

if (device != nullptr) {
    printf("type: %s\n", dmi_cooling_device_type_name(device->type));
    printf("status: %s\n", dmi_status_name(device->status));
    printf("description: %s\n",
           device->description ? device->description : "unspecified");

    if (device->nominal_speed != SHRT_MIN)
        printf("speed: %d rpm\n", device->nominal_speed);
}
```

See also

`dmi_entity_info`, `dmi_cooling_device_type`, `dmi_status`, `dmi_probe`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

dmi_cooling_device_type

Synopsis

```
#include <opendmi/entity/cooling-device.h>

typedef enum dmi_cooling_device_type {
    DMI_COOLING_DEVICE_TYPE_UNSPEC                  = 0x00,
```

```
    DMI_COOLING_DEVICE_TYPE_OTHER                    = 0x01,
    DMI_COOLING_DEVICE_TYPE_UNKNOWN                  = 0x02,
    DMI_COOLING_DEVICE_TYPE_FAN                      = 0x03,
    DMI_COOLING_DEVICE_TYPE_CENTRIFUGAL_BLOWER       = 0x04,
    DMI_COOLING_DEVICE_TYPE_CHIP_FAN                 = 0x05,
    DMI_COOLING_DEVICE_TYPE_CABINET_FAN              = 0x06,
    DMI_COOLING_DEVICE_TYPE_POWER_SUPPLY_FAN         = 0x07,
    DMI_COOLING_DEVICE_TYPE_HEAT_PIPE                = 0x08,
    DMI_COOLING_DEVICE_TYPE_INTEGRATED_REFRIGERATION = 0x09,
    DMI_COOLING_DEVICE_TYPE_ACTIVE_COOLING           = 0x10,
    DMI_COOLING_DEVICE_TYPE_PASSIVE_COOLING          = 0x11,
} dmi_cooling_device_type_t;

const char *dmi_cooling_device_type_name(dmi_cooling_device_type_t value);
```

Description

The `dmi_cooling_device_type_t` enumeration tells the kind of a cooling device, and is the type of the `type` field of `dmi_cooling_device_t`. The data carries it in 5 bits; values 0x0A to 0x0F and above 0x11 are not assigned by the specification.

The `dmi_cooling_device_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_COOLING_DEVICE_TYPE_UNSPEC`
: Unspecified.

`DMI_COOLING_DEVICE_TYPE_OTHER`
: Other type.

`DMI_COOLING_DEVICE_TYPE_UNKNOWN`
: Unknown type.

`DMI_COOLING_DEVICE_TYPE_FAN`
: Fan.

`DMI_COOLING_DEVICE_TYPE_CENTRIFUGAL_BLOWER`
: Centrifugal blower.

`DMI_COOLING_DEVICE_TYPE_CHIP_FAN`
: Chip fan.

`DMI_COOLING_DEVICE_TYPE_CABINET_FAN`
: Cabinet fan.

`DMI_COOLING_DEVICE_TYPE_POWER_SUPPLY_FAN`
: Power supply fan.

`DMI_COOLING_DEVICE_TYPE_HEAT_PIPE`
: Heat pipe.

`DMI_COOLING_DEVICE_TYPE_INTEGRATED_REFRIGERATION`
: Integrated refrigeration.

`DMI_COOLING_DEVICE_TYPE_ACTIVE_COOLING`
: Active cooling.

`DMI_COOLING_DEVICE_TYPE_PASSIVE_COOLING`
: Passive cooling.

Return value

`dmi_cooling_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_cooling_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/firmware-inventory.h>

dmi_firmware_ident

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef struct dmi_firmware_ident {
    dmi_firmware_ident_format_t format;
    dmi_uuid_t guid;
} dmi_firmware_ident_t;
```

Description

The `dmi_firmware_ident_t` structure holds the identifier string of a firmware parsed according to the identifier format of its firmware inventory information structure. It is the type of the `parsed_ident` field of `dmi_firmware_inventory_t`.

The decoder parses the string only if it conforms to the format the structure declares. The string itself stays available in the `ident` field of `dmi_firmware_inventory_t` in any case.

Fields

`format`

Format the string has been parsed in: `DMI_FIRMWARE_IDENT_FORMAT_GUID` if `guid` holds the identifier, `DMI_FIRMWARE_IDENT_FORMAT_FREE` if the string is absent, is free-form, uses another format, or is not a well-formed GUID.

`guid`

Identifier, if `format` is `DMI_FIRMWARE_IDENT_FORMAT_GUID`. Its bytes follow the order of the digits in the string.

See also

`dmi_firmware_inventory`, `dmi_firmware_ident_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_ident_format

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>
```

```
typedef enum dmi_firmware_ident_format {
    DMI_FIRMWARE_IDENT_FORMAT_FREE = 0x00,
    DMI_FIRMWARE_IDENT_FORMAT_GUID = 0x01,
} dmi_firmware_ident_format_t;

const char *dmi_firmware_ident_format_name(dmi_firmware_ident_format_t value);
```

Description

The `dmi_firmware_ident_format_t` enumeration tells how the identifier of a firmware is written, and is the type of the `ident_format` field of `dmi_firmware_inventory_t` and of the `format` field of `dmi_firmware_ident_t`. Values 0x02 to 0x7F are reserved by the specification, and values 0x80 to 0xFF are for firmware vendors and OEMs; the enumeration has no constants for them.

The `dmi_firmware_ident_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_FIRMWARE_IDENT_FORMAT_FREE`

Free-form string, whose meaning depends on the implementation, for example "35EQP72B".

`DMI_FIRMWARE_IDENT_FORMAT_GUID`

GUID in the 36-character form of RFC 4122, for example "1624a9df-5e13-47fc-874a-df3aff143089": the firmware class of the UEFI ESRT, or the image type of the UEFI Firmware Management Protocol.

Return value

`dmi_firmware_ident_format_name` returns the name of the value, or `nullptr` if the value has no name. Values 0x02 to 0x7F are named "Reserved", and values 0x80 to 0xFF "Vendor/OEM-specific". The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_firmware_inventory`, `dmi_firmware_ident`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_inventory

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef struct dmi_firmware_inventory {
    const char *name;
    const char *version;
    dmi_firmware_version_format_t version_format;
    const char *ident;
    dmi_firmware_ident_format_t ident_format;
    const char *release_date;
    const char *vendor;
    const char *lowest_version;
    dmi_size_t image_size;
    dmi_firmware_version_t parsed_version;
    dmi_firmware_version_t parsed_lowest_version;
    dmi_firmware_ident_t parsed_ident;
    dmi_firmware_inventory_chars_t characteristics;
    dmi_firmware_inventory_state_t state;
    size_t component_count;
    dmi_firmware_inventory_component_t *components;
} dmi_firmware_inventory_t;
```

Description

The `dmi_firmware_inventory_t` structure holds the decoded firmware inventory information structure (type 45) of SMBIOS: the name, version, identifier, vendor, release date, image size and state of one firmware component of the system, such as the platform firmware, the firmware of a BMC or the firmware of another device, together with the structures of the devices the firmware belongs to. A table carries one such structure for each firmware component.

The version strings and the identifier string are written in the formats the structure declares. The decoder parses them according to these formats into `parsed_version`, `parsed_lowest_version` and `parsed_ident`, and keeps the strings as they are.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(firmware_inventory)`, and lives as long as the context it has been decoded in.

Fields

`name`

Name of the firmware component, for example "BMC Firmware", or `nullptr` if the structure has none.

`version`

Version of the firmware, written in the format `version_format` gives, or `nullptr` if the structure has none.

`version_format`

Format of `version` and `lowest_version`, see `dmi_firmware_version_format`.

`ident`

Identifier of the firmware, written in the format `ident_format` gives, or `nullptr` if the structure has none.

`ident_format`

Format of `ident`, see `dmi_firmware_ident_format`.

`release_date`

Release date of the firmware, as a date-time string in the format of DSP0266 (for example "2021-05-15T04:14:33+06:00", with a time of "T00:00:00Z" when the time is unknown), or `nullptr` if the structure has none.

`vendor`

Manufacturer or producer of the firmware, or `nullptr` if the structure has none.

`lowest_version`

Lowest version the firmware can be rolled back to, written in the format `version_format` gives, or `nullptr` if the structure has none.

`image_size`

Size of the firmware image currently programmed in the device, in bytes, or `DMI_SIZE_MAX` if unknown.

`parsed_version`

`version` parsed according to `version_format`, see `dmi_firmware_version`. Its format is `DMI_FIRMWARE_VERSION_FORMAT_FREE` if the string is absent or does not conform to the format.

`parsed_lowest_version`

`lowest_version` parsed according to `version_format`, see `dmi_firmware_version`.

`parsed_ident`

`ident` parsed according to `ident_format`, see `dmi_firmware_ident`.

`characteristics`

Characteristics of the firmware, see `dmi_firmware_inventory_chars`.

`state`

State of the firmware, see `dmi_firmware_inventory_state`.

`component_count`

Number of the elements of `components`: the number of associated components the structure declares, less the ones the data does not hold completely. 0 if the structure ends before the count.

`components`

Array of `component_count` components associated with the firmware, or `nullptr` if there are none, see `dmi_firmware_inventory_component`. These are the structures of the devices the firmware belongs to, such as system slots (type 9), memory devices (type 17), onboard devices (type 41) or TPM devices (type 43).

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                         DMI_TYPE(firmware_inventory), false);
dmi_firmware_inventory_t *firmware = dmi_entity_info(entity,
                                         DMI_TYPE(firmware_inventory));

if (firmware != nullptr) {
    printf("%s: %s\n", firmware->name ? firmware->name : "?",
           firmware->version ? firmware->version : "?");

    for (size_t i = 0; i < firmware->component_count; i++)
        printf("  component 0x%04X\n", firmware->components[i].handle);
}
```

See also

`dmi_entity_info`, `dmi_firmware_ident_format`, `dmi_firmware_inventory_state`, `dmi_firmware_inventory_chars`, `dmi_firmware_inventory_component`, `dmi_firmware_version`, `dmi_firmware_version_format`, `dmi_firmware_ident`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_inventory_component

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef struct dmi_firmware_inventory_component {
    dmi_handle_t handle;
    dmi_entity_t *entity;
} dmi_firmware_inventory_component_t;
```

Description

The `dmi_firmware_inventory_component_t` structure identifies one structure associated with a firmware inventory information structure, that is one device the firmware belongs to. The `components` array of `dmi_firmware_inventory_t` holds these structures.

Fields

`handle`

Handle of the associated structure.

`entity`

Associated structure, of any type, or `nullptr` if `handle` is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or if the table has no structure with this handle.

See also

`dmi_firmware_inventory`, `dmi_entity_info`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_inventory_chars

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef union dmi_firmware_inventory_chars {
    dmi_word_t __value;

    struct {
        dmi_word_t is_updatable       : 1;
        dmi_word_t is_write_protected : 1;
        dmi_word_t __reserved         : 14;
    };
} dmi_firmware_inventory_chars_t;
```

Description

The `dmi_firmware_inventory_chars_t` union holds the characteristics of a firmware component, one bit per characteristic, and is the type of the `characteristics` field of `dmi_firmware_inventory_t`. A bit is 1 if the firmware has the characteristic. The union gives access to the whole word as well as to the single bits.

Fields

`__value`

Whole 16-bit word, as the data carries it.

`is_updatable`

Software can update the firmware.

`is_write_protected`

The firmware is write-protected.

`__reserved`

Bits 2 to 15, reserved by the specification.

See also

`dmi_firmware_inventory`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_inventory_state

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef enum dmi_firmware_inventory_state {
    DMI_FIRMWARE_INVENTORY_STATE_UNSPEC          = 0x00,
    DMI_FIRMWARE_INVENTORY_STATE_OTHER           = 0x01,
    DMI_FIRMWARE_INVENTORY_STATE_UNKNOWN         = 0x02,
    DMI_FIRMWARE_INVENTORY_STATE_DISABLED        = 0x03,
    DMI_FIRMWARE_INVENTORY_STATE_ENABLED         = 0x04,
    DMI_FIRMWARE_INVENTORY_STATE_ABSENT          = 0x05,
    DMI_FIRMWARE_INVENTORY_STATE_STANDBY_OFFLINE = 0x06,
    DMI_FIRMWARE_INVENTORY_STATE_STANDBY_SPARE   = 0x07,
    DMI_FIRMWARE_INVENTORY_STATE_UNAVAIL_OFFLINE = 0x08,
} dmi_firmware_inventory_state_t;

const char *dmi_firmware_inventory_state_name(
        dmi_firmware_inventory_state_t value);
```

Description

The `dmi_firmware_inventory_state_t` enumeration tells the state of a firmware component, and is the type of the `state` field of `dmi_firmware_inventory_t`. The values follow the `Status.State` property of Redfish.

The `dmi_firmware_inventory_state_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_FIRMWARE_INVENTORY_STATE_UNSPEC`

Unspecified.

`DMI_FIRMWARE_INVENTORY_STATE_OTHER`

Other state.

`DMI_FIRMWARE_INVENTORY_STATE_UNKNOWN`
: Unknown state.

`DMI_FIRMWARE_INVENTORY_STATE_DISABLED`
: The firmware component is disabled.

`DMI_FIRMWARE_INVENTORY_STATE_ENABLED`
: The firmware component is enabled.

`DMI_FIRMWARE_INVENTORY_STATE_ABSENT`
: The firmware component is not present, or has not been detected.

`DMI_FIRMWARE_INVENTORY_STATE_STANDBY_OFFLINE`
: The firmware is enabled, and waits for an external action to become active.

`DMI_FIRMWARE_INVENTORY_STATE_STANDBY_SPARE`
: The firmware belongs to a redundancy set, and waits for a failover or another external action to become active.

`DMI_FIRMWARE_INVENTORY_STATE_UNAVAIL_OFFLINE`
: The firmware component is present, but cannot be used.

Return value

`dmi_firmware_inventory_state_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_firmware_inventory`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_version

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef struct dmi_firmware_version {
    dmi_firmware_version_format_t format;
    dmi_firmware_version_number_t number;
    uint64_t value;
```

```
} dmi_firmware_version_t;
```

Description

The `dmi_firmware_version_t` structure holds a firmware version string parsed according to the version format of its firmware inventory information structure. It is the type of the `parsed_version` and `parsed_lowest_version` fields of `dmi_firmware_inventory_t`.

The decoder parses a string only if it conforms to the format the structure declares. The string itself stays available in `version` or `lowest_version` of `dmi_firmware_inventory_t` in any case.

Fields

`format`

Format the string has been parsed in, which tells the field holding the version: `DMI_FIRMWARE_VERSION_FORMAT_SEMANTIC` for `number`, `DMI_FIRMWARE_VERSION_FORMAT_HEX_32` and `DMI_FIRMWARE_VERSION_FORMAT_HEX_64` for `value`. `DMI_FIRMWARE_VERSION_FORMAT_FREE` if the string is absent, is free-form, uses a vendor-specific format, or does not conform to the format the structure declares; neither field is used then. See `dmi_firmware_version_format`.

`number`

Major and minor version numbers, if `format` is `DMI_FIRMWARE_VERSION_FORMAT_SEMANTIC`, see `dmi_firmware_version_number`.

`value`

Numeric value of the version, if `format` is `DMI_FIRMWARE_VERSION_FORMAT_HEX_32` or `DMI_FIRMWARE_VERSION_FORMAT_HEX_64`: the string is "0x" followed by at most 8 or 16 hexadecimal digits.

See also

`dmi_firmware_inventory`, `dmi_firmware_version_format`, `dmi_firmware_version_number`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_version_format

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef enum dmi_firmware_version_format {
    DMI_FIRMWARE_VERSION_FORMAT_FREE     = 0x00,
    DMI_FIRMWARE_VERSION_FORMAT_SEMANTIC = 0x01,
    DMI_FIRMWARE_VERSION_FORMAT_HEX_32   = 0x02,
    DMI_FIRMWARE_VERSION_FORMAT_HEX_64   = 0x03,
} dmi_firmware_version_format_t;
```

```
const char *dmi_firmware_version_format_name(
        dmi_firmware_version_format_t value);
```

Description

The `dmi_firmware_version_format_t` enumeration tells how the version of a firmware is written, and is the type of the `version_format` field of `dmi_firmware_inventory_t` and of the `format` field of `dmi_firmware_version_t`. Values 0x04 to 0x7F are reserved by the specification, and values 0x80 to 0xFF are for firmware vendors and OEMs; the enumeration has no constants for them.

The `dmi_firmware_version_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_FIRMWARE_VERSION_FORMAT_FREE`

Free-form string, whose meaning depends on the implementation, for example "1.45.455b66-rev4".

`DMI_FIRMWARE_VERSION_FORMAT_SEMANTIC`

Major and minor version numbers in decimal, separated by a dot, for example "1.45".

`DMI_FIRMWARE_VERSION_FORMAT_HEX_32`

Numeric value of 32 bits, written in hexadecimal the way C does: "0x" followed by 8 digits, for example "0x0001002d".

`DMI_FIRMWARE_VERSION_FORMAT_HEX_64`

Numeric value of 64 bits, written in hexadecimal the way C does: "0x" followed by 16 digits, for example "0x000000010000002d".

Return value

`dmi_firmware_version_format_name` returns the name of the value, or `nullptr` if the value has no name. Values 0x04 to 0x7F are named "Reserved", and values 0x80 to 0xFF "Vendor/OEM-specific". The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_firmware_inventory`, `dmi_firmware_version`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_firmware_version_number

Synopsis

```
#include <opendmi/entity/firmware-inventory.h>

typedef struct dmi_firmware_version_number {
    uint32_t major;
    uint32_t minor;
} dmi_firmware_version_number_t;
```

Description

The `dmi_firmware_version_number_t` structure holds a firmware version written in the "MAJOR.MINOR" format (`DMI_FIRMWARE_VERSION_FORMAT_SEMANTIC`), as two numbers. It is the type of the `number` field of `dmi_firmware_version_t`.

Fields

`major`
: Major version number: the number before the dot.

`minor`
: Minor version number: the number after the dot.

See also

`dmi_firmware_version`, `dmi_firmware_inventory`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

<opendmi/entity/firmware-language.h>

dmi_firmware_language

Synopsis

```
#include <opendmi/entity/firmware-language.h>

typedef struct dmi_firmware_language {
    size_t language_count;
    const char **languages;
    dmi_firmware_language_flags_t flags;
    const char *current_language;
} dmi_firmware_language_t;
```

Description

The `dmi_firmware_language_t` structure holds the decoded firmware language information structure (type 13) of SMBIOS: the languages the firmware can show its messages in, and the one which is installed.

The names of the languages come in one of two formats, which the flags of the structure tell apart (see `dmi_firmware_language_flags`): the long one gives an ISO 639-1 language code, an ISO 3166-1 alpha-2 territory code and an encoding, separated by vertical bars, e.g. "de|DE|iso8859-1"; the abbreviated one glues the two codes together, e.g. "deDE".

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(firmware_language)`, and lives as long as the context it has been decoded in.

Fields

`language_count`

Number of languages the firmware can install, as the structure declares it, and number of elements of `languages`.

`languages`

Names of the languages, one per string of the structure, in order; `nullptr` if `language_count` is 0. An element is `nullptr` if the string it refers to is missing.

`flags`

Flags, see `dmi_firmware_language_flags`. Defined since SMBIOS 2.1; the byte is reserved, and zero, in older data.

`current_language`

Name of the language installed, one of `languages`; `nullptr` if the structure does not carry it.

Examples

```
dmi_registry_t          *registry = dmi_get_registry(context);
dmi_entity_t            *entity   = dmi_registry_lookup_first(registry,
                                        DMI_TYPE(firmware_language), true);
dmi_firmware_language_t *info     = dmi_entity_info(entity,
                                        DMI_TYPE(firmware_language));

for (size_t i = 0; (info != nullptr) && (i < info->language_count); i++) {
    const char *language = info->languages[i];

    if (language == nullptr)
        continue;

    bool current = (info->current_language != nullptr) &&
                   (strcmp(language, info->current_language) == 0);

    printf("%s%s\n", language, current ? " (installed)" : "");
}
```

See also

`dmi_entity_info`, `dmi_firmware_language_flags`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_firmware_language_flags

Synopsis

```
#include <opendmi/entity/firmware-language.h>

typedef union dmi_firmware_language_flags {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_abbreviated : 1;
        dmi_byte_t __reserved     : 7;
    };
} dmi_firmware_language_flags_t;
```

Description

The `dmi_firmware_language_flags_t` union holds the flags of the firmware language information structure, and is the type of the `flags` field of `dmi_firmware_language_t`.

Fields

`__value`
: Raw value of the byte, `is_abbreviated` being bit 0.

`is_abbreviated`
: Set if the names of the languages are in the abbreviated format, a language code followed by a territory code, e.g. "deDE"; clear if they are in the long format, e.g. "de|DE|iso8859-1".

`__reserved`
: Reserved bits, zero.

See also

`dmi_firmware_language`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/firmware.h>

dmi_firmware

Synopsis

```
#include <opendmi/entity/firmware.h>

typedef struct dmi_firmware {
    const char                *vendor;
    const char                *version;
    uint16_t                   bios_segment;
    dmi_date_t                 release_date;
    dmi_size_t                 rom_size;
    dmi_firmware_features_t    features;
    dmi_firmware_features_ex_t features_ex;
    dmi_version_t              platform_version;
    dmi_version_t              controller_version;
} dmi_firmware_t;
```

Description

The `dmi_firmware_t` structure holds the decoded platform firmware information structure (type 0) of SMBIOS: the vendor, version and release date of the platform firmware (BIOS or UEFI), the size of the ROM holding it, and the features it supports. A table has exactly one such structure.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(firmware)`, and lives as long as the context it has been decoded in. The fields a structure is too short to carry are 0 or `nullptr`.

Fields

`vendor`

Name of the firmware vendor, or `nullptr`.

`version`

Firmware version, a free-form string, or `nullptr`.

`bios_segment`

Segment of the starting address of the BIOS, for example 0xE800; the size of the runtime BIOS image is (0x10000 - `bios_segment`) * 16 bytes. 0 where it does not apply, such as on UEFI systems.

`release_date`

Release date of the firmware, which the `dmi_date_year`, `dmi_date_month` and `dmi_date_day` functions of `<opendmi/utils/datetime.h>` read. `DMI_DATE_NONE` if the structure carries no date, or one which cannot be parsed. Two-digit years are taken as 19xx.

`rom_size`

Size of the physical device(s) holding the firmware, in bytes, a multiple of 64 KiB. Sizes of 16 MiB and above are taken from the extended ROM size field, present since SMBIOS 3.1, in units of MiB or GiB; 0 if that field uses a reserved unit.

`features`

Features the firmware supports, see `dmi_firmware_features`.

`features_ex`

Extended features the firmware supports, see `dmi_firmware_features_ex`. Present since SMBIOS 2.1.

`platform_version`

Major and minor release of the platform firmware, for example 10.22, which the `dmi_version_major` and `dmi_version_minor` functions of `<opendmi/utils/version.h>` read. `DMI_VERSION_NONE` if the system does not report it. Present since SMBIOS 2.4.

`controller_version`

Major and minor release of the embedded controller firmware. `DMI_VERSION_NONE` if the system has no embedded controller firmware which can be upgraded in the field. Present since SMBIOS 2.4.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                     DMI_TYPE(firmware),
                                                     false);
dmi_firmware_t *firmware = dmi_entity_info(entity, DMI_TYPE(firmware));

if ((firmware != nullptr) && (firmware->release_date != DMI_DATE_NONE))
    printf("Firmware released on %04u-%02u-%02u\n",
           dmi_date_year(firmware->release_date),
           dmi_date_month(firmware->release_date),
           dmi_date_day(firmware->release_date));
```

See also

`dmi_entity_info`, `dmi_firmware_features`, `dmi_firmware_features_ex`, `dmi_firmware_language`, `dmi_firmware_inventory`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_firmware_features

Synopsis

```
#include <opendmi/entity/firmware.h>

typedef union dmi_firmware_features {
    dmi_qword_t __value;
    struct {
        dmi_qword_t __reserved          : 2;
        dmi_qword_t unknown             : 1;
        dmi_qword_t unsupported         : 1;
        dmi_qword_t isa_support         : 1;
        dmi_qword_t mca_support         : 1;
        dmi_qword_t eisa_support        : 1;
        dmi_qword_t pci_support         : 1;
        dmi_qword_t pcmcia_support      : 1;
        dmi_qword_t pnp_support         : 1;
        dmi_qword_t apm_support         : 1;
        dmi_qword_t upgradeable         : 1;
        dmi_qword_t shadowing           : 1;
        dmi_qword_t vesa_support        : 1;
        dmi_qword_t escd_support        : 1;
        dmi_qword_t boot_cd             : 1;
        dmi_qword_t boot_selectable     : 1;
        dmi_qword_t socketed            : 1;
        dmi_qword_t boot_pcmcia         : 1;
        dmi_qword_t edd_support         : 1;
        dmi_qword_t floppy_svc_nec      : 1;
        dmi_qword_t floppy_svc_toshiba  : 1;
        dmi_qword_t floppy_svc_525_360k : 1;
        dmi_qword_t floppy_svc_525_1m2  : 1;
        dmi_qword_t floppy_svc_35_720k  : 1;
        dmi_qword_t floppy_svc_35_2m88  : 1;
        dmi_qword_t print_screen_svc    : 1;
        dmi_qword_t keyboard_svc        : 1;
        dmi_qword_t serial_svc          : 1;
        dmi_qword_t printer_svc         : 1;
        dmi_qword_t video_svc_cga_mono  : 1;
        dmi_qword_t nec_pc_98           : 1;
    };
} dmi_firmware_features_t;
```

Description

The `dmi_firmware_features_t` union holds the set of the features the platform firmware supports, as the platform firmware information structure (type 0) of SMBIOS gives them. It is the type of the `features` field of `dmi_firmware_t`. Each bit, when set, tells that the feature is supported.

Bits 32 to 47 of the value are reserved for the firmware vendor, and bits 48 to 63 for the system vendor;

they have no members, and are only available in `__value`.

Fields

`__value`
All the bits as one quad word, as stored.

`__reserved`
Reserved bits 0 and 1.

`unknown`
The features are unknown.

`unsupported`
The firmware does not report its features.

`isa_support`
ISA.

`mca_support`
MCA.

`eisa_support`
EISA.

`pci_support`
PCI.

`pcmcia_support`
PC card (PCMCIA).

`pnp_support`
Plug and Play.

`apm_support`
APM.

`upgradeable`
The firmware can be upgraded (flash).

`shadowing`
The firmware can be shadowed.

`vesa_support`
VL-VESA.

`escd_support`
ESCD.

`boot_cd`
Boot from CD.

`boot_selectable`
: Selectable boot device.

`socketed`
: The firmware ROM is socketed.

`boot_pcmcia`
: Boot from PC card (PCMCIA).

`edd_support`
: Enhanced Disk Drive (EDD) specification.

`floppy_svc_nec`
: Int 13h services for the 1.2 MB Japanese floppy of NEC 9800.

`floppy_svc_toshiba`
: Int 13h services for the 1.2 MB Japanese floppy of Toshiba.

`floppy_svc_525_360k`
: Int 13h services for 5.25" 360 KB floppies.

`floppy_svc_525_1m2`
: Int 13h services for 5.25" 1.2 MB floppies.

`floppy_svc_35_720k`
: Int 13h services for 3.5" 720 KB floppies.

`floppy_svc_35_2m88`
: Int 13h services for 3.5" 2.88 MB floppies.

`print_screen_svc`
: Int 5h print screen service.

`keyboard_svc`
: Int 9h 8042 keyboard services.

`serial_svc`
: Int 14h serial services.

`printer_svc`
: Int 17h printer services.

`video_svc_cga_mono`
: Int 10h CGA and monochrome video services.

`nec_pc_98`
: NEC PC-98.

See also

`dmi_firmware`, `dmi_firmware_features_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_firmware_features_ex

Synopsis

```
#include <opendmi/entity/firmware.h>

typedef union dmi_firmware_features_ex {
    dmi_byte_t __value[2];
    struct {
        bool acpi_support         : 1;
        bool usb_legacy_support   : 1;
        bool agp_support          : 1;
        bool boot_i2o             : 1;
        bool boot_ls120           : 1;
        bool boot_atapi_zip       : 1;
        bool boot_ieee1394        : 1;
        bool smart_battery        : 1;

        bool bios_boot_spec       : 1;
        bool boot_network_fn_key  : 1;
        bool content_distribution : 1;
        bool uefi_spec            : 1;
        bool virtual_machine      : 1;
        bool mfg_mode_support     : 1;
        bool mfg_mode_enabled     : 1;
    };
} dmi_firmware_features_ex_t;
```

Description

The `dmi_firmware_features_ex_t` union holds the features of the platform firmware which the extension bytes of the platform firmware information structure (type 0) of SMBIOS give. It is the type of the `features_ex` field of `dmi_firmware_t`. Each bit, when set, tells that the feature is supported.

The first byte is present since SMBIOS 2.1, and the second one since SMBIOS 2.3; the bits of a byte the structure does not carry are 0.

Fields

`__value`
: Both bytes, as stored.

`acpi_support`
: ACPI.

`usb_legacy_support`
: USB legacy support.

`agp_support`
: AGP.

`boot_i2o`
: Boot from I2O devices.

`boot_ls120`
: Boot from LS-120 SuperDisk.

`boot_atapi_zip`
: Boot from ATAPI ZIP drives.

`boot_ieee1394`
: Boot from IEEE 1394 devices.

`smart_battery`
: Smart battery.

`bios_boot_spec`
: BIOS Boot Specification. Present since SMBIOS 2.3, as all the following bits.

`boot_network_fn_key`
: Network boot started by a function key.

`content_distribution`
: The SMBIOS data identifies the system well enough for the targeted distribution of software and firmware.

`uefi_spec`
: UEFI specification.

`virtual_machine`
: The system is a virtual machine. A clear bit tells nothing.

`mfg_mode_support`
: Manufacturing mode, a boot mode used while the system is manufactured and tested.

`mfg_mode_enabled`
: Manufacturing mode is enabled.

See also

`dmi_firmware`, `dmi_firmware_features`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/group-assoc.h>

dmi_group_assoc

Synopsis

```
#include <opendmi/entity/group-assoc.h>

typedef struct dmi_group_assoc {
    const char             *group_name;
    size_t                  item_count;
    dmi_group_assoc_item_t *items;
} dmi_group_assoc_t;
```

Description

The `dmi_group_assoc_t` structure holds the decoded group associations structure (type 14) of SMBIOS: it gathers other structures, possibly other groups, under a name, to show how components of the system are arranged. For example, a group may hold two processors and the cache they share.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(group_assoc)`, and lives as long as the context it has been decoded in.

Fields

`group_name`

Name of the group, `nullptr` if the structure has none.

`item_count`

Number of elements of `items`. The structure does not store it: it is the number of items which fit into the length of the structure.

`items`

Array of the members of the group, see `dmi_group_assoc_item`; `nullptr` if there are none.

Examples

```
dmi_registry_t    *registry = dmi_get_registry(context);
dmi_entity_t      *entity   = dmi_registry_lookup_first(registry,
                                  DMI_TYPE(group_assoc), false);
dmi_group_assoc_t *group    = dmi_entity_info(entity, DMI_TYPE(group_assoc));

if (group != nullptr) {
    printf("%s:\n",
           group->group_name != nullptr ? group->group_name : "Unnamed");

    for (size_t i = 0; i < group->item_count; i++)
```

```
        printf("Type %d, handle 0x%04x\n", (int)group->items[i].type,
               (unsigned)group->items[i].handle);
}
```

See also

`dmi_entity_info`, `dmi_group_assoc_item`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_group_assoc_item

Synopsis

```
#include <opendmi/entity/group-assoc.h>

typedef struct dmi_group_assoc_item {
    dmi_type_id_t type;
    dmi_handle_t  handle;
    dmi_entity_t *entity;
} dmi_group_assoc_item_t;
```

Description

The `dmi_group_assoc_item_t` structure describes one member of a group association, and is the type of the elements of the `items` array of `dmi_group_assoc_t`.

Fields

`type`

Type number of the member structure, as stored.

`handle`

Handle of the member structure.

`entity`

Entity of the member structure, set when the structures are linked; `nullptr` if `handle` does not refer to a structure found at type number `type`.

See also

`dmi_group_assoc`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

`<opendmi/entity/hardware-security.h>`

dmi_hardware_security

Synopsis

```
#include <opendmi/entity/hardware-security.h>

typedef struct dmi_hardware_security {
    dmi_hardware_security_status_t front_panel_reset;
    dmi_hardware_security_status_t admin_password;
    dmi_hardware_security_status_t keyboard_password;
    dmi_hardware_security_status_t poweron_password;
} dmi_hardware_security_t;
```

Description

The `dmi_hardware_security_t` structure holds the decoded hardware security structure (type 24) of SMBIOS: the state of the passwords and of the front panel reset, which the firmware sets for the whole system.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(hardware_security)` and lives as long as the context it has been decoded in.

Fields

`front_panel_reset`

State of the front panel reset.

`admin_password`

State of the administrator password.

`keyboard_password`

State of the keyboard password.

`poweron_password`

State of the power-on password.

Examples

```
dmi_registry_t          *registry = dmi_get_registry(context);
dmi_entity_t            *entity   = dmi_registry_lookup_first(registry,
                                        DMI_TYPE(hardware_security), true);
dmi_hardware_security_t *security = dmi_entity_info(entity,
                                        DMI_TYPE(hardware_security));

if (security != nullptr) {
```

```
    const char *state =
        dmi_hardware_security_status_name(security->poweron_password);

    printf("Power-on password: %s\n", state != nullptr ? state : "?");
}
```

See also

`dmi_entity_info`, `dmi_hardware_security_status`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

dmi_hardware_security_status

Synopsis

```
#include <opendmi/entity/hardware-security.h>

typedef enum dmi_hardware_security_status {
    DMI_HARDWARE_SECURITY_STATUS_DISABLED        = 0x00,
    DMI_HARDWARE_SECURITY_STATUS_ENABLED         = 0x01,
    DMI_HARDWARE_SECURITY_STATUS_NOT_IMPLEMENTED = 0x02,
    DMI_HARDWARE_SECURITY_STATUS_UNKNOWN         = 0x03,
} dmi_hardware_security_status_t;

const char *dmi_hardware_security_status_name(
        dmi_hardware_security_status_t value);
```

Description

The `dmi_hardware_security_status_t` enumeration tells the state of a hardware security setting, and is the type of the fields of `dmi_hardware_security_t`. The structure carries each state in two bits, so every value it can hold is defined.

The `dmi_hardware_security_status_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_HARDWARE_SECURITY_STATUS_DISABLED`
: Setting is disabled.

`DMI_HARDWARE_SECURITY_STATUS_ENABLED`
: Setting is enabled.

`DMI_HARDWARE_SECURITY_STATUS_NOT_IMPLEMENTED`

System does not implement the setting.

`DMI_HARDWARE_SECURITY_STATUS_UNKNOWN`

State of the setting is unknown.

Return value

`dmi_hardware_security_status_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_hardware_security`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/ipmi-device.h>

dmi_ipmi_addr_type

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef enum dmi_ipmi_addr_type {
    DMI_IPMI_ADDR_TYPE_MEMORY = 0x0,
    DMI_IPMI_ADDR_TYPE_IO     = 0x1,
    DMI_IPMI_ADDR_TYPE_SMBUS  = 0x2
} dmi_ipmi_addr_type_t;

const char *dmi_ipmi_addr_type_name(dmi_ipmi_addr_type_t value);
```

Description

The `dmi_ipmi_addr_type_t` enumeration tells in which address space the base address of an IPMI baseboard management controller (BMC) lies, and is the type of the `base_address_type` field of `dmi_ipmi_device_t`. The decoder derives it from the interface type and from the least-significant bit of the stored base address.

The `dmi_ipmi_addr_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_IPMI_ADDR_TYPE_MEMORY`
: Memory-mapped address.

`DMI_IPMI_ADDR_TYPE_IO`
: Address in the I/O space.

`DMI_IPMI_ADDR_TYPE_SMBUS`
: SMBus target address, used by the SSIF interface.

Return value

`dmi_ipmi_addr_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the language of the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_ipmi_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_device

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef struct dmi_ipmi_device {
    dmi_ipmi_iface_type_t        interface_type;
    dmi_version_t                specification_version;
    dmi_i2c_addr_t               i2c_target_address;
    uint8_t                      nv_storage_address;
    dmi_size_t                   base_address;
    dmi_ipmi_addr_type_t         base_address_type;
    bool                         base_address_lsb;
    dmi_ipmi_intr_trigger_mode_t interrupt_trigger_mode;
    dmi_ipmi_intr_polarity_t     interrupt_polarity;
    unsigned short               register_spacing;
    unsigned short               interrupt_number;
} dmi_ipmi_device_t;
```

Description

The `dmi_ipmi_device_t` structure holds the decoded IPMI device information structure (type 38) of SMBIOS: it describes the baseboard management controller (BMC) of the Intelligent Platform Management Interface (IPMI), and how software reaches its system interface. The IPMI specifications tell how to use this information.

The byte which follows the base address in the structure, holding the base address modifier and the interrupt information, is spread over the `base_address_lsb`, `register_spacing`, `interrupt_trigger_mode` and `interrupt_polarity` fields; see `dmi_ipmi_device_details` for its layout.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(ipmi_device)`, and lives as long as the context it has been decoded in.

Fields

`interface_type`

Type of the system interface of the BMC, see `dmi_ipmi_iface_type`.

`specification_version`

Revision of the IPMI specification the BMC has been designed for, such as 1.5 or 2.0. The structure stores the major revision in the high nibble and the minor one in the low nibble; use `dmi_version_major` and `dmi_version_minor` to read it.

`i2c_target_address`

Target address of the BMC on the I2C bus, as stored.

`nv_storage_address`

Bus ID of the non-volatile storage device of the BMC, `UINT8_MAX` (0xFF) if the BMC has none.

`base_address`

Base address of the BMC. For the memory and I/O address spaces, this is the full address, with its least-significant bit taken from the base address modifier; `base_address_type` tells which space it is in. For the SSIF interface, it is the 7-bit SMBus target address of the BMC.

`base_address_type`

Address space of `base_address`, see `dmi_ipmi_addr_type`: `DMI_IPMI_ADDR_TYPE_SMBUS` for the SSIF interface, otherwise memory or I/O space, as the least-significant bit of the stored address says.

`base_address_lsb`

Least-significant bit of the base address, as given by the base address modifier.

`interrupt_trigger_mode`

Trigger mode of the interrupt of the system interface, see `dmi_ipmi_intr_trigger_mode`; `DMI_IPMI_INTR_TRIGGER_MODE_UNSPEC` when the structure does not specify the interrupt information.

`interrupt_polarity`

Polarity of the interrupt of the system interface, see `dmi_ipmi_intr_polarity`; `DMI_IPMI_INTR_POLARITY_UNSPEC` when the structure does not specify the interrupt information.

`register_spacing`

Distance between the interface registers, in bytes: 1, 4 or 16, or 0 when the structure holds the reserved code.

`interrupt_number`

Interrupt number of the system interface, 0 if it is unspecified or not supported.

Examples

```
dmi_registry_t    *registry = dmi_get_registry(context);
dmi_entity_t      *entity   = dmi_registry_lookup_first(registry,
                                  DMI_TYPE(ipmi_device), false);
dmi_ipmi_device_t *ipmi     = dmi_entity_info(entity, DMI_TYPE(ipmi_device));

if (ipmi != nullptr) {
    const char *interface = dmi_ipmi_iface_type_name(ipmi->interface_type);

    printf("IPMI %u.%u, %s interface at 0x%llx (%s)\n",
```

```
            dmi_version_major(ipmi->specification_version),
            dmi_version_minor(ipmi->specification_version),
            interface != nullptr ? interface : "unknown",
            (unsigned long long)ipmi->base_address,
            dmi_ipmi_addr_type_name(ipmi->base_address_type));
}
```

See also

`dmi_entity_info`, `dmi_ipmi_iface_type`, `dmi_ipmi_addr_type`, `dmi_ipmi_intr_trigger_mode`, `dmi_ipmi_intr_polarity`, `dmi_ipmi_register_spacing`, `dmi_ipmi_device_details`, `dmi_version_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_device_details

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef union dmi_ipmi_device_details {
    uint8_t __value;

    struct {
        uint8_t is_intr_level_triggered : 1;
        uint8_t is_intr_active_high     : 1;
        uint8_t __reserved_1            : 1;
        uint8_t is_intr_info_specified  : 1;
        uint8_t base_address_lsb           : 1;
        uint8_t __reserved_2            : 1;
        uint8_t register_spacing        : 2;
    };
} dmi_ipmi_device_details_t;
```

Description

The `dmi_ipmi_device_details_t` union describes the byte of the IPMI device information structure which follows the base address: the base address modifier and the interrupt information. The byte can be read as a whole through `__value`, or bit by bit through the named members.

The decoder does not keep the byte: it spreads it over the fields of `dmi_ipmi_device_t`, and the union serves to read or build the raw byte. For the SSIF interface, the byte is unused and set to 0.

Fields

`__value`
: Raw value of the byte.

`is_intr_level_triggered`
: Set if the interrupt is level-triggered, clear if it is edge-triggered. Meaningful only when `is_intr_info_specified` is set.

`is_intr_active_high`
: Set if the interrupt is active high, clear if it is active low. Meaningful only when `is_intr_info_specified` is set.

`__reserved_1`
: Reserved bit, set to 0.

`is_intr_info_specified`
: Set if the byte specifies the trigger mode and the polarity of the interrupt.

`base_address_lsb`
: Least-significant bit of the base address of the BMC.

`__reserved_2`
: Reserved bit, set to 0.

`register_spacing`
: Distance between the interface registers, as a code of `dmi_ipmi_register_spacing`.

See also

`dmi_ipmi_device`, `dmi_ipmi_register_spacing`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_iface_type

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef enum dmi_ipmi_iface_type {
    DMI_IPMI_IFACE_TYPE_UNKNOWN = 0x00,
    DMI_IPMI_IFACE_TYPE_KCS     = 0x01,
    DMI_IPMI_IFACE_TYPE_SMIC    = 0x02,
    DMI_IPMI_IFACE_TYPE_BT      = 0x03,
    DMI_IPMI_IFACE_TYPE_SSIF    = 0x04,
} dmi_ipmi_iface_type_t;
```

```
const char *dmi_ipmi_iface_type_name(dmi_ipmi_iface_type_t value);
```

Description

The `dmi_ipmi_iface_type_t` enumeration tells the type of the system interface of an IPMI baseboard management controller (BMC), and is the type of the `interface_type` field of `dmi_ipmi_device_t`. Values 0x05 to 0xFF are reserved by the specification.

The `dmi_ipmi_iface_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_IPMI_IFACE_TYPE_UNKNOWN`
: Unknown.

`DMI_IPMI_IFACE_TYPE_KCS`
: Keyboard Controller Style (KCS) interface.

`DMI_IPMI_IFACE_TYPE_SMIC`
: Server Management Interface Chip (SMIC) interface.

`DMI_IPMI_IFACE_TYPE_BT`
: Block Transfer (BT) interface.

`DMI_IPMI_IFACE_TYPE_SSIF`
: SMBus System Interface (SSIF).

Return value

`dmi_ipmi_iface_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the language of the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_ipmi_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_intr_polarity

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef enum dmi_ipmi_intr_polarity {
    DMI_IPMI_INTR_POLARITY_UNSPEC = 0x0,
    DMI_IPMI_INTR_POLARITY_LOW    = 0x1,
    DMI_IPMI_INTR_POLARITY_HIGH   = 0x2
} dmi_ipmi_intr_polarity_t;

const char *dmi_ipmi_intr_polarity_name(dmi_ipmi_intr_polarity_t value);
```

Description

The `dmi_ipmi_intr_polarity_t` enumeration tells the polarity of the interrupt of the system interface of an IPMI baseboard management controller, and is the type of the `intr_polarity` field of `dmi_ipmi_device_t`.

The `dmi_ipmi_intr_polarity_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_IPMI_INTR_POLARITY_UNSPEC`
: Unspecified: the structure does not give the interrupt information.

`DMI_IPMI_INTR_POLARITY_LOW`
: Active low.

`DMI_IPMI_INTR_POLARITY_HIGH`
: Active high.

Return value

`dmi_ipmi_intr_polarity_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the language of the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_ipmi_device`, `dmi_ipmi_intr_trigger_mode`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_intr_trigger_mode

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef enum dmi_ipmi_intr_trigger_mode {
    DMI_IPMI_INTR_TRIGGER_MODE_UNSPEC = 0x0,
    DMI_IPMI_INTR_TRIGGER_MODE_EDGE   = 0x1,
    DMI_IPMI_INTR_TRIGGER_MODE_LEVEL  = 0x2
} dmi_ipmi_intr_trigger_mode_t;

const char *dmi_ipmi_intr_trigger_mode_name(dmi_ipmi_intr_trigger_mode_t
value);
```

Description

The `dmi_ipmi_intr_trigger_mode_t` enumeration tells how the interrupt of the system interface of an IPMI baseboard management controller is triggered, and is the type of the `intr_trigger` field of `dmi_ipmi_device_t`.

The `dmi_ipmi_intr_trigger_mode_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_IPMI_INTR_TRIGGER_MODE_UNSPEC`

Unspecified: the structure does not give the interrupt information.

`DMI_IPMI_INTR_TRIGGER_MODE_EDGE`

Edge-triggered interrupt.

`DMI_IPMI_INTR_TRIGGER_MODE_LEVEL`

Level-triggered interrupt.

Return value

`dmi_ipmi_intr_trigger_mode_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the language of the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_ipmi_device`, `dmi_ipmi_intr_polarity`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_ipmi_register_spacing

Synopsis

```
#include <opendmi/entity/ipmi-device.h>

typedef enum dmi_ipmi_register_spacing {
    DMI_IPMI_REGISTER_SPACING_1        = 0x0,
    DMI_IPMI_REGISTER_SPACING_4        = 0x1,
    DMI_IPMI_REGISTER_SPACING_16       = 0x2,
    DMI_IPMI_REGISTER_SPACING_RESERVED = 0x3,
} dmi_ipmi_register_spacing_t;
```

Description

The `dmi_ipmi_register_spacing_t` enumeration lists the codes of the distance between the interface registers of an IPMI baseboard management controller, as the `register_spacing` bits of `dmi_ipmi_device_details_t` hold them. The `register_spacing` field of `dmi_ipmi_device_t` holds the distance in bytes instead.

Values

`DMI_IPMI_REGISTER_SPACING_1`
: Registers on successive byte boundaries.

`DMI_IPMI_REGISTER_SPACING_4`
: Registers on 32-bit boundaries.

`DMI_IPMI_REGISTER_SPACING_16`
: Registers on 16-byte boundaries.

`DMI_IPMI_REGISTER_SPACING_RESERVED`
: Reserved code; decoded as a spacing of 0 bytes.

See also

`dmi_ipmi_device`, `dmi_ipmi_device_details`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/memory-array-addr.h>

dmi_memory_array_addr

Synopsis

```
#include <opendmi/entity/memory-array-addr.h>

typedef struct dmi_memory_array_addr {
    uint64_t        start_address;
    uint64_t        end_address;
    uint64_t        range_size;
    dmi_handle_t    array_handle;
    dmi_entity_t   *array;
    unsigned short  partition_width;
} dmi_memory_array_addr_t;
```

Description

The `dmi_memory_array_addr_t` structure holds the decoded memory array mapped address structure (type 19) of SMBIOS: it maps one contiguous range of physical addresses to a physical memory array. The table holds one such structure for each range, and several ranges may map to the same array. Since SMBIOS 2.3, a table must hold at least one of them.

The structure gives the addresses in kilobytes, or, since SMBIOS 2.7, in bytes in extended fields for the ranges which do not fit; the decoder converts both to bytes.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_array_addr)`, and lives as long as the context it has been decoded in.

Fields

`start_address`

Physical address of the start of the range, in bytes.

`end_address`

Physical address of the last byte of the range. When the structure gives the addresses in kilobytes, this is the last byte of the last kilobyte of the range.

`range_size`

Size of the range in bytes, computed from `start_address` and `end_address`, whichever order they come in, both of which belong to the range.

`array_handle`

Handle of the physical memory array structure (type 16) the range maps to.

`array`

Entity of the physical memory array structure, set when the structures are linked; `nullptr` if

`array_handle` does not refer to a physical memory array.

`partition_width`

Number of memory devices which make up one row of memory for the partition of addresses this structure defines.

Examples

```
dmi_registry_t          *registry = dmi_get_registry(context);
dmi_entity_t            *entity   = dmi_registry_lookup_first(registry,
                                        DMI_TYPE(memory_array_addr), false);
dmi_memory_array_addr_t *range    = dmi_entity_info(entity,
                                        DMI_TYPE(memory_array_addr));

if (range != nullptr)
    printf("0x%llx-0x%llx, array 0x%04x\n",
           (unsigned long long)range->start_address,
           (unsigned long long)range->end_address,
           (unsigned)range->array_handle);
```

See also

`dmi_entity_info`, `dmi_memory_array`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-array.h>

dmi_memory_array

Synopsis

```
#include <opendmi/entity/memory-array.h>

typedef struct dmi_memory_array {
    dmi_memory_array_location_t location;
    dmi_memory_array_use_t use;
    dmi_error_correct_type_t error_correction;
    dmi_size_t maximum_capacity;
    dmi_handle_t error_handle;
    dmi_entity_t *error_info;
    unsigned int device_count;
} dmi_memory_array_t;
```

Description

The `dmi_memory_array_t` structure holds the decoded physical memory array structure (type 16) of SMBIOS: a group of memory devices which work together to make up one memory address space, such as the system memory on the motherboard, or the memory of an add-on card.

A table has one such structure per memory array. The memory device structures (type 17) of the array and the memory array mapped address structures (type 19) refer to it by its handle. The structure is required since SMBIOS 2.3.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_array)`, and lives as long as the context it has been decoded in.

Fields

`location`

Where the array resides: on the motherboard, or on an add-on card of some kind. See `dmi_memory_array_location`.

`use`

What the array is used for, e.g. system memory, video memory or flash memory. See `dmi_memory_array_use`.

`error_correction`

Main method of error detection or correction the array supports. See `dmi_error_correct_type`.

`maximum_capacity`

Largest amount of memory the array can hold, in bytes. The structure carries it in kilobytes, or in bytes in an extended field present since SMBIOS 2.7 for the arrays of 2 TiB or more; both are converted to bytes. `DMI_SIZE_MAX` if the capacity is unknown, which is the case when the structure points to the

extended field and is too short to hold it.

`error_handle`

Handle of the memory error information structure (type 18 or 33) describing the last error detected in the array. `DMI_HANDLE_INVALID` if no error has been detected, and `DMI_HANDLE_UNSUPPORTED` if the system provides no error information.

`error_info`

Memory error information structure of the array, resolved from `error_handle` once the table has been linked. `nullptr` if the handle is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or does not refer to a memory error information structure.

`device_count`

Number of the slots or sockets for memory devices the array has, which is the number of the memory device structures belonging to the array, whether the slots are populated or not.

Examples

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_entity_t       *entity   = dmi_registry_lookup_first(registry,
                                       DMI_TYPE(memory_array), false);
dmi_memory_array_t *array    = dmi_entity_info(entity,
                                            DMI_TYPE(memory_array));

if ((array != nullptr) && (array->maximum_capacity != DMI_SIZE_MAX))
    printf("%u slots, up to %llu bytes\n", array->device_count,
           (unsigned long long)array->maximum_capacity);
```

See also

`dmi_entity_info`, `dmi_memory_array_location`, `dmi_memory_array_use`, `dmi_error_correct_type`, `dmi_memory_device`, `dmi_memory_array_addr`, `dmi_memory_error_32`, `dmi_memory_error_64`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_array_location

Synopsis

```
#include <opendmi/entity/memory-array.h>

typedef enum dmi_memory_array_location {
    DMI_MEMORY_ARRAY_LOCATION_UNSPEC          = 0x00,
    DMI_MEMORY_ARRAY_LOCATION_OTHER           = 0x01,
    DMI_MEMORY_ARRAY_LOCATION_UNKNOWN         = 0x02,
```

```
    DMI_MEMORY_ARRAY_LOCATION_MOTHERBOARD     = 0x03,
    DMI_MEMORY_ARRAY_LOCATION_ISA             = 0x04,
    DMI_MEMORY_ARRAY_LOCATION_EISA            = 0x05,
    DMI_MEMORY_ARRAY_LOCATION_PCI             = 0x06,
    DMI_MEMORY_ARRAY_LOCATION_MCA             = 0x07,
    DMI_MEMORY_ARRAY_LOCATION_PCMCIA          = 0x08,
    DMI_MEMORY_ARRAY_LOCATION_PROPRIETARY     = 0x09,
    DMI_MEMORY_ARRAY_LOCATION_NUBUS           = 0x0A,
    DMI_MEMORY_ARRAY_LOCATION_PC_98_C20       = 0xA0,
    DMI_MEMORY_ARRAY_LOCATION_PC_98_C24       = 0xA1,
    DMI_MEMORY_ARRAY_LOCATION_PC_98_E         = 0xA2,
    DMI_MEMORY_ARRAY_LOCATION_PC_98_LOCAL_BUS = 0xA3,
    DMI_MEMORY_ARRAY_LOCATION_CXL             = 0xA4,
} dmi_memory_array_location_t;

const char *dmi_memory_array_location_name(dmi_memory_array_location_t value);
```

Description

The `dmi_memory_array_location_t` enumeration tells where a physical memory array resides, and is the type of the `location` field of `dmi_memory_array_t`. The values from 0x0B to 0x9F are not assigned by the specification.

The `dmi_memory_array_location_name` function returns the name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Values

`DMI_MEMORY_ARRAY_LOCATION_UNSPEC`

Unspecified. The specification defines no location of zero.

`DMI_MEMORY_ARRAY_LOCATION_OTHER`

Other location, which none of the values below names.

`DMI_MEMORY_ARRAY_LOCATION_UNKNOWN`

Unknown.

`DMI_MEMORY_ARRAY_LOCATION_MOTHERBOARD`

System board or motherboard.

`DMI_MEMORY_ARRAY_LOCATION_ISA`

ISA add-on card.

`DMI_MEMORY_ARRAY_LOCATION_EISA`

EISA add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PCI`

PCI add-on card.

`DMI_MEMORY_ARRAY_LOCATION_MCA`
: MCA add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PCMCIA`
: PCMCIA add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PROPRIETARY`
: Proprietary add-on card.

`DMI_MEMORY_ARRAY_LOCATION_NUBUS`
: NuBus.

`DMI_MEMORY_ARRAY_LOCATION_PC_98_C20`
: PC-98/C20 add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PC_98_C24`
: PC-98/C24 add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PC_98_E`
: PC-98/E add-on card.

`DMI_MEMORY_ARRAY_LOCATION_PC_98_LOCAL_BUS`
: PC-98/Local bus add-on card.

`DMI_MEMORY_ARRAY_LOCATION_CXL`
: CXL add-on card.

Return value

`dmi_memory_array_location_name` returns the name of the value, or `nullptr` if the value has no name, e.g. a value the specification reserves. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None.

See also

`dmi_memory_array`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_array_use

Synopsis

```
#include <opendmi/entity/memory-array.h>
```

```
typedef enum dmi_memory_array_use {
    DMI_MEMORY_ARRAY_USE_UNSPEC  = 0x00,
    DMI_MEMORY_ARRAY_USE_OTHER   = 0x01,
    DMI_MEMORY_ARRAY_USE_UNKNOWN = 0x02,
    DMI_MEMORY_ARRAY_USE_SYSTEM  = 0x03,
    DMI_MEMORY_ARRAY_USE_VIDEO   = 0x04,
    DMI_MEMORY_ARRAY_USE_FLASH   = 0x05,
    DMI_MEMORY_ARRAY_USE_NVRAM   = 0x06,
    DMI_MEMORY_ARRAY_USE_CACHE   = 0x07,
} dmi_memory_array_use_t;

const char *dmi_memory_array_use_name(dmi_memory_array_use_t value);
```

Description

The `dmi_memory_array_use_t` enumeration tells what a physical memory array is used for, and is the type of the `use` field of `dmi_memory_array_t`.

The `dmi_memory_array_use_name` function returns the name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Values

`DMI_MEMORY_ARRAY_USE_UNSPEC`
: Unspecified. The specification defines no use of zero.

`DMI_MEMORY_ARRAY_USE_OTHER`
: Other use, which none of the values below names.

`DMI_MEMORY_ARRAY_USE_UNKNOWN`
: Unknown.

`DMI_MEMORY_ARRAY_USE_SYSTEM`
: System memory.

`DMI_MEMORY_ARRAY_USE_VIDEO`
: Video memory.

`DMI_MEMORY_ARRAY_USE_FLASH`
: Flash memory.

`DMI_MEMORY_ARRAY_USE_NVRAM`
: Non-volatile RAM.

`DMI_MEMORY_ARRAY_USE_CACHE`
: Cache memory.

Return value

`dmi_memory_array_use_name` returns the name of the value, or `nullptr` if the value has no name, e.g. a value the specification reserves. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None.

See also

`dmi_memory_array`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-channel.h>

dmi_memory_channel

Synopsis

```
#include <opendmi/entity/memory-channel.h>

typedef struct dmi_memory_channel {
    dmi_memory_channel_type_t    type;
    unsigned short               maximum_load;
    size_t                       device_count;
    dmi_memory_channel_device_t *devices;
} dmi_memory_channel_t;
```

Description

The `dmi_memory_channel_t` structure holds the decoded memory channel structure (type 37) of SMBIOS: it lists the memory devices attached to a memory channel, and the load each of them puts on the channel. The loads of all the devices together must not exceed the maximum load of the channel.

When the structures are linked, the library also sets the `channel` field of each memory device of the list (see `dmi_memory_device`) to the entity of this structure.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_channel)`, and lives as long as the context it has been decoded in.

Fields

`type`

Type of memory of the channel, see `dmi_memory_channel_type`.

`maximum_load`

Maximum load the channel supports; the sum of the `load` fields of the devices must not exceed it.

`device_count`

Number of elements of `devices`: the devices the structure holds completely, out of the number it declares.

`devices`

Array of the memory devices attached to the channel, with their loads, see `dmi_memory_channel_device`; `nullptr` if there are none.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
```

```
                                    DMI_TYPE(memory_channel), false);
dmi_memory_channel_t *channel  = dmi_entity_info(entity,
                                    DMI_TYPE(memory_channel));

if (channel != nullptr) {
    const char *type = dmi_memory_channel_type_name(channel->type);

    printf("%s channel, maximum load %u\n",
           type != nullptr ? type : "Unknown", channel->maximum_load);

    for (size_t i = 0; i < channel->device_count; i++)
        printf("Device 0x%04x: load %u\n",
               (unsigned)channel->devices[i].handle,
               (unsigned)channel->devices[i].load);
}
```

See also

`dmi_entity_info`, `dmi_memory_channel_type`, `dmi_memory_channel_device`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_memory_channel_device

Synopsis

```
#include <opendmi/entity/memory-channel.h>

typedef struct dmi_memory_channel_device {
    dmi_byte_t    load;
    dmi_handle_t  handle;
    dmi_entity_t *device;
} dmi_memory_channel_device_t;
```

Description

The `dmi_memory_channel_device_t` structure describes one memory device attached to a memory channel, and is the type of the elements of the `devices` array of `dmi_memory_channel_t`.

Fields

`load`

Load the memory device puts on the channel.

`handle`
: Handle of the memory device structure (type 17).

`device`
: Entity of the memory device structure, set when the structures are linked; `nullptr` if `handle` does not refer to a memory device.

See also

`dmi_memory_channel`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_memory_channel_type

Synopsis

```
#include <opendmi/entity/memory-channel.h>

typedef enum dmi_memory_channel_type {
    DMI_MEMORY_CHANNEL_TYPE_UNSPEC   = 0x00,
    DMI_MEMORY_CHANNEL_TYPE_OTHER    = 0x01,
    DMI_MEMORY_CHANNEL_TYPE_UNKNOWN  = 0x02,
    DMI_MEMORY_CHANNEL_TYPE_RAMBUS   = 0x03,
    DMI_MEMORY_CHANNEL_TYPE_SYNCLINK = 0x04,
} dmi_memory_channel_type_t;

const char *dmi_memory_channel_type_name(dmi_memory_channel_type_t value);
```

Description

The `dmi_memory_channel_type_t` enumeration tells the type of memory of a memory channel, and is the type of the `type` field of `dmi_memory_channel_t`. The specification leaves these values to the DMTF.

The `dmi_memory_channel_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_MEMORY_CHANNEL_TYPE_UNSPEC`
: Unspecified. Not defined by the specification.

`DMI_MEMORY_CHANNEL_TYPE_OTHER`
: Other.

`DMI_MEMORY_CHANNEL_TYPE_UNKNOWN`
: Unknown.

`DMI_MEMORY_CHANNEL_TYPE_RAMBUS`
: Rambus.

`DMI_MEMORY_CHANNEL_TYPE_SYNCLINK`
: SyncLink.

Return value

`dmi_memory_channel_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the language of the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_memory_channel`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/memory-controller.h>

dmi_error_correct_caps

Synopsis

```
#include <opendmi/entity/memory-controller.h>

typedef union dmi_error_correct_caps {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_other      : 1;
        dmi_byte_t is_unknown    : 1;
        dmi_byte_t is_none       : 1;
        dmi_byte_t is_single_bit : 1;
        dmi_byte_t is_double_bit : 1;
        dmi_byte_t is_scrubbing  : 1;
        dmi_byte_t __reserved    : 2;
    };
} dmi_error_correct_caps_t;
```

Description

The `dmi_error_correct_caps_t` union holds the error correcting capabilities of a memory controller as a set of flags, and is the type of the `error_correction` and `enabled_error_correction` fields of `dmi_memory_controller_t`. A bit is set when the controller has the capability.

Fields

`__value`
: Raw value of the byte, `is_other` being bit 0.

`is_other`
: Other capability.

`is_unknown`
: The capabilities are unknown.

`is_none`
: No error correction.

`is_single_bit`
: Correction of single-bit errors.

`is_double_bit`
: Correction of double-bit errors.

`is_scrubbing`
: Error scrubbing.

`__reserved`
: Reserved bits, zero.

See also

`dmi_memory_controller`, `dmi_error_detect_method`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

dmi_error_detect_method

Synopsis

```
#include <opendmi/entity/memory-controller.h>

typedef enum dmi_error_detect_method {
    DMI_ERROR_DETECT_METHOD_UNSPEC  = 0x00,
    DMI_ERROR_DETECT_METHOD_OTHER   = 0x01,
    DMI_ERROR_DETECT_METHOD_UNKNOWN = 0x02,
    DMI_ERROR_DETECT_METHOD_NONE    = 0x03,
    DMI_ERROR_DETECT_METHOD_PARITY  = 0x04,
    DMI_ERROR_DETECT_METHOD_ECC_32  = 0x05,
    DMI_ERROR_DETECT_METHOD_ECC_64  = 0x06,
    DMI_ERROR_DETECT_METHOD_ECC_128 = 0x07,
    DMI_ERROR_DETECT_METHOD_CRC     = 0x08,
} dmi_error_detect_method_t;

const char *dmi_error_detect_method_name(dmi_error_detect_method_t value);
```

Description

The `dmi_error_detect_method_t` enumeration tells how a memory controller detects memory errors, and is the type of the `error_detection` field of `dmi_memory_controller_t`.

The `dmi_error_detect_method_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_ERROR_DETECT_METHOD_UNSPEC`
: Unspecified.

`DMI_ERROR_DETECT_METHOD_OTHER`
: Other.

`DMI_ERROR_DETECT_METHOD_UNKNOWN`
: Unknown.

`DMI_ERROR_DETECT_METHOD_NONE`
: No error detection.

`DMI_ERROR_DETECT_METHOD_PARITY`
: 8-bit parity.

`DMI_ERROR_DETECT_METHOD_ECC_32`
: 32-bit ECC.

`DMI_ERROR_DETECT_METHOD_ECC_64`
: 64-bit ECC.

`DMI_ERROR_DETECT_METHOD_ECC_128`
: 128-bit ECC.

`DMI_ERROR_DETECT_METHOD_CRC`
: CRC.

Return value

`dmi_error_detect_method_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_memory_controller`, `dmi_error_correct_caps`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

dmi_memory_controller

Synopsis

```
#include <opendmi/entity/memory-controller.h>
```

```
typedef struct dmi_memory_controller {
    dmi_error_detect_method_t error_detection;
    dmi_error_correct_caps_t error_correction;
    dmi_memory_interleave_t supported_interleave;
    dmi_memory_interleave_t current_interleave;
    uint64_t maximum_module_size;
    uint64_t maximum_memory_size;
    dmi_memory_module_speed_t supported_speeds;
    dmi_memory_module_type_t supported_types;
    dmi_memory_module_voltage_t required_voltages;
    size_t slot_count;
    dmi_handle_t *module_handles;
    dmi_entity_t **modules;
    dmi_error_correct_caps_t enabled_error_correction;
} dmi_memory_controller_t;
```

Description

The `dmi_memory_controller_t` structure holds the decoded memory controller information structure (type 5) of SMBIOS: how a memory controller detects and corrects errors, how it interleaves memory, which memory modules it takes, and the memory module information structures (type 6) of its sockets.

The structure is obsolete since SMBIOS 2.1, which describes memory with the physical memory array (type 16) and memory device (type 17) structures instead. Firmware may still provide both, for the sake of older software.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_controller)`, and lives as long as the context it has been decoded in.

Fields

`error_detection`

Error detecting method of the controller, see `dmi_error_detect_method`.

`error_correction`

Error correcting capabilities of the controller, see `dmi_error_correct_caps`.

`supported_interleave`

Interleave mode the controller supports, see `dmi_memory_interleave`.

`current_interleave`

Interleave mode in use, see `dmi_memory_interleave`.

`maximum_module_size`

Size of the largest memory module a socket takes, in bytes. The structure stores it as a power of two of mebibytes; a power too large for 64 bits gives `DMI_SIZE_MAX`, which the output shows as unknown.

`maximum_memory_size`

Largest amount of memory the controller supports, in bytes: `maximum_module_size` times `slot_count`, or `DMI_SIZE_MAX` if the module size is unknown or the product is too large for 64 bits.

The library computes it; the structure does not store it.

`supported_speeds`

Speeds of the memory modules the controller supports, see `dmi_memory_module_speed`.

`supported_types`

Types of the memory modules the controller supports, see `dmi_memory_module_type`.

`required_voltages`

Voltages the sockets of the controller require, see `dmi_memory_module_voltage`.

`slot_count`

Number of elements of `module_handles` and `modules`: the handles the structure holds in full, which may be fewer than it declares when it is truncated.

`module_handles`

Handles of the memory module information structures (type 6) of the sockets the controller drives; `nullptr` if the structure declares none.

`modules`

Memory module information entities the handles of `module_handles` refer to, element by element, see `dmi_memory_module`. An element is `nullptr` if its handle refers to no memory module information structure, and the array is `nullptr` if `slot_count` is 0. Linking also sets the `controller` field of each of these modules to this entity.

`enabled_error_correction`

Error correcting capabilities enabled when the structure was built, see `dmi_error_correct_caps`. Present since SMBIOS 2.1; zero in older data.

Examples

```
dmi_registry_t          *registry = dmi_get_registry(context);
dmi_entity_t            *entity   = dmi_registry_lookup_first(registry,
                                        DMI_TYPE(memory_controller), true);
dmi_memory_controller_t *ctrl     = dmi_entity_info(entity,
                                        DMI_TYPE(memory_controller));

if (ctrl != nullptr)
    printf("%zu sockets, up to %llu MiB each, %s\n", ctrl->slot_count,
           (unsigned long long)(ctrl->maximum_module_size >> 20),
           dmi_memory_interleave_name(ctrl->current_interleave));
```

See also

`dmi_entity_info`, `dmi_error_detect_method`, `dmi_error_correct_caps`, `dmi_memory_interleave`, `dmi_memory_module_speed`, `dmi_memory_module_type`, `dmi_memory_module_voltage`, `dmi_memory_module`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

dmi_memory_interleave

Synopsis

```
#include <opendmi/entity/memory-controller.h>

typedef enum dmi_memory_interleave {
    DMI_MEMORY_INTERLEAVE_UNSPEC  = 0x00,
    DMI_MEMORY_INTERLEAVE_OTHER   = 0x01,
    DMI_MEMORY_INTERLEAVE_UNKNOWN = 0x02,
    DMI_MEMORY_INTERLEAVE_1WAY    = 0x03,
    DMI_MEMORY_INTERLEAVE_2WAY    = 0x04,
    DMI_MEMORY_INTERLEAVE_4WAY    = 0x05,
    DMI_MEMORY_INTERLEAVE_8WAY    = 0x06,
    DMI_MEMORY_INTERLEAVE_16WAY   = 0x07,
} dmi_memory_interleave_t;

const char *dmi_memory_interleave_name(dmi_memory_interleave_t value);
```

Description

The `dmi_memory_interleave_t` enumeration tells how many ways a memory controller interleaves memory accesses, and is the type of the `supported_interleave` and `current_interleave` fields of `dmi_memory_controller_t`.

The `dmi_memory_interleave_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_MEMORY_INTERLEAVE_UNSPEC`
: Unspecified.

`DMI_MEMORY_INTERLEAVE_OTHER`
: Other.

`DMI_MEMORY_INTERLEAVE_UNKNOWN`
: Unknown.

`DMI_MEMORY_INTERLEAVE_1WAY`
: One-way interleave.

`DMI_MEMORY_INTERLEAVE_2WAY`
: Two-way interleave.

`DMI_MEMORY_INTERLEAVE_4WAY`
: Four-way interleave.

`DMI_MEMORY_INTERLEAVE_8WAY`
: Eight-way interleave.

`DMI_MEMORY_INTERLEAVE_16WAY`
: Sixteen-way interleave.

Return value

`dmi_memory_interleave_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_memory_controller`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

dmi_memory_module_speed

Synopsis

```
#include <opendmi/entity/memory-controller.h>

typedef union dmi_memory_module_speed {
    dmi_word_t __value;
    struct {
        dmi_word_t is_other   : 1;
        dmi_word_t is_unknown : 1;
        dmi_word_t is_70ns    : 1;
        dmi_word_t is_60ns    : 1;
        dmi_word_t is_50ns    : 1;
        dmi_word_t __reserved : 11;
    };
} dmi_memory_module_speed_t;
```

Description

The `dmi_memory_module_speed_t` union holds the speeds of the memory modules a memory controller supports as a set of flags, a speed being the access time of the module. It is the type of the `supported_speeds` field of `dmi_memory_controller_t`. A bit is set when the controller supports the speed.

Fields

`__value`
Raw value of the word, `is_other` being bit 0.

`is_other`
Other speed.

`is_unknown`
The speeds are unknown.

`is_70ns`
70 ns.

`is_60ns`
60 ns.

`is_50ns`
50 ns.

`__reserved`
Reserved bits, zero.

See also

`dmi_memory_controller`, `dmi_memory_module_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

dmi_memory_module_voltage

Synopsis

```
#include <opendmi/entity/memory-controller.h>

typedef union dmi_memory_module_voltage {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_5v      : 1;
        dmi_byte_t is_3v3     : 1;
```

```
        dmi_byte_t is_2v9     : 1;
        dmi_byte_t __reserved : 5;
    };
} dmi_memory_module_voltage_t;
```

Description

The `dmi_memory_module_voltage_t` union holds the voltages the memory module sockets of a memory controller require as a set of flags, and is the type of the `required_voltages` field of `dmi_memory_controller_t`. Several bits set mean that the voltage of the sockets can be configured.

Fields

`__value`
: Raw value of the byte, `is_5v` being bit 0.

`is_5v`
: 5 V.

`is_3v3`
: 3.3 V.

`is_2v9`
: 2.9 V.

`__reserved`
: Reserved bits, zero.

See also

`dmi_memory_controller`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer. The memory controller information structure is obsolete since version 2.1.

<opendmi/entity/memory-device-addr.h>

dmi_memory_device_addr

Synopsis

```
#include <opendmi/entity/memory-device-addr.h>

typedef struct dmi_memory_device_addr {
    uint64_t start_address;
    uint64_t end_address;
    uint64_t range_size;
    dmi_handle_t device_handle;
    dmi_entity_t *device;
    dmi_handle_t array_address_handle;
    dmi_entity_t *array_addr;
    unsigned short partition_pos;
    unsigned short interleave_pos;
    unsigned short interleave_depth;
} dmi_memory_device_addr_t;
```

Description

The `dmi_memory_device_addr_t` structure holds the decoded memory device mapped address structure (type 20) of SMBIOS: a contiguous range of physical addresses mapped to a memory device, the memory array mapped address structure (type 19) whose range it is a part of, and the position of the device in the partition and in the interleave. A table holds one structure for each such range; devices without a mapped address have none, since the structure cannot describe an empty range.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_device_addr)`, and lives as long as the context it has been decoded in.

Fields

`start_address`

Physical address of the start of the range, in bytes. The SMBIOS 2.1 field counts in kibibytes; ranges it cannot express are carried by an extended field in bytes, present since SMBIOS 2.7.

`end_address`

Physical address of the last byte of the range. Taken from the SMBIOS 2.1 field, which counts in kibibytes, it is the last byte of the last kibibyte of the range; the extended field (SMBIOS 2.7), which is used when the 2.1 fields hold FFFFFFFFh, gives the last byte itself.

`range_size`

Size of the range, in bytes, computed from `start_address` and `end_address`, both of which belong to the range.

`device_handle`

Handle of the memory device structure (type 17) the range is mapped to. Several ranges can be mapped to the same device.

`device`

The memory device structure `device_handle` refers to, resolved while linking. `nullptr` if the handle is not set or does not refer to a memory device, or before the structures are linked.

`array_address_handle`

Handle of the memory array mapped address structure (type 19) the range is a part of. Several device ranges can belong to the same array range.

`array_address`

The memory array mapped address structure `array_address_handle` refers to, resolved while linking. `nullptr` if the handle is not set or does not refer to such a structure, or before the structures are linked.

`partition_pos`

Position of the device within a row of the address partition, starting at 1: with two 8-bit devices forming a 16-bit row, 1 or 2. Zero is reserved, `USHRT_MAX` means unknown.

`interleave_pos`

Position of the device in the interleave: 0 if the range is not interleaved, 1 for the first position, 2 for the second, and so on. With a 2:1 interleave, 1 is the "even" position; with a 4:1 interleave, 1 is the first of four. `USHRT_MAX` if unknown.

`interleave_depth`

Largest number of consecutive rows of the device read in a single interleaved transfer. Zero if the device is not interleaved, `USHRT_MAX` if the interleave configuration is unknown.

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                           DMI_TYPE(memory_device_addr), false);
dmi_memory_device_addr_t *range    = dmi_entity_info(entity,
                                           DMI_TYPE(memory_device_addr));

if (range != nullptr)
    printf("0x%llX-0x%llX: memory device 0x%04X\n",
           (unsigned long long)range->start_address,
           (unsigned long long)range->end_address,
           (unsigned)range->device_handle);
```

See also

`dmi_entity_info`, `dmi_entity_link`, `dmi_memory_device`, `dmi_memory_array_addr`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-device.h>

dmi_memory_device

Synopsis

```
#include <opendmi/entity/memory-device.h>

typedef struct dmi_memory_device {
    dmi_handle_t                     array_handle;
    dmi_entity_t                    *array;
    dmi_handle_t                     error_info_handle;
    dmi_entity_t                    *error_info;
    unsigned short                   total_width;
    unsigned short                   data_width;
    dmi_size_t                       size;
    dmi_memory_device_form_factor_t  form_factor;
    unsigned short                   device_set;
    const char                      *device_locator;
    const char                      *bank_locator;
    dmi_memory_device_type_t         memory_type;
    dmi_memory_device_type_details_t memory_type_details;
    unsigned long                    maximum_speed;
    const char                      *vendor;
    const char                      *serial_number;
    const char                      *asset_tag;
    const char                      *part_number;
    unsigned short                   rank;
    bool                             is_disabled;
    bool                             is_unmapped;
    unsigned long                    configured_speed;
    unsigned short                   minimum_voltage;
    unsigned short                   maximum_voltage;
    unsigned short                   configured_voltage;
    dmi_memory_device_tech_t         memory_technology;
    uint16_t                         memory_mode_caps;
    const char                      *firmware_version;
    uint16_t                         module_vendor_id;
    uint16_t                         module_product_id;
    uint16_t                         controller_vendor_id;
    uint16_t                         controller_product_id;
    dmi_size_t                       non_volatile_size;
    dmi_size_t                       volatile_size;
    dmi_size_t                       cache_size;
    dmi_size_t                       logical_size;
    uint16_t                         pmic0_vendor_id;
    uint16_t                         pmic0_revision;
```

```
    uint16_t                        rcd_vendor_id;
    uint16_t                        rcd_revision;
    dmi_entity_t                   *channel;
} dmi_memory_device_t;
```

Description

The `dmi_memory_device_t` structure holds the decoded memory device structure (type 17) of SMBIOS: the attributes of one memory device, such as a memory module, belonging to a physical memory array (type 16). A table has one such structure for every memory socket of the system, populated or not.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_device)`, and lives as long as the context it has been decoded in. The structure first appeared in SMBIOS 2.1, and has grown with many later versions; the fields a structure is too short to carry are 0 or `nullptr`.

Fields

`array_handle`

Handle of the physical memory array structure (type 16) the device belongs to.

`array`

Physical memory array entity `array_handle` refers to, set when the table is linked, `nullptr` if there is none.

`error_info_handle`

Handle of the memory error information structure (type 18 or 33) describing the last error detected in the device. 0xFFFE if the firmware provides no error information, `DMI_HANDLE_INVALID` if no error has been detected.

`error_info`

Memory error information entity `error_info_handle` refers to, set when the table is linked, `nullptr` if there is none.

`total_width`

Width of the device, in bits, error-correction bits included; equal to `data_width` if there are none. `USHRT_MAX` if unknown.

`data_width`

Width of the data of the device, in bits, `USHRT_MAX` if unknown. A data width of 0 with a total width of 8 describes a device holding error-correction bits only.

`size`

Size of the device, in bytes. 0 if the socket is empty, `DMI_SIZE_MAX` if unknown. Sizes of 32 GiB - 1 MiB and above are taken from the extended size field, present since SMBIOS 2.7. For a logical non-volatile device, the size is unknown, and `logical_size` gives it.

`form_factor`

Form factor of the device, see `dmi_memory_device_form_factor`.

`device_set`

Number of the set of devices which must all be of the same type and size, unique within the memory

array. 0 if the device belongs to no set, `USHRT_MAX` if unknown.

`device_locator`

Label of the socket or board position holding the device, for example `SIMM 3`, or `nullptr`.

`bank_locator`

Label of the bank holding the device, for example `Bank 0`, or `nullptr`.

`memory_type`

Type of memory the device is made of, see `dmi_memory_device_type`.

`memory_type_details`

Additional details on the type of memory, see `dmi_memory_device_type_details`.

`maximum_speed`

Highest speed the device is capable of, in megatransfers per second (MT/s), 0 if unknown. Present since SMBIOS 2.3; speeds of 65535 MT/s and above are taken from the extended speed field, present since SMBIOS 3.3.

`vendor`

Manufacturer of the device, or `nullptr`. Present since SMBIOS 2.3.

`serial_number`

Serial number of the device, or `nullptr`. Present since SMBIOS 2.3.

`asset_tag`

Asset tag of the device, or `nullptr`. Present since SMBIOS 2.3.

`part_number`

Part number of the device, or `nullptr`. Present since SMBIOS 2.3.

`rank`

Physical or package rank of the device, 0 if unknown. Present since SMBIOS 2.6. The field is four bits wide in the tables of earlier versions than SMBIOS 3.10, which widens it to five bits, so ranks above 15 are found only in the tables of SMBIOS 3.10 and later.

`is_disabled`

`true` if the device has been disabled because of an error, which a 32-bit memory error information structure may describe. Present since SMBIOS 3.10, and `false` in the tables of earlier versions.

`is_unmapped`

`true` if the device is functioning, but has been left out of the memory map to prevent a non-standard configuration, which a memory error information structure of the `DMI_MEMORY_ERROR_TYPE_OK` kind may describe. Present since SMBIOS 3.10, and `false` in the tables of earlier versions.

`configured_speed`

Speed the device is configured to run at, in megatransfers per second (MT/s), 0 if unknown. Present since SMBIOS 2.7; speeds of 65535 MT/s and above are taken from the extended configured speed field, present since SMBIOS 3.3.

`minimum_voltage`

Lowest operating voltage of the device, in millivolts, 0 if unknown. Present since SMBIOS 2.8.

`maximum_voltage`

Highest operating voltage of the device, in millivolts, 0 if unknown. Present since SMBIOS 2.8.

`configured_voltage`

Voltage the device is configured to run at, in millivolts, 0 if unknown. Present since SMBIOS 2.8.

`memory_technology`

Memory technology of the device, see `dmi_memory_device_tech`. Present since SMBIOS 3.2.

`memory_mode_caps`

Operating modes the device supports, as stored: bit 1 for other modes, bit 2 for unknown ones, bit 3 for volatile memory, bit 4 for byte-accessible persistent memory, and bit 5 for block-accessible persistent memory. Present since SMBIOS 3.2.

`firmware_version`

Firmware version of the device, or `nullptr`. Present since SMBIOS 3.2.

`module_vendor_id`

JEDEC JEP106 manufacturer code of the module, as read from its SPD, with the first of its two bytes in the low byte, 0 if unknown. Present since SMBIOS 3.2.

`module_product_id`

Product identifier of the module, as read from its SPD, 0 if unknown. Present since SMBIOS 3.2.

`controller_vendor_id`

JEDEC JEP106 manufacturer code of the memory subsystem controller, as read from the SPD, 0 if unknown. Present since SMBIOS 3.2.

`controller_product_id`

Product identifier of the memory subsystem controller, as read from the SPD, 0 if unknown. Present since SMBIOS 3.2.

`non_volatile_size`

Size of the non-volatile part of the device, in bytes, 0 if there is none, `DMI_SIZE_MAX` if unknown. Present since SMBIOS 3.2.

`volatile_size`

Size of the volatile part of the device, in bytes, 0 if there is none, `DMI_SIZE_MAX` if unknown. Present since SMBIOS 3.2.

`cache_size`

Size of the part of the device used as cache, in bytes, 0 if there is none, `DMI_SIZE_MAX` if unknown. Present since SMBIOS 3.2.

`logical_size`

Size of a logical memory device, in bytes, `DMI_SIZE_MAX` if unknown; 0 for the other devices. Present since SMBIOS 3.2.

`pmic0_vendor_id`

JEDEC JEP106 manufacturer code of the power management IC 0 of the device, as read from the SPD, 0 if unknown. Present since SMBIOS 3.7.

`pmic0_revision`

Revision of the power management IC 0, as read from the SPD, 0xFF00 if unknown. Present since SMBIOS 3.7.

`rcd_vendor_id`

JEDEC JEP106 manufacturer code of the registering clock driver (RCD) of the device, as read from the SPD, 0 if unknown. Present since SMBIOS 3.7.

`rcd_revision`

Revision of the registering clock driver, as read from the SPD, 0xFF00 if unknown. Present since SMBIOS 3.7.

`channel`

Memory channel entity (type 37) listing the device, set when the memory channel structures are linked, `nullptr` if no channel lists it.

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_entity_t        *entity   = dmi_registry_lookup_first(
        registry, DMI_TYPE(memory_device), false);
dmi_memory_device_t *device   = dmi_entity_info(entity,
                                                DMI_TYPE(memory_device));

if ((device != nullptr) && (device->size != 0) &&
    (device->size != DMI_SIZE_MAX)) {
    const char *type = dmi_memory_device_type_name(device->memory_type);

    printf("%s: %llu MiB of %s\n",
           device->device_locator ? device->device_locator : "Unknown",
           (unsigned long long)(device->size >> 20),
           type ? type : "unknown memory");
}
```

See also

`dmi_entity_info`, `dmi_memory_device_type`, `dmi_memory_device_type_details`, `dmi_memory_device_form_factor`, `dmi_memory_device_tech`, `dmi_memory_array`, `dmi_memory_error_32`, `dmi_memory_error_64`, `dmi_memory_channel`, `dmi_memory_device_addr`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_device_form_factor

Synopsis

```
#include <opendmi/entity/memory-device.h>

typedef enum dmi_memory_device_form_factor {
    DMI_MEMORY_DEVICE_FORM_FACTOR_UNSPEC   = 0x00,
    DMI_MEMORY_DEVICE_FORM_FACTOR_OTHER    = 0x01,
    DMI_MEMORY_DEVICE_FORM_FACTOR_UNKNOWN  = 0x02,
    DMI_MEMORY_DEVICE_FORM_FACTOR_SIMM     = 0x03,
    DMI_MEMORY_DEVICE_FORM_FACTOR_SIP      = 0x04,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CHIP     = 0x05,
    DMI_MEMORY_DEVICE_FORM_FACTOR_DIP      = 0x06,
    DMI_MEMORY_DEVICE_FORM_FACTOR_ZIP      = 0x07,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CARD     = 0x08,
    DMI_MEMORY_DEVICE_FORM_FACTOR_DIMM     = 0x09,
    DMI_MEMORY_DEVICE_FORM_FACTOR_TSOP     = 0x0A,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CHIP_ROW = 0x0B,
    DMI_MEMORY_DEVICE_FORM_FACTOR_RIMM     = 0x0C,
    DMI_MEMORY_DEVICE_FORM_FACTOR_SODIMM   = 0x0D,
    DMI_MEMORY_DEVICE_FORM_FACTOR_SRIMM    = 0x0E,
    DMI_MEMORY_DEVICE_FORM_FACTOR_FBDIMM   = 0x0F,
    DMI_MEMORY_DEVICE_FORM_FACTOR_DIE      = 0x10,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CAMM     = 0x11,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CUDIMM   = 0x12,
    DMI_MEMORY_DEVICE_FORM_FACTOR_CSODIMM  = 0x13
} dmi_memory_device_form_factor_t;

const char *dmi_memory_device_form_factor_name(
        dmi_memory_device_form_factor_t value);
```

Description

The `dmi_memory_device_form_factor_t` enumeration tells the physical form of a memory device, and is the type of the `form_factor` field of `dmi_memory_device_t`.

The `dmi_memory_device_form_factor_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_MEMORY_DEVICE_FORM_FACTOR_UNSPEC`

Unspecified.

`DMI_MEMORY_DEVICE_FORM_FACTOR_OTHER`

Other.

`DMI_MEMORY_DEVICE_FORM_FACTOR_UNKNOWN`
: Unknown.

`DMI_MEMORY_DEVICE_FORM_FACTOR_SIMM`
: SIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_SIP`
: SIP.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CHIP`
: Chip.

`DMI_MEMORY_DEVICE_FORM_FACTOR_DIP`
: DIP.

`DMI_MEMORY_DEVICE_FORM_FACTOR_ZIP`
: ZIP.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CARD`
: Proprietary card.

`DMI_MEMORY_DEVICE_FORM_FACTOR_DIMM`
: DIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_TSOP`
: TSOP.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CHIP_ROW`
: Row of chips.

`DMI_MEMORY_DEVICE_FORM_FACTOR_RIMM`
: RIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_SODIMM`
: SODIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_SRIMM`
: SRIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_FBDIMM`
: FBDIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_DIE`
: Die.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CAMM`
: CAMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CUDIMM`
: CUDIMM.

`DMI_MEMORY_DEVICE_FORM_FACTOR_CSODIMM`
: CSODIMM.

Return value

`dmi_memory_device_form_factor_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_device_tech

Synopsis

```
#include <opendmi/entity/memory-device.h>

typedef enum dmi_memory_device_tech {
    DMI_MEMORY_DEVICE_TECH_UNSPEC   = 0x00,
    DMI_MEMORY_DEVICE_TECH_OTHER    = 0x01,
    DMI_MEMORY_DEVICE_TECH_UNKNOWN  = 0x02,
    DMI_MEMORY_DEVICE_TECH_DRAM     = 0x03,
    DMI_MEMORY_DEVICE_TECH_NVDIMM_N = 0x04,
    DMI_MEMORY_DEVICE_TECH_NVDIMM_F = 0x05,
    DMI_MEMORY_DEVICE_TECH_NVDIMM_P = 0x06,
    DMI_MEMORY_DEVICE_TECH_OPTANE   = 0x07,
    DMI_MEMORY_DEVICE_TECH_MRDIMM   = 0x08
} dmi_memory_device_tech_t;

const char *dmi_memory_device_tech_name(dmi_memory_device_tech_t value);
```

Description

The `dmi_memory_device_tech_t` enumeration tells the memory technology of a memory device, volatile or persistent, and is the type of the `memory_technology` field of `dmi_memory_device_t`, present since SMBIOS 3.2.

The `dmi_memory_device_tech_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_MEMORY_DEVICE_TECH_UNSPEC`
: Unspecified.

`DMI_MEMORY_DEVICE_TECH_OTHER`
: Other.

`DMI_MEMORY_DEVICE_TECH_UNKNOWN`
: Unknown.

`DMI_MEMORY_DEVICE_TECH_DRAM`
: DRAM.

`DMI_MEMORY_DEVICE_TECH_NVDIMM_N`
: NVDIMM-N.

`DMI_MEMORY_DEVICE_TECH_NVDIMM_F`
: NVDIMM-F.

`DMI_MEMORY_DEVICE_TECH_NVDIMM_P`
: NVDIMM-P.

`DMI_MEMORY_DEVICE_TECH_OPTANE`
: Intel Optane persistent memory.

`DMI_MEMORY_DEVICE_TECH_MRDIMM`
: MRDIMM. The specification deprecates this value in favour of `DMI_MEMORY_DEVICE_TYPE_MRDIMM` of `dmi_memory_device_type`.

Return value

`dmi_memory_device_tech_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_memory_device`, `dmi_memory_device_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.2 or newer.

dmi_memory_device_type

Synopsis

```
#include <opendmi/entity/memory-device.h>

typedef enum dmi_memory_device_type {
    DMI_MEMORY_DEVICE_TYPE_UNSPEC             = 0x00,
    DMI_MEMORY_DEVICE_TYPE_OTHER              = 0x01,
    DMI_MEMORY_DEVICE_TYPE_UNKNOWN            = 0x02,
    DMI_MEMORY_DEVICE_TYPE_DRAM               = 0x03,
    DMI_MEMORY_DEVICE_TYPE_EDRAM              = 0x04,
    DMI_MEMORY_DEVICE_TYPE_VRAM               = 0x05,
    DMI_MEMORY_DEVICE_TYPE_SRAM               = 0x06,
    DMI_MEMORY_DEVICE_TYPE_RAM                = 0x07,
    DMI_MEMORY_DEVICE_TYPE_ROM                = 0x08,
    DMI_MEMORY_DEVICE_TYPE_FLASH              = 0x09,
    DMI_MEMORY_DEVICE_TYPE_EEPROM             = 0x0A,
    DMI_MEMORY_DEVICE_TYPE_FEPROM             = 0x0B,
    DMI_MEMORY_DEVICE_TYPE_EPROM              = 0x0C,
    DMI_MEMORY_DEVICE_TYPE_CDRAM              = 0x0D,
    DMI_MEMORY_DEVICE_TYPE_3DRAM              = 0x0E,
    DMI_MEMORY_DEVICE_TYPE_SDRAM              = 0x0F,
    DMI_MEMORY_DEVICE_TYPE_SGRAM              = 0x10,
    DMI_MEMORY_DEVICE_TYPE_RDRAM              = 0x11,
    DMI_MEMORY_DEVICE_TYPE_DDR                = 0x12,
    DMI_MEMORY_DEVICE_TYPE_DDR2               = 0x13,
    DMI_MEMORY_DEVICE_TYPE_DDR2_FBDIMM        = 0x14,
    __DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_START = 0x15,
    __DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_END   = 0x17,
    DMI_MEMORY_DEVICE_TYPE_DDR3               = 0x18,
    DMI_MEMORY_DEVICE_TYPE_FBD2               = 0x19,
    DMI_MEMORY_DEVICE_TYPE_DDR4               = 0x1A,
    DMI_MEMORY_DEVICE_TYPE_LPDDR              = 0x1B,
    DMI_MEMORY_DEVICE_TYPE_LPDDR2             = 0x1C,
    DMI_MEMORY_DEVICE_TYPE_LPDDR3             = 0x1D,
    DMI_MEMORY_DEVICE_TYPE_LPDDR4             = 0x1E,
    DMI_MEMORY_DEVICE_TYPE_LOGICAL_NV         = 0x1F,
    DMI_MEMORY_DEVICE_TYPE_HBM                = 0x20,
    DMI_MEMORY_DEVICE_TYPE_HBM2               = 0x21,
    DMI_MEMORY_DEVICE_TYPE_DDR5               = 0x22,
    DMI_MEMORY_DEVICE_TYPE_LPDDR5             = 0x23,
    DMI_MEMORY_DEVICE_TYPE_HBM3               = 0x24,
    DMI_MEMORY_DEVICE_TYPE_MRDIMM             = 0x25,
    DMI_MEMORY_DEVICE_TYPE_LPDDR6             = 0x26
} dmi_memory_device_type_t;
```

```
const char *dmi_memory_device_type_name(dmi_memory_device_type_t value);
```

Description

The `dmi_memory_device_type_t` enumeration tells the type of memory a memory device is made of, and is the type of the `memory_type` field of `dmi_memory_device_t`. The values 0x15 to 0x17 are reserved; the hidden enumerators *`DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_START`* and`DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_END` bound them.

The `dmi_memory_device_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_MEMORY_DEVICE_TYPE_UNSPEC`
: Unspecified.

`DMI_MEMORY_DEVICE_TYPE_OTHER`
: Other.

`DMI_MEMORY_DEVICE_TYPE_UNKNOWN`
: Unknown.

`DMI_MEMORY_DEVICE_TYPE_DRAM`
: DRAM.

`DMI_MEMORY_DEVICE_TYPE_EDRAM`
: EDRAM.

`DMI_MEMORY_DEVICE_TYPE_VRAM`
: VRAM.

`DMI_MEMORY_DEVICE_TYPE_SRAM`
: SRAM.

`DMI_MEMORY_DEVICE_TYPE_RAM`
: RAM.

`DMI_MEMORY_DEVICE_TYPE_ROM`
: ROM.

`DMI_MEMORY_DEVICE_TYPE_FLASH`
: Flash.

`DMI_MEMORY_DEVICE_TYPE_EEPROM`
: EEPROM.

`DMI_MEMORY_DEVICE_TYPE_FEPROM`
: FEPROM.

`DMI_MEMORY_DEVICE_TYPE_EPROM`
: EPROM.

`DMI_MEMORY_DEVICE_TYPE_CDRAM`
: CDRAM.

`DMI_MEMORY_DEVICE_TYPE_3DRAM`
: 3DRAM.

`DMI_MEMORY_DEVICE_TYPE_SDRAM`
: SDRAM.

`DMI_MEMORY_DEVICE_TYPE_SGRAM`
: SGRAM.

`DMI_MEMORY_DEVICE_TYPE_RDRAM`
: RDRAM.

`DMI_MEMORY_DEVICE_TYPE_DDR`
: DDR.

`DMI_MEMORY_DEVICE_TYPE_DDR2`
: DDR2.

`DMI_MEMORY_DEVICE_TYPE_DDR2_FBDIMM`
: DDR2 FB-DIMM.

`__DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_START`
: First reserved value.

`__DMI_MEMORY_DEVICE_TYPE_UNASSIGNED_END`
: Last reserved value.

`DMI_MEMORY_DEVICE_TYPE_DDR3`
: DDR3.

`DMI_MEMORY_DEVICE_TYPE_FBD2`
: FBD2.

`DMI_MEMORY_DEVICE_TYPE_DDR4`
: DDR4.

`DMI_MEMORY_DEVICE_TYPE_LPDDR`
: LPDDR.

`DMI_MEMORY_DEVICE_TYPE_LPDDR2`
: LPDDR2.

`DMI_MEMORY_DEVICE_TYPE_LPDDR3`
: LPDDR3.

`DMI_MEMORY_DEVICE_TYPE_LPDDR4`

LPDDR4.

`DMI_MEMORY_DEVICE_TYPE_LOGICAL_NV`

Logical non-volatile device, which is not installed physically but made of the capacity of volatile devices. Its size is given by the `logical_size` and `non_volatile_size` fields of `dmi_memory_device_t`.

`DMI_MEMORY_DEVICE_TYPE_HBM`

HBM (High Bandwidth Memory).

`DMI_MEMORY_DEVICE_TYPE_HBM2`

HBM2 (High Bandwidth Memory Generation 2).

`DMI_MEMORY_DEVICE_TYPE_DDR5`

DDR5.

`DMI_MEMORY_DEVICE_TYPE_LPDDR5`

LPDDR5.

`DMI_MEMORY_DEVICE_TYPE_HBM3`

HBM3 (High Bandwidth Memory Generation 3).

`DMI_MEMORY_DEVICE_TYPE_MRDIMM`

MRDIMM (multiplexed rank DIMM).

`DMI_MEMORY_DEVICE_TYPE_LPDDR6`

LPDDR6.

Return value

`dmi_memory_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_memory_device`, `dmi_memory_device_type_details`, `dmi_memory_device_tech`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_device_type_details

Synopsis

```
#include <opendmi/entity/memory-device.h>

typedef union dmi_memory_device_type_details {
    dmi_word_t __value;
    struct {
        dmi_word_t __reserved    : 1;
        dmi_word_t other         : 1;
        dmi_word_t unknown       : 1;
        dmi_word_t fast_paged    : 1;
        dmi_word_t static_column : 1;
        dmi_word_t pseudo_static : 1;
        dmi_word_t rambus        : 1;
        dmi_word_t synchronous   : 1;
        dmi_word_t cmos          : 1;
        dmi_word_t edo           : 1;
        dmi_word_t window_dram   : 1;
        dmi_word_t cache_dram    : 1;
        dmi_word_t non_volatile  : 1;
        dmi_word_t registered    : 1;
        dmi_word_t unbuffered    : 1;
        dmi_word_t lrdimm        : 1;
    };
} dmi_memory_device_type_details_t;
```

Description

The `dmi_memory_device_type_details_t` union holds additional details on the type of the memory of a memory device, as the memory device structure (type 17) of SMBIOS gives them. It is the type of the `memory_type_details` field of `dmi_memory_device_t`. Several bits are set when several details apply.

Fields

`__value`
: All the bits as one word, as stored.

`__reserved`
: Reserved bit 0, always 0.

`other`
: Other detail.

`unknown`
: The details are unknown.

`fast_paged`
: Fast-paged memory.

`static_column`
: Static column memory.

`pseudo_static`
: Pseudo-static memory.

`rambus`
: RAMBUS memory.

`synchronous`
: Synchronous memory.

`cmos`
: CMOS memory.

`edo`
: EDO memory.

`window_dram`
: Window DRAM.

`cache_dram`
: Cache DRAM.

`non_volatile`
: Non-volatile memory. The device has non-volatile capacity, which the `non_volatile_size` field of `dmi_memory_device_t` gives.

`registered`
: Registered (buffered) memory.

`unbuffered`
: Unbuffered (unregistered) memory.

`lrdimm`
: Load-reduced DIMM.

See also

`dmi_memory_device`, `dmi_memory_device_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-error-32.h>

dmi_memory_error_32

Synopsis

```
#include <opendmi/entity/memory-error-32.h>

typedef struct dmi_memory_error_32 {
    dmi_memory_error_type_t type;
    dmi_memory_error_granularity_t granularity;
    dmi_memory_error_operation_t operation;
    uint32_t vendor_syndrome;
    uint32_t array_address;
    uint32_t device_address;
    uint32_t resolution;
} dmi_memory_error_32_t;
```

Description

The `dmi_memory_error_32_t` structure holds the decoded 32-bit memory error information structure (type 18) of SMBIOS: the details of an error detected in a physical memory array or in a memory device, i.e. its kind, the access which caused it, and where it happened, with 32-bit addresses. The physical memory array (type 16) and memory device (type 17) structures point at it through their `error_info` fields. The 64-bit memory error information structure (type 33, `dmi_memory_error_64`) carries the same information with 64-bit addresses.

The values of the fields are the ones the structure carries: the special values telling that a value is unknown are kept as they are.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_error_32)` and lives as long as the context it has been decoded in.

Fields

`type`

Kind of error reported for the memory array or device.

`granularity`

How precisely the error can be located: to a device, or to a partition.

`operation`

Memory access operation which caused the error.

`vendor_syndrome`

Vendor-specific ECC syndrome or CRC data of the failing access. 0 when unknown.

`array_address`

Physical address of the error, as the bus the memory array is connected to sees it. `0x80000000` when unknown.

`device_address`

Physical address of the error from the start of the failing memory device, in bytes. `0x80000000` when unknown.

`resolution`

Size of the range, in bytes, within which the error can be located when an address is given. `0x80000000` when unknown.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(memory_error_32), true);
dmi_memory_error_32_t *error    = dmi_entity_info(entity,
                                      DMI_TYPE(memory_error_32));

if (error != nullptr) {
    const char *type = dmi_memory_error_type_name(error->type);

    printf("Memory error: %s\n", type != nullptr ? type : "?");
    if (error->array_address != 0x80000000)
        printf("Array address: 0x%08x\n", (unsigned)error->array_address);
}
```

See also

`dmi_entity_info`, `dmi_memory_error_type`, `dmi_memory_error_granularity`, `dmi_memory_error_operation`, `dmi_memory_error_64`, `dmi_memory_array`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-error-64.h>

dmi_memory_error_64

Synopsis

```
#include <opendmi/entity/memory-error-64.h>

typedef struct dmi_memory_error_64 {
    dmi_memory_error_type_t type;
    dmi_memory_error_granularity_t granularity;
    dmi_memory_error_operation_t operation;
    uint32_t vendor_syndrome;
    uint64_t array_address;
    uint64_t device_address;
    uint32_t resolution;
} dmi_memory_error_64_t;
```

Description

The `dmi_memory_error_64_t` structure holds the decoded 64-bit memory error information structure (type 33) of SMBIOS: the last error detected in a memory array or a memory device, for the systems whose error addresses need 64 bits. The physical memory array structures (type 16) and the memory device structures (type 17) refer to it by its handle. The 32-bit memory error information structure (type 18) is its counterpart for 32-bit addresses, see `dmi_memory_error_32`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_error_64)`, and lives as long as the context it has been decoded in.

Fields

`type`

Kind of the error reported for the array or the device, `DMI_MEMORY_ERROR_TYPE_OK` if there is none. See `dmi_memory_error_type`.

`granularity`

Level the error can be tracked down to, e.g. a device or a partition. See `dmi_memory_error_granularity`.

`operation`

Memory access which caused the error. See `dmi_memory_error_operation`.

`vendor_syndrome`

Vendor-specific ECC syndrome or CRC data of the failed access. Zero if unknown.

`array_address`

Physical address of the error, as the bus the memory array is attached to addresses it. `0x8000000000000000` if unknown.

`device_address`

Physical address of the error, in bytes from the beginning of the failed memory device. `0x8000000000000000` if unknown.

`resolution`

Size of the range the error address is accurate to, in bytes. `0x80000000` if unknown.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(memory_error_64), false);
dmi_memory_error_64_t *error    = dmi_entity_info(entity,
                                      DMI_TYPE(memory_error_64));

if ((error != nullptr) && (error->type != DMI_MEMORY_ERROR_TYPE_OK) &&
    (error->array_address != 0x8000000000000000u))
    printf("%s at 0x%016llx\n", dmi_memory_error_type_name(error->type),
           (unsigned long long)error->array_address);
```

See also

`dmi_entity_info`, `dmi_memory_error_type`, `dmi_memory_error_granularity`, `dmi_memory_error_operation`, `dmi_memory_error_32`, `dmi_memory_array`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/memory-error.h>

dmi_memory_error_granularity

Synopsis

```
#include <opendmi/entity/memory-error.h>

typedef enum dmi_memory_error_granularity {
    DMI_MEMORY_ERROR_GRANULARITY_UNSPEC    = 0x00,
    DMI_MEMORY_ERROR_GRANULARITY_OTHER     = 0x01,
    DMI_MEMORY_ERROR_GRANULARITY_UNKNOWN   = 0x02,
    DMI_MEMORY_ERROR_GRANULARITY_DEVICE    = 0x03,
    DMI_MEMORY_ERROR_GRANULARITY_PARTITION = 0x04,
} dmi_memory_error_granularity_t;

const char *dmi_memory_error_granularity_name(
        dmi_memory_error_granularity_t value);
```

Description

The `dmi_memory_error_granularity_t` enumeration tells how precisely a memory error can be located, and is the type of the `granularity` field of `dmi_memory_error_32_t` and `dmi_memory_error_64_t`.

The `dmi_memory_error_granularity_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_MEMORY_ERROR_GRANULARITY_UNSPEC`

Unspecified. The specification does not define the value 0.

`DMI_MEMORY_ERROR_GRANULARITY_OTHER`

Other granularity.

`DMI_MEMORY_ERROR_GRANULARITY_UNKNOWN`

Unknown granularity.

`DMI_MEMORY_ERROR_GRANULARITY_DEVICE`

Error can be resolved to a memory device.

`DMI_MEMORY_ERROR_GRANULARITY_PARTITION`

Error can be resolved to a memory partition.

Return value

`dmi_memory_error_granularity_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_memory_error_32`, `dmi_memory_error_64`, `dmi_memory_error_type`, `dmi_memory_error_operation`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_error_operation

Synopsis

```
#include <opendmi/entity/memory-error.h>

typedef enum dmi_memory_error_operation {
    DMI_MEMORY_ERROR_OPERATION_UNSPEC        = 0x00,
    DMI_MEMORY_ERROR_OPERATION_OTHER         = 0x01,
    DMI_MEMORY_ERROR_OPERATION_UNKNOWN       = 0x02,
    DMI_MEMORY_ERROR_OPERATION_READ          = 0x03,
    DMI_MEMORY_ERROR_OPERATION_WRITE         = 0x04,
    DMI_MEMORY_ERROR_OPERATION_PARTIAL_WRITE = 0x05,
} dmi_memory_error_operation_t;

const char *dmi_memory_error_operation_name(
        dmi_memory_error_operation_t value);
```

Description

The `dmi_memory_error_operation_t` enumeration tells the memory access operation which caused a memory error, and is the type of the `operation` field of `dmi_memory_error_32_t` and `dmi_memory_error_64_t`.

The `dmi_memory_error_operation_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_MEMORY_ERROR_OPERATION_UNSPEC`
: Unspecified. The specification does not define the value 0.

`DMI_MEMORY_ERROR_OPERATION_OTHER`
: Other operation.

`DMI_MEMORY_ERROR_OPERATION_UNKNOWN`
: Unknown operation.

`DMI_MEMORY_ERROR_OPERATION_READ`
: Read.

`DMI_MEMORY_ERROR_OPERATION_WRITE`
: Write.

`DMI_MEMORY_ERROR_OPERATION_PARTIAL_WRITE`
: Partial write.

Return value

`dmi_memory_error_operation_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_memory_error_32`, `dmi_memory_error_64`, `dmi_memory_error_type`, `dmi_memory_error_granularity`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_memory_error_type

Synopsis

```
#include <opendmi/entity/memory-error.h>

typedef enum dmi_memory_error_type {
    DMI_MEMORY_ERROR_TYPE_UNSPEC               = 0x00,
    DMI_MEMORY_ERROR_TYPE_OTHER                = 0x01,
    DMI_MEMORY_ERROR_TYPE_UNKNOWN              = 0x02,
    DMI_MEMORY_ERROR_TYPE_OK                   = 0x03,
```

```
    DMI_MEMORY_ERROR_TYPE_BAD_READ             = 0x04,
    DMI_MEMORY_ERROR_TYPE_PARITY               = 0x05,
    DMI_MEMORY_ERROR_TYPE_SINGLE_BIT           = 0x06,
    DMI_MEMORY_ERROR_TYPE_DOUBLE_BIT           = 0x07,
    DMI_MEMORY_ERROR_TYPE_MULTI_BIT            = 0x08,
    DMI_MEMORY_ERROR_TYPE_NIBBLE               = 0x09,
    DMI_MEMORY_ERROR_TYPE_CHECKSUM             = 0x0A,
    DMI_MEMORY_ERROR_TYPE_CRC                  = 0x0B,
    DMI_MEMORY_ERROR_TYPE_CORRECTED_SINGLE_BIT = 0x0C,
    DMI_MEMORY_ERROR_TYPE_CORRECTED            = 0x0D,
    DMI_MEMORY_ERROR_TYPE_UNCORRECTABLE        = 0x0E,
} dmi_memory_error_type_t;

const char *dmi_memory_error_type_name(dmi_memory_error_type_t value);
```

Description

The `dmi_memory_error_type_t` enumeration tells the kind of error reported for a memory array or a memory device, and is the type of the `type` field of `dmi_memory_error_32_t` and `dmi_memory_error_64_t`.

The `dmi_memory_error_type_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_MEMORY_ERROR_TYPE_UNSPEC`

Unspecified. The specification does not define the value 0.

`DMI_MEMORY_ERROR_TYPE_OTHER`

Other error.

`DMI_MEMORY_ERROR_TYPE_UNKNOWN`

Unknown error.

`DMI_MEMORY_ERROR_TYPE_OK`

No error. Since SMBIOS 3.10 this value may also describe a memory device which is healthy, but has been left out of the memory map; the memory device structure flags such a device as unmapped. A device flagged as disabled because of an error is not described by this value.

`DMI_MEMORY_ERROR_TYPE_BAD_READ`

Bad read.

`DMI_MEMORY_ERROR_TYPE_PARITY`

Parity error.

`DMI_MEMORY_ERROR_TYPE_SINGLE_BIT`

Single-bit error.

`DMI_MEMORY_ERROR_TYPE_DOUBLE_BIT`
: Double-bit error.

`DMI_MEMORY_ERROR_TYPE_MULTI_BIT`
: Multi-bit error.

`DMI_MEMORY_ERROR_TYPE_NIBBLE`
: Nibble error.

`DMI_MEMORY_ERROR_TYPE_CHECKSUM`
: Checksum error.

`DMI_MEMORY_ERROR_TYPE_CRC`
: CRC error.

`DMI_MEMORY_ERROR_TYPE_CORRECTED_SINGLE_BIT`
: Single-bit error, which has been corrected.

`DMI_MEMORY_ERROR_TYPE_CORRECTED`
: Error, which has been corrected.

`DMI_MEMORY_ERROR_TYPE_UNCORRECTABLE`
: Error, which could not be corrected.

Return value

`dmi_memory_error_type_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_memory_error_32`, `dmi_memory_error_64`, `dmi_memory_error_granularity`, `dmi_memory_error_operation`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/memory-module.h>

dmi_memory_module

Synopsis

```
#include <opendmi/entity/memory-module.h>

typedef struct dmi_memory_module {
    dmi_entity_t *controller;
    const char *socket;
    unsigned short bank_connections[2];
    unsigned short current_speed;
    dmi_memory_module_type_t current_type;
    dmi_memory_module_size_t installed_size;
    dmi_memory_module_size_t enabled_size;
    dmi_memory_module_error_t error_status;
} dmi_memory_module_t;
```

Description

The `dmi_memory_module_t` structure holds the decoded memory module information structure (type 6) of SMBIOS: the speed, type, size and error status of the memory module in one socket. A table carries one such structure for each memory module socket, and the memory controller information structure (type 5) lists them and tells what they support.

Both structures are obsolete since SMBIOS 2.1, which replaces them with the physical memory array (type 16) and memory device (type 17) structures. Some firmware still provides both kinds, for older software.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(memory_module)`, and lives as long as the context it has been decoded in.

Fields

`controller`

Memory controller information structure (type 5) listing this module, or `nullptr` if no memory controller lists it. It is set when the memory controller is linked.

`socket`

Reference designation of the socket, for example "J202", or `nullptr` if the structure has none.

`bank_connections`

Banks (RAS#) the socket is connected to, one per element: element 0 comes from the low nibble of the data, element 1 from the high nibble. 0x0F means no connection. For example, a socket connected to banks 1 and 3 (data 0x13) gives { 3, 1 }, and a socket connected to bank 2 only (data 0x2F) gives { 0x0F, 2 }.

`current_speed`

Speed of the module, in nanoseconds (70 for a 70 ns module), or 0 if unknown.

`current_type`

Physical characteristics of the module, see `dmi_memory_module_type`.

`installed_size`

Size of the module installed in the socket, as its presence-detect information gives it, see `dmi_memory_module_size`. The status is `DMI_MEMORY_MODULE_SIZE_STATUS_NOT_DETERMINABLE` if the system cannot read that information.

`enabled_size`

Amount of memory of the module enabled for the use of the system, see `dmi_memory_module_size`. The status is `DMI_MEMORY_MODULE_SIZE_STATUS_NOT_ENABLED` if a module is installed but all of its memory is disabled, for example because of errors.

`error_status`

Error status of the module, see `dmi_memory_module_error`.

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_entity_t        *entity   = dmi_registry_lookup_first(registry,
                                    DMI_TYPE(memory_module), false);
dmi_memory_module_t *module   = dmi_entity_info(entity,
                                    DMI_TYPE(memory_module));

if ((module != nullptr) &&
    (module->installed_size.status == DMI_MEMORY_MODULE_SIZE_STATUS_PRESENT))
    printf("%s: %llu MiB\n", module->socket ? module->socket : "?",
           (unsigned long long)(module->installed_size.value >> 20));
```

See also

`dmi_entity_info`, `dmi_memory_module_type`, `dmi_memory_module_size`, `dmi_memory_module_size_status`, `dmi_memory_module_error`, `dmi_memory_controller`, `dmi_memory_array`, `dmi_memory_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_memory_module_error

Synopsis

```
#include <opendmi/entity/memory-module.h>
```

```
typedef union dmi_memory_module_error {
    dmi_byte_t __value;

    struct {
        dmi_byte_t has_uncorrectable : 1;
        dmi_byte_t has_correctable   : 1;
        dmi_byte_t has_event_log     : 1;
        dmi_byte_t __reserved        : 5;
    };
} dmi_memory_module_error_t;
```

Description

The `dmi_memory_module_error_t` union holds the error status of a memory module as a set of bits, and is the type of the `error_status` field of `dmi_memory_module_t`. The union gives access to the whole byte as well as to the single bits.

Fields

`__value`
: Whole byte, as the data carries it.

`has_uncorrectable`
: 1 if the module has had uncorrectable errors, and all or part of it has been disabled. The bit is cleared only at power-on.

`has_correctable`
: 1 if the module has had correctable errors. The bit is cleared only at system reset.

`has_event_log`
: 1 if the error status is to be read from the event log. The two bits above carry no information then.

`__reserved`
: Bits 3 to 7, reserved, and zero.

See also

`dmi_memory_module`, `dmi_system_event_log`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_memory_module_size

Synopsis

```
#include <opendmi/entity/memory-module.h>
```

```
typedef struct dmi_memory_module_size {
    dmi_size_t value;
    unsigned bank_count;
    dmi_memory_module_size_status_t status;
} dmi_memory_module_size_t;
```

Description

The `dmi_memory_module_size_t` structure holds a decoded memory size of a memory module, and is the type of the `installed_size` and `enabled_size` fields of `dmi_memory_module_t`. The data carries the size as a power of two of mebibytes, or a special value telling why there is no size, together with the bank connection of the module.

Fields

`value`

Size, in bytes, or 0 unless `status` is `DMI_MEMORY_MODULE_SIZE_STATUS_PRESENT`.

`bank_count`

Number of bank connections of the module: 1 for a single-bank connection, 2 for a double-bank one. Set whatever the status.

`status`

Whether the size is known, and why not if it is not, see `dmi_memory_module_size_status`.

See also

`dmi_memory_module`, `dmi_memory_module_size_status`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_memory_module_size_status

Synopsis

```
#include <opendmi/entity/memory-module.h>

typedef enum dmi_memory_module_size_status {
    DMI_MEMORY_MODULE_SIZE_STATUS_PRESENT,
    DMI_MEMORY_MODULE_SIZE_STATUS_INVALID,
    DMI_MEMORY_MODULE_SIZE_STATUS_NOT_DETERMINABLE,
    DMI_MEMORY_MODULE_SIZE_STATUS_NOT_ENABLED,
    DMI_MEMORY_MODULE_SIZE_STATUS_NOT_INSTALLED,
} dmi_memory_module_size_status_t;
```

```
const char *dmi_memory_module_size_status_name(
        dmi_memory_module_size_status_t value);
```

Description

The `dmi_memory_module_size_status_t` enumeration tells whether the size of a memory module is known, and why not if it is not. It is the type of the `status` field of `dmi_memory_module_size_t`. The data carries these statuses as special values of the size; the enumeration and its values belong to the library.

The `dmi_memory_module_size_status_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MEMORY_MODULE_SIZE_STATUS_PRESENT`
: The size is known, and is in the `value` field.

`DMI_MEMORY_MODULE_SIZE_STATUS_INVALID`
: The data holds a size too large to be represented: 2 to the power of 44 MiB or more, other than the special values below.

`DMI_MEMORY_MODULE_SIZE_STATUS_NOT_DETERMINABLE`
: The size cannot be determined, for example because the system has no means of reading the presence-detect information of the module (data 0x7D). Used for the installed size only.

`DMI_MEMORY_MODULE_SIZE_STATUS_NOT_ENABLED`
: A module is installed, but none of its memory is enabled (data 0x7E).

`DMI_MEMORY_MODULE_SIZE_STATUS_NOT_INSTALLED`
: No module is installed (data 0x7F).

Return value

`dmi_memory_module_size_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_memory_module_size`, `dmi_memory_module`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_memory_module_type

Synopsis

```
#include <opendmi/entity/memory-module.h>

typedef union dmi_memory_module_type {
    dmi_word_t __value;

    struct {
        dmi_word_t is_other      : 1;
        dmi_word_t is_unknown    : 1;
        dmi_word_t is_standard   : 1;
        dmi_word_t is_fast_page  : 1;
        dmi_word_t is_edo        : 1;
        dmi_word_t has_parity    : 1;
        dmi_word_t has_ecc       : 1;
        dmi_word_t is_simm       : 1;
        dmi_word_t is_dimm       : 1;
        dmi_word_t is_burst_edo  : 1;
        dmi_word_t is_sdram      : 1;
        dmi_word_t __reserved    : 5;
    };
} dmi_memory_module_type_t;
```

Description

The `dmi_memory_module_type_t` union holds a set of the physical characteristics of memory modules, one bit per characteristic. It is the type of the `current_type` field of `dmi_memory_module_t`, and of the `supported_types` field of `dmi_memory_controller_t`. A bit is 1 if the characteristic is in the set. The union gives access to the whole word as well as to the single bits.

Fields

`__value`
: Whole 16-bit word, as the data carries it.

`is_other`
: Other type.

`is_unknown`
: Unknown type.

`is_standard`
: Standard.

`is_fast_page`
: Fast page mode.

`is_edo`
: EDO (extended data out).

`has_parity`
: Parity.

`has_ecc`
: ECC.

`is_simm`
: SIMM.

`is_dimm`
: DIMM.

`is_burst_edo`
: Burst EDO.

`is_sdram`
: SDRAM.

`__reserved`
: Bits 11 to 15, reserved, and zero.

See also

`dmi_memory_module`, `dmi_memory_controller`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/mgmt-controller.h>

dmi_mgmt_controller

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_controller {
    dmi_mgmt_iface_type_t interface_type;
    dmi_binary_t interface_data;
    bool has_nhi;
    dmi_mgmt_nhi_t nhi;
    size_t proto_records_count;
    dmi_mgmt_proto_record_t *proto_records;
} dmi_mgmt_controller_t;
```

Description

The `dmi_mgmt_controller_t` structure holds the decoded management controller host interface structure (type 42) of SMBIOS: an interface the host cannot discover by itself, through which it talks to a management controller such as a BMC, and the protocols running over that interface.

A table has one such structure per interface. An interface carrying IPMI alone may be described by the IPMI device information structure (type 38) instead, or by both of them, see `dmi_ipmi_device`.

The layout of the structure has been changed in SMBIOS 3.2, which added the length of the interface data. Structures whose interface data does not fit the length they declare, such as the ones written to the older layout, are decoded up to the interface type only.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(mgmt_controller)`, and lives as long as the context it has been decoded in.

Fields

`interface_type`

Type of the interface: one of the MCTP host interface types, the network host interface, or an OEM-defined one. See `dmi_mgmt_iface_type`.

`interface_data`

Data of the interface, as the structure carries it; its layout depends on `interface_type`. Empty if the structure has none, or if it cannot hold the length it declares.

`has_nhi`

`true` if the interface is a network host interface (`DMI_MGMT_IFACE_TYPE_NETWORK_HOST`) and its data is not empty, in which case `nhi` holds the data decoded.

`nhi`

Data of the interface, decoded as network host interface data, see `dmi_mgmt_nhi`. Zeroed unless `has_nhi` is set.

`proto_records_count`

Number of the elements of `proto_records`, which is the number of the protocol records the structure holds in full. Smaller than the number the structure declares if the structure ends in the middle of a record. Present since SMBIOS 3.2; zero for the older structures.

`proto_records`

Protocol records of the interface, one per protocol running over it, see `dmi_mgmt_proto_record`. `nullptr` if the structure declares none. Present since SMBIOS 3.2.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(mgmt_controller),
                                      false);
dmi_mgmt_controller_t *mc       = dmi_entity_info(entity,
                                      DMI_TYPE(mgmt_controller));

for (size_t i = 0; (mc != nullptr) && (i < mc->proto_records_count); i++) {
    const dmi_mgmt_proto_record_t *record = &mc->proto_records[i];

    if (record->has_redfish && (record->redfish.service_hostname != nullptr))
        printf("Redfish service: %s\n", record->redfish.service_hostname);
}
```

See also

`dmi_entity_info`, `dmi_mgmt_iface_type`, `dmi_mgmt_nhi`, `dmi_mgmt_proto_record`, `dmi_ipmi_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.7 or newer.

dmi_mgmt_iface_type

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_iface_type {
    DMI_MGMT_IFACE_TYPE_MCTP_KCS        = 0x02,
    DMI_MGMT_IFACE_TYPE_MCTP_8250_UART  = 0x03,
```

```
    DMI_MGMT_IFACE_TYPE_MCTP_16450_UART = 0x04,
    DMI_MGMT_IFACE_TYPE_MCTP_16550_UART = 0x05,
    DMI_MGMT_IFACE_TYPE_MCTP_16650_UART = 0x06,
    DMI_MGMT_IFACE_TYPE_MCTP_16750_UART = 0x07,
    DMI_MGMT_IFACE_TYPE_MCTP_16850_UART = 0x08,
    DMI_MGMT_IFACE_TYPE_MCTP_I2C_SMBUS  = 0x09,
    DMI_MGMT_IFACE_TYPE_MCTP_I3C        = 0x0A,
    DMI_MGMT_IFACE_TYPE_MCTP_PCIE_VDM   = 0x0B,
    DMI_MGMT_IFACE_TYPE_MCTP_MMBI       = 0x0C,
    DMI_MGMT_IFACE_TYPE_MCTP_PCC        = 0x0D,
    DMI_MGMT_IFACE_TYPE_MCTP_UCIE       = 0x0E,
    DMI_MGMT_IFACE_TYPE_MCTP_USB        = 0x0F,
    DMI_MGMT_IFACE_TYPE_NETWORK_HOST    = 0x40,
    DMI_MGMT_IFACE_TYPE_OEM             = 0xF0
} dmi_mgmt_iface_type_t;

const char *dmi_mgmt_iface_type_name(dmi_mgmt_iface_type_t value);
```

Description

The `dmi_mgmt_iface_type_t` enumeration tells the type of a management controller host interface, and is the type of the `interface_type` field of `dmi_mgmt_controller_t`. The values from 0x00 to 0x3F are MCTP host interface types, which the MCTP host interface specification (DSP0239) assigns; the enumeration names the ones assigned so far. Every value the enumeration does not name is reserved, and the field holds it as the structure carries it.

The `dmi_mgmt_iface_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_IFACE_TYPE_MCTP_KCS`
MCTP over a keyboard controller style (KCS) interface.

`DMI_MGMT_IFACE_TYPE_MCTP_8250_UART`
MCTP over a serial port compatible with the 8250 UART.

`DMI_MGMT_IFACE_TYPE_MCTP_16450_UART`
MCTP over a serial port compatible with the 16450 UART.

`DMI_MGMT_IFACE_TYPE_MCTP_16550_UART`
MCTP over a serial port compatible with the 16550 or 16550A UART.

`DMI_MGMT_IFACE_TYPE_MCTP_16650_UART`
MCTP over a serial port compatible with the 16650 or 16650A UART.

`DMI_MGMT_IFACE_TYPE_MCTP_16750_UART`
MCTP over a serial port compatible with the 16750 or 16750A UART.

`DMI_MGMT_IFACE_TYPE_MCTP_16850_UART`
: MCTP over a serial port compatible with the 16850 or 16850A UART.

`DMI_MGMT_IFACE_TYPE_MCTP_I2C_SMBUS`
: MCTP over I2C or SMBus.

`DMI_MGMT_IFACE_TYPE_MCTP_I3C`
: MCTP over I3C.

`DMI_MGMT_IFACE_TYPE_MCTP_PCIE_VDM`
: MCTP over PCIe vendor-defined messages.

`DMI_MGMT_IFACE_TYPE_MCTP_MMBI`
: MCTP over a memory-mapped BMC interface (MMBI).

`DMI_MGMT_IFACE_TYPE_MCTP_PCC`
: MCTP over an ACPI platform communication channel (PCC).

`DMI_MGMT_IFACE_TYPE_MCTP_UCIE`
: MCTP over UCIe.

`DMI_MGMT_IFACE_TYPE_MCTP_USB`
: MCTP over USB.

`DMI_MGMT_IFACE_TYPE_NETWORK_HOST`
: Network host interface, defined by the Redfish host interface specification (DSP0270). Its data is decoded into `dmi_mgmt_nhi_t`.

`DMI_MGMT_IFACE_TYPE_OEM`
: OEM-defined interface.

Return value

`dmi_mgmt_iface_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_controller`, `dmi_mgmt_nhi`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.7 or newer.

dmi_mgmt_nhi

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi {
    dmi_mgmt_nhi_device_type_t device_type;
    dmi_mgmt_nhi_format_t format;
    dmi_binary_t descriptor;
    dmi_mgmt_nhi_usb_t usb;
    dmi_mgmt_nhi_pci_t pci;
    dmi_mgmt_nhi_usb_v2_t usb_v2;
    dmi_mgmt_nhi_pci_v2_t pci_v2;
    dmi_mgmt_nhi_oem_t oem;
} dmi_mgmt_nhi_t;
```

Description

The `dmi_mgmt_nhi_t` structure holds the data of a management controller host interface of the network host interface type (`DMI_MGMT_IFACE_TYPE_NETWORK_HOST`), as the Redfish host interface specification (DSP0270) lays it out: the type of the device through which the host reaches the management controller, and the descriptor identifying that device. It is the type of the `nhi` field of `dmi_mgmt_controller_t`, which is valid when `has_nhi` is set.

The descriptor is decoded into the one of the members `usb`, `pci`, `usb_v2`, `pci_v2` and `oem` which `format` names. A descriptor of an unknown type, or one too short for its type, is not decoded, and `format` is `DMI_MGMT_NHI_FORMAT_RAW`.

Fields

`device_type`

Type of the device, see `dmi_mgmt_nhi_device_type`. The values from 0x80 to 0xFF are OEM-defined.

`format`

Member holding the decoded descriptor, see `dmi_mgmt_nhi_format`. `DMI_MGMT_NHI_FORMAT_RAW` if the descriptor has not been decoded, in which case only `descriptor` is to be used.

`descriptor`

Descriptor of the device, the bytes following the device type, as the structure carries them.

`usb`

Descriptor of a USB network interface, valid if `format` is `DMI_MGMT_NHI_FORMAT_USB`. See `dmi_mgmt_nhi_usb`.

`pci`

Descriptor of a PCI or PCIe network interface, valid if `format` is `DMI_MGMT_NHI_FORMAT_PCI`. See `dmi_mgmt_nhi_pci`.

`usb_v2`

Descriptor of a USB network interface, second version, valid if `format` is `DMI_MGMT_NHI_FORMAT_USB_V2`. See `dmi_mgmt_nhi_usb_v2`.

`pci_v2`

Descriptor of a PCI or PCIe network interface, second version, valid if `format` is `DMI_MGMT_NHI_FORMAT_PCI_V2`. See `dmi_mgmt_nhi_pci_v2`.

`oem`

Descriptor of an OEM-defined device, valid if `format` is `DMI_MGMT_NHI_FORMAT_OEM`. See `dmi_mgmt_nhi_oem`.

See also

`dmi_mgmt_controller`, `dmi_mgmt_nhi_device_type`, `dmi_mgmt_nhi_format`, `dmi_mgmt_nhi_usb`, `dmi_mgmt_nhi_pci`, `dmi_mgmt_nhi_usb_v2`, `dmi_mgmt_nhi_pci_v2`, `dmi_mgmt_nhi_oem`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_char

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_nhi_char {
    DMI_MGMT_NHI_CHAR_CREDENTIAL_BOOTSTRAPPING = 0
} dmi_mgmt_nhi_char_t;

const char *dmi_mgmt_nhi_char_name(dmi_mgmt_nhi_char_t value);
```

Description

The `dmi_mgmt_nhi_char_t` enumeration numbers the bits of the `characteristics` field of `dmi_mgmt_nhi_usb_v2_t` and `dmi_mgmt_nhi_pci_v2_t`, which tell the characteristics of the device through which the host reaches the management controller. The bits the enumeration does not name are reserved.

A characteristic is present if the bit it numbers is set, e.g. `(characteristics & (1u << DMI_MGMT_NHI_CHAR_CREDENTIAL_BOOTSTRAPPING)) != 0`.

The `dmi_mgmt_nhi_char_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_NHI_CHAR_CREDENTIAL_BOOTSTRAPPING`
: Credential bootstrapping via IPMI commands is supported; the `credential_bootstrapping_handle` field of the descriptor names the interface to use for it.

Return value

`dmi_mgmt_nhi_char_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_nhi_usb_v2`, `dmi_mgmt_nhi_pci_v2`

Standards

DMTF Redfish Host Interface Specification (DSP0270), version 1.3 or newer.

dmi_mgmt_nhi_device_type

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_nhi_device_type {
    DMI_MGMT_NHI_DEVICE_TYPE_USB       = 0x02,
    DMI_MGMT_NHI_DEVICE_TYPE_PCI       = 0x03,
    DMI_MGMT_NHI_DEVICE_TYPE_USB_V2    = 0x04,
    DMI_MGMT_NHI_DEVICE_TYPE_PCI_V2    = 0x05,
    DMI_MGMT_NHI_DEVICE_TYPE_OEM_START = 0x80,
    DMI_MGMT_NHI_DEVICE_TYPE_OEM_END   = 0xFF
} dmi_mgmt_nhi_device_type_t;

const char *dmi_mgmt_nhi_device_type_name(dmi_mgmt_nhi_device_type_t value);
```

Description

The `dmi_mgmt_nhi_device_type_t` enumeration tells the type of the device through which the host reaches the management controller over a network host interface, and is the type of the `device_type` field of `dmi_mgmt_nhi_t`. The values from 0x80 to 0xFF are OEM-defined, and the other values the enumeration does not name are reserved.

The `dmi_mgmt_nhi_device_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built

with ICU4C support and has a translation of it.

Values

`DMI_MGMT_NHI_DEVICE_TYPE_USB`
: USB network interface, see `dmi_mgmt_nhi_usb`.

`DMI_MGMT_NHI_DEVICE_TYPE_PCI`
: PCI or PCIe network interface, see `dmi_mgmt_nhi_pci`.

`DMI_MGMT_NHI_DEVICE_TYPE_USB_V2`
: USB network interface, second version of the descriptor, see `dmi_mgmt_nhi_usb_v2`.

`DMI_MGMT_NHI_DEVICE_TYPE_PCI_V2`
: PCI or PCIe network interface, second version of the descriptor, see `dmi_mgmt_nhi_pci_v2`.

`DMI_MGMT_NHI_DEVICE_TYPE_OEM_START`
: First OEM-defined device type, see `dmi_mgmt_nhi_oem`.

`DMI_MGMT_NHI_DEVICE_TYPE_OEM_END`
: Last OEM-defined device type.

Return value

`dmi_mgmt_nhi_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_format`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_format

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_nhi_format {
    DMI_MGMT_NHI_FORMAT_RAW    = 0,
    DMI_MGMT_NHI_FORMAT_USB    = 1,
    DMI_MGMT_NHI_FORMAT_PCI    = 2,
```

```
    DMI_MGMT_NHI_FORMAT_USB_V2 = 3,
    DMI_MGMT_NHI_FORMAT_PCI_V2 = 4,
    DMI_MGMT_NHI_FORMAT_OEM    = 5
} dmi_mgmt_nhi_format_t;
```

Description

The `dmi_mgmt_nhi_format_t` enumeration tells which member of `dmi_mgmt_nhi_t` holds the decoded descriptor of a network host interface device, and is the type of the `format` field of `dmi_mgmt_nhi_t`. The values are not carried by the data: they are chosen while the descriptor is decoded, after its device type and its length.

Values

`DMI_MGMT_NHI_FORMAT_RAW`
: The descriptor has not been decoded, because its device type is unknown or it is malformed. Only the `descriptor` field holds it.

`DMI_MGMT_NHI_FORMAT_USB`
: USB network interface, decoded into the `usb` field.

`DMI_MGMT_NHI_FORMAT_PCI`
: PCI or PCIe network interface, decoded into the `pci` field.

`DMI_MGMT_NHI_FORMAT_USB_V2`
: USB network interface, second version of the descriptor, decoded into the `usb_v2` field.

`DMI_MGMT_NHI_FORMAT_PCI_V2`
: PCI or PCIe network interface, second version of the descriptor, decoded into the `pci_v2` field.

`DMI_MGMT_NHI_FORMAT_OEM`
: OEM-defined device, decoded into the `oem` field.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_device_type`

Standards

Not part of any specification. The formats are defined by the library, after the device types of the DMTF Redfish Host Interface Specification (DSP0270).

dmi_mgmt_nhi_oem

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi_oem {
```

```
    uint32_t vendor_iana;
    dmi_binary_t vendor_data;
} dmi_mgmt_nhi_oem_t;
```

Description

The `dmi_mgmt_nhi_oem_t` structure holds the descriptor of an OEM-defined device (device types 0x80 to 0xFF) through which the host reaches the management controller. It is the type of the `oem` field of `dmi_mgmt_nhi_t`, which is valid when the `format` of it is `DMI_MGMT_NHI_FORMAT_OEM`.

Fields

`vendor_iana`

IANA enterprise number of the vendor defining the device. The descriptor carries it with the most significant byte first; the field holds it as a number of the host.

`vendor_data`

Rest of the descriptor, defined by the vendor, referring to the data of the structure in place.

See also

`dmi_mgmt_nhi`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_pci

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi_pci {
    uint16_t vendor_id;
    uint16_t device_id;
    uint16_t subsystem_vendor_id;
    uint16_t subsystem_id;
} dmi_mgmt_nhi_pci_t;
```

Description

The `dmi_mgmt_nhi_pci_t` structure holds the descriptor of a PCI or PCIe network interface (device type 0x03) through which the host reaches the management controller. It is the type of the `pci` field of `dmi_mgmt_nhi_t`, which is valid when the `format` of it is `DMI_MGMT_NHI_FORMAT_PCI`.

Fields

`vendor_id`
: PCI vendor ID of the device.

`device_id`
: PCI device ID of the device.

`subsystem_vendor_id`
: PCI subsystem vendor ID of the device.

`subsystem_id`
: PCI subsystem ID of the device.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_pci_v2`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_pci_v2

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi_pci_v2 {
    uint16_t vendor_id;
    uint16_t device_id;
    uint16_t subsystem_vendor_id;
    uint16_t subsystem_id;
    dmi_binary_t mac_address;
    uint16_t segment_group;
    uint8_t bus_number;
    uint8_t device_number;
    uint8_t function_number;
    uint16_t characteristics;
    dmi_handle_t credential_bootstrapping_handle;
} dmi_mgmt_nhi_pci_v2_t;
```

Description

The `dmi_mgmt_nhi_pci_v2_t` structure holds the second version of the descriptor of a PCI or PCIe network interface (device type 0x05) through which the host reaches the management controller, which adds the MAC address and the location of the device on the bus. It is the type of the `pci_v2` field of

`dmi_mgmt_nhi_t`, which is valid when the `format` of it is `DMI_MGMT_NHI_FORMAT_PCI_V2`.

Fields

`vendor_id`
PCI vendor ID of the device.

`device_id`
PCI device ID of the device.

`subsystem_vendor_id`
PCI subsystem vendor ID of the device.

`subsystem_id`
PCI subsystem ID of the device.

`mac_address`
MAC address of the device, six bytes referring to the data of the structure in place.

`segment_group`
PCI segment group number of the device, zero on the systems with a single segment.

`bus_number`
PCI bus number of the device.

`device_number`
PCI device number of the device, 0 to 31.

`function_number`
PCI function number of the device, 0 to 7.

`characteristics`
Device characteristics, a set of the bits `dmi_mgmt_nhi_char` numbers. Zero if the descriptor is too short to hold them, as the descriptors older than DSP0270 1.3 are.

`credential_bootstrapping_handle`
Handle of the interface to be used for credential bootstrapping via IPMI commands. `DMI_HANDLE_INVALID` if bootstrapping is not supported, or if the descriptor is too short to hold the handle, as the descriptors older than DSP0270 1.3 are.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_char`, `dmi_mgmt_nhi_pci`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_usb

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi_usb {
    uint16_t vendor_id;
    uint16_t product_id;
    char *serial_number;
} dmi_mgmt_nhi_usb_t;
```

Description

The `dmi_mgmt_nhi_usb_t` structure holds the descriptor of a USB network interface (device type 0x02) through which the host reaches the management controller. It is the type of the `usb` field of `dmi_mgmt_nhi_t`, which is valid when the `format` of it is `DMI_MGMT_NHI_FORMAT_USB`.

Fields

`vendor_id`

Vendor ID of the device, as its USB device descriptor gives it (`idVendor`).

`product_id`

Product ID of the device, as its USB device descriptor gives it (`idProduct`).

`serial_number`

Serial number of the device, converted from the UTF-16 string the descriptor carries to UTF-8, unpaired surrogates being replaced with U+FFFD. `nullptr` if the string is empty. The string is allocated by the library, and is freed along with the structure.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_usb_v2`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_nhi_usb_v2

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_nhi_usb_v2 {
```

```
    uint16_t vendor_id;
    uint16_t product_id;
    const char *serial_number;
    dmi_binary_t mac_address;
    uint16_t characteristics;
    dmi_handle_t credential_bootstrapping_handle;
} dmi_mgmt_nhi_usb_v2_t;
```

Description

The `dmi_mgmt_nhi_usb_v2_t` structure holds the second version of the descriptor of a USB network interface (device type 0x04) through which the host reaches the management controller. It is the type of the `usb_v2` field of `dmi_mgmt_nhi_t`, which is valid when the `format` of it is `DMI_MGMT_NHI_FORMAT_USB_V2`.

Fields

`vendor_id`

Vendor ID of the device, as its USB device descriptor gives it (`idVendor`).

`product_id`

Product ID of the device, as its USB device descriptor gives it (`idProduct`).

`serial_number`

Serial number of the device, one of the strings of the SMBIOS structure, or `nullptr` if the descriptor refers to none.

`mac_address`

MAC address of the device, six bytes referring to the data of the structure in place.

`characteristics`

Device characteristics, a set of the bits `dmi_mgmt_nhi_char` numbers. Zero if the descriptor is too short to hold them, as the descriptors older than DSP0270 1.3 are.

`credential_bootstrapping_handle`

Handle of the interface to be used for credential bootstrapping via IPMI commands. `DMI_HANDLE_INVALID` if bootstrapping is not supported, or if the descriptor is too short to hold the handle, as the descriptors older than DSP0270 1.3 are.

See also

`dmi_mgmt_nhi`, `dmi_mgmt_nhi_char`, `dmi_mgmt_nhi_usb`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_proto

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_proto {
    DMI_MGMT_PROTO_IPMI            = 0x02,
    DMI_MGMT_PROTO_MCTP            = 0x03,
    DMI_MGMT_PROTO_REDFISH_OVER_IP = 0x04,
    DMI_MGMT_PROTO_OEM             = 0xF0
} dmi_mgmt_proto_t;

const char *dmi_mgmt_proto_name(dmi_mgmt_proto_t value);
```

Description

The `dmi_mgmt_proto_t` enumeration tells the protocol a protocol record of a management controller host interface describes, and is the type of the `type` field of `dmi_mgmt_proto_record_t`. Every value the enumeration does not name is reserved, and the field holds it as the structure carries it.

The `dmi_mgmt_proto_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_PROTO_IPMI`

Intelligent Platform Management Interface (IPMI).

`DMI_MGMT_PROTO_MCTP`

Management Component Transport Protocol (MCTP), defined by DSP0236.

`DMI_MGMT_PROTO_REDFISH_OVER_IP`

Redfish over IP, defined by the Redfish host interface specification (DSP0270). Its data is decoded into `dmi_mgmt_redfish_over_ip_t`.

`DMI_MGMT_PROTO_OEM`

OEM-defined protocol.

Return value

`dmi_mgmt_proto_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_proto_record`, `dmi_mgmt_redfish_over_ip`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.2 or newer.

dmi_mgmt_proto_record

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_proto_record {
    dmi_mgmt_proto_t type;
    dmi_binary_t data;
    bool has_redfish;
    dmi_mgmt_redfish_over_ip_t redfish;
} dmi_mgmt_proto_record_t;
```

Description

The `dmi_mgmt_proto_record_t` structure holds a protocol record of a management controller host interface: a protocol running over the interface, and the data describing it. The records are the elements of the `proto_records` field of `dmi_mgmt_controller_t`, and are present since SMBIOS 3.2.

Fields

`type`
: Protocol, see `dmi_mgmt_proto`.

`data`
: Data of the protocol, as the record carries it; its layout depends on `type`.

`has_redfish`
: `true` if the protocol is Redfish over IP (`DMI_MGMT_PROTO_REDFISH_OVER_IP`) and its data is long enough to be decoded, in which case `redfish` holds the data decoded.

`redfish`
: Data of the protocol, decoded as Redfish over IP data, see `dmi_mgmt_redfish_over_ip`. Valid only if `has_redfish` is set.

See also

`dmi_mgmt_controller`, `dmi_mgmt_proto`, `dmi_mgmt_redfish_over_ip`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.2 or newer.

dmi_mgmt_redfish_ip_assignment

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_redfish_ip_assignment {
    DMI_MGMT_REDFISH_IP_ASSIGNMENT_UNKNOWN        = 0x00,
    DMI_MGMT_REDFISH_IP_ASSIGNMENT_STATIC         = 0x01,
    DMI_MGMT_REDFISH_IP_ASSIGNMENT_DHCP           = 0x02,
    DMI_MGMT_REDFISH_IP_ASSIGNMENT_AUTO_CONFIGURE = 0x03,
    DMI_MGMT_REDFISH_IP_ASSIGNMENT_HOST_SELECTED  = 0x04
} dmi_mgmt_redfish_ip_assignment_t;

const char *dmi_mgmt_redfish_ip_assignment_name(
        dmi_mgmt_redfish_ip_assignment_t value);
```

Description

The `dmi_mgmt_redfish_ip_assignment_t` enumeration tells how the host gets its IP address on a Redfish host interface, and how it finds the IP address of the Redfish service. It is the type of the `host_ip_assignment` and `service_ip_discovery` fields of `dmi_mgmt_redfish_over_ip_t`. The values the enumeration does not name are reserved.

The `dmi_mgmt_redfish_ip_assignment_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_REDFISH_IP_ASSIGNMENT_UNKNOWN`
: Unknown.

`DMI_MGMT_REDFISH_IP_ASSIGNMENT_STATIC`
: Static: the address is the one the record gives.

`DMI_MGMT_REDFISH_IP_ASSIGNMENT_DHCP`
: The address is obtained with DHCP.

`DMI_MGMT_REDFISH_IP_ASSIGNMENT_AUTO_CONFIGURE`
: The address is configured automatically.

`DMI_MGMT_REDFISH_IP_ASSIGNMENT_HOST_SELECTED`
: The address is selected by the host.

Return value

`dmi_mgmt_redfish_ip_assignment_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_redfish_over_ip`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_redfish_ip_format

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef enum dmi_mgmt_redfish_ip_format {
    DMI_MGMT_REDFISH_IP_FORMAT_UNKNOWN = 0x00,
    DMI_MGMT_REDFISH_IP_FORMAT_IPV4    = 0x01,
    DMI_MGMT_REDFISH_IP_FORMAT_IPV6    = 0x02
} dmi_mgmt_redfish_ip_format_t;

const char *dmi_mgmt_redfish_ip_format_name(
        dmi_mgmt_redfish_ip_format_t value);
```

Description

The `dmi_mgmt_redfish_ip_format_t` enumeration tells the format of the IP addresses and masks of a Redfish over IP protocol record, and is the type of the `host_ip_format` and `service_ip_format` fields of `dmi_mgmt_redfish_over_ip_t`. The values the enumeration does not name are reserved.

The `dmi_mgmt_redfish_ip_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_REDFISH_IP_FORMAT_UNKNOWN`

Unknown. The address and the mask take 16 bytes.

`DMI_MGMT_REDFISH_IP_FORMAT_IPV4`
: IPv4. The address and the mask take 4 bytes.

`DMI_MGMT_REDFISH_IP_FORMAT_IPV6`
: IPv6. The address and the mask take 16 bytes.

Return value

`dmi_mgmt_redfish_ip_format_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_redfish_over_ip`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

dmi_mgmt_redfish_over_ip

Synopsis

```
#include <opendmi/entity/mgmt-controller.h>

typedef struct dmi_mgmt_redfish_over_ip {
    dmi_uuid_t service_uuid;
    dmi_mgmt_redfish_ip_assignment_t host_ip_assignment;
    dmi_mgmt_redfish_ip_format_t host_ip_format;
    dmi_binary_t host_ip_address;
    dmi_binary_t host_ip_mask;
    dmi_mgmt_redfish_ip_assignment_t service_ip_discovery;
    dmi_mgmt_redfish_ip_format_t service_ip_format;
    dmi_binary_t service_ip_address;
    dmi_binary_t service_ip_mask;
    uint16_t service_ip_port;
    uint32_t service_vlan_id;
    char *service_hostname;
} dmi_mgmt_redfish_over_ip_t;
```

Description

The `dmi_mgmt_redfish_over_ip_t` structure holds the data of a Redfish over IP protocol record, as the

Redfish host interface specification (DSP0270) lays it out: how the host and the Redfish service of the management controller get their IP addresses on the interface, and how the host finds the service. It is the type of the `redfish` field of `dmi_mgmt_proto_record_t`, which is valid when `has_redfish` is set.

Fields

`service_uuid`

UUID of the Redfish service, all zeroes if the service does not provide one or it is unknown.

`host_ip_assignment`

How the host gets its IP address, see `dmi_mgmt_redfish_ip_assignment`.

`host_ip_format`

Format of `host_ip_address` and `host_ip_mask`, see `dmi_mgmt_redfish_ip_format`.

`host_ip_address`

IP address of the host, 4 bytes for IPv4 and 16 bytes otherwise, referring to the data of the structure in place.

`host_ip_mask`

IP mask of the host, 4 bytes for IPv4 and 16 bytes otherwise, referring to the data of the structure in place.

`service_ip_discovery`

How the host finds the IP address of the Redfish service, with the same values as `host_ip_assignment`.

`service_ip_format`

Format of `service_ip_address` and `service_ip_mask`, see `dmi_mgmt_redfish_ip_format`.

`service_ip_address`

IP address of the Redfish service, 4 bytes for IPv4 and 16 bytes otherwise, referring to the data of the structure in place.

`service_ip_mask`

IP mask of the Redfish service, 4 bytes for IPv4 and 16 bytes otherwise, referring to the data of the structure in place.

`service_ip_port`

IP port of the Redfish service.

`service_vlan_id`

VLAN ID of the Redfish service.

`service_hostname`

Host name of the Redfish service, without the null characters padding it, or `nullptr` if it is empty. The string is allocated by the library, and is freed along with the structure.

See also

`dmi_mgmt_proto_record`, `dmi_mgmt_redfish_ip_assignment`, `dmi_mgmt_redfish_ip_format`

Standards

DMTF Redfish Host Interface Specification (DSP0270), which defines this data of the management controller host interface structure (type 42) of the DMTF SMBIOS Specification (DSP0134).

`<opendmi/entity/mgmt-device-component.h>`

dmi_mgmt_device_component

Synopsis

```
#include <opendmi/entity/mgmt-device-component.h>

typedef struct dmi_mgmt_device_component {
    const char *description;
    dmi_handle_t device_handle;
    dmi_entity_t *device;
    dmi_handle_t component_handle;
    dmi_entity_t *component;
    dmi_handle_t threshold_handle;
    dmi_entity_t *threshold;
} dmi_mgmt_device_component_t;
```

Description

The `dmi_mgmt_device_component_t` structure holds the decoded management device component structure (type 35) of SMBIOS, which ties a probe or a cooling device to the management device controlling it, and optionally to the threshold data of the component. There is one structure for each component of a management device.

The handles are resolved into the structures they refer to when the table is linked. Since the threshold data depend on the component, linking also tells the threshold structure the type of the component, which gives the units of its values.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(mgmt_device_component)`, and lives as long as the context it has been decoded in.

Fields

`description`

Additional description of the component, or `nullptr` if there is none.

`device_handle`

Handle of the management device structure (type 34) containing the component.

`device`

Management device structure containing the component, whose information is a `dmi_mgmt_device_t`. Set when linking; `nullptr` if the handle refers to no management device, which linking reports as an error.

`component_handle`

Handle of the structure defining the component: a voltage probe (type 26), a cooling device (type 27), a temperature probe (type 28) or an electrical current probe (type 29).

`component`

Structure defining the component, whose information is a `dmi_probe_t` or a `dmi_cooling_device_t`, depending on its type. Set when linking; `nullptr` if the handle refers to no such structure, which linking reports as an error.

`threshold_handle`

Handle of the management device threshold data structure (type 36) of the component, or `DMI_HANDLE_INVALID` if there is none.

`threshold`

Management device threshold data structure of the component, whose information is a `dmi_mgmt_device_threshold_t`. Set when linking; `nullptr` if the component has no thresholds, or if the handle refers to no such structure, which linking reports as an error.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   =
        dmi_registry_lookup_first(registry,
                                  DMI_TYPE(mgmt_device_component), true);

dmi_mgmt_device_component_t *info =
        dmi_entity_info(entity, DMI_TYPE(mgmt_device_component));

if ((info != nullptr) && (info->component != nullptr))
    printf("Component 0x%04x of device 0x%04x, type %d\n",
           info->component_handle, info->device_handle,
           (int)dmi_entity_type(info->component));
```

See also

`dmi_entity_info`, `dmi_mgmt_device`, `dmi_mgmt_device_threshold`, `dmi_probe`, `dmi_cooling_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/mgmt-device-threshold.h>

dmi_mgmt_device_threshold

Synopsis

```
#include <opendmi/entity/mgmt-device-threshold.h>

typedef struct dmi_mgmt_device_threshold {
    short      lower_non_critical;
    short      upper_non_critical;
    short      lower_critical;
    short      upper_critical;
    short      lower_non_recoverable;
    short      upper_non_recoverable;
    dmi_type_id_t component_type;
    bool       is_ambiguous;
} dmi_mgmt_device_threshold_t;
```

Description

The `dmi_mgmt_device_threshold_t` structure holds the decoded management device threshold data structure (type 36) of SMBIOS: the thresholds of a component of a management device, that is, of a voltage, temperature or electrical current probe, or of a cooling device. A management device component structure (type 35) refers to it.

The thresholds are signed, and their units are the ones of the component: millivolts for a voltage probe, tenths of degree Celsius for a temperature probe, milliamperes for an electrical current probe, and revolutions per minute for a cooling device. A threshold which is not available is `SHRT_MIN` (0x8000 as stored).

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(mgmt_device_threshold)`, and lives as long as the context it has been decoded in.

Fields

`lower_non_critical`
: Lower non-critical threshold, or `SHRT_MIN`.

`upper_non_critical`
: Upper non-critical threshold, or `SHRT_MIN`.

`lower_critical`
: Lower critical threshold, or `SHRT_MIN`.

`upper_critical`

Upper critical threshold, or `SHRT_MIN`.

`lower_non_recoverable`

Lower non-recoverable threshold, or `SHRT_MIN`.

`upper_non_recoverable`

Upper non-recoverable threshold, or `SHRT_MIN`.

`component_type`

Type number of the component using the thresholds, such as `DMI_TYPE_ID_VOLTAGE_PROBE`, which tells the units of the values. Set when the management device components are linked. `DMI_TYPE_ID_INVALID` if no component uses the thresholds, or if components of different types do.

`is_ambiguous`

`true` if components of different types use the thresholds, so that their units are unknown.

Examples

```
dmi_registry_t              *registry  = dmi_get_registry(context);
dmi_entity_t                *entity    = dmi_registry_lookup_first(
        registry, DMI_TYPE(mgmt_device_threshold), true);
dmi_mgmt_device_threshold_t *threshold = dmi_entity_info(
        entity, DMI_TYPE(mgmt_device_threshold));

if ((threshold != nullptr) &&
    (threshold->component_type == DMI_TYPE_ID_TEMPERATURE_PROBE) &&
    (threshold->upper_critical != SHRT_MIN))
    printf("Critical above %.1f C\n", threshold->upper_critical / 10.0);
```

See also

`dmi_entity_info`, `dmi_mgmt_device_component`, `dmi_mgmt_device`, `dmi_probe`, `dmi_cooling_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/mgmt-device.h>

dmi_mgmt_device

Synopsis

```
#include <opendmi/entity/mgmt-device.h>

typedef struct dmi_mgmt_device {
    const char *description;
    dmi_mgmt_device_type_t type;
    uint32_t address;
    dmi_mgmt_device_addr_type_t address_type;
} dmi_mgmt_device_t;
```

Description

The `dmi_mgmt_device_t` structure holds the decoded management device structure (type 34) of SMBIOS: a hardware device, typically a sensor chip, which reads probes or controls cooling devices, and how it is addressed. Management device component structures (`dmi_mgmt_device_component_t`) tie the probes and cooling devices to the management device.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(mgmt_device)`, and lives as long as the context it has been decoded in.

Fields

`description`
: Additional description of the device or of its location, or `nullptr` if there is none.

`type`
: Type of the device.

`address`
: Address of the device, to be read according to `address_type`.

`address_type`
: Type of the address the device is accessed at.

Examples

```
dmi_registry_t    *registry = dmi_get_registry(context);
dmi_entity_t      *entity   =
        dmi_registry_lookup_first(registry, DMI_TYPE(mgmt_device), true);
dmi_mgmt_device_t *device   =
        dmi_entity_info(entity, DMI_TYPE(mgmt_device));
```

```
if (device != nullptr)
    printf("%s at %s 0x%08" PRIx32 "\n",
           dmi_mgmt_device_type_name(device->type),
           dmi_mgmt_device_addr_type_name(device->address_type),
           device->address);
```

See also

`dmi_entity_info`, `dmi_mgmt_device_type`, `dmi_mgmt_device_addr_type`, `dmi_mgmt_device_component`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_mgmt_device_addr_type

Synopsis

```
#include <opendmi/entity/mgmt-device.h>

typedef enum dmi_mgmt_device_addr_type {
    DMI_MGMT_DEVICE_ADDR_TYPE_UNSPEC  = 0x00,
    DMI_MGMT_DEVICE_ADDR_TYPE_OTHER   = 0x01,
    DMI_MGMT_DEVICE_ADDR_TYPE_UNKNOWN = 0x02,
    DMI_MGMT_DEVICE_ADDR_TYPE_PORT    = 0x03,
    DMI_MGMT_DEVICE_ADDR_TYPE_MEMORY  = 0x04,
    DMI_MGMT_DEVICE_ADDR_TYPE_SMBUS   = 0x05,
} dmi_mgmt_device_addr_type_t;

const char *dmi_mgmt_device_addr_type_name(dmi_mgmt_device_addr_type_t value);
```

Description

The `dmi_mgmt_device_addr_type_t` enumeration tells how a management device is accessed, and so how its address is to be read. It is the type of the `address_type` field of `dmi_mgmt_device_t`.

The `dmi_mgmt_device_addr_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_DEVICE_ADDR_TYPE_UNSPEC`
: Unspecified.

`DMI_MGMT_DEVICE_ADDR_TYPE_OTHER`
: Other address type.

`DMI_MGMT_DEVICE_ADDR_TYPE_UNKNOWN`
: Unknown address type.

`DMI_MGMT_DEVICE_ADDR_TYPE_PORT`
: I/O port address.

`DMI_MGMT_DEVICE_ADDR_TYPE_MEMORY`
: Memory address.

`DMI_MGMT_DEVICE_ADDR_TYPE_SMBUS`
: SMBus address.

Return value

`dmi_mgmt_device_addr_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_mgmt_device_type

Synopsis

```
#include <opendmi/entity/mgmt-device.h>

typedef enum dmi_mgmt_device_type {
    DMI_MGMT_DEVICE_TYPE_UNSPEC          = 0x00,
    DMI_MGMT_DEVICE_TYPE_OTHER           = 0x01,
    DMI_MGMT_DEVICE_TYPE_UNKNOWN         = 0x02,
    DMI_MGMT_DEVICE_TYPE_NATIONAL_LM75   = 0x03,
    DMI_MGMT_DEVICE_TYPE_NATIONAL_LM78   = 0x04,
    DMI_MGMT_DEVICE_TYPE_NATIONAL_LM79   = 0x05,
    DMI_MGMT_DEVICE_TYPE_NATIONAL_LM80   = 0x06,
    DMI_MGMT_DEVICE_TYPE_NATIONAL_LM81   = 0x07,
    DMI_MGMT_DEVICE_TYPE_ANALOG_ADM9240  = 0x08,
    DMI_MGMT_DEVICE_TYPE_DALLAS_DS1780   = 0x09,
```

```
    DMI_MGMT_DEVICE_TYPE_MAXIM_1617      = 0x0A,
    DMI_MGMT_DEVICE_TYPE_GENESYS_GL518SM = 0x0B,
    DMI_MGMT_DEVICE_TYPE_WINBOND_W83781D = 0x0C,
    DMI_MGMT_DEVICE_TYPE_HOLTEK_HT82H791 = 0x0D,
} dmi_mgmt_device_type_t;

const char *dmi_mgmt_device_type_name(dmi_mgmt_device_type_t value);
```

Description

The `dmi_mgmt_device_type_t` enumeration tells the model of a management device, and is the type of the `type` field of `dmi_mgmt_device_t`.

The `dmi_mgmt_device_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_MGMT_DEVICE_TYPE_UNSPEC`
: Unspecified.

`DMI_MGMT_DEVICE_TYPE_OTHER`
: Other device.

`DMI_MGMT_DEVICE_TYPE_UNKNOWN`
: Unknown device.

`DMI_MGMT_DEVICE_TYPE_NATIONAL_LM75`
: National Semiconductor LM75.

`DMI_MGMT_DEVICE_TYPE_NATIONAL_LM78`
: National Semiconductor LM78.

`DMI_MGMT_DEVICE_TYPE_NATIONAL_LM79`
: National Semiconductor LM79.

`DMI_MGMT_DEVICE_TYPE_NATIONAL_LM80`
: National Semiconductor LM80.

`DMI_MGMT_DEVICE_TYPE_NATIONAL_LM81`
: National Semiconductor LM81.

`DMI_MGMT_DEVICE_TYPE_ANALOG_ADM9240`
: Analog Devices ADM9240.

`DMI_MGMT_DEVICE_TYPE_DALLAS_DS1780`
: Dallas Semiconductor DS1780.

`DMI_MGMT_DEVICE_TYPE_MAXIM_1617`
: Maxim 1617.

`DMI_MGMT_DEVICE_TYPE_GENESYS_GL518SM`
: Genesys GL518SM.

`DMI_MGMT_DEVICE_TYPE_WINBOND_W83781D`
: Winbond W83781D.

`DMI_MGMT_DEVICE_TYPE_HOLTEK_HT82H791`
: Holtek HT82H791.

Return value

`dmi_mgmt_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_mgmt_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/oem-strings.h>

dmi_oem_strings

Synopsis

```
#include <opendmi/entity/oem-strings.h>

typedef struct dmi_oem_strings {
    size_t string_count;
    const char **strings;
} dmi_oem_strings_t;
```

Description

The `dmi_oem_strings_t` structure holds the decoded OEM strings structure (type 11) of SMBIOS: free-form strings defined by the OEM, such as part numbers of reference documents or contact information of the manufacturer.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(oem_strings)`, and lives as long as the context it has been decoded in.

Fields

`string_count`
: Number of strings the structure declares.

`strings`
: Array of `string_count` strings, element `i` holding the string number `i + 1`, or `nullptr` if `string_count` is 0. An element is `nullptr` if the structure has fewer strings than it declares.

Examples

```
dmi_registry_t    *registry = dmi_get_registry(context);
dmi_entity_t      *entity   = dmi_registry_lookup_first(registry,
                                  DMI_TYPE(oem_strings), false);
dmi_oem_strings_t *oem      = dmi_entity_info(entity,
                                  DMI_TYPE(oem_strings));

if (oem != nullptr) {
    for (size_t i = 0; i < oem->string_count; i++)
        printf("%zu: %s\n", i + 1, oem->strings[i] ? oem->strings[i] : "");
}
```

See also

`dmi_entity_info`, `dmi_entity_string`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/onboard-device-ex.h>

dmi_onboard_device_ex

Synopsis

```
#include <opendmi/entity/onboard-device-ex.h>

typedef struct dmi_onboard_device_ex {
    const char *designator;
    dmi_onboard_device_type_t type;
    bool is_enabled;
    unsigned short instance;
    dmi_pci_addr_t address;
} dmi_onboard_device_ex_t;
```

Description

The `dmi_onboard_device_ex_t` structure holds the decoded onboard devices extended information structure (type 41) of SMBIOS: a device soldered onto a board of the system, usually the baseboard, which the firmware can typically enable or disable. It gives the kind of the device, whether it is enabled, and its PCI address. A table carries one such structure for each device, or for each function of a multi-function device, whose functions a group association structure (type 14) then ties together.

The structure replaces the onboard devices information structure (type 10), obsolete since SMBIOS 2.6, which firmware may still provide beside it.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(onboard_device_ex)`, and lives as long as the context it has been decoded in.

Fields

`designator`

Reference designator of the device, usually its label on the board; `nullptr` if the structure does not carry it.

`type`

Kind of the device, see `dmi_onboard_device_type`.

`is_enabled`

`true` if the device is enabled.

`instance`

Rank of the device among the onboard devices of the same type, as the system numbers them: 1 for the first network adapter, 2 for the second, and so on. The order need not follow the PCI addresses of the devices. The `onboard-device-ex.instance` lint rule reports two devices of a type sharing an instance.

`address`

PCI segment group, bus, device and function of the device, see `dmi_pci_addr`. Devices without such an address carry `0xFFFF` as segment group and `UINT8_MAX` as bus, device and function numbers.

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_registry_iter_t  iter;
dmi_entity_t        *entity;

dmi_registry_iter_initialize(&iter, registry, nullptr);

while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
    if (dmi_entity_type(entity) != DMI_TYPE(onboard_device_ex))
        continue;

    dmi_onboard_device_ex_t *device =
            dmi_entity_info(entity, DMI_TYPE(onboard_device_ex));
    if ((device != nullptr) && (device->designator != nullptr))
        printf("%s: %s #%u, %s\n", device->designator,
               dmi_onboard_device_type_name(device->type), device->instance,
               device->is_enabled ? "enabled" : "disabled");
}
```

See also

`dmi_entity_info`, `dmi_onboard_device_type`, `dmi_pci_addr`, `dmi_onboard_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.6 or newer.

<opendmi/entity/onboard-device.h>

dmi_onboard_device

Synopsis

```
#include <opendmi/entity/onboard-device.h>

typedef struct dmi_onboard_device {
    size_t                       instance_count;
    dmi_onboard_device_instance_t *instances;
} dmi_onboard_device_t;
```

Description

The `dmi_onboard_device_t` structure holds the decoded onboard devices information structure (type 10) of SMBIOS: a list of the devices soldered onto a board of the system, usually the baseboard, which the firmware can enable or disable. A table may have one such structure for all the devices, or one for each of them.

The structure is obsolete since SMBIOS 2.6: the onboard devices extended information structure (type 41), see `dmi_onboard_device_ex`, replaces it, although firmware may still provide both.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(onboard_device)`, and lives as long as the context it has been decoded in.

Fields

`instance_count`

Number of devices, told by the length of the structure.

`instances`

Array of `instance_count` devices, see `dmi_onboard_device_instance`.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(
        registry, DMI_TYPE(onboard_device), false);
dmi_onboard_device_t *info     = dmi_entity_info(entity,
                                                 DMI_TYPE(onboard_device));

for (size_t i = 0; (info != nullptr) && (i < info->instance_count); i++) {
    const dmi_onboard_device_instance_t *device = &info->instances[i];

    printf("%s (%s)\n",
           device->description ? device->description : "Unnamed device",
```

```
            device->is_enabled ? "enabled" : "disabled");
}
```

See also

`dmi_entity_info`, `dmi_onboard_device_instance`, `dmi_onboard_device_type`, `dmi_onboard_device_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_onboard_device_instance

Synopsis

```
#include <opendmi/entity/onboard-device.h>

typedef struct dmi_onboard_device_instance {
    dmi_onboard_device_type_t  type;
    bool                       is_enabled;
    const char                *description;
} dmi_onboard_device_instance_t;
```

Description

The `dmi_onboard_device_instance_t` structure describes one of the devices of the onboard devices information structure (type 10) of SMBIOS. The `instances` field of `dmi_onboard_device_t` is an array of them.

Fields

`type`
: Kind of the device, see `dmi_onboard_device_type`.

`is_enabled`
: `true` if the device is enabled.

`description`
: Description of the device, or `nullptr`.

See also

`dmi_onboard_device`, `dmi_onboard_device_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_onboard_device_type

Synopsis

```
#include <opendmi/entity/onboard-device.h>

typedef enum dmi_onboard_device_type {
    DMI_ONBOARD_DEVICE_TYPE_UNSPEC                = 0x00,
    DMI_ONBOARD_DEVICE_TYPE_OTHER                 = 0x01,
    DMI_ONBOARD_DEVICE_TYPE_UNKNOWN               = 0x02,
    DMI_ONBOARD_DEVICE_TYPE_VIDEO                 = 0x03,
    DMI_ONBOARD_DEVICE_TYPE_SCSI_CONTROLLER       = 0x04,
    DMI_ONBOARD_DEVICE_TYPE_ETHERNET              = 0x05,
    DMI_ONBOARD_DEVICE_TYPE_TOKEN_RING            = 0x06,
    DMI_ONBOARD_DEVICE_TYPE_SOUND                 = 0x07,
    DMI_ONBOARD_DEVICE_TYPE_PATA_CONTROLLER       = 0x08,
    DMI_ONBOARD_DEVICE_TYPE_SATA_CONTROLLER       = 0x09,
    DMI_ONBOARD_DEVICE_TYPE_SAS_CONTROLLER        = 0x0A,
    DMI_ONBOARD_DEVICE_TYPE_WIRELESS_LAN          = 0x0B,
    DMI_ONBOARD_DEVICE_TYPE_BLUETOOTH             = 0x0C,
    DMI_ONBOARD_DEVICE_TYPE_WIRELESS_WAN          = 0x0D,
    DMI_ONBOARD_DEVICE_TYPE_EMM_CONTROLLER        = 0x0E,
    DMI_ONBOARD_DEVICE_TYPE_NVME_CONTROLLER       = 0x0F,
    DMI_ONBOARD_DEVICE_TYPE_UFS_CONTROLLER        = 0x10,
    DMI_ONBOARD_DEVICE_TYPE_INFINIBAND_CONTROLLER = 0x11,
    DMI_ONBOARD_DEVICE_TYPE_GPU_CONTROLLER        = 0x12
} dmi_onboard_device_type_t;

const char *dmi_onboard_device_type_name(dmi_onboard_device_type_t value);
```

Description

The `dmi_onboard_device_type_t` enumeration tells the kind of a device soldered onto a board of the system. It is the type of the `type` field of `dmi_onboard_device_instance_t`, and of the one of `dmi_onboard_device_ex_t`. The values above 0x0A were added for the onboard devices extended information structure (type 41), and do not fit the obsolete onboard devices information structure (type 10), which has no newer versions.

The `dmi_onboard_device_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_ONBOARD_DEVICE_TYPE_UNSPEC`
: Unspecified.

`DMI_ONBOARD_DEVICE_TYPE_OTHER`
: Other.

`DMI_ONBOARD_DEVICE_TYPE_UNKNOWN`
: Unknown.

`DMI_ONBOARD_DEVICE_TYPE_VIDEO`
: Video.

`DMI_ONBOARD_DEVICE_TYPE_SCSI_CONTROLLER`
: SCSI Controller.

`DMI_ONBOARD_DEVICE_TYPE_ETHERNET`
: Ethernet.

`DMI_ONBOARD_DEVICE_TYPE_TOKEN_RING`
: Token Ring.

`DMI_ONBOARD_DEVICE_TYPE_SOUND`
: Sound.

`DMI_ONBOARD_DEVICE_TYPE_PATA_CONTROLLER`
: PATA Controller.

`DMI_ONBOARD_DEVICE_TYPE_SATA_CONTROLLER`
: SATA Controller.

`DMI_ONBOARD_DEVICE_TYPE_SAS_CONTROLLER`
: SAS Controller.

`DMI_ONBOARD_DEVICE_TYPE_WIRELESS_LAN`
: Wireless LAN.

`DMI_ONBOARD_DEVICE_TYPE_BLUETOOTH`
: Bluetooth.

`DMI_ONBOARD_DEVICE_TYPE_WIRELESS_WAN`
: Wireless WAN.

`DMI_ONBOARD_DEVICE_TYPE_EMM_CONTROLLER`
: eMMC (embedded multimedia card) controller.

`DMI_ONBOARD_DEVICE_TYPE_NVME_CONTROLLER`
: NVMe controller.

`DMI_ONBOARD_DEVICE_TYPE_UFS_CONTROLLER`
: UFS controller.

`DMI_ONBOARD_DEVICE_TYPE_INFINIBAND_CONTROLLER`

InfiniBand controller.

`DMI_ONBOARD_DEVICE_TYPE_GPU_CONTROLLER`

GPU controller.

Return value

`dmi_onboard_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_onboard_device`, `dmi_onboard_device_instance`, `dmi_onboard_device_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/oob-remote-access.h>

dmi_oob_connections

Synopsis

```
#include <opendmi/entity/oob-remote-access.h>

typedef union dmi_oob_connections {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_inbound_enabled  : 1;
        dmi_byte_t is_outbound_enabled : 1;
        dmi_byte_t __reserved          : 6;
    };
} dmi_oob_connections_t;
```

Description

The `dmi_oob_connections_t` union holds the connections an out-of-band remote access facility is allowed to accept and to make. It is the type of the `connections` field of `dmi_oob_remote_access_t`.

The union is packed into a single byte: the flags can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the flags, as the byte the structure carries them in, `is_inbound_enabled` being bit 0.

`is_inbound_enabled`
: Set if the facility accepts incoming connections, for remote operation or problem management.

`is_outbound_enabled`
: Set if the facility may open connections to an alert management facility when critical conditions occur.

`__reserved`
: Reserved bits, zero.

See also

`dmi_oob_remote_access`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

dmi_oob_remote_access

Synopsis

```
#include <opendmi/entity/oob-remote-access.h>

typedef struct dmi_oob_remote_access {
    const char *vendor;
    dmi_oob_connections_t connections;
} dmi_oob_remote_access_t;
```

Description

The `dmi_oob_remote_access_t` structure holds the decoded out-of-band remote access structure (type 30) of SMBIOS: a hardware facility through which the system can be reached remotely when its operating system cannot run, e.g. because the system is powered down or fails to boot, and the connections the facility is allowed to make and to accept.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(oob_remote_access)`, and lives as long as the context it has been decoded in.

Fields

`vendor`

Name of the manufacturer of the facility, or `nullptr`.

`connections`

Connections the facility is allowed to accept and to make, see `dmi_oob_connections`.

Examples

```
dmi_registry_t          *registry = dmi_get_registry(context);
dmi_entity_t            *entity   = dmi_registry_lookup_first(registry,
                                        DMI_TYPE(oob_remote_access), false);
dmi_oob_remote_access_t *access   = dmi_entity_info(entity,
                                        DMI_TYPE(oob_remote_access));

if (access != nullptr)
    printf("Inbound: %s, outbound: %s\n",
           access->connections.is_inbound_enabled ? "yes" : "no",
           access->connections.is_outbound_enabled ? "yes" : "no");
```

See also

`dmi_entity_info`, `dmi_oob_connections`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/pointing-device.h>

dmi_pointing_device

Synopsis

```
#include <opendmi/entity/pointing-device.h>

typedef struct dmi_pointing_device {
    dmi_pointing_device_type_t type;
    dmi_pointing_device_iface_t interface;
    unsigned short button_count;
} dmi_pointing_device_t;
```

Description

The `dmi_pointing_device_t` structure holds the decoded built-in pointing device structure (type 21) of SMBIOS: the type, interface and number of buttons of the pointing device built into the system, such as the touch pad of a laptop. The structure being present does not mean that the device is in use.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(pointing_device)`, and lives as long as the context it has been decoded in.

Fields

`type`

Type of the pointing device, see `dmi_pointing_device_type`.

`interface`

Interface the pointing device is connected through, see `dmi_pointing_device_iface`.

`button_count`

Number of buttons of the pointing device, for example 3 for a three-button mouse.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(pointing_device), false);
dmi_pointing_device_t *device   = dmi_entity_info(entity,
                                      DMI_TYPE(pointing_device));

if (device != nullptr)
    printf("%s (%s), %u buttons\n",
           dmi_pointing_device_type_name(device->type),
           dmi_pointing_device_iface_name(device->interface),
```

```
            device->button_count);
```

See also

`dmi_entity_info`, `dmi_pointing_device_type`, `dmi_pointing_device_iface`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_pointing_device_iface

Synopsis

```
#include <opendmi/entity/pointing-device.h>

typedef enum dmi_pointing_device_iface {
    DMI_POINTING_DEVICE_IFACE_UNSPEC        = 0x00,
    DMI_POINTING_DEVICE_IFACE_OTHER         = 0x01,
    DMI_POINTING_DEVICE_IFACE_UNKNOWN       = 0x02,
    DMI_POINTING_DEVICE_IFACE_SERIAL        = 0x03,
    DMI_POINTING_DEVICE_IFACE_PS2           = 0x04,
    DMI_POINTING_DEVICE_IFACE_INFRARED      = 0x05,
    DMI_POINTING_DEVICE_IFACE_HP_HIL        = 0x06,
    DMI_POINTING_DEVICE_IFACE_BUS_MOUSE     = 0x07,
    DMI_POINTING_DEVICE_IFACE_ADB           = 0x08,
    DMI_POINTING_DEVICE_IFACE_BUS_MOUSE_DB9 = 0xA0,
    DMI_POINTING_DEVICE_IFACE_BUS_MOUSE_DIN = 0xA1,
    DMI_POINTING_DEVICE_IFACE_USB           = 0xA2,
    DMI_POINTING_DEVICE_IFACE_I2C           = 0xA3,
    DMI_POINTING_DEVICE_IFACE_SPI           = 0xA4,
} dmi_pointing_device_iface_t;

const char *dmi_pointing_device_iface_name(dmi_pointing_device_iface_t value);
```

Description

The `dmi_pointing_device_iface_t` enumeration tells the interface a built-in pointing device is connected through, and is the type of the `interface` field of `dmi_pointing_device_t`. Values 0x09 to 0x9F are not assigned by the specification.

The `dmi_pointing_device_iface_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_POINTING_DEVICE_IFACE_UNSPEC`
: Unspecified.

`DMI_POINTING_DEVICE_IFACE_OTHER`
: Other interface.

`DMI_POINTING_DEVICE_IFACE_UNKNOWN`
: Unknown interface.

`DMI_POINTING_DEVICE_IFACE_SERIAL`
: Serial port.

`DMI_POINTING_DEVICE_IFACE_PS2`
: PS/2.

`DMI_POINTING_DEVICE_IFACE_INFRARED`
: Infrared.

`DMI_POINTING_DEVICE_IFACE_HP_HIL`
: HP-HIL.

`DMI_POINTING_DEVICE_IFACE_BUS_MOUSE`
: Bus mouse.

`DMI_POINTING_DEVICE_IFACE_ADB`
: ADB (Apple Desktop Bus).

`DMI_POINTING_DEVICE_IFACE_BUS_MOUSE_DB9`
: Bus mouse, DB-9.

`DMI_POINTING_DEVICE_IFACE_BUS_MOUSE_DIN`
: Bus mouse, micro-DIN.

`DMI_POINTING_DEVICE_IFACE_USB`
: USB.

`DMI_POINTING_DEVICE_IFACE_I2C`
: I2C.

`DMI_POINTING_DEVICE_IFACE_SPI`
: SPI.

Return value

`dmi_pointing_device_iface_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_pointing_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_pointing_device_type

Synopsis

```
#include <opendmi/entity/pointing-device.h>

typedef enum dmi_pointing_device_type {
    DMI_POINTING_DEVICE_TYPE_UNSPEC         = 0x00,
    DMI_POINTING_DEVICE_TYPE_OTHER          = 0x01,
    DMI_POINTING_DEVICE_TYPE_UNKNOWN        = 0x02,
    DMI_POINTING_DEVICE_TYPE_MOUSE          = 0x03,
    DMI_POINTING_DEVICE_TYPE_TRACK_BALL     = 0x04,
    DMI_POINTING_DEVICE_TYPE_TRACK_POINT    = 0x05,
    DMI_POINTING_DEVICE_TYPE_GLIDE_POINT    = 0x06,
    DMI_POINTING_DEVICE_TYPE_TOUCH_PAD      = 0x07,
    DMI_POINTING_DEVICE_TYPE_TOUCH_SCREEN   = 0x08,
    DMI_POINTING_DEVICE_TYPE_OPTICAL_SENSOR = 0x09,
} dmi_pointing_device_type_t;

const char *dmi_pointing_device_type_name(dmi_pointing_device_type_t value);
```

Description

The `dmi_pointing_device_type_t` enumeration tells the kind of a built-in pointing device, and is the type of the `type` field of `dmi_pointing_device_t`.

The `dmi_pointing_device_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_POINTING_DEVICE_TYPE_UNSPEC`
: Unspecified.

`DMI_POINTING_DEVICE_TYPE_OTHER`
: Other type.

`DMI_POINTING_DEVICE_TYPE_UNKNOWN`
: Unknown type.

`DMI_POINTING_DEVICE_TYPE_MOUSE`
: Mouse.

`DMI_POINTING_DEVICE_TYPE_TRACK_BALL`
: Track ball.

`DMI_POINTING_DEVICE_TYPE_TRACK_POINT`
: Track point.

`DMI_POINTING_DEVICE_TYPE_GLIDE_POINT`
: Glide point.

`DMI_POINTING_DEVICE_TYPE_TOUCH_PAD`
: Touch pad.

`DMI_POINTING_DEVICE_TYPE_TOUCH_SCREEN`
: Touch screen.

`DMI_POINTING_DEVICE_TYPE_OPTICAL_SENSOR`
: Optical sensor.

Return value

`dmi_pointing_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_pointing_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/port-connector.h>

dmi_connector_type

Synopsis

```
#include <opendmi/entity/port-connector.h>

typedef enum dmi_connector_type {
    DMI_CONNECTOR_TYPE_NONE             = 0x00,
    DMI_CONNECTOR_TYPE_CN_36            = 0x01,
    DMI_CONNECTOR_TYPE_HPCN_36          = 0x02,
    DMI_CONNECTOR_TYPE_PROPRIETARY      = 0x03,
    DMI_CONNECTOR_TYPE_DB_25_M          = 0x04,
    DMI_CONNECTOR_TYPE_DB_25_F          = 0x05,
    DMI_CONNECTOR_TYPE_DB_15_M          = 0x06,
    DMI_CONNECTOR_TYPE_DB_15_F          = 0x07,
    DMI_CONNECTOR_TYPE_DB_9_M           = 0x08,
    DMI_CONNECTOR_TYPE_DB_9_F           = 0x09,
    DMI_CONNECTOR_TYPE_RJ_11            = 0x0A,
    DMI_CONNECTOR_TYPE_RJ_45            = 0x0B,
    DMI_CONNECTOR_TYPE_MINI_SCSI_50PIN  = 0x0C,
    DMI_CONNECTOR_TYPE_MINI_DIN         = 0x0D,
    DMI_CONNECTOR_TYPE_MICRO_DIN        = 0x0E,
    DMI_CONNECTOR_TYPE_PS2              = 0x0F,
    DMI_CONNECTOR_TYPE_INFRARED         = 0x10,
    DMI_CONNECTOR_TYPE_HP_HIL           = 0x11,
    DMI_CONNECTOR_TYPE_ACCESS_BUS       = 0x12,
    DMI_CONNECTOR_TYPE_SSA_SCSI         = 0x13,
    DMI_CONNECTOR_TYPE_CIRCULAR_DIN_8_M = 0x14,
    DMI_CONNECTOR_TYPE_CIRCULAR_DIN_8_F = 0x15,
    DMI_CONNECTOR_TYPE_ONBOARD_IDE      = 0x16,
    DMI_CONNECTOR_TYPE_ONBOARD_FLOPPY   = 0x17,
    DMI_CONNECTOR_TYPE_DIL_9PIN         = 0x18,
    DMI_CONNECTOR_TYPE_DIL_25PIN        = 0x19,
    DMI_CONNECTOR_TYPE_DIL_50PIN        = 0x1A,
    DMI_CONNECTOR_TYPE_DIL_68PIN        = 0x1B,
    DMI_CONNECTOR_TYPE_ONBOARD_SOUND    = 0x1C,
    DMI_CONNECTOR_TYPE_HPCN_14          = 0x1D,
    DMI_CONNECTOR_TYPE_HPCN_26          = 0x1E,
    DMI_CONNECTOR_TYPE_MINI_JACK        = 0x1F,
    DMI_CONNECTOR_TYPE_BNC              = 0x20,
    DMI_CONNECTOR_TYPE_IEEE_1394        = 0x21,
    DMI_CONNECTOR_TYPE_SAS_SATA         = 0x22,
    DMI_CONNECTOR_TYPE_USB_C            = 0x23,
    DMI_CONNECTOR_TYPE_PC_98            = 0xA0,
    DMI_CONNECTOR_TYPE_PC_98_HIRESO     = 0xA1,
```

```
    DMI_CONNECTOR_TYPE_PC_H98           = 0xA2,
    DMI_CONNECTOR_TYPE_PC_98_NOTE       = 0xA3,
    DMI_CONNECTOR_TYPE_PC_98_FULL       = 0xA4,
    DMI_CONNECTOR_TYPE_OTHER            = 0xFF,
} dmi_connector_type_t;

const char *dmi_connector_type_name(dmi_connector_type_t value);
```

Description

The `dmi_connector_type_t` enumeration tells the physical type of a port connector, and is the type of the `internal_connector` and `external_connector` fields of `dmi_port_connector_t`. The specification assigns no values from `0x24` to `0x9F` and from `0xA5` to `0xFE`.

The `dmi_connector_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_CONNECTOR_TYPE_NONE`
: No connector.

`DMI_CONNECTOR_TYPE_CN_36`
: Centronics.

`DMI_CONNECTOR_TYPE_HPCN_36`
: Mini-Centronics.

`DMI_CONNECTOR_TYPE_PROPRIETARY`
: Proprietary.

`DMI_CONNECTOR_TYPE_DB_25_M`
: DB-25, male.

`DMI_CONNECTOR_TYPE_DB_25_F`
: DB-25, female.

`DMI_CONNECTOR_TYPE_DB_15_M`
: DB-15, male.

`DMI_CONNECTOR_TYPE_DB_15_F`
: DB-15, female.

`DMI_CONNECTOR_TYPE_DB_9_M`
: DB-9, male.

`DMI_CONNECTOR_TYPE_DB_9_F`
: DB-9, female.

`DMI_CONNECTOR_TYPE_RJ_11`
: RJ-11.

`DMI_CONNECTOR_TYPE_RJ_45`
: RJ-45.

`DMI_CONNECTOR_TYPE_MINI_SCSI_50PIN`
: MiniSCSI, 50 pins.

`DMI_CONNECTOR_TYPE_MINI_DIN`
: Mini-DIN.

`DMI_CONNECTOR_TYPE_MICRO_DIN`
: Micro-DIN.

`DMI_CONNECTOR_TYPE_PS2`
: PS/2.

`DMI_CONNECTOR_TYPE_INFRARED`
: Infrared.

`DMI_CONNECTOR_TYPE_HP_HIL`
: HP-HIL.

`DMI_CONNECTOR_TYPE_ACCESS_BUS`
: Access Bus (USB).

`DMI_CONNECTOR_TYPE_SSA_SCSI`
: SSA SCSI.

`DMI_CONNECTOR_TYPE_CIRCULAR_DIN_8_M`
: Circular DIN-8, male.

`DMI_CONNECTOR_TYPE_CIRCULAR_DIN_8_F`
: Circular DIN-8, female.

`DMI_CONNECTOR_TYPE_ONBOARD_IDE`
: IDE connector on the board.

`DMI_CONNECTOR_TYPE_ONBOARD_FLOPPY`
: Floppy connector on the board.

`DMI_CONNECTOR_TYPE_DIL_9PIN`
: Dual inline, 9 pins (pin 10 cut).

`DMI_CONNECTOR_TYPE_DIL_25PIN`
: Dual inline, 25 pins (pin 26 cut).

`DMI_CONNECTOR_TYPE_DIL_50PIN`
: Dual inline, 50 pins.

`DMI_CONNECTOR_TYPE_DIL_68PIN`
: Dual inline, 68 pins.

`DMI_CONNECTOR_TYPE_ONBOARD_SOUND`
: Sound input from a CD-ROM drive, on the board.

`DMI_CONNECTOR_TYPE_HPCN_14`
: Mini-Centronics type 14.

`DMI_CONNECTOR_TYPE_HPCN_26`
: Mini-Centronics type 26.

`DMI_CONNECTOR_TYPE_MINI_JACK`
: Mini-jack, as for headphones.

`DMI_CONNECTOR_TYPE_BNC`
: BNC.

`DMI_CONNECTOR_TYPE_IEEE_1394`
: IEEE 1394 (FireWire).

`DMI_CONNECTOR_TYPE_SAS_SATA`
: SAS/SATA plug receptacle.

`DMI_CONNECTOR_TYPE_USB_C`
: USB Type-C receptacle.

`DMI_CONNECTOR_TYPE_PC_98`
: PC-98.

`DMI_CONNECTOR_TYPE_PC_98_HIRESO`
: PC-98 Hireso.

`DMI_CONNECTOR_TYPE_PC_H98`
: PC-H98.

`DMI_CONNECTOR_TYPE_PC_98_NOTE`
: PC-98 Note.

`DMI_CONNECTOR_TYPE_PC_98_FULL`
: PC-98 Full.

`DMI_CONNECTOR_TYPE_OTHER`
: Other; the reference designators tell more.

Return value

`dmi_connector_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_port_connector`, `dmi_port_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_port_connector

Synopsis

```
#include <opendmi/entity/port-connector.h>

typedef struct dmi_port_connector {
    const char *internal_designator;
    dmi_connector_type_t internal_connector;
    const char *external_designator;
    dmi_connector_type_t external_connector;
    dmi_port_type_t port_type;
} dmi_port_connector_t;
```

Description

The `dmi_port_connector_t` structure holds the decoded port connector information structure (type 8) of SMBIOS: what a port of the system is for (serial, parallel, keyboard, USB, network...), and the connectors it has inside and outside the enclosure. A table carries one such structure for each port.

A port without an internal connector, such as one soldered onto the board and reaching out of the chassis, has no internal designator and the internal connector type `DMI_CONNECTOR_TYPE_NONE`; a port without an external connector, such as the sound input of an optical drive, has the same for the external side.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(port_connector)`, and lives as long as the context it has been decoded in.

Fields

`internal_designator`
: Reference designator of the connector inside the enclosure, such as "J12"; `nullptr` if the structure does not carry it.

`internal_connector`
: Type of the connector inside the enclosure, see `dmi_connector_type`.

`external_designator`

Reference designator of the connector on the outside of the enclosure, such as "USB 1"; `nullptr` if the structure does not carry it.

`external_connector`

Type of the connector on the outside of the enclosure, see `dmi_connector_type`.

`port_type`

Function of the port, see `dmi_port_type`.

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_registry_iter_t  iter;
dmi_entity_t        *entity;

dmi_registry_iter_initialize(&iter, registry, nullptr);

while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
    if (dmi_entity_type(entity) != DMI_TYPE(port_connector))
        continue;

    dmi_port_connector_t *port = dmi_entity_info(entity,
                                                 DMI_TYPE(port_connector));
    if ((port != nullptr) && (port->external_designator != nullptr))
        printf("%s: %s\n", port->external_designator,
               dmi_port_type_name(port->port_type));
}
```

See also

`dmi_entity_info`, `dmi_connector_type`, `dmi_port_type`, `dmi_sun_port_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_port_type

Synopsis

```
#include <opendmi/entity/port-connector.h>

typedef enum dmi_port_type {
    DMI_PORT_TYPE_NONE             = 0x00,
    DMI_PORT_TYPE_PARALLEL_XT_AT   = 0x01,
    DMI_PORT_TYPE_PARALLEL_PS2     = 0x02,
```

```
    DMI_PORT_TYPE_PARALLEL_ECP     = 0x03,
    DMI_PORT_TYPE_PARALLEL_EPP     = 0x04,
    DMI_PORT_TYPE_PARALLEL_ECP_EPP = 0x05,
    DMI_PORT_TYPE_SERIAL_XT_AT     = 0x06,
    DMI_PORT_TYPE_SERIAL_16450     = 0x07,
    DMI_PORT_TYPE_SERIAL_16550     = 0x08,
    DMI_PORT_TYPE_SERIAL_16550A    = 0x09,
    DMI_PORT_TYPE_SCSI             = 0x0A,
    DMI_PORT_TYPE_MIDI             = 0x0B,
    DMI_PORT_TYPE_JOYSTICK         = 0x0C,
    DMI_PORT_TYPE_KEYBOARD         = 0x0D,
    DMI_PORT_TYPE_MOUSE            = 0x0E,
    DMI_PORT_TYPE_SCSI_SSA         = 0x0F,
    DMI_PORT_TYPE_USB              = 0x10,
    DMI_PORT_TYPE_FIREWIRE         = 0x11,
    DMI_PORT_TYPE_PCMCIA_I         = 0x12,
    DMI_PORT_TYPE_PCMCIA_II        = 0x13,
    DMI_PORT_TYPE_PCMCIA_III       = 0x14,
    DMI_PORT_TYPE_CARDBUS          = 0x15,
    DMI_PORT_TYPE_ACCESS_BUS       = 0x16,
    DMI_PORT_TYPE_SCSI_2           = 0x17,
    DMI_PORT_TYPE_SCSI_WIDE        = 0x18,
    DMI_PORT_TYPE_PC_98            = 0x19,
    DMI_PORT_TYPE_PC_98_HIRESO     = 0x1A,
    DMI_PORT_TYPE_PC_H98           = 0x1B,
    DMI_PORT_TYPE_VIDEO            = 0x1C,
    DMI_PORT_TYPE_AUDIO            = 0x1D,
    DMI_PORT_TYPE_MODEM            = 0x1E,
    DMI_PORT_TYPE_NETWORK          = 0x1F,
    DMI_PORT_TYPE_SATA             = 0x20,
    DMI_PORT_TYPE_SAS              = 0x21,
    DMI_PORT_TYPE_MFDP             = 0x22,
    DMI_PORT_TYPE_THUNDERBOLT      = 0x23,
    DMI_PORT_TYPE_8251             = 0xA0,
    DMI_PORT_TYPE_8251_FIFO        = 0xA1,
    DMI_PORT_TYPE_OTHER            = 0xFF,
} dmi_port_type_t;

const char *dmi_port_type_name(dmi_port_type_t value);
```

Description

The `dmi_port_type_t` enumeration tells what a port is for, and is the type of the `port_type` field of `dmi_port_connector_t`. The specification assigns no values from `0x24` to `0x9F` and from `0xA2` to `0xFE`.

The `dmi_port_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_PORT_TYPE_NONE`
No port.

`DMI_PORT_TYPE_PARALLEL_XT_AT`
Parallel port, XT/AT compatible.

`DMI_PORT_TYPE_PARALLEL_PS2`
Parallel port, PS/2.

`DMI_PORT_TYPE_PARALLEL_ECP`
Parallel port, ECP.

`DMI_PORT_TYPE_PARALLEL_EPP`
Parallel port, EPP.

`DMI_PORT_TYPE_PARALLEL_ECP_EPP`
Parallel port, ECP/EPP.

`DMI_PORT_TYPE_SERIAL_XT_AT`
Serial port, XT/AT compatible.

`DMI_PORT_TYPE_SERIAL_16450`
Serial port, 16450 compatible.

`DMI_PORT_TYPE_SERIAL_16550`
Serial port, 16550 compatible.

`DMI_PORT_TYPE_SERIAL_16550A`
Serial port, 16550A compatible.

`DMI_PORT_TYPE_SCSI`
SCSI port.

`DMI_PORT_TYPE_MIDI`
MIDI port.

`DMI_PORT_TYPE_JOYSTICK`
Joystick port.

`DMI_PORT_TYPE_KEYBOARD`
Keyboard port.

`DMI_PORT_TYPE_MOUSE`
Mouse port.

`DMI_PORT_TYPE_SCSI_SSA`
SSA SCSI.

`DMI_PORT_TYPE_USB`
: USB.

`DMI_PORT_TYPE_FIREWIRE`
: FireWire (IEEE P1394).

`DMI_PORT_TYPE_PCMCIA_I`
: PC Card (PCMCIA) type I.

`DMI_PORT_TYPE_PCMCIA_II`
: PC Card (PCMCIA) type II.

`DMI_PORT_TYPE_PCMCIA_III`
: PC Card (PCMCIA) type III.

`DMI_PORT_TYPE_CARDBUS`
: CardBus.

`DMI_PORT_TYPE_ACCESS_BUS`
: ACCESS.bus port.

`DMI_PORT_TYPE_SCSI_2`
: SCSI-2.

`DMI_PORT_TYPE_SCSI_WIDE`
: Wide SCSI.

`DMI_PORT_TYPE_PC_98`
: PC-98.

`DMI_PORT_TYPE_PC_98_HIRESO`
: PC-98 Hireso.

`DMI_PORT_TYPE_PC_H98`
: PC-H98.

`DMI_PORT_TYPE_VIDEO`
: Video port.

`DMI_PORT_TYPE_AUDIO`
: Audio port.

`DMI_PORT_TYPE_MODEM`
: Modem port.

`DMI_PORT_TYPE_NETWORK`
: Network port.

`DMI_PORT_TYPE_SATA`
: SATA.

`DMI_PORT_TYPE_SAS`
: SAS.

`DMI_PORT_TYPE_MFDP`
: Multi-Function Display Port (MFDP).

`DMI_PORT_TYPE_THUNDERBOLT`
: Thunderbolt.

`DMI_PORT_TYPE_8251`
: 8251 compatible.

`DMI_PORT_TYPE_8251_FIFO`
: 8251 FIFO compatible.

`DMI_PORT_TYPE_OTHER`
: Other.

Return value

`dmi_port_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_port_connector`, `dmi_connector_type`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/power-controls.h>

dmi_power_controls

Synopsis

```
#include <opendmi/entity/power-controls.h>

typedef struct dmi_power_controls {
    unsigned short poweron_month;
    unsigned short poweron_day;
    unsigned short poweron_hour;
    unsigned short poweron_minute;
    unsigned short poweron_second;
} dmi_power_controls_t;
```

Description

The `dmi_power_controls_t` structure holds the decoded system power controls structure (type 25) of SMBIOS. Its presence tells that the system can be powered on at a set time; the fields give the date and time of the next scheduled power-on, from which software computes the number of seconds left until it.

The structure stores the values as binary-coded decimals; the decoder converts them to plain numbers. A field which is not set holds 0xFF, which is no decimal, and is decoded into `USHRT_MAX`. Such a field, and any value out of the range of its field, takes no part in the schedule: for example, a month which is not set means that the power-on falls within the next month.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(power_controls)`, and lives as long as the context it has been decoded in.

Fields

`poweron_month`
: Month of the next scheduled power-on, 1 to 12.

`poweron_day`
: Day of the month of the next scheduled power-on, 1 to 31.

`poweron_hour`
: Hour of the next scheduled power-on, 0 to 23.

`poweron_minute`
: Minute of the next scheduled power-on, 0 to 59.

`poweron_second`
: Second of the next scheduled power-on, 0 to 59.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
                                     DMI_TYPE(power_controls), false);
dmi_power_controls_t *power    = dmi_entity_info(entity,
                                     DMI_TYPE(power_controls));

if (power != nullptr)
    printf("Next power-on: %02u-%02u %02u:%02u:%02u\n",
           power->poweron_month, power->poweron_day, power->poweron_hour,
           power->poweron_minute, power->poweron_second);
```

See also

`dmi_entity_info`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/power-supply.h>

dmi_power_supply

Synopsis

```
#include <opendmi/entity/power-supply.h>

typedef struct dmi_power_supply {
    unsigned short group;
    const char *location;
    const char *name;
    const char *vendor;
    const char *serial_number;
    const char *asset_tag;
    const char *part_number;
    const char *revision;
    short maximum_capacity;
    bool hot_swappable;
    bool present;
    bool unplugged;
    dmi_range_switching_type_t range_switching;
    dmi_status_t status;
    dmi_power_supply_type_t type;
    dmi_handle_t voltage_probe_handle;
    dmi_entity_t *voltage_probe;
    dmi_handle_t cooling_device_handle;
    dmi_entity_t *cooling_device;
    dmi_handle_t current_probe_handle;
    dmi_entity_t *current_probe;
} dmi_power_supply_t;
```

Description

The `dmi_power_supply_t` structure holds the decoded system power supply structure (type 39) of SMBIOS: the identity, the capacity and the state of a power supply of the system, and the probes and the cooling device attached to it.

A table has one such structure per power supply the system can hold, whether the supply is installed or not. Supplies backing each other up share the same non-zero `group`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(power_supply)`, and lives as long as the context it has been decoded in.

Fields

`group`

Number of the redundant power unit the supply belongs to. Supplies sharing a non-zero group back each other up, and there are at least two of them. Zero if the supply is not part of a redundant unit.

`location`

Where the supply resides, e.g. the name of its bay, or `nullptr`.

`name`

Name of the supply device, e.g. its model, or `nullptr`.

`vendor`

Name of the manufacturer of the supply, or `nullptr`.

`serial_number`

Serial number of the supply, or `nullptr`.

`asset_tag`

Asset tag of the supply, or `nullptr`.

`part_number`

Part number the OEM orders the supply by, or `nullptr`.

`revision`

Revision of the supply, or `nullptr`.

`maximum_capacity`

Largest power the supply can deliver without interruption, in watts. `SHRT_MIN` if unknown. The specification also mentions milliwatts as the unit of the field; the value is taken as it is, in watts.

`hot_swappable`

`true` if the supply can be replaced while the system is running.

`present`

`true` if the supply is installed.

`unplugged`

`true` if the supply is disconnected from the mains.

`range_switching`

How the supply selects the range of its input voltage, see `dmi_range_switching_type`.

`status`

State of the supply, see `dmi_status`. `DMI_STATUS_CRITICAL` means that the supply has failed and has been taken off-line.

`type`

Kind of the supply, see `dmi_power_supply_type`.

`voltage_probe_handle`

Handle of the voltage probe structure (type 26) measuring the input voltage of the supply. `DMI_HANDLE_INVALID` if there is none, or if the structure is too short to hold it: the three handles are

optional, and the structures of 0x10 bytes end before them.

`voltage_probe`

Voltage probe of the supply, resolved from `voltage_probe_handle` once the table has been linked. `nullptr` if the handle is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or does not refer to a voltage probe structure.

`cooling_device_handle`

Handle of the cooling device structure (type 27) cooling the supply. `DMI_HANDLE_INVALID` if there is none, or if the structure is too short to hold it.

`cooling_device`

Cooling device of the supply, resolved from `cooling_device_handle` once the table has been linked. `nullptr` if the handle is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or does not refer to a cooling device structure.

`current_probe_handle`

Handle of the electrical current probe structure (type 29) measuring the input current of the supply. `DMI_HANDLE_INVALID` if there is none, or if the structure is too short to hold it.

`current_probe`

Electrical current probe of the supply, resolved from `current_probe_handle` once the table has been linked. `nullptr` if the handle is `DMI_HANDLE_INVALID` or `DMI_HANDLE_UNSUPPORTED`, or does not refer to an electrical current probe structure.

Examples

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_entity_t       *entity   = dmi_registry_lookup_first(registry,
                                       DMI_TYPE(power_supply), false);
dmi_power_supply_t *supply   = dmi_entity_info(entity,
                                            DMI_TYPE(power_supply));

if ((supply != nullptr) && supply->present &&
    (supply->maximum_capacity != SHRT_MIN))
    printf("Power supply: %d W\n", supply->maximum_capacity);
```

See also

`dmi_entity_info`,`dmi_power_supply_type`,`dmi_range_switching_type`,`dmi_status`,`dmi_probe`,`dmi_cooling_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3.1 or newer.

dmi_power_supply_type

Synopsis

```
#include <opendmi/entity/power-supply.h>

typedef enum dmi_power_supply_type {
    DMI_POWER_SUPPLY_TYPE_UNSPEC    = 0x00,
    DMI_POWER_SUPPLY_TYPE_OTHER     = 0x01,
    DMI_POWER_SUPPLY_TYPE_UNKNOWN   = 0x02,
    DMI_POWER_SUPPLY_TYPE_LINEAR    = 0x03,
    DMI_POWER_SUPPLY_TYPE_SWITCHING = 0x04,
    DMI_POWER_SUPPLY_TYPE_BATTERY   = 0x05,
    DMI_POWER_SUPPLY_TYPE_UPS       = 0x06,
    DMI_POWER_SUPPLY_TYPE_CONVERTER = 0x07,
    DMI_POWER_SUPPLY_TYPE_REGULATOR = 0x08,
} dmi_power_supply_type_t;

const char *dmi_power_supply_type_name(dmi_power_supply_type_t value);
```

Description

The `dmi_power_supply_type_t` enumeration tells what kind of power supply a system power supply structure describes, and is the type of the `type` field of `dmi_power_supply_t`. The field is four bits wide; the values from 0x09 to 0x0F are reserved by the specification.

The `dmi_power_supply_type_name` function returns the name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Values

`DMI_POWER_SUPPLY_TYPE_UNSPEC`

Unspecified. The specification defines no type of zero.

`DMI_POWER_SUPPLY_TYPE_OTHER`

Other type, which none of the values below names.

`DMI_POWER_SUPPLY_TYPE_UNKNOWN`

Unknown.

`DMI_POWER_SUPPLY_TYPE_LINEAR`

Linear power supply.

`DMI_POWER_SUPPLY_TYPE_SWITCHING`

Switching power supply.

`DMI_POWER_SUPPLY_TYPE_BATTERY`

Battery.

`DMI_POWER_SUPPLY_TYPE_UPS`
: Uninterruptible power supply (UPS).

`DMI_POWER_SUPPLY_TYPE_CONVERTER`
: Converter.

`DMI_POWER_SUPPLY_TYPE_REGULATOR`
: Regulator.

Return value

`dmi_power_supply_type_name` returns the name of the value, or `nullptr` if the value has no name, e.g. a value the specification reserves. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None.

See also

`dmi_power_supply`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3.1 or newer.

dmi_range_switching_type

Synopsis

```
#include <opendmi/entity/power-supply.h>

typedef enum dmi_range_switching_type {
    DMI_RANGE_SWITCHING_TYPE_UNSPEC         = 0x00,
    DMI_RANGE_SWITCHING_TYPE_OTHER          = 0x01,
    DMI_RANGE_SWITCHING_TYPE_UNKNOWN        = 0x02,
    DMI_RANGE_SWITCHING_TYPE_MANUAL         = 0x03,
    DMI_RANGE_SWITCHING_TYPE_AUTO           = 0x04,
    DMI_RANGE_SWITCHING_TYPE_WIDE           = 0x05,
    DMI_RANGE_SWITCHING_TYPE_NOT_APPLICABLE = 0x06,
} dmi_range_switching_type_t;

const char *dmi_range_switching_type_name(dmi_range_switching_type_t value);
```

Description

The `dmi_range_switching_type_t` enumeration tells how a power supply selects the range of its input

voltage, and is the type of the `range_switching` field of `dmi_power_supply_t`. The field is four bits wide; the values from 0x07 to 0x0F are reserved by the specification.

The `dmi_range_switching_type_name` function returns the name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale` if the library is built with ICU4C support and the locale has a translation of it, and is in English otherwise.

Values

`DMI_RANGE_SWITCHING_TYPE_UNSPEC`

Unspecified. The specification defines no value of zero.

`DMI_RANGE_SWITCHING_TYPE_OTHER`

Other way, which none of the values below names.

`DMI_RANGE_SWITCHING_TYPE_UNKNOWN`

Unknown.

`DMI_RANGE_SWITCHING_TYPE_MANUAL`

The range is selected by hand.

`DMI_RANGE_SWITCHING_TYPE_AUTO`

The range is selected automatically.

`DMI_RANGE_SWITCHING_TYPE_WIDE`

The supply accepts a wide range of input voltages, with no switching.

`DMI_RANGE_SWITCHING_TYPE_NOT_APPLICABLE`

Range switching does not apply to the supply.

Return value

`dmi_range_switching_type_name` returns the name of the value, or `nullptr` if the value has no name, e.g. a value the specification reserves. The string belongs to the library and must not be modified or freed; a translated name may become invalid once the locale is changed.

Errors

None.

See also

`dmi_power_supply`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3.1 or newer.

<opendmi/entity/probe.h>

dmi_probe

Synopsis

```
#include <opendmi/entity/probe.h>

typedef struct dmi_probe {
    const char *description;
    dmi_probe_location_t location;
    dmi_status_t status;
    short maximum_value;
    short minimum_value;
    short resolution;
    short tolerance;
    short accuracy;
    uint32_t oem_defined;
    short nominal_value;
} dmi_probe_t;

#include <opendmi/entity/voltage-probe.h>

typedef struct dmi_probe dmi_voltage_probe_t;

#include <opendmi/entity/temperature-probe.h>

typedef struct dmi_probe dmi_temperature_probe_t;

#include <opendmi/entity/current-probe.h>

typedef struct dmi_probe dmi_current_probe_t;
```

Description

The `dmi_probe_t` structure holds a decoded voltage probe (type 26), temperature probe (type 28) or electrical current probe (type 29) structure of SMBIOS: a sensor of the system, where it is, its status, and the range and precision of its readings. There is one structure for each probe. The three types of structures are laid out the same way and differ only in the units of their values, so they share the structure, which each of the headers of the probes names after its type: `dmi_voltage_probe_t`, `dmi_temperature_probe_t` and `dmi_current_probe_t`.

The values are in millivolts for voltage probes, in milliamperes for electrical current probes, and in tenths of a degree Celsius for temperature probes, unless said otherwise below. A management device component structure (`dmi_mgmt_device_component_t`) may tie a probe to the management device reading it, and to its thresholds.

The structure is returned by `dmi_entity_info` for the structures of types `DMI_TYPE(voltage_probe)`, `DMI_TYPE(temperature_probe)` and `DMI_TYPE(current_probe)`, and lives as long as the context it has been decoded in.

Fields

`description`

Additional description of the probe or of its location, or `nullptr` if there is none.

`location`

Physical location of the probe.

`status`

Status of the voltage, temperature or current the probe monitors.

`maximum_value`

Largest value the probe can read, or `SHRT_MIN` if unknown.

`minimum_value`

Smallest value the probe can read, or `SHRT_MIN` if unknown.

`resolution`

Resolution of the readings of the probe, or `SHRT_MIN` if unknown. The unit is ten times as fine as the one of the other values for voltage and current probes (tenths of a millivolt or of a milliampere), and a thousandth of a degree Celsius for temperature probes.

`tolerance`

Tolerance of the readings of the probe, plus or minus, or `SHRT_MIN` if unknown.

`accuracy`

Accuracy of the readings of the probe, plus or minus, in hundredths of a percent, or `SHRT_MIN` if unknown.

`oem_defined`

Information specific to the OEM or to the firmware vendor.

`nominal_value`

Nominal reading of the probe, or `SHRT_MIN` if unknown or if the structure is too short to carry it.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   =
        dmi_registry_lookup_first(registry, DMI_TYPE(temperature_probe),
                                  true);
dmi_probe_t    *probe    =
        dmi_entity_info(entity, DMI_TYPE(temperature_probe));

if (probe != nullptr) {
    printf("%s: %s\n", dmi_probe_location_name(probe->location),
```

```
            dmi_status_name(probe->status));

    if (probe->nominal_value != SHRT_MIN)
        printf("Nominal: %.1f C\n", probe->nominal_value / 10.0);
}
```

See also

`dmi_entity_info`, `dmi_probe_location`, `dmi_status`, `dmi_mgmt_device_component`, `dmi_cooling_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

dmi_probe_location

Synopsis

```
#include <opendmi/entity/probe.h>

typedef enum dmi_probe_location {
    DMI_PROBE_LOCATION_UNSPEC             = 0x00,
    DMI_PROBE_LOCATION_OTHER              = 0x01,
    DMI_PROBE_LOCATION_UNKNOWN            = 0x02,
    DMI_PROBE_LOCATION_PROCESSOR          = 0x03,
    DMI_PROBE_LOCATION_DISK               = 0x04,
    DMI_PROBE_LOCATION_PERIPHERAL_BAY     = 0x05,
    DMI_PROBE_LOCATION_SYSTEM_MGMT_MODULE = 0x06,
    DMI_PROBE_LOCATION_MOTHERBOARD        = 0x07,
    DMI_PROBE_LOCATION_MEMORY_MODULE      = 0x08,
    DMI_PROBE_LOCATION_PROCESSOR_MODULE   = 0x09,
    DMI_PROBE_LOCATION_POWER_UNIT         = 0x0A,
    DMI_PROBE_LOCATION_ADDIN_CARD         = 0x0B,
    DMI_PROBE_LOCATION_FRONT_PANEL_BOARD  = 0x0C,
    DMI_PROBE_LOCATION_BACK_PANEL_BOARD   = 0x0D,
    DMI_PROBE_LOCATION_POWER_SYSTEM_BOARD = 0x0E,
    DMI_PROBE_LOCATION_DRIVE_BACK_PLANE   = 0x0F,
} dmi_probe_location_t;

const char *dmi_probe_location_name(dmi_probe_location_t value);
```

Description

The `dmi_probe_location_t` enumeration tells where a probe is in the system, and is the type of the `location` field of `dmi_probe_t`, the structure of voltage, temperature and electrical current probes. The values from `DMI_PROBE_LOCATION_FRONT_PANEL_BOARD` on are defined for temperature probes only.

The `dmi_probe_location_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_PROBE_LOCATION_UNSPEC`
: Unspecified.

`DMI_PROBE_LOCATION_OTHER`
: Other location.

`DMI_PROBE_LOCATION_UNKNOWN`
: Unknown location.

`DMI_PROBE_LOCATION_PROCESSOR`
: Processor.

`DMI_PROBE_LOCATION_DISK`
: Disk.

`DMI_PROBE_LOCATION_PERIPHERAL_BAY`
: Peripheral bay.

`DMI_PROBE_LOCATION_SYSTEM_MGMT_MODULE`
: System management module.

`DMI_PROBE_LOCATION_MOTHERBOARD`
: Motherboard.

`DMI_PROBE_LOCATION_MEMORY_MODULE`
: Memory module.

`DMI_PROBE_LOCATION_PROCESSOR_MODULE`
: Processor module.

`DMI_PROBE_LOCATION_POWER_UNIT`
: Power unit.

`DMI_PROBE_LOCATION_ADDIN_CARD`
: Add-in card.

`DMI_PROBE_LOCATION_FRONT_PANEL_BOARD`
: Front panel board (temperature probes only).

`DMI_PROBE_LOCATION_BACK_PANEL_BOARD`
: Back panel board (temperature probes only).

`DMI_PROBE_LOCATION_POWER_SYSTEM_BOARD`
: Power system board (temperature probes only).

`DMI_PROBE_LOCATION_DRIVE_BACK_PLANE`
: Drive back plane (temperature probes only).

Return value

`dmi_probe_location_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_probe`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/processor-ex.h>

dmi_processor_aarch64_arch_data

Synopsis

```
#include <opendmi/entity/processor-ex.h>

struct dmi_processor_aarch64_arch_data {
    dmi_word_t version;
    dmi_byte_t length;
    dmi_byte_t __reserved;
    dmi_dword_t __reserved2;
    dmi_qword_t id_aa64afr0_el1;
    dmi_qword_t id_aa64afr1_el1;
    dmi_qword_t id_aa64dfr0_el1;
    dmi_qword_t id_aa64dfr1_el1;
    dmi_qword_t id_aa64dfr2_el1;
    dmi_qword_t id_aa64fpfr0_el1;
    dmi_qword_t id_aa64isar0_el1;
    dmi_qword_t id_aa64isar1_el1;
    dmi_qword_t id_aa64isar2_el1;
    dmi_qword_t id_aa64isar3_el1;
    dmi_qword_t id_aa64mmfr0_el1;
    dmi_qword_t id_aa64mmfr1_el1;
    dmi_qword_t id_aa64mmfr2_el1;
    dmi_qword_t id_aa64mmfr3_el1;
    dmi_qword_t id_aa64mmfr4_el1;
    dmi_qword_t id_aa64pfr0_el1;
    dmi_qword_t id_aa64pfr1_el1;
    dmi_qword_t id_aa64pfr2_el1;
    dmi_qword_t id_aa64smfr0_el1;
    dmi_qword_t id_aa64zfr0_el1;
};
```

Description

The `dmi_processor_aarch64_arch_data` structure describes the AArch64 architecture data, which the 64-bit Arm processor-specific data hold when their sub-type is `DMI_PROCESSOR_AARCH64_DATA_SUBTYPE_AARCH64`: the values of the identification registers of the processor, which tell the features it implements. The bit fields of the registers are defined by the Arm A-profile architecture. The structure has no typedef.

The structure is packed: it has no padding, and its fields are laid out as in the data, whose multi-byte values are little-endian. The decoder does not use it: it may be laid over the `subtype_specific_data` field of `dmi_processor_aarch64_data_t` to read the processor-specific data, once its architecture and length have been checked.

Fields

`version`
Version of the format of the data, 1 at present.

`length`
Length of the data, in bytes (168 at present).

`__reserved`
Reserved, zero.

`__reserved2`
Reserved, zero.

`id_aa64afr0_el1`
Auxiliary feature register 0.

`id_aa64afr1_el1`
Auxiliary feature register 1.

`id_aa64dfr0_el1`
Debug feature register 0.

`id_aa64dfr1_el1`
Debug feature register 1.

`id_aa64dfr2_el1`
Debug feature register 2.

`id_aa64fpfr0_el1`
Floating-point feature register 0.

`id_aa64isar0_el1`
Instruction set attribute register 0.

`id_aa64isar1_el1`
Instruction set attribute register 1.

`id_aa64isar2_el1`
Instruction set attribute register 2.

`id_aa64isar3_el1`
Instruction set attribute register 3.

`id_aa64mmfr0_el1`
Memory model feature register 0. Processors of a system may support different physical address ranges; the register then gives the smallest of them.

`id_aa64mmfr1_el1`
Memory model feature register 1.

`id_aa64mmfr2_el1`
Memory model feature register 2.

`id_aa64mmfr3_el1`
Memory model feature register 3.

`id_aa64mmfr4_el1`
Memory model feature register 4.

`id_aa64pfr0_el1`
Processor feature register 0.

`id_aa64pfr1_el1`
Processor feature register 1.

`id_aa64pfr2_el1`
Processor feature register 2.

`id_aa64smfr0_el1`
SME feature identification register 0.

`id_aa64zfr0_el1`
SVE feature identification register 0.

See also

`dmi_processor_aarch64_data`, `dmi_processor_aarch64_data_subtype`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.8 or newer.

dmi_processor_aarch64_data

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef struct dmi_processor_aarch64_data {
    dmi_processor_revision_t revision;
    dmi_byte_t length;
    dmi_byte_t __reserved;
    dmi_word_t vendor_id;
    dmi_byte_t subtype;
    dmi_byte_t __reserved2;
    dmi_byte_t subtype_specific_data[];
} dmi_processor_aarch64_data_t;
```

Description

The `dmi_processor_aarch64_data_t` structure describes the processor-specific data of the processor additional information structures (type 44) of 64-bit Arm (AArch64) processors: a header naming the vendor of the processor and the sub-type of the data, followed by the data of the sub-type. A system has at most one structure of each sub-type.

The structure is packed: it has no padding, and its fields are laid out as in the data, whose multi-byte values are little-endian. The decoder does not use it: it may be laid over the `data` field of `dmi_processor_ex_t` to read the processor-specific data, once its architecture and length have been checked.

Fields

`revision`

Revision of the format of the data, 1.0 at present.

`length`

Length of the processor-specific data, in bytes: the 8 bytes of the header and the length of the sub-type specific data.

`__reserved`

Reserved, zero.

`vendor_id`

JEP-106 code of the vendor of the processor or of the silicon provider: the bank index in bits 14:8, and the identification code with its parity bit in bits 7:0. Bit 15 is zero.

`subtype`

Sub-type of the data, one of the `dmi_processor_aarch64_data_subtype_t` values.

`__reserved2`

Reserved, zero.

`subtype_specific_data`

Data of the sub-type: AArch64 architecture data (`dmi_processor_aarch64_arch_data`), or data defined by the vendor.

See also

`dmi_processor_ex`, `dmi_processor_revision`, `dmi_processor_aarch64_data_subtype`, `dmi_processor_aarch64_arch_data`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.8 or newer.

dmi_processor_aarch64_data_subtype

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef enum dmi_processor_aarch64_data_subtype {
    DMI_PROCESSOR_AARCH64_DATA_SUBTYPE_AARCH64         = 0x00,
    DMI_PROCESSOR_AARCH64_DATA_SUBTYPE_VENDOR_SPECIFIC = 0x01,
} dmi_processor_aarch64_data_subtype_t;
```

Description

The `dmi_processor_aarch64_data_subtype_t` enumeration tells what the 64-bit Arm (AArch64) processor-specific data hold after their header, and is the values of the `subtype` field of `dmi_processor_aarch64_data_t`. The other values are reserved.

Values

`DMI_PROCESSOR_AARCH64_DATA_SUBTYPE_AARCH64`
: AArch64 architecture data, see `dmi_processor_aarch64_arch_data`.

`DMI_PROCESSOR_AARCH64_DATA_SUBTYPE_VENDOR_SPECIFIC`
: Data defined by the vendor of the processor.

See also

`dmi_processor_aarch64_data`, `dmi_processor_aarch64_arch_data`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.8 or newer.

dmi_processor_amd64_attribute

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef struct dmi_processor_amd64_attribute {
    dmi_byte_t use_condition;
    dmi_byte_t temperature_range;
    dmi_word_t __reserved;
} dmi_processor_amd64_attribute_t;
```

Description

The `dmi_processor_amd64_attribute_t` structure holds the attributes of the processor use condition data of x64 processors, and is the type of the `attr` field of `dmi_processor_amd64_data`. The structure is

packed, and is laid out as in the data, a little-endian double word.

Fields

`use_condition`
: Use condition of the processor, bits 7:0, one of the `dmi_processor_amd64_use_condition_t` values.

`temperature_range`
: Temperature range of the processor, bits 15:8, one of the `dmi_processor_amd64_temperature_range_t` values.

`__reserved`
: Reserved, bits 31:16.

See also

`dmi_processor_amd64_data`, `dmi_processor_amd64_use_condition`, `dmi_processor_amd64_temperature_range`

Standards

Not part of the SMBIOS specification itself, which leaves the format of the x64 processor-specific data to a separate document maintained by Intel (https://github.com/intel/smbios).

dmi_processor_amd64_data

Synopsis

```
#include <opendmi/entity/processor-ex.h>

struct dmi_processor_amd64_data {
    dmi_byte_t id;
    dmi_byte_t length;
    dmi_processor_revision_t revision;
    dmi_processor_amd64_attribute_t attr;
};
```

Description

The `dmi_processor_amd64_data` structure describes the processor use condition data, which the processor-specific data of x64 processors (x86-64, Intel 64, AMD64) may hold: the conditions, such as the temperature range, the processor is made to be used in. The structure has no typedef.

The structure is packed: it has no padding, and its fields are laid out as in the data, whose multi-byte values are little-endian. The decoder does not use it: it may be laid over the `data` field of `dmi_processor_ex_t` to read the processor-specific data, once its architecture and length have been checked.

Fields

`id`
: Identifier of the processor use condition data, 1.

`length`
: Length of the processor use condition data, in bytes.

`revision`
: Revision of the format of the data.

`attr`
: Use condition and temperature range of the processor.

See also

`dmi_processor_ex`, `dmi_processor_revision`, `dmi_processor_amd64_attribute`

Standards

Not part of the SMBIOS specification itself, which leaves the format of the x64 processor-specific data to a separate document maintained by Intel (https://github.com/intel/smbios).

dmi_processor_amd64_temperature_range

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef enum dmi_processor_amd64_temperature_range {
    DMI_PROCESSOR_AMD64_TEMPERATURE_RANGE_COMMERCIAL = 0x00,
    DMI_PROCESSOR_AMD64_TEMPERATURE_RANGE_EXTENDED   = 0x01
} dmi_processor_amd64_temperature_range_t;
```

Description

The `dmi_processor_amd64_temperature_range_t` enumeration tells the range of temperatures an x64 processor is made to work in, and is the values of the `temperature_range` field of `dmi_processor_amd64_attribute_t`.

Values

`DMI_PROCESSOR_AMD64_TEMPERATURE_RANGE_COMMERCIAL`
: Commercial temperature range.

`DMI_PROCESSOR_AMD64_TEMPERATURE_RANGE_EXTENDED`
: Extended temperature range.

See also

`dmi_processor_amd64_attribute`, `dmi_processor_amd64_data`

Standards

Not part of the SMBIOS specification itself, which leaves the format of the x64 processor-specific data to a separate document maintained by Intel (https://github.com/intel/smbios).

dmi_processor_amd64_use_condition

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef enum dmi_processor_amd64_use_condition {
    DMI_PROCESSOR_AMD64_USE_CONDITION_CLIENT     = 0x00,
    DMI_PROCESSOR_AMD64_USE_CONDITION_EMBEDDED   = 0x01,
    DMI_PROCESSOR_AMD64_USE_CONDITION_INDUSTRIAL = 0x02
} dmi_processor_amd64_use_condition_t;
```

Description

The `dmi_processor_amd64_use_condition_t` enumeration tells the kind of use an x64 processor is made for, and is the values of the `use_condition` field of `dmi_processor_amd64_attribute_t`.

Values

`DMI_PROCESSOR_AMD64_USE_CONDITION_CLIENT`
: Client systems.

`DMI_PROCESSOR_AMD64_USE_CONDITION_EMBEDDED`
: Embedded systems.

`DMI_PROCESSOR_AMD64_USE_CONDITION_INDUSTRIAL`
: Industrial systems.

See also

`dmi_processor_amd64_attribute`, `dmi_processor_amd64_data`

Standards

Not part of the SMBIOS specification itself, which leaves the format of the x64 processor-specific data to a separate document maintained by Intel (https://github.com/intel/smbios).

dmi_processor_arch

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef enum dmi_processor_arch {
    DMI_PROCESSOR_ARCH_RESERVED     = 0x00,
    DMI_PROCESSOR_ARCH_IA32         = 0x01,
    DMI_PROCESSOR_ARCH_AMD64        = 0x02,
    DMI_PROCESSOR_ARCH_ITANIUM      = 0x03,
    DMI_PROCESSOR_ARCH_AARCH32      = 0x04,
    DMI_PROCESSOR_ARCH_AARCH64      = 0x05,
    DMI_PROCESSOR_ARCH_RV32         = 0x06,
    DMI_PROCESSOR_ARCH_RV64         = 0x07,
    DMI_PROCESSOR_ARCH_RV128        = 0x08,
    DMI_PROCESSOR_ARCH_LOONG_ARCH32 = 0x09,
    DMI_PROCESSOR_ARCH_LOONG_ARCH64 = 0x0A,
} dmi_processor_arch_t;

const char *dmi_processor_arch_name(dmi_processor_arch_t value);
```

Description

The `dmi_processor_arch_t` enumeration tells the processor architecture of the processor-specific data of a processor additional information structure, and so their format. It is the type of the `arch` field of `dmi_processor_ex_t`, and the values of the `arch` field of `dmi_processor_specific_block_t`.

The `dmi_processor_arch_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_PROCESSOR_ARCH_RESERVED`

Reserved.

`DMI_PROCESSOR_ARCH_IA32`

IA-32 (32-bit x86).

`DMI_PROCESSOR_ARCH_AMD64`

x64 (x86-64, Intel 64, AMD64, EM64T). See `dmi_processor_amd64_data`.

`DMI_PROCESSOR_ARCH_ITANIUM`

Intel Itanium.

`DMI_PROCESSOR_ARCH_AARCH32`

32-bit Arm (AArch32).

`DMI_PROCESSOR_ARCH_AARCH64`
64-bit Arm (AArch64). See `dmi_processor_aarch64_data`.

`DMI_PROCESSOR_ARCH_RV32`
32-bit RISC-V (RV32). See `dmi_processor_rv64_data`.

`DMI_PROCESSOR_ARCH_RV64`
64-bit RISC-V (RV64). See `dmi_processor_rv64_data`.

`DMI_PROCESSOR_ARCH_RV128`
128-bit RISC-V (RV128). See `dmi_processor_rv64_data`.

`DMI_PROCESSOR_ARCH_LOONG_ARCH32`
32-bit LoongArch (LoongArch32).

`DMI_PROCESSOR_ARCH_LOONG_ARCH64`
64-bit LoongArch (LoongArch64).

Return value

`dmi_processor_arch_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_processor_ex`, `dmi_processor_specific_block`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.3 or newer.

dmi_processor_ex

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef struct dmi_processor_ex {
    dmi_handle_t processor_handle;
    dmi_entity_t *processor;
    dmi_processor_arch_t architecture;
    dmi_binary_t data;
} dmi_processor_ex_t;
```

Description

The `dmi_processor_ex_t` structure holds a decoded processor additional information structure (type 44) of SMBIOS, which completes a processor structure (type 4) with data specific to the architecture of the processor. Several such structures may refer to the same processor, for example to describe cores which are not all alike.

The data are kept as the structure carries them. Their format is defined for each architecture by the architecture workgroups or the vendors; `dmi_processor_aarch64_data_t`, `dmi_processor_amd64_data` and `dmi_processor_rv64_data` describe the layouts of some of them.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(processor_ex)`, and lives as long as the context it has been decoded in.

Fields

`processor_handle`

Handle of the processor structure (type 4) the information is about.

`processor`

Processor structure the information is about, whose information is a `dmi_processor_t`. Set when linking; `nullptr` if the handle refers to no processor structure.

`architecture`

Processor architecture the data are about.

`data`

Processor-specific data: a pointer into the structure, and their length in bytes, as the processor-specific block gives it. The pointer is `nullptr` if the block holds no data.

Examples

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_entity_t       *entity   =
        dmi_registry_lookup_first(registry, DMI_TYPE(processor_ex), true);
dmi_processor_ex_t *info     =
        dmi_entity_info(entity, DMI_TYPE(processor_ex));

if ((info != nullptr) && (info->architecture == DMI_PROCESSOR_ARCH_AARCH64) &&
    (info->data.length >= sizeof(dmi_processor_aarch64_data_t))) {
    const dmi_processor_aarch64_data_t *data =
            (const dmi_processor_aarch64_data_t *)info->data.data;

    printf("AArch64 data, revision %u.%u, sub-type %u\n",
           data->revision.major, data->revision.minor, data->subtype);
}
```

See also

`dmi_entity_info`, `dmi_processor`, `dmi_processor_arch`, `dmi_processor_specific_block`, `dmi_processor_aarch64_data`, `dmi_processor_amd64_data`, `dmi_processor_rv64_data`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.3 or newer.

dmi_processor_revision

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef union dmi_processor_revision {
    dmi_word_t __value;
    struct {
        dmi_byte_t minor;
        dmi_byte_t major;
    };
} dmi_processor_revision_t;
```

Description

The `dmi_processor_revision_t` union holds the revision of the format of processor-specific data, as a word or as its major and minor numbers. It is the type of the `revision` fields of `dmi_processor_aarch64_data_t`, `dmi_processor_amd64_data` and `dmi_processor_rv64_data`.

The union is packed, and is laid out as in the data: the minor number is the low byte of the little-endian word, and the major number its high byte.

Fields

`__value`
: Revision as a word, as the data stores it (little-endian).

`minor`
: Minor revision, bits 7:0 of the word.

`major`
: Major revision, bits 15:8 of the word.

See also

`dmi_processor_aarch64_data`, `dmi_processor_amd64_data`, `dmi_processor_rv64_data`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.8 or newer.

dmi_processor_rv64_data

Synopsis

```
#include <opendmi/entity/processor-ex.h>

struct dmi_processor_rv64_data {
    dmi_processor_revision_t revision;
    dmi_qword_t hart_id;
    dmi_qword_t vendor_id;
    dmi_qword_t arch_id;
    dmi_qword_t machine_impl_id;
};
```

Description

The `dmi_processor_rv64_data` structure describes the processor-specific data of the processor additional information structures (type 44) of RISC-V processors: the hardware thread (hart) described, and the identification registers of its machine mode. The structure has no typedef.

The structure is packed: it has no padding, and its fields are laid out as in the data, whose multi-byte values are little-endian. The decoder does not use it: it may be laid over the `data` field of `dmi_processor_ex_t` to read the processor-specific data, once its architecture and length have been checked.

Fields

`revision`
: Revision of the format of the data.

`hart_id`
: Identifier of the hart the data are about.

`vendor_id`
: Vendor of the core, as the `mvendorid` register holds it.

`arch_id`
: Base microarchitecture of the hart, as the `marchid` register holds it.

`machine_impl_id`
: Version of the implementation of the processor, as the `mimpid` register holds it.

See also

`dmi_processor_ex`, `dmi_processor_revision`, `dmi_processor_arch`

Standards

Not part of the SMBIOS specification itself, which leaves the format of the RISC-V processor-specific data to the SMBIOS requirements of the RISC-V Boot and Runtime Services specification (https://github.com/riscv-non-isa/riscv-brs).

dmi_processor_specific_block

Synopsis

```
#include <opendmi/entity/processor-ex.h>

typedef struct dmi_processor_specific_block {
    dmi_byte_t length;
    dmi_byte_t arch;
    dmi_byte_t __data[];
} dmi_processor_specific_block_t;
```

Description

The `dmi_processor_specific_block_t` structure describes the processor-specific block, which ends a processor additional information structure (type 44): a header naming the length and the architecture of the processor-specific data, followed by the data.

The decoder reads the block into the `arch` and `data` fields of `dmi_processor_ex_t`. The structure is packed, and is laid out as in the data.

Fields

`length`
: Length of the processor-specific data, in bytes.

`arch`
: Processor architecture of the data, one of the `dmi_processor_arch_t` values.

`__data`
: Processor-specific data, `length` bytes.

See also

`dmi_processor_ex`, `dmi_processor_arch`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.3 or newer.

<opendmi/entity/processor.h>

dmi_processor

Synopsis

```
#include <opendmi/entity/processor.h>

typedef struct dmi_processor {
    const char                  *socket_designation;
    dmi_processor_type_t         type;
    dmi_processor_family_t       family;
    const char                  *vendor;
    dmi_binary_t                 id;
    dmi_processor_id_format_t    id_format;
    dmi_processor_x86_id_t       x86_id;
    dmi_processor_arm_id_t       arm_id;
    dmi_processor_soc_id_t       soc_id;
    const char                  *version;
    uint8_t                      voltage;
    dmi_processor_voltages_t     supported_voltages;
    uint16_t                     external_clock;
    uint16_t                     maximum_speed;
    uint16_t                     current_speed;
    bool                         is_populated;
    dmi_processor_status_t       status;
    dmi_processor_upgrade_t      upgrade;
    dmi_handle_t                 l1_cache_handle;
    dmi_entity_t                *l1_cache;
    dmi_handle_t                 l2_cache_handle;
    dmi_entity_t                *l2_cache;
    dmi_handle_t                 l3_cache_handle;
    dmi_entity_t                *l3_cache;
    const char                  *serial_number;
    const char                  *asset_tag;
    const char                  *part_number;
    dmi_processor_features_t     features;
    uint16_t                     core_count;
    uint16_t                     core_enabled;
    uint16_t                     thread_count;
    uint16_t                     thread_enabled;
    const char                  *socket_type;
} dmi_processor_t;
```

Description

The `dmi_processor_t` structure holds the decoded processor information structure (type 4) of SMBIOS: the attributes of one processor socket or slot, and of the processor installed in it. A table has one such structure for every socket, populated or not, so that their number gives the largest number of processors the system can have. A coprocessor, such as a math processor, has a structure of its own.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(processor)`, and lives as long as the context it has been decoded in. The fields a structure is too short to carry are 0 or `nullptr`, except the cache handles, which are `DMI_HANDLE_INVALID`.

Fields

`socket_designation`

Reference designation of the socket, as labelled on the board, for example `J202`, or `nullptr`.

`type`

Processor type, see `dmi_processor_type`.

`family`

Processor family, see `dmi_processor_family`. When the one-byte family field holds `DMI_PROCESSOR_FAMILY_EXTENDED`, the value is taken from the two-byte family field, present since SMBIOS 2.6.

`vendor`

Processor manufacturer, or `nullptr`.

`id`

Processor identifier, the eight bytes the structure carries, as they are stored. Their meaning depends on the architecture of the processor.

`id_format`

Format the identifier has been decoded in, see `dmi_processor_id_format`, telling which of `x86_id`, `arm_id` and `soc_id` holds it. `DMI_PROCESSOR_ID_FORMAT_RAW` if the architecture has not been recognized; the identifier is then only available in `id`.

`x86_id`

Identifier of an x86 processor, see `dmi_processor_x86_id`. Set when `id_format` is `DMI_PROCESSOR_ID_FORMAT_X86`.

`arm_id`

Main ID Register of an Arm processor, see `dmi_processor_arm_id`. Set when `id_format` is `DMI_PROCESSOR_ID_FORMAT_MIDR`.

`soc_id`

SoC identifier of an Arm64 processor, see `dmi_processor_soc_id`. Set when `id_format` is `DMI_PROCESSOR_ID_FORMAT_SOC_ID`.

`version`

Processor version, the name of the model as the vendor gives it, or `nullptr`.

`voltage`

Current voltage of the processor, in tenths of volt. 0 if the structure gives the supported voltages instead, or no voltage at all. The specification deprecates the voltage field since SMBIOS 3.8.

`supported_voltages`

Voltages the socket accepts, see `dmi_processor_voltages`. Set when the structure gives them instead of the current voltage, 0 otherwise.

`external_clock`

External clock frequency, in MHz, 0 if unknown.

`maximum_speed`

Highest processor speed the system supports in this socket, in MHz, 0 if unknown. It tells the capability of the system, not of the installed processor.

`current_speed`

Speed of the processor at system boot, in MHz, 0 if unknown.

`is_populated`

`true` if a processor is installed in the socket.

`status`

State of the processor, see `dmi_processor_status`.

`upgrade`

Upgrade method of the processor, most often the type of the socket, see `dmi_processor_upgrade`.

`l1_cache_handle`, `l2_cache_handle`, `l3_cache_handle`

Handles of the cache information structures (type 7) describing the level 1, 2 and 3 caches of the processor, `DMI_HANDLE_INVALID` if the processor has no such cache or the table has no structure for it. Present since SMBIOS 2.1.

`l1_cache`, `l2_cache`, `l3_cache`

Cache entities the handles refer to, set when the table is linked, `nullptr` if there is none. Present since SMBIOS 2.1.

`serial_number`

Serial number of the processor, or `nullptr`. Present since SMBIOS 2.3.

`asset_tag`

Asset tag of the processor, or `nullptr`. Present since SMBIOS 2.3.

`part_number`

Part number of the processor, or `nullptr`. Present since SMBIOS 2.3.

`features`

Processor characteristics, see `dmi_processor_features`. Present since SMBIOS 2.5.

`core_count`

Number of cores the firmware has detected in the socket, 0 if unknown. It may be less than the processor has, if the hardware limits it. Present since SMBIOS 2.5; counts above 255 are taken from the

two-byte field present since SMBIOS 3.0.

`core_enabled`

Number of cores the firmware has enabled for the operating system, 0 if unknown. Present since SMBIOS 2.5; counts above 255 are taken from the two-byte field present since SMBIOS 3.0.

`thread_count`

Number of threads the firmware has detected in the socket, for the whole processor rather than per core, 0 if unknown. Present since SMBIOS 2.5; counts above 255 are taken from the two-byte field present since SMBIOS 3.0.

`thread_enabled`

Number of threads the firmware has enabled for the operating system, 0 if unknown. Present since SMBIOS 3.6.

`socket_type`

Type of the socket, for example `Socket LGA4710`, or `nullptr`. It names the socket in particular when `upgrade` has no value for it. Present since SMBIOS 3.8.

Examples

```
dmi_registry_t  *registry  = dmi_get_registry(context);
dmi_entity_t    *entity    = dmi_registry_lookup_first(registry,
                                                       DMI_TYPE(processor),
                                                       false);
dmi_processor_t *processor = dmi_entity_info(entity, DMI_TYPE(processor));

if ((processor != nullptr) && processor->is_populated) {
    const char *family = dmi_processor_family_name(processor->family);

    printf("%s, %u MHz, %u cores\n", family ? family : "Unknown family",
           (unsigned)processor->current_speed,
           (unsigned)processor->core_count);
}
```

See also

`dmi_entity_info`, `dmi_processor_type`, `dmi_processor_family`, `dmi_processor_upgrade`, `dmi_processor_status`, `dmi_processor_id_format`, `dmi_processor_x86_id`, `dmi_processor_arm_id`, `dmi_processor_soc_id`, `dmi_processor_voltages`, `dmi_processor_voltage_data`, `dmi_processor_features`, `dmi_cache`, `dmi_processor_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_arm_id

Synopsis

```
#include <opendmi/entity/processor.h>

typedef struct dmi_processor_arm_id {
    uint8_t  implementer;
    uint8_t  variant;
    uint8_t  architecture;
    uint16_t part_number;
    uint8_t  revision;
} dmi_processor_arm_id_t;
```

Description

The `dmi_processor_arm_id_t` structure holds the processor identifier of an Arm processor, decoded from the processor information structure (type 4) of SMBIOS: the Main ID Register (MIDR, or MIDR_EL1 on 64-bit processors).

It is the `arm_id` field of `dmi_processor_t`, set when the `id_format` field of it is `DMI_PROCESSOR_ID_FORMAT_MIDR`: for 32-bit Arm processors, and for 64-bit ones which do not report a SoC identifier.

Fields

`implementer`
: Implementer code, bits 24 to 31 of the register, for example 0x41 for Arm.

`variant`
: Variant number, the major revision of the processor, bits 20 to 23.

`architecture`
: Architecture code, bits 16 to 19.

`part_number`
: Primary part number, bits 4 to 15.

`revision`
: Revision number, the minor revision of the processor, bits 0 to 3.

See also

`dmi_processor`, `dmi_processor_id_format`, `dmi_processor_soc_id`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.1 or newer.

dmi_processor_family

Synopsis

```
#include <opendmi/entity/processor.h>

typedef enum dmi_processor_family {
    DMI_PROCESSOR_FAMILY_UNSPEC                 = 0x00,
    DMI_PROCESSOR_FAMILY_OTHER                  = 0x01,
    DMI_PROCESSOR_FAMILY_UNKNOWN                = 0x02,
    DMI_PROCESSOR_FAMILY_INTEL_8086             = 0x03,
    DMI_PROCESSOR_FAMILY_INTEL_80286            = 0x04,
    DMI_PROCESSOR_FAMILY_INTEL_80386            = 0x05,
    DMI_PROCESSOR_FAMILY_INTEL_80486            = 0x06,
    DMI_PROCESSOR_FAMILY_INTEL_8087             = 0x07,
    DMI_PROCESSOR_FAMILY_INTEL_80287            = 0x08,
    DMI_PROCESSOR_FAMILY_INTEL_80387            = 0x09,
    DMI_PROCESSOR_FAMILY_INTEL_80487            = 0x0A,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM          = 0x0B,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_PRO      = 0x0C,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_2        = 0x0D,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_MMX      = 0x0E,
    DMI_PROCESSOR_FAMILY_INTEL_CELERON          = 0x0F,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_2_XEON   = 0x10,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3        = 0x11,
    DMI_PROCESSOR_FAMILY_APPLE_M1               = 0x12,
    DMI_PROCESSOR_FAMILY_APPLE_M2               = 0x13,
    DMI_PROCESSOR_FAMILY_INTEL_CELERON_M        = 0x14,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_4_HT     = 0x15,
    DMI_PROCESSOR_FAMILY_INTEL                  = 0x16,
    DMI_PROCESSOR_FAMILY_AMD_DURON              = 0x18,
    DMI_PROCESSOR_FAMILY_AMD_K5                 = 0x19,
    DMI_PROCESSOR_FAMILY_AMD_K6                 = 0x1A,
    DMI_PROCESSOR_FAMILY_AMD_K6_2               = 0x1B,
    DMI_PROCESSOR_FAMILY_AMD_K6_3               = 0x1C,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON             = 0x1D,
    DMI_PROCESSOR_FAMILY_AMD_29000              = 0x1E,
    DMI_PROCESSOR_FAMILY_AMD_K6_2_PLUS          = 0x1F,
    DMI_PROCESSOR_FAMILY_POWERPC                = 0x20,
    DMI_PROCESSOR_FAMILY_POWERPC_601            = 0x21,
    DMI_PROCESSOR_FAMILY_POWERPC_603            = 0x22,
    DMI_PROCESSOR_FAMILY_POWERPC_603_PLUS       = 0x23,
    DMI_PROCESSOR_FAMILY_POWERPC_604            = 0x24,
    DMI_PROCESSOR_FAMILY_POWERPC_620            = 0x25,
    DMI_PROCESSOR_FAMILY_POWERPC_X704           = 0x26,
    DMI_PROCESSOR_FAMILY_POWERPC_750            = 0x27,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_DUO         = 0x28,
```

```
    DMI_PROCESSOR_FAMILY_INTEL_CORE_DUO_M        = 0x29,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_SOLO_M       = 0x2A,
    DMI_PROCESSOR_FAMILY_INTEL_ATOM              = 0x2B,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_M            = 0x2C,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_M3           = 0x2D,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_M5           = 0x2E,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_M7           = 0x2F,
    DMI_PROCESSOR_FAMILY_ALPHA                   = 0x30,
    DMI_PROCESSOR_FAMILY_ALPHA_21064             = 0x31,
    DMI_PROCESSOR_FAMILY_ALPHA_21066             = 0x32,
    DMI_PROCESSOR_FAMILY_ALPHA_21164             = 0x33,
    DMI_PROCESSOR_FAMILY_ALPHA_21164PC           = 0x34,
    DMI_PROCESSOR_FAMILY_ALPHA_21164A            = 0x35,
    DMI_PROCESSOR_FAMILY_ALPHA_21264             = 0x36,
    DMI_PROCESSOR_FAMILY_ALPHA_21364             = 0x37,
    DMI_PROCESSOR_FAMILY_AMD_TURION_2_ULTRA_2C_M = 0x38,
    DMI_PROCESSOR_FAMILY_AMD_TURION_2_2C_M       = 0x39,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_2_2C_M       = 0x3A,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_6100        = 0x3B,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_4100        = 0x3C,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_6200        = 0x3D,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_4200        = 0x3E,
    DMI_PROCESSOR_FAMILY_AMD_FX                  = 0x3F,
    DMI_PROCESSOR_FAMILY_MIPS                    = 0x40,
    DMI_PROCESSOR_FAMILY_MIPS_R4000              = 0x41,
    DMI_PROCESSOR_FAMILY_MIPS_R4200              = 0x42,
    DMI_PROCESSOR_FAMILY_MIPS_R4400              = 0x43,
    DMI_PROCESSOR_FAMILY_MIPS_R4600              = 0x44,
    DMI_PROCESSOR_FAMILY_MIPS_R10000             = 0x45,
    DMI_PROCESSOR_FAMILY_AMD_C                   = 0x46,
    DMI_PROCESSOR_FAMILY_AMD_E                   = 0x47,
    DMI_PROCESSOR_FAMILY_AMD_A                   = 0x48,
    DMI_PROCESSOR_FAMILY_AMD_G                   = 0x49,
    DMI_PROCESSOR_FAMILY_AMD_Z                   = 0x4A,
    DMI_PROCESSOR_FAMILY_AMD_R                   = 0x4B,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_4300        = 0x4C,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_6300        = 0x4D,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_3300        = 0x4E,
    DMI_PROCESSOR_FAMILY_AMD_FIREPRO             = 0x4F,
    DMI_PROCESSOR_FAMILY_SPARC                   = 0x50,
    DMI_PROCESSOR_FAMILY_SUPERSPARC              = 0x51,
    DMI_PROCESSOR_FAMILY_MICROSPARC_2            = 0x52,
    DMI_PROCESSOR_FAMILY_MICROSPARC_2EP          = 0x53,
    DMI_PROCESSOR_FAMILY_ULTRASPARC              = 0x54,
    DMI_PROCESSOR_FAMILY_ULTRASPARC_2            = 0x55,
    DMI_PROCESSOR_FAMILY_ULTRASPARC_2I           = 0x56,
    DMI_PROCESSOR_FAMILY_ULTRASPARC_3            = 0x57,
    DMI_PROCESSOR_FAMILY_ULTRASPARC_3I           = 0x58,
```

```
    DMI_PROCESSOR_FAMILY_MOTOROLA_68040          = 0x60,
    DMI_PROCESSOR_FAMILY_MOTOROLA_68XXX          = 0x61,
    DMI_PROCESSOR_FAMILY_MOTOROLA_68000          = 0x62,
    DMI_PROCESSOR_FAMILY_MOTOROLA_68010          = 0x63,
    DMI_PROCESSOR_FAMILY_MOTOROLA_68020          = 0x64,
    DMI_PROCESSOR_FAMILY_MOTOROLA_68030          = 0x65,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_X4_4C        = 0x66,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_X1000       = 0x67,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_X2000       = 0x68,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_A           = 0x69,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_X3000       = 0x6A,
    DMI_PROCESSOR_FAMILY_AMD_ZEN                 = 0x6B,
    DMI_PROCESSOR_FAMILY_HOBBIT                  = 0x70,
    DMI_PROCESSOR_FAMILY_CRUSOE_TM5000           = 0x78,
    DMI_PROCESSOR_FAMILY_CRUSOE_TM3000           = 0x79,
    DMI_PROCESSOR_FAMILY_EFFICEON_TM8000         = 0x7A,
    DMI_PROCESSOR_FAMILY_WEITEK                  = 0x80,
    DMI_PROCESSOR_FAMILY_INTEL_ITANIUM           = 0x82,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_64           = 0x83,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON             = 0x84,
    DMI_PROCESSOR_FAMILY_AMD_SEMPRON             = 0x85,
    DMI_PROCESSOR_FAMILY_AMD_TURION_64_M         = 0x86,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_2C          = 0x87,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_64_X2_2C     = 0x88,
    DMI_PROCESSOR_FAMILY_AMD_TURION_64_X2_M      = 0x89,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_4C          = 0x8A,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_G3          = 0x8B,
    DMI_PROCESSOR_FAMILY_AMD_PHENOM_FX_4C        = 0x8C,
    DMI_PROCESSOR_FAMILY_AMD_PHENOM_X4_4C        = 0x8D,
    DMI_PROCESSOR_FAMILY_AMD_PHENOM_X2_2C        = 0x8E,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_X2_2C        = 0x8F,
    DMI_PROCESSOR_FAMILY_PARISC                  = 0x90,
    DMI_PROCESSOR_FAMILY_PARISC_8500             = 0x91,
    DMI_PROCESSOR_FAMILY_PARISC_8000             = 0x92,
    DMI_PROCESSOR_FAMILY_PARISC_7300LC           = 0x93,
    DMI_PROCESSOR_FAMILY_PARISC_7200             = 0x94,
    DMI_PROCESSOR_FAMILY_PARISC_7100LC           = 0x95,
    DMI_PROCESSOR_FAMILY_PARISC_7100             = 0x96,
    DMI_PROCESSOR_FAMILY_V30                     = 0xA0,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_3200_4C      = 0xA1,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_3000_2C      = 0xA2,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5300_4C      = 0xA3,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5100_2C      = 0xA4,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5000_2C      = 0xA5,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_LV_2C        = 0xA6,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_ULV_2C       = 0xA7,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7100_2C      = 0xA8,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5400_4C      = 0xA9,
```

```
    DMI_PROCESSOR_FAMILY_INTEL_XEON_4C           = 0xAA,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5200_2C      = 0xAB,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7200_2C      = 0xAC,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7300_4C      = 0xAD,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7400_4C      = 0xAE,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7400_MC      = 0xAF,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3_XEON    = 0xB0,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3_SS      = 0xB1,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_4         = 0xB2,
    DMI_PROCESSOR_FAMILY_INTEL_XEON              = 0xB3,
    DMI_PROCESSOR_FAMILY_IBM_AS400               = 0xB4,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_MP           = 0xB5,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_XP           = 0xB6,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_MP           = 0xB7,
    DMI_PROCESSOR_FAMILY_INTEL_ITANIUM_2         = 0xB8,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_M         = 0xB9,
    DMI_PROCESSOR_FAMILY_INTEL_CELERON_D         = 0xBA,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_D         = 0xBB,
    DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_EXTREME   = 0xBC,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_SOLO         = 0xBD,
    DMI_PROCESSOR_FAMILY_RESERVED_1              = 0xBE,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_DUO        = 0xBF,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_SOLO       = 0xC0,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_EXTREME    = 0xC1,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_QUAD       = 0xC2,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_EXTREME_M  = 0xC3,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_DUO_M      = 0xC4,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_2_SOLO_M     = 0xC5,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_I7           = 0xC6,
    DMI_PROCESSOR_FAMILY_INTEL_CELERON_2C        = 0xC7,
    DMI_PROCESSOR_FAMILY_IBM_ESA390              = 0xC8,
    DMI_PROCESSOR_FAMILY_POWERPC_G4              = 0xC9,
    DMI_PROCESSOR_FAMILY_POWERPC_G5              = 0xCA,
    DMI_PROCESSOR_FAMILY_IBM_ESA390_G6           = 0xCB,
    DMI_PROCESSOR_FAMILY_IBM_ZARCH               = 0xCC,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_I5           = 0xCD,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_I3           = 0xCE,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_I9           = 0xCF,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_D            = 0xD0,
    DMI_PROCESSOR_FAMILY_VIA_C7_M                = 0xD2,
    DMI_PROCESSOR_FAMILY_VIA_C7_D                = 0xD3,
    DMI_PROCESSOR_FAMILY_VIA_C7                  = 0xD4,
    DMI_PROCESSOR_FAMILY_VIA_EDEN                = 0xD5,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_MC           = 0xD6,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_3XXX_2C      = 0xD7,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_3XXX_4C      = 0xD8,
    DMI_PROCESSOR_FAMILY_VIA_NANO                = 0xD9,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5XXX_2C      = 0xDA,
```

```
    DMI_PROCESSOR_FAMILY_INTEL_XEON_5XXX_4C      = 0xDB,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_2C      = 0xDD,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_4C      = 0xDE,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_MC      = 0xDF,
    DMI_PROCESSOR_FAMILY_INTEL_XEON_3400_MC      = 0xE0,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_3000        = 0xE4,
    DMI_PROCESSOR_FAMILY_AMD_SEMPRON_2           = 0xE5,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_EMBEDDED_4C = 0xE6,
    DMI_PROCESSOR_FAMILY_AMD_PHENOM_3C           = 0xE7,
    DMI_PROCESSOR_FAMILY_AMD_TURION_ULTRA_M_2C   = 0xE8,
    DMI_PROCESSOR_FAMILY_AMD_TURION_M_2C         = 0xE9,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_2C           = 0xEA,
    DMI_PROCESSOR_FAMILY_AMD_SEMPRON_SI          = 0xEB,
    DMI_PROCESSOR_FAMILY_AMD_PHENOM_2            = 0xEC,
    DMI_PROCESSOR_FAMILY_AMD_ATHLON_2            = 0xED,
    DMI_PROCESSOR_FAMILY_AMD_OPTERON_6C          = 0xEE,
    DMI_PROCESSOR_FAMILY_AMD_SEMPRON_M           = 0xEF,
    DMI_PROCESSOR_FAMILY_INTEL_I860              = 0xFA,
    DMI_PROCESSOR_FAMILY_INTEL_I960              = 0xFB,
    DMI_PROCESSOR_FAMILY_EXTENDED                = 0xFE,
    DMI_PROCESSOR_FAMILY_RESERVED_2              = 0xFF,
    DMI_PROCESSOR_FAMILY_ARM_V7                  = 0x100,
    DMI_PROCESSOR_FAMILY_ARM_V8                  = 0x101,
    DMI_PROCESSOR_FAMILY_ARM_V9                  = 0x102,
    DMI_PROCESSOR_FAMILY_ARM_RESERVED            = 0x103,
    DMI_PROCESSOR_FAMILY_HITACHI_SH_3            = 0x104,
    DMI_PROCESSOR_FAMILY_HITACHI_SH_4            = 0x105,
    DMI_PROCESSOR_FAMILY_ARM                     = 0x118,
    DMI_PROCESSOR_FAMILY_STRONGARM               = 0x119,
    DMI_PROCESSOR_FAMILY_CYRIX_6X86              = 0x12C,
    DMI_PROCESSOR_FAMILY_CYRIX_MEDIAGX           = 0x12D,
    DMI_PROCESSOR_FAMILY_CYRIX_M2                = 0x12E,
    DMI_PROCESSOR_FAMILY_WINCHIP                 = 0x140,
    DMI_PROCESSOR_FAMILY_DSP                     = 0x15E,
    DMI_PROCESSOR_FAMILY_VIDEO                   = 0x1F4,
    DMI_PROCESSOR_FAMILY_RISCV_RV32              = 0x200,
    DMI_PROCESSOR_FAMILY_RISCV_RV64              = 0x201,
    DMI_PROCESSOR_FAMILY_RISCV_RV128             = 0x202,
    DMI_PROCESSOR_FAMILY_LOONGARCH               = 0x258,
    DMI_PROCESSOR_FAMILY_LOONGSON_1              = 0x259,
    DMI_PROCESSOR_FAMILY_LOONGSON_2              = 0x25A,
    DMI_PROCESSOR_FAMILY_LOONGSON_3              = 0x25B,
    DMI_PROCESSOR_FAMILY_LOONGSON_2K             = 0x25C,
    DMI_PROCESSOR_FAMILY_LOONGSON_3A             = 0x25D,
    DMI_PROCESSOR_FAMILY_LOONGSON_3B             = 0x25E,
    DMI_PROCESSOR_FAMILY_LOONGSON_3C             = 0x25F,
    DMI_PROCESSOR_FAMILY_LOONGSON_3D             = 0x260,
    DMI_PROCESSOR_FAMILY_LOONGSON_3E             = 0x261,
```

```
    DMI_PROCESSOR_FAMILY_LOONGSON_2K_2XXX_2C     = 0x262,
    DMI_PROCESSOR_FAMILY_LOONGSON_3A_5XXX_4C     = 0x26C,
    DMI_PROCESSOR_FAMILY_LOONGSON_3A_5XXX_MC     = 0x26D,
    DMI_PROCESSOR_FAMILY_LOONGSON_3B_5XXX_4C     = 0x26E,
    DMI_PROCESSOR_FAMILY_LOONGSON_3B_5XXX_MC     = 0x26F,
    DMI_PROCESSOR_FAMILY_LOONGSON_3C_5XXX_MC     = 0x270,
    DMI_PROCESSOR_FAMILY_LOONGSON_3D_5XXX_MC     = 0x271,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_3            = 0x300,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_5            = 0x301,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_7            = 0x302,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_9            = 0x303,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_3      = 0x304,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_5      = 0x305,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_7      = 0x306,
    DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_9      = 0x307
} dmi_processor_family_t;

const char *dmi_processor_family_name(dmi_processor_family_t value);
```

Description

The `dmi_processor_family_t` enumeration tells the family of a processor, and is the type of the `family` field of `dmi_processor_t`.

Values up to 0xFD are carried by the one-byte family field of the structure. Values of 0x100 and above only fit the two-byte family field added in SMBIOS 2.6, and the one-byte field holds `DMI_PROCESSOR_FAMILY_EXTENDED` instead. Values missing from the enumeration are not assigned by the specification; 0xFFFE and 0xFFFF are reserved.

The `dmi_processor_family_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_PROCESSOR_FAMILY_UNSPEC`
: Unspecified.

`DMI_PROCESSOR_FAMILY_OTHER`
: Other.

`DMI_PROCESSOR_FAMILY_UNKNOWN`
: Unknown.

`DMI_PROCESSOR_FAMILY_INTEL_8086`
: Intel 8086.

`DMI_PROCESSOR_FAMILY_INTEL_80286`
: Intel 80286.

`DMI_PROCESSOR_FAMILY_INTEL_80386`
: Intel 80386.

`DMI_PROCESSOR_FAMILY_INTEL_80486`
: Intel 80486.

`DMI_PROCESSOR_FAMILY_INTEL_8087`
: Intel 8087.

`DMI_PROCESSOR_FAMILY_INTEL_80287`
: Intel 80287.

`DMI_PROCESSOR_FAMILY_INTEL_80387`
: Intel 80387.

`DMI_PROCESSOR_FAMILY_INTEL_80487`
: Intel 80487.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM`
: Intel Pentium.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_PRO`
: Intel Pentium Pro.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_2`
: Intel Pentium II.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_MMX`
: Intel Pentium MMX.

`DMI_PROCESSOR_FAMILY_INTEL_CELERON`
: Intel Celeron.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_2_XEON`
: Intel Pentium II Xeon.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3`
: Intel Pentium III.

`DMI_PROCESSOR_FAMILY_APPLE_M1`
: Apple M1.

`DMI_PROCESSOR_FAMILY_APPLE_M2`
: Apple M2.

`DMI_PROCESSOR_FAMILY_INTEL_CELERON_M`
: Intel Celeron M.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_4_HT`
: Intel Pentium 4 HT.

`DMI_PROCESSOR_FAMILY_INTEL`
: Intel.

`DMI_PROCESSOR_FAMILY_AMD_DURON`
: AMD Duron.

`DMI_PROCESSOR_FAMILY_AMD_K5`
: AMD K5.

`DMI_PROCESSOR_FAMILY_AMD_K6`
: AMD K6.

`DMI_PROCESSOR_FAMILY_AMD_K6_2`
: AMD K6-2.

`DMI_PROCESSOR_FAMILY_AMD_K6_3`
: AMD K6-3.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON`
: AMD Athlon.

`DMI_PROCESSOR_FAMILY_AMD_29000`
: AMD 29000.

`DMI_PROCESSOR_FAMILY_AMD_K6_2_PLUS`
: AMD K6-2+.

`DMI_PROCESSOR_FAMILY_POWERPC`
: PowerPC.

`DMI_PROCESSOR_FAMILY_POWERPC_601`
: PowerPC 601.

`DMI_PROCESSOR_FAMILY_POWERPC_603`
: PowerPC 603.

`DMI_PROCESSOR_FAMILY_POWERPC_603_PLUS`
: PowerPC 603+.

`DMI_PROCESSOR_FAMILY_POWERPC_604`
: PowerPC 604.

`DMI_PROCESSOR_FAMILY_POWERPC_620`
: PowerPC 620.

`DMI_PROCESSOR_FAMILY_POWERPC_X704`
: PowerPC x704.

`DMI_PROCESSOR_FAMILY_POWERPC_750`
: PowerPC 750.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_DUO`
: Intel Core Duo.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_DUO_M`
: Intel Core Duo M.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_SOLO_M`
: Intel Core Solo M.

`DMI_PROCESSOR_FAMILY_INTEL_ATOM`
: Intel Atom.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_M`
: Intel Core M.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_M3`
: Intel Core m3.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_M5`
: Intel Core m5.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_M7`
: Intel Core m7.

`DMI_PROCESSOR_FAMILY_ALPHA`
: Alpha.

`DMI_PROCESSOR_FAMILY_ALPHA_21064`
: Alpha 21064.

`DMI_PROCESSOR_FAMILY_ALPHA_21066`
: Alpha 21066.

`DMI_PROCESSOR_FAMILY_ALPHA_21164`
: Alpha 21164.

`DMI_PROCESSOR_FAMILY_ALPHA_21164PC`
: Alpha 21164PC.

`DMI_PROCESSOR_FAMILY_ALPHA_21164A`
: Alpha 21164a.

`DMI_PROCESSOR_FAMILY_ALPHA_21264`
: Alpha 21264.

`DMI_PROCESSOR_FAMILY_ALPHA_21364`
: Alpha 21364.

`DMI_PROCESSOR_FAMILY_AMD_TURION_2_ULTRA_2C_M`
: AMD Turion II Ultra Dual-Core M.

`DMI_PROCESSOR_FAMILY_AMD_TURION_2_2C_M`
: AMD Turion II Dual-Core M.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_2_2C_M`
: AMD Athlon II Dual-Core M.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_6100`
: AMD Opteron 6100 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_4100`
: AMD Opteron 4100 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_6200`
: AMD Opteron 6200 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_4200`
: AMD Opteron 4200 series.

`DMI_PROCESSOR_FAMILY_AMD_FX`
: AMD FX series.

`DMI_PROCESSOR_FAMILY_MIPS`
: MIPS.

`DMI_PROCESSOR_FAMILY_MIPS_R4000`
: MIPS R4000.

`DMI_PROCESSOR_FAMILY_MIPS_R4200`
: MIPS R4200.

`DMI_PROCESSOR_FAMILY_MIPS_R4400`
: MIPS R4400.

`DMI_PROCESSOR_FAMILY_MIPS_R4600`
: MIPS R4600.

`DMI_PROCESSOR_FAMILY_MIPS_R10000`
: MIPS R10000.

`DMI_PROCESSOR_FAMILY_AMD_C`
: AMD C-series.

`DMI_PROCESSOR_FAMILY_AMD_E`
: AMD E-series.

`DMI_PROCESSOR_FAMILY_AMD_A`
: AMD A-series.

`DMI_PROCESSOR_FAMILY_AMD_G`
: AMD G-series.

`DMI_PROCESSOR_FAMILY_AMD_Z`
: AMD Z-series.

`DMI_PROCESSOR_FAMILY_AMD_R`
: AMD R-series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_4300`
: AMD Opteron 4300 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_6300`
: AMD Opteron 6300 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_3300`
: AMD Opteron 3300 series.

`DMI_PROCESSOR_FAMILY_AMD_FIREPRO`
: AMD FirePro series.

`DMI_PROCESSOR_FAMILY_SPARC`
: SPARC.

`DMI_PROCESSOR_FAMILY_SUPERSPARC`
: SuperSPARC.

`DMI_PROCESSOR_FAMILY_MICROSPARC_2`
: microSPARC II.

`DMI_PROCESSOR_FAMILY_MICROSPARC_2EP`
: microSPARC IIep.

`DMI_PROCESSOR_FAMILY_ULTRASPARC`
: UltraSPARC.

`DMI_PROCESSOR_FAMILY_ULTRASPARC_2`
: UltraSPARC II.

`DMI_PROCESSOR_FAMILY_ULTRASPARC_2I`
: UltraSPARC IIi.

`DMI_PROCESSOR_FAMILY_ULTRASPARC_3`
: UltraSPARC III.

`DMI_PROCESSOR_FAMILY_ULTRASPARC_3I`
: UltraSPARC IIIi.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68040`
: Motorola 68040.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68XXX`
: Motorola 68xxx.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68000`
: Motorola 68000.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68010`
: Motorola 68010.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68020`
: Motorola 68020.

`DMI_PROCESSOR_FAMILY_MOTOROLA_68030`
: Motorola 68030.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_X4_4C`
: AMD Athlon X4 Quad-Core.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_X1000`
: AMD Opteron X1000 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_X2000`
: AMD Opteron X2000 series APU.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_A`
: AMD Opteron A-series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_X3000`
: AMD Opteron X3000 series APU.

`DMI_PROCESSOR_FAMILY_AMD_ZEN`
: AMD Zen.

`DMI_PROCESSOR_FAMILY_HOBBIT`
: Hobbit.

`DMI_PROCESSOR_FAMILY_CRUSOE_TM5000`
: Transmeta Crusoe TM5000.

`DMI_PROCESSOR_FAMILY_CRUSOE_TM3000`
: Transmeta Crusoe TM3000.

`DMI_PROCESSOR_FAMILY_EFFICEON_TM8000`
: Transmeta Efficeon TM8000.

`DMI_PROCESSOR_FAMILY_WEITEK`
: Weitek.

`DMI_PROCESSOR_FAMILY_INTEL_ITANIUM`
: Intel Itanium.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_64`
: AMD Athlon 64.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON`
: AMD Opteron.

`DMI_PROCESSOR_FAMILY_AMD_SEMPRON`
: AMD Sempron.

`DMI_PROCESSOR_FAMILY_AMD_TURION_64_M`
: AMD Turion 64 Mobile.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_2C`
: Dual-Core AMD Opteron.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_64_X2_2C`
: AMD Athlon 64 X2 Dual-Core.

`DMI_PROCESSOR_FAMILY_AMD_TURION_64_X2_M`
: AMD Turion 64 X2 Mobile.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_4C`
: Quad-Core AMD Opteron.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_G3`
: Third-Generation AMD Opteron.

`DMI_PROCESSOR_FAMILY_AMD_PHENOM_FX_4C`
: AMD Phenom FX Quad-Core.

`DMI_PROCESSOR_FAMILY_AMD_PHENOM_X4_4C`
: AMD Phenom X4 Quad-Core.

`DMI_PROCESSOR_FAMILY_AMD_PHENOM_X2_2C`
: AMD Phenom X2 Dual-Core.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_X2_2C`
: AMD Athlon X2 Dual-Core.

`DMI_PROCESSOR_FAMILY_PARISC`
: PA-RISC.

`DMI_PROCESSOR_FAMILY_PARISC_8500`
: PA-RISC 8500.

`DMI_PROCESSOR_FAMILY_PARISC_8000`
: PA-RISC 8000.

`DMI_PROCESSOR_FAMILY_PARISC_7300LC`
: PA-RISC 7300LC.

`DMI_PROCESSOR_FAMILY_PARISC_7200`
: PA-RISC 7200.

`DMI_PROCESSOR_FAMILY_PARISC_7100LC`
: PA-RISC 7100LC.

`DMI_PROCESSOR_FAMILY_PARISC_7100`
: PA-RISC 7100.

`DMI_PROCESSOR_FAMILY_V30`
: V30.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_3200_4C`
: Quad-Core Intel Xeon 3200 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_3000_2C`
: Dual-Core Intel Xeon 3000 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5300_4C`
: Quad-Core Intel Xeon 5300 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5100_2C`
: Dual-Core Intel Xeon 5100 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5000_2C`
: Dual-Core Intel Xeon 5000 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_LV_2C`
: Dual-Core Intel Xeon LV.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_ULV_2C`
: Dual-Core Intel Xeon ULV.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7100_2C`
: Dual-Core Intel Xeon 7100 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5400_4C`
: Quad-Core Intel Xeon 5400 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_4C`
: Quad-Core Intel Xeon.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5200_2C`
: Dual-Core Intel Xeon 5200 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7200_2C`
: Dual-Core Intel Xeon 7200 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7300_4C`
: Quad-Core Intel Xeon 7300 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7400_4C`
: Quad-Core Intel Xeon 7400 series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7400_MC`
: Multi-Core Intel Xeon 7400 series.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3_XEON`
: Intel Pentium III Xeon.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_3_SS`
: Intel Pentium III with SpeedStep technology.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_4`
: Intel Pentium 4.

`DMI_PROCESSOR_FAMILY_INTEL_XEON`
: Intel Xeon.

`DMI_PROCESSOR_FAMILY_IBM_AS400`
: IBM AS/400.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_MP`
: Intel Xeon processor MP.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_XP`
: AMD Athlon XP.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_MP`
: AMD Athlon MP.

`DMI_PROCESSOR_FAMILY_INTEL_ITANIUM_2`
: Intel Itanium 2.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_M`
: Intel Pentium M.

`DMI_PROCESSOR_FAMILY_INTEL_CELERON_D`
: Intel Celeron D.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_D`
: Intel Pentium D.

`DMI_PROCESSOR_FAMILY_INTEL_PENTIUM_EXTREME`
: Intel Pentium Extreme Edition.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_SOLO`
: Intel Core Solo.

`DMI_PROCESSOR_FAMILY_RESERVED_1`
: Reserved. Some firmware uses it for Intel Core 2 processors, and some software for AMD K7 processors.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_DUO`
: Intel Core 2 Duo.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_SOLO`
: Intel Core 2 Solo.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_EXTREME`
: Intel Core 2 Extreme.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_QUAD`
: Intel Core 2 Quad.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_EXTREME_M`
: Intel Core 2 Extreme mobile.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_DUO_M`
: Intel Core 2 Duo mobile.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_2_SOLO_M`
: Intel Core 2 Solo mobile.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_I7`
: Intel Core i7.

`DMI_PROCESSOR_FAMILY_INTEL_CELERON_2C`
: Dual-Core Intel Celeron.

`DMI_PROCESSOR_FAMILY_IBM_ESA390`
: IBM ESA/390.

`DMI_PROCESSOR_FAMILY_POWERPC_G4`
: PowerPC G4.

`DMI_PROCESSOR_FAMILY_POWERPC_G5`
: PowerPC G5.

`DMI_PROCESSOR_FAMILY_IBM_ESA390_G6`
: IBM ESA/390 G6.

`DMI_PROCESSOR_FAMILY_IBM_ZARCH`
: z/Architecture.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_I5`
: Intel Core i5.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_I3`
: Intel Core i3.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_I9`
: Intel Core i9.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_D`
: Intel Xeon D.

`DMI_PROCESSOR_FAMILY_VIA_C7_M`
: VIA C7-M.

`DMI_PROCESSOR_FAMILY_VIA_C7_D`
: VIA C7-D.

`DMI_PROCESSOR_FAMILY_VIA_C7`
: VIA C7.

`DMI_PROCESSOR_FAMILY_VIA_EDEN`
: VIA Eden.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_MC`
: Multi-Core Intel Xeon processor.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_3XXX_2C`
: Dual-Core Intel Xeon 3xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_3XXX_4C`
: Quad-Core Intel Xeon 3xxx series.

`DMI_PROCESSOR_FAMILY_VIA_NANO`
: VIA Nano.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5XXX_2C`
: Dual-Core Intel Xeon 5xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_5XXX_4C`
: Quad-Core Intel Xeon 5xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_2C`
: Dual-Core Intel Xeon 7xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_4C`
: Quad-Core Intel Xeon 7xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_7XXX_MC`
: Multi-Core Intel Xeon 7xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_XEON_3400_MC`
: Multi-Core Intel Xeon 3400 series.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_3000`
: AMD Opteron 3000 series.

`DMI_PROCESSOR_FAMILY_AMD_SEMPRON_2`
: AMD Sempron II.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_EMBEDDED_4C`
: Embedded AMD Opteron Quad-Core.

`DMI_PROCESSOR_FAMILY_AMD_PHENOM_3C`

AMD Phenom Triple-Core.

`DMI_PROCESSOR_FAMILY_AMD_TURION_ULTRA_M_2C`

AMD Turion Ultra Dual-Core mobile.

`DMI_PROCESSOR_FAMILY_AMD_TURION_M_2C`

AMD Turion Dual-Core mobile.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_2C`

AMD Athlon Dual-Core.

`DMI_PROCESSOR_FAMILY_AMD_SEMPRON_SI`

AMD Sempron SI.

`DMI_PROCESSOR_FAMILY_AMD_PHENOM_2`

AMD Phenom II.

`DMI_PROCESSOR_FAMILY_AMD_ATHLON_2`

AMD Athlon II.

`DMI_PROCESSOR_FAMILY_AMD_OPTERON_6C`

Six-Core AMD Opteron.

`DMI_PROCESSOR_FAMILY_AMD_SEMPRON_M`

AMD Sempron M.

`DMI_PROCESSOR_FAMILY_INTEL_I860`

Intel i860.

`DMI_PROCESSOR_FAMILY_INTEL_I960`

Intel i960.

`DMI_PROCESSOR_FAMILY_EXTENDED`

The family is carried by the two-byte family field. The `family` field of `dmi_processor_t` only holds this value if the structure is too short to carry the two-byte field.

`DMI_PROCESSOR_FAMILY_RESERVED_2`

Reserved: the value of erased flash memory.

`DMI_PROCESSOR_FAMILY_ARM_V7`

ARMv7.

`DMI_PROCESSOR_FAMILY_ARM_V8`

ARMv8.

`DMI_PROCESSOR_FAMILY_ARM_V9`

ARMv9.

`DMI_PROCESSOR_FAMILY_ARM_RESERVED`

Reserved for future use by ARM.

`DMI_PROCESSOR_FAMILY_HITACHI_SH_3`
: Hitachi SH-3.

`DMI_PROCESSOR_FAMILY_HITACHI_SH_4`
: Hitachi SH-4.

`DMI_PROCESSOR_FAMILY_ARM`
: ARM.

`DMI_PROCESSOR_FAMILY_STRONGARM`
: StrongARM.

`DMI_PROCESSOR_FAMILY_CYRIX_6X86`
: Cyrix 6x86.

`DMI_PROCESSOR_FAMILY_CYRIX_MEDIAGX`
: Cyrix MediaGX.

`DMI_PROCESSOR_FAMILY_CYRIX_M2`
: Cyrix MII.

`DMI_PROCESSOR_FAMILY_WINCHIP`
: WinChip.

`DMI_PROCESSOR_FAMILY_DSP`
: DSP.

`DMI_PROCESSOR_FAMILY_VIDEO`
: Video.

`DMI_PROCESSOR_FAMILY_RISCV_RV32`
: RISC-V RV32.

`DMI_PROCESSOR_FAMILY_RISCV_RV64`
: RISC-V RV64.

`DMI_PROCESSOR_FAMILY_RISCV_RV128`
: RISC-V RV128.

`DMI_PROCESSOR_FAMILY_LOONGARCH`
: LoongArch.

`DMI_PROCESSOR_FAMILY_LOONGSON_1`
: Loongson 1.

`DMI_PROCESSOR_FAMILY_LOONGSON_2`
: Loongson 2.

`DMI_PROCESSOR_FAMILY_LOONGSON_3`
: Loongson 3.

`DMI_PROCESSOR_FAMILY_LOONGSON_2K`
: Loongson 2K.

`DMI_PROCESSOR_FAMILY_LOONGSON_3A`
: Loongson 3A.

`DMI_PROCESSOR_FAMILY_LOONGSON_3B`
: Loongson 3B.

`DMI_PROCESSOR_FAMILY_LOONGSON_3C`
: Loongson 3C.

`DMI_PROCESSOR_FAMILY_LOONGSON_3D`
: Loongson 3D.

`DMI_PROCESSOR_FAMILY_LOONGSON_3E`
: Loongson 3E.

`DMI_PROCESSOR_FAMILY_LOONGSON_2K_2XXX_2C`
: Dual-Core Loongson 2K 2xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3A_5XXX_4C`
: Quad-Core Loongson 3A 5xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3A_5XXX_MC`
: Multi-Core Loongson 3A 5xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3B_5XXX_4C`
: Quad-Core Loongson 3B 5xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3B_5XXX_MC`
: Multi-Core Loongson 3B 5xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3C_5XXX_MC`
: Multi-Core Loongson 3C 5xxx series.

`DMI_PROCESSOR_FAMILY_LOONGSON_3D_5XXX_MC`
: Multi-Core Loongson 3D 5xxx series.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_3`
: Intel Core 3.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_5`
: Intel Core 5.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_7`
: Intel Core 7.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_9`
: Intel Core 9.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_3`
: Intel Core Ultra 3.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_5`
: Intel Core Ultra 5.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_7`
: Intel Core Ultra 7.

`DMI_PROCESSOR_FAMILY_INTEL_CORE_ULTRA_9`
: Intel Core Ultra 9.

Return value

`dmi_processor_family_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_processor`, `dmi_processor_id_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_features

Synopsis

```
#include <opendmi/entity/processor.h>

typedef union dmi_processor_features {
    uint16_t __value;
    struct {
        uint16_t __reserved              : 1;
        uint16_t is_unknown              : 1;
        uint16_t capable_64bit           : 1;
        uint16_t multicore               : 1;
        uint16_t hardware_thread         : 1;
        uint16_t execute_protection      : 1;
        uint16_t enhanced_virtualization : 1;
        uint16_t power_perf_control      : 1;
        uint16_t capable_128bit          : 1;
```

```
        uint16_t arm64_soc_id           : 1;
        uint16_t __reserved2            : 6;
    };
} dmi_processor_features_t;
```

Description

The `dmi_processor_features_t` union holds the characteristics of a processor, as the processor information structure (type 4) of SMBIOS gives them since SMBIOS 2.5. It is the type of the `features` field of `dmi_processor_t`.

The bits tell what the processor is capable of, not whether the capabilities are currently enabled, nor how many cores or threads are in use.

Fields

`__value`
All the bits as one word, as stored.

`__reserved`
Reserved bit 0.

`is_unknown`
The characteristics are unknown.

`capable_64bit`
The processor can process 64-bit data, whether this mode is enabled or not.

`multicore`
The processor has more than one core.

`hardware_thread`
The processor runs more than one hardware thread per core.

`execute_protection`
The processor can mark memory regions as not executable, for example with the NX or XD feature.

`enhanced_virtualization`
The processor has enhanced virtualization instructions.

`power_perf_control`
The processor can save power depending on the load.

`capable_128bit`
The processor can process 128-bit data.

`arm64_soc_id`
The processor returns a SoC identifier through the SMCCC_ARCH_SOC_ID call, and the processor identifier holds it; see `dmi_processor_soc_id`.

`__reserved2`
: Reserved bits 10 to 15.

See also

`dmi_processor`, `dmi_processor_soc_id`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.5 or newer.

dmi_processor_id_format

Synopsis

```
#include <opendmi/entity/processor.h>

typedef enum dmi_processor_id_format {
    DMI_PROCESSOR_ID_FORMAT_RAW    = 0,
    DMI_PROCESSOR_ID_FORMAT_X86    = 1,
    DMI_PROCESSOR_ID_FORMAT_MIDR   = 2,
    DMI_PROCESSOR_ID_FORMAT_SOC_ID = 3
} dmi_processor_id_format_t;
```

Description

The `dmi_processor_id_format_t` enumeration tells how the processor identifier of a processor information structure has been decoded, and is the type of the `id_format` field of `dmi_processor_t`. It selects which of the `x86_id`, `arm_id` and `soc_id` fields of the structure holds the decoded identifier.

The format is chosen from the processor family, and, for the families which do not tell the architecture (other, unknown, and the Apple M1 and M2 families, which older specifications gave to Cyrix x86 processors), from the vendor and version strings as well.

Values

`DMI_PROCESSOR_ID_FORMAT_RAW`
: Architecture not recognized: the identifier is only available as stored, in the `id` field.

`DMI_PROCESSOR_ID_FORMAT_X86`
: x86 processor: signature and feature flags returned by CPUID leaf 1, decoded into the `x86_id` field.

`DMI_PROCESSOR_ID_FORMAT_MIDR`
: Arm processor: Main ID Register (MIDR or MIDR_EL1), decoded into the `arm_id` field.

`DMI_PROCESSOR_ID_FORMAT_SOC_ID`
: Arm64 processor supporting the SMCCC_ARCH_SOC_ID call: SoC identifier and revision, decoded into the `soc_id` field.

See also

`dmi_processor`, `dmi_processor_x86_id`, `dmi_processor_arm_id`, `dmi_processor_soc_id`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_soc_id

Synopsis

```
#include <opendmi/entity/processor.h>

typedef struct dmi_processor_soc_id {
    uint8_t  jep106_bank;
    uint8_t  jep106_id;
    uint16_t soc_id;
    uint32_t soc_revision;
} dmi_processor_soc_id_t;
```

Description

The `dmi_processor_soc_id_t` structure holds the processor identifier of an Arm64 processor which reports a SoC identifier, decoded from the processor information structure (type 4) of SMBIOS. The values are the ones the SMCCC_ARCH_SOC_ID call of the Arm SMC Calling Convention returns, and identify the system in package (SiP) and its vendor.

It is the `soc_id` field of `dmi_processor_t`, set when the `id_format` field of it is `DMI_PROCESSOR_ID_FORMAT_SOC_ID`, that is, when the processor family is an Arm64 one and the `arm64_soc_id` characteristic is set.

Fields

`jep106_bank`
JEDEC JEP106 bank index (continuation code) of the SiP vendor, 7 bits.

`jep106_id`
JEDEC JEP106 identification code of the SiP vendor, 7 bits.

`soc_id`
SoC identifier, defined by the SiP vendor.

`soc_revision`
SoC revision, defined by the SiP vendor, 31 bits.

See also

`dmi_processor`, `dmi_processor_id_format`, `dmi_processor_arm_id`, `dmi_processor_features`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.4 or newer.

dmi_processor_status

Synopsis

```
#include <opendmi/entity/processor.h>

typedef enum dmi_processor_status {
    DMI_PROCESSOR_STATUS_UNKNOWN          = 0x00,
    DMI_PROCESSOR_STATUS_ENABLED          = 0x01,
    DMI_PROCESSOR_STATUS_DISABLED_BY_USER = 0x02,
    DMI_PROCESSOR_STATUS_DISABLED_BY_FW   = 0x03,
    DMI_PROCESSOR_STATUS_IDLE             = 0x04,
    DMI_PROCESSOR_STATUS_OTHER            = 0x07
} dmi_processor_status_t;

const char *dmi_processor_status_name(dmi_processor_status_t value);
```

Description

The `dmi_processor_status_t` enumeration tells the state of a processor, and is the type of the `status` field of `dmi_processor_t`. The values 0x05 and 0x06 are reserved.

The `dmi_processor_status_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_PROCESSOR_STATUS_UNKNOWN`
: Unknown.

`DMI_PROCESSOR_STATUS_ENABLED`
: Processor is enabled.

`DMI_PROCESSOR_STATUS_DISABLED_BY_USER`
: Processor is disabled by the user, in the firmware setup.

`DMI_PROCESSOR_STATUS_DISABLED_BY_FW`
: Processor is disabled by the firmware, after a POST error.

`DMI_PROCESSOR_STATUS_IDLE`
: Processor is idle, waiting to be enabled.

`DMI_PROCESSOR_STATUS_OTHER`
: Other.

Return value

`dmi_processor_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_processor`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_type

Synopsis

```
#include <opendmi/entity/processor.h>

typedef enum dmi_processor_type {
    DMI_PROCESSOR_TYPE_UNSPEC  = 0x00,
    DMI_PROCESSOR_TYPE_OTHER   = 0x01,
    DMI_PROCESSOR_TYPE_UNKNOWN = 0x02,
    DMI_PROCESSOR_TYPE_CENTRAL = 0x03,
    DMI_PROCESSOR_TYPE_MATH    = 0x04,
    DMI_PROCESSOR_TYPE_DSP     = 0x05,
    DMI_PROCESSOR_TYPE_VIDEO   = 0x06
} dmi_processor_type_t;

const char *dmi_processor_type_name(dmi_processor_type_t value);
```

Description

The `dmi_processor_type_t` enumeration tells the role of a processor, and is the type of the `type` field of `dmi_processor_t`.

The `dmi_processor_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_PROCESSOR_TYPE_UNSPEC`
: Unspecified.

`DMI_PROCESSOR_TYPE_OTHER`
: Other.

`DMI_PROCESSOR_TYPE_UNKNOWN`
: Unknown.

`DMI_PROCESSOR_TYPE_CENTRAL`
: Central processor.

`DMI_PROCESSOR_TYPE_MATH`
: Math processor.

`DMI_PROCESSOR_TYPE_DSP`
: DSP processor.

`DMI_PROCESSOR_TYPE_VIDEO`
: Video processor.

Return value

`dmi_processor_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_processor`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_upgrade

Synopsis

```
#include <opendmi/entity/processor.h>

typedef enum dmi_processor_upgrade {
    DMI_PROCESSOR_UPGRADE_UNSPEC           = 0x00,
    DMI_PROCESSOR_UPGRADE_OTHER            = 0x01,
    DMI_PROCESSOR_UPGRADE_UNKNOWN          = 0x02,
    DMI_PROCESSOR_UPGRADE_DAUGHTER_BOARD   = 0x03,
    DMI_PROCESSOR_UPGRADE_SOCKET_ZIF       = 0x04,
```

```
    DMI_PROCESSOR_UPGRADE_PIGGY_BACK       = 0x05,
    DMI_PROCESSOR_UPGRADE_NONE             = 0x06,
    DMI_PROCESSOR_UPGRADE_SOCKET_LIF       = 0x07,
    DMI_PROCESSOR_UPGRADE_SLOT_1           = 0x08,
    DMI_PROCESSOR_UPGRADE_SLOT_2           = 0x09,
    DMI_PROCESSOR_UPGRADE_SOCKET_370PIN    = 0x0A,
    DMI_PROCESSOR_UPGRADE_SLOT_A           = 0x0B,
    DMI_PROCESSOR_UPGRADE_SLOT_M           = 0x0C,
    DMI_PROCESSOR_UPGRADE_SOCKET_423       = 0x0D,
    DMI_PROCESSOR_UPGRADE_SOCKET_A         = 0x0E,
    DMI_PROCESSOR_UPGRADE_SOCKET_478       = 0x0F,
    DMI_PROCESSOR_UPGRADE_SOCKET_754       = 0x10,
    DMI_PROCESSOR_UPGRADE_SOCKET_940       = 0x11,
    DMI_PROCESSOR_UPGRADE_SOCKET_939       = 0x12,
    DMI_PROCESSOR_UPGRADE_SOCKET_MPGA604   = 0x13,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA771    = 0x14,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA775    = 0x15,
    DMI_PROCESSOR_UPGRADE_SOCKET_S1        = 0x16,
    DMI_PROCESSOR_UPGRADE_SOCKET_AM2       = 0x17,
    DMI_PROCESSOR_UPGRADE_SOCKET_F         = 0x18,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1366   = 0x19,
    DMI_PROCESSOR_UPGRADE_SOCKET_G34       = 0x1A,
    DMI_PROCESSOR_UPGRADE_SOCKET_AM3       = 0x1B,
    DMI_PROCESSOR_UPGRADE_SOCKET_C32       = 0x1C,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1156   = 0x1D,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1567   = 0x1E,
    DMI_PROCESSOR_UPGRADE_SOCKET_PGA988A   = 0x1F,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1288   = 0x20,
    DMI_PROCESSOR_UPGRADE_SOCKET_RPGA988B  = 0x21,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1023   = 0x22,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1224   = 0x23,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1155   = 0x24,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1356   = 0x25,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA2011   = 0x26,
    DMI_PROCESSOR_UPGRADE_SOCKET_FS1       = 0x27,
    DMI_PROCESSOR_UPGRADE_SOCKET_FS2       = 0x28,
    DMI_PROCESSOR_UPGRADE_SOCKET_FM1       = 0x29,
    DMI_PROCESSOR_UPGRADE_SOCKET_FM2       = 0x2A,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA2011_3 = 0x2B,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1356_3 = 0x2C,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1150   = 0x2D,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1168   = 0x2E,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1234   = 0x2F,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1364   = 0x30,
    DMI_PROCESSOR_UPGRADE_SOCKET_AM4       = 0x31,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1151   = 0x32,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1356   = 0x33,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1440   = 0x34,
```

```
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1515   = 0x35,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA3647_1 = 0x36,
    DMI_PROCESSOR_UPGRADE_SOCKET_SP3       = 0x37,
    DMI_PROCESSOR_UPGRADE_SOCKET_SP3R2     = 0x38,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA2066   = 0x39,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1392   = 0x3A,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1510   = 0x3B,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1528   = 0x3C,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA4189   = 0x3D,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1200   = 0x3E,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA4677   = 0x3F,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1700   = 0x40,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1744   = 0x41,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1781   = 0x42,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1211   = 0x43,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA2422   = 0x44,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1211   = 0x45,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA2422   = 0x46,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA5773   = 0x47,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA5773   = 0x48,
    DMI_PROCESSOR_UPGRADE_SOCKET_AM5       = 0x49,
    DMI_PROCESSOR_UPGRADE_SOCKET_SP5       = 0x4A,
    DMI_PROCESSOR_UPGRADE_SOCKET_SP6       = 0x4B,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA883    = 0x4C,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1190   = 0x4D,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA4129   = 0x4E,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA4710   = 0x4F,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA7529   = 0x50,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1964   = 0x51,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA1792   = 0x52,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA2049   = 0x53,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA2551   = 0x54,
    DMI_PROCESSOR_UPGRADE_SOCKET_LGA1851   = 0x55,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA2114   = 0x56,
    DMI_PROCESSOR_UPGRADE_SOCKET_BGA2833   = 0x57,
    DMI_PROCESSOR_UPGRADE_SEE_SOCKET_TYPE  = 0xFF
} dmi_processor_upgrade_t;

const char *dmi_processor_upgrade_name(dmi_processor_upgrade_t value);
```

Description

The `dmi_processor_upgrade_t` enumeration tells how the processor of a socket can be upgraded, which is most often the type of the socket, and is the type of the `upgrade` field of `dmi_processor_t`.

The specification also defines the value 0xFF, which the enumeration does not name, for the sockets no value fits; the `socket_type` field of `dmi_processor_t` names the socket then.

The `dmi_processor_upgrade_name` function returns the name of a value, as the command line tool

shows it.

Values

`DMI_PROCESSOR_UPGRADE_UNSPEC`
: Unspecified.

`DMI_PROCESSOR_UPGRADE_OTHER`
: Other.

`DMI_PROCESSOR_UPGRADE_UNKNOWN`
: Unknown.

`DMI_PROCESSOR_UPGRADE_DAUGHTER_BOARD`
: Daughter board.

`DMI_PROCESSOR_UPGRADE_SOCKET_ZIF`
: ZIF socket.

`DMI_PROCESSOR_UPGRADE_PIGGY_BACK`
: Replaceable piggy-back.

`DMI_PROCESSOR_UPGRADE_NONE`
: None.

`DMI_PROCESSOR_UPGRADE_SOCKET_LIF`
: LIF socket.

`DMI_PROCESSOR_UPGRADE_SLOT_1`
: Slot 1.

`DMI_PROCESSOR_UPGRADE_SLOT_2`
: Slot 2.

`DMI_PROCESSOR_UPGRADE_SOCKET_370PIN`
: 370-pin socket.

`DMI_PROCESSOR_UPGRADE_SLOT_A`
: Slot A.

`DMI_PROCESSOR_UPGRADE_SLOT_M`
: Slot M.

`DMI_PROCESSOR_UPGRADE_SOCKET_423`
: Socket 423.

`DMI_PROCESSOR_UPGRADE_SOCKET_A`
: Socket A (462).

`DMI_PROCESSOR_UPGRADE_SOCKET_478`
: Socket 478.

`DMI_PROCESSOR_UPGRADE_SOCKET_754`
Socket 754.

`DMI_PROCESSOR_UPGRADE_SOCKET_940`
Socket 940.

`DMI_PROCESSOR_UPGRADE_SOCKET_939`
Socket 939.

`DMI_PROCESSOR_UPGRADE_SOCKET_MPGA604`
Socket mPGA604.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA771`
Socket LGA771.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA775`
Socket LGA775.

`DMI_PROCESSOR_UPGRADE_SOCKET_S1`
Socket S1.

`DMI_PROCESSOR_UPGRADE_SOCKET_AM2`
Socket AM2.

`DMI_PROCESSOR_UPGRADE_SOCKET_F`
Socket F (1207).

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1366`
Socket LGA1366.

`DMI_PROCESSOR_UPGRADE_SOCKET_G34`
Socket G34.

`DMI_PROCESSOR_UPGRADE_SOCKET_AM3`
Socket AM3.

`DMI_PROCESSOR_UPGRADE_SOCKET_C32`
Socket C32.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1156`
Socket LGA1156.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1567`
Socket LGA1567.

`DMI_PROCESSOR_UPGRADE_SOCKET_PGA988A`
Socket PGA988A.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1288`
Socket BGA1288.

`DMI_PROCESSOR_UPGRADE_SOCKET_RPGA988B`
: Socket rPGA988B.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1023`
: Socket BGA1023.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1224`
: Socket BGA1224.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1155`
: Socket LGA1155.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1356`
: Socket LGA1356.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA2011`
: Socket LGA2011.

`DMI_PROCESSOR_UPGRADE_SOCKET_FS1`
: Socket FS1.

`DMI_PROCESSOR_UPGRADE_SOCKET_FS2`
: Socket FS2.

`DMI_PROCESSOR_UPGRADE_SOCKET_FM1`
: Socket FM1.

`DMI_PROCESSOR_UPGRADE_SOCKET_FM2`
: Socket FM2.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA2011_3`
: Socket LGA2011-3.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1356_3`
: Socket LGA1356-3.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1150`
: Socket LGA1150.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1168`
: Socket BGA1168.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1234`
: Socket BGA1234.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1364`
: Socket BGA1364.

`DMI_PROCESSOR_UPGRADE_SOCKET_AM4`
: Socket AM4.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1151`
: Socket LGA1151.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1356`
: Socket BGA1356.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1440`
: Socket BGA1440.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1515`
: Socket BGA1515.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA3647_1`
: Socket LGA3647-1.

`DMI_PROCESSOR_UPGRADE_SOCKET_SP3`
: Socket SP3.

`DMI_PROCESSOR_UPGRADE_SOCKET_SP3R2`
: Socket SP3r2.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA2066`
: Socket LGA2066.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1392`
: Socket BGA1392.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1510`
: Socket BGA1510.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1528`
: Socket BGA1528.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA4189`
: Socket LGA4189.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1200`
: Socket LGA1200.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA4677`
: Socket LGA4677.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1700`
: Socket LGA1700.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1744`
: Socket BGA1744.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1781`
: Socket BGA1781.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1211`
: Socket BGA1211.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA2422`
: Socket BGA2422.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1211`
: Socket LGA1211.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA2422`
: Socket LGA2422.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA5773`
: Socket LGA5773.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA5773`
: Socket BGA5773.

`DMI_PROCESSOR_UPGRADE_SOCKET_AM5`
: Socket AM5.

`DMI_PROCESSOR_UPGRADE_SOCKET_SP5`
: Socket SP5.

`DMI_PROCESSOR_UPGRADE_SOCKET_SP6`
: Socket SP6.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA883`
: Socket BGA883.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1190`
: Socket BGA1190.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA4129`
: Socket BGA4129.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA4710`
: Socket LGA4710.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA7529`
: Socket LGA7529.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1964`
: Socket BGA1964.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA1792`
: Socket BGA1792.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA2049`
: Socket BGA2049.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA2551`
: Socket BGA2551.

`DMI_PROCESSOR_UPGRADE_SOCKET_LGA1851`
: Socket LGA1851.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA2114`
: Socket BGA2114.

`DMI_PROCESSOR_UPGRADE_SOCKET_BGA2833`
: Socket BGA2833.

`DMI_PROCESSOR_UPGRADE_SEE_SOCKET_TYPE`
: No other value applies, and the `socket_type` string of the processor names the socket. Defined since SMBIOS 3.8.

Return value

`dmi_processor_upgrade_name` returns the name of the value, or `nullptr` if the value has no name. The name is translated to the current locale when the library is built with localization support and has a translation for it, and is in English otherwise. The string is owned by the library and must not be modified or freed.

Errors

None.

See also

`dmi_processor`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_voltage_data

Synopsis

```
#include <opendmi/entity/processor.h>

typedef union dmi_processor_voltage_data {
    dmi_byte_t __value;
    struct {
        dmi_byte_t value      : 7;
        dmi_byte_t is_current : 1;
    };
} dmi_processor_voltage_data_t;
```

Description

The `dmi_processor_voltage_data_t` union describes the voltage field of the processor information structure (type 4) of SMBIOS as it is stored: one byte holding either the current voltage of the processor or the voltages its socket accepts.

The decoder splits the field into the `voltage` and `supported_voltages` fields of `dmi_processor_t`, so the union is only needed to read or build the stored byte. The specification deprecates the field since SMBIOS 3.8.

Fields

`__value`
: Stored byte.

`value`
: Current voltage, in tenths of volt, if `is_current` is set, and the set of the supported voltages in the layout of `dmi_processor_voltages` otherwise.

`is_current`
: Set if `value` holds the current voltage, clear in the legacy mode, where it holds the supported voltages.

See also

`dmi_processor`, `dmi_processor_voltages`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_voltages

Synopsis

```
#include <opendmi/entity/processor.h>

typedef union dmi_processor_voltages {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_5v      : 1;
        dmi_byte_t is_3v3     : 1;
        dmi_byte_t is_2v9     : 1;
        dmi_byte_t __reserved : 5;
    };
} dmi_processor_voltages_t;
```

Description

The `dmi_processor_voltages_t` union holds the set of the voltages a processor socket accepts, as the processor information structure (type 4) of SMBIOS gives it in the legacy mode of its voltage field. It is the type of the `supported_voltages` field of `dmi_processor_t`, which is 0 when the structure gives the current voltage instead.

Fields

`__value`
: All the bits as one byte.

`is_5v`
: 5 V is supported.

`is_3v3`
: 3.3 V is supported.

`is_2v9`
: 2.9 V is supported.

`__reserved`
: Reserved, always 0.

See also

`dmi_processor`, `dmi_processor_voltage_data`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_processor_x86_id

Synopsis

```
#include <opendmi/entity/processor.h>

typedef struct dmi_processor_x86_id {
    uint8_t  type;
    uint16_t family;
    uint16_t model;
    uint8_t  stepping;
    uint32_t features;
} dmi_processor_x86_id_t;
```

Description

The `dmi_processor_x86_id_t` structure holds the processor identifier of an x86 processor, decoded from the processor information structure (type 4) of SMBIOS: the processor signature and the feature flags which the CPUID instruction returns for leaf 1 in the EAX and EDX registers.

It is the `x86_id` field of `dmi_processor_t`, set when the `id_format` field of it is `DMI_PROCESSOR_ID_FORMAT_X86`. Some firmware stores the two words of the identifier in the reverse order; the decoder recognizes them by the reserved bits of the signature and puts them back in order.

Fields

`type`
: Processor type, bits 12 and 13 of the signature: 0 for an original OEM processor, 1 for an OverDrive processor, 2 for the second processor of a dual-processor system.

`family`
: Processor family, bits 8 to 11 of the signature. For the base family 0xF, the extended family (bits 20 to 27) is added to it.

`model`
: Processor model, bits 4 to 7 of the signature. For the base families 0x6 and 0xF, the extended model (bits 16 to 19) gives its upper four bits.

`stepping`
: Stepping, bits 0 to 3 of the signature.

`features`
: Feature flags, the value of the EDX register.

See also

`dmi_processor`, `dmi_processor_id_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/slot.h>

dmi_slot

Synopsis

```
#include <opendmi/entity/slot.h>

typedef struct dmi_slot {
    const char *designator;
    dmi_slot_type_t type;
    dmi_slot_width_t bus_width;
    dmi_slot_usage_t usage;
    dmi_slot_length_t length;
    uint16_t ident;
    dmi_slot_features_t features;
    dmi_slot_features_ex_t features_ex;
    dmi_pci_addr_t base_address;
    dmi_slot_width_t base_bus_width;
    size_t peer_group_count;
    dmi_slot_peer_group_t *peer_groups;
    uint8_t information;
    dmi_slot_width_t physical_width;
    uint16_t pitch;
    dmi_slot_height_t height;
} dmi_slot_t;
```

Description

The `dmi_slot_t` structure holds the decoded system slots structure (type 9) of SMBIOS: the kind, size and state of one expansion slot of the system, and the PCI addresses of the devices plugged into it. A table carries one such structure for each slot.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system_slots)`, and lives as long as the context it has been decoded in.

Fields

`designator`
: Reference designator of the slot, such as "PCIE2"; `nullptr` if the structure does not carry it.

`type`
: Bus or form factor of the slot, see `dmi_slot_type`.

`bus_width`
: Electrical width of the data bus of the slot, see `dmi_slot_width`.

`usage`

Whether the slot is in use or free, see `dmi_slot_usage`.

`length`

Length of the slot, see `dmi_slot_length`.

`ident`

Slot identifier, as stored; its meaning depends on `type`, and the library does not interpret it. For MCA and EISA slots, the low byte is the logical slot number, 1 to 15. For PCI, AGP, PCI-X, PCI Express, M.2, U.2, OCP and EDSFF slots, it is the number ACPI (`_SUN`) or the PCI interrupt routing table gives the slot. For PC Card slots, the low byte is the adapter number and the high byte the socket number.

`features`

Characteristics of the slot, see `dmi_slot_features`.

`features_ex`

Extended characteristics of the slot, see `dmi_slot_features_ex`. Present since SMBIOS 2.1; zero in older data.

`base_address`

PCI segment group, bus, device and function of the device in the slot (for PCI Express, the endpoint rather than a switch above it), see `dmi_pci_addr`. Slots without such an address carry `0xFFFF` as segment group and `UINT8_MAX` as bus, device and function numbers. Present since SMBIOS 2.6; all zeros in older data.

`base_bus_width`

Electrical width of the bus of the device at `base_address`, see `dmi_slot_width`. Present since SMBIOS 3.2; `DMI_SLOT_WIDTH_UNSPEC` in older data.

`peer_group_count`

Number of elements of `peer_groups`, counting the peer groups the structure holds in full. Present since SMBIOS 3.2; 0 in older data and for a slot without peer devices.

`peer_groups`

Peer devices of the slot, for a slot whose lanes are split between several devices, see `dmi_slot_peer_group`. The device at `base_address` is not one of them. `nullptr` if the structure declares no peer group.

`information`

Additional information, depending on `type`. For `DMI_SLOT_TYPE_PCI_E_G6`, the PCI Express generation of the slot, e.g. 6 for Gen 6. Other PCI Express slot types may use it the same way, and the other slot types leave it at 0. Present since SMBIOS 3.4; 0 in older data.

`physical_width`

Mechanical width of the slot, while `bus_width` is its electrical width, see `dmi_slot_width`. Present since SMBIOS 3.4; `DMI_SLOT_WIDTH_UNSPEC` in older data.

`pitch`

Distance between the cards of neighbouring slots, in hundredths of a millimeter (1250 stands for 12.5 mm); 0 if unknown. Present since SMBIOS 3.4; 0 in older data.

`height`
: Height of the tallest card the slot takes, see `dmi_slot_height`. Present since SMBIOS 3.5; `DMI_SLOT_HEIGHT_UNSPEC` in older data.

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_registry_iter_t  iter;
dmi_entity_t        *entity;

dmi_registry_iter_initialize(&iter, registry, nullptr);

while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
    if (dmi_entity_type(entity) != DMI_TYPE(system_slots))
        continue;

    dmi_slot_t *slot = dmi_entity_info(entity, DMI_TYPE(system_slots));
    if ((slot != nullptr) && (slot->designator != nullptr))
        printf("%s: %s, %s\n", slot->designator,
               dmi_slot_type_name(slot->type),
               dmi_slot_usage_name(slot->usage));
}
```

See also

`dmi_entity_info`, `dmi_slot_type`, `dmi_slot_width`, `dmi_slot_usage`, `dmi_slot_length`, `dmi_slot_height`, `dmi_slot_features`, `dmi_slot_features_ex`, `dmi_slot_peer_group`, `dmi_pci_addr`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_slot_features

Synopsis

```
#include <opendmi/entity/slot.h>

typedef union dmi_slot_features {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_unknown                   : 1;
        dmi_byte_t has_5v0_support              : 1;
        dmi_byte_t has_3v3_support              : 1;
        dmi_byte_t has_shared_opening           : 1;
        dmi_byte_t has_pccard_16_support        : 1;
```

```
        dmi_byte_t has_cardbus_support           : 1;
        dmi_byte_t has_zoom_video_support        : 1;
        dmi_byte_t has_modem_ring_resume_support : 1;
    };
} dmi_slot_features_t;
```

Description

The `dmi_slot_features_t` union holds the first set of characteristics of a system slot: the voltages it provides, whether its opening is shared, and what its PC Card socket supports. It is the type of the `features` field of `dmi_slot_t`. A bit is set when the slot has the property.

Fields

`__value`
: Raw value of the byte, `is_unknown` being bit 0.

`is_unknown`
: The characteristics of the slot are unknown.

`has_5v0_support`
: The slot provides 5.0 V.

`has_3v3_support`
: The slot provides 3.3 V.

`has_shared_opening`
: The opening of the slot is shared with another slot, as for a combined PCI/EISA slot.

`has_pccard_16_support`
: The PC Card slot takes 16-bit PC Cards.

`has_cardbus_support`
: The PC Card slot supports CardBus.

`has_zoom_video_support`
: The PC Card slot supports Zoom Video.

`has_modem_ring_resume_support`
: The PC Card slot supports Modem Ring Resume.

See also

`dmi_slot`, `dmi_slot_features_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_slot_features_ex

Synopsis

```
#include <opendmi/entity/slot.h>

typedef union dmi_slot_features_ex {
    dmi_byte_t __value;
    struct {
        dmi_byte_t has_pme_support           : 1;
        dmi_byte_t has_hotplug_support       : 1;
        dmi_byte_t has_smbus_support         : 1;
        dmi_byte_t has_bifurcation_support   : 1;
        dmi_byte_t has_async_removal_support : 1;
        dmi_byte_t is_cxl_10_capable         : 1;
        dmi_byte_t is_cxl_20_capable         : 1;
        dmi_byte_t is_cxl_30_capable         : 1;
    };
} dmi_slot_features_ex_t;
```

Description

The `dmi_slot_features_ex_t` union holds the second set of characteristics of a system slot: the signals and removal modes it supports, lane bifurcation and CXL capabilities. It is the type of the `features_ex` field of `dmi_slot_t`, present since SMBIOS 2.1 and zero in older data. A bit is set when the slot has the property.

Fields

`__value`

Raw value of the byte, `has_pme_support` being bit 0.

`has_pme_support`

The PCI slot supports the power management event (PME#) signal.

`has_hotplug_support`

Devices can be plugged into the slot and removed from it while the system runs.

`has_smbus_support`

The PCI slot supports the SMBus signal.

`has_bifurcation_support`

The PCI Express slot can split its lanes between two or more devices plugged into it. The bit does not tell which splits the slot supports.

`has_async_removal_support`

Devices can be removed from the slot without telling the operating system, the driver or the applications first (surprise removal).

`is_cxl_10_capable`
: The slot is a Flex Bus slot capable of CXL 1.0.

`is_cxl_20_capable`
: The slot is a Flex Bus slot capable of CXL 2.0.

`is_cxl_30_capable`
: The slot is a Flex Bus slot capable of CXL 3.0.

See also

`dmi_slot`, `dmi_slot_features`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_slot_height

Synopsis

```
#include <opendmi/entity/slot.h>

typedef enum dmi_slot_height {
    DMI_SLOT_HEIGHT_UNSPEC      = 0x00,
    DMI_SLOT_HEIGHT_OTHER       = 0x01,
    DMI_SLOT_HEIGHT_UNKNOWN     = 0x02,
    DMI_SLOT_HEIGHT_FULL        = 0x03,
    DMI_SLOT_HEIGHT_LOW_PROFILE = 0x04,
} dmi_slot_height_t;

const char *dmi_slot_height_name(dmi_slot_height_t value);
```

Description

The `dmi_slot_height_t` enumeration tells the height of the tallest card a system slot takes, and is the type of the `height` field of `dmi_slot_t`, present since SMBIOS 3.5.

The `dmi_slot_height_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SLOT_HEIGHT_UNSPEC`
: Not applicable, or not given.

`DMI_SLOT_HEIGHT_OTHER`
: Other.

`DMI_SLOT_HEIGHT_UNKNOWN`
: Unknown.

`DMI_SLOT_HEIGHT_FULL`
: Full-height cards.

`DMI_SLOT_HEIGHT_LOW_PROFILE`
: Low-profile cards.

Return value

`dmi_slot_height_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_slot`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

dmi_slot_length

Synopsis

```
#include <opendmi/entity/slot.h>

typedef enum dmi_slot_length {
    DMI_SLOT_LENGTH_UNSPEC   = 0x00,
    DMI_SLOT_LENGTH_OTHER    = 0x01,
    DMI_SLOT_LENGTH_UNKNOWN  = 0x02,
    DMI_SLOT_LENGTH_SHORT    = 0x03,
    DMI_SLOT_LENGTH_LONG     = 0x04,
    DMI_SLOT_LENGTH_SFF_8200 = 0x05,
    DMI_SLOT_LENGTH_SFF_8300 = 0x06,
} dmi_slot_length_t;

const char *dmi_slot_length_name(dmi_slot_length_t value);
```

Description

The `dmi_slot_length_t` enumeration tells the length of a system slot, and is the type of the `length` field of `dmi_slot_t`.

The `dmi_slot_length_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SLOT_LENGTH_UNSPEC`
: Unspecified.

`DMI_SLOT_LENGTH_OTHER`
: Other.

`DMI_SLOT_LENGTH_UNKNOWN`
: Unknown.

`DMI_SLOT_LENGTH_SHORT`
: Short; also used for EDSFF E1.S and E3.S slots.

`DMI_SLOT_LENGTH_LONG`
: Long; also used for EDSFF E1.L and E3.L slots.

`DMI_SLOT_LENGTH_SFF_8200`
: Form factor of a 2.5-inch drive.

`DMI_SLOT_LENGTH_SFF_8300`
: Form factor of a 3.5-inch drive.

Return value

`dmi_slot_length_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_slot`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_slot_peer_group

Synopsis

```
#include <opendmi/entity/slot.h>

typedef struct dmi_slot_peer_group {
    dmi_pci_addr_t address;
    dmi_slot_width_t bus_width;
} dmi_slot_peer_group_t;
```

Description

The `dmi_slot_peer_group_t` structure describes a peer device of a system slot. A slot whose lanes can be split into narrower links holds a base device, given by the `base_address` and `base_bus_width` fields of `dmi_slot_t`, and a peer group for each other device. The peer groups of a slot are the elements of the `peer_groups` array of `dmi_slot_t`. Devices behind a bridge in the slot are not listed.

Fields

`address`

PCI segment group, bus, device and function of the peer device, see `dmi_pci_addr`.

`bus_width`

Electrical width of the bus of the peer device, see `dmi_slot_width`.

See also

`dmi_slot`, `dmi_slot_width`, `dmi_pci_addr`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.2 or newer.

dmi_slot_type

Synopsis

```
#include <opendmi/entity/slot.h>

typedef enum dmi_slot_type {
    DMI_SLOT_TYPE_UNSPEC                = 0x00,
    DMI_SLOT_TYPE_OTHER                 = 0x01,
    DMI_SLOT_TYPE_UNKNOWN               = 0x02,
    DMI_SLOT_TYPE_ISA                   = 0x03,
    DMI_SLOT_TYPE_MCA                   = 0x04,
    DMI_SLOT_TYPE_EISA                  = 0x05,
```

```
    DMI_SLOT_TYPE_PCI                   = 0x06,
    DMI_SLOT_TYPE_PCMCIA                = 0x07,
    DMI_SLOT_TYPE_VESA                  = 0x08,
    DMI_SLOT_TYPE_PROPRIETARY           = 0x09,
    DMI_SLOT_TYPE_PROCESSOR_CARD        = 0x0A,
    DMI_SLOT_TYPE_MEMORY_CARD           = 0x0B,
    DMI_SLOT_TYPE_IO_RISER_CARD         = 0x0C,
    DMI_SLOT_TYPE_NUBUS                 = 0x0D,
    DMI_SLOT_TYPE_PCI_66                = 0x0E,
    DMI_SLOT_TYPE_AGP                   = 0x0F,
    DMI_SLOT_TYPE_AGP_2X                = 0x10,
    DMI_SLOT_TYPE_AGP_4X                = 0x11,
    DMI_SLOT_TYPE_PCI_X                 = 0x12,
    DMI_SLOT_TYPE_AGP_8X                = 0x13,
    DMI_SLOT_TYPE_M2_S1_DP              = 0x14,
    DMI_SLOT_TYPE_M2_S1_SD              = 0x15,
    DMI_SLOT_TYPE_M2_S2                 = 0x16,
    DMI_SLOT_TYPE_M2_S3                 = 0x17,
    DMI_SLOT_TYPE_MXM_I                 = 0x18,
    DMI_SLOT_TYPE_MXM_II                = 0x19,
    DMI_SLOT_TYPE_MXM_III_STD           = 0x1A,
    DMI_SLOT_TYPE_MXM_III_HE            = 0x1B,
    DMI_SLOT_TYPE_MXM_IV                = 0x1C,
    DMI_SLOT_TYPE_MXM_30_A              = 0x1D,
    DMI_SLOT_TYPE_MXM_30_B              = 0x1E,
    DMI_SLOT_TYPE_PCI_E_G2_SFF8639      = 0x1F,
    DMI_SLOT_TYPE_PCI_E_G3_SFF8639      = 0x20,
    DMI_SLOT_TYPE_PCI_E_MINI_52PIN_BSKO = 0x21,
    DMI_SLOT_TYPE_PCI_E_MINI_52PIN      = 0x22,
    DMI_SLOT_TYPE_PCI_E_MINI_76PIN      = 0x23,
    DMI_SLOT_TYPE_PCI_E_G4_SFF8639      = 0x24,
    DMI_SLOT_TYPE_PCI_E_G5_SFF8639      = 0x25,
    DMI_SLOT_TYPE_OCP_NIC_30_SFF        = 0x26,
    DMI_SLOT_TYPE_OCP_NIC_30_LFF        = 0x27,
    DMI_SLOT_TYPE_OCP_NIC_LEGACY        = 0x28,
    DMI_SLOT_TYPE_CXL_FLEXBUS_10        = 0x30,
    DMI_SLOT_TYPE_PC_98_C20             = 0xA0,
    DMI_SLOT_TYPE_PC_98_C24             = 0xA1,
    DMI_SLOT_TYPE_PC_98_E               = 0xA2,
    DMI_SLOT_TYPE_PC_98_LOCAL_BUS       = 0xA3,
    DMI_SLOT_TYPE_PC_98_CARD            = 0xA4,
    DMI_SLOT_TYPE_PCI_E                 = 0xA5,
    DMI_SLOT_TYPE_PCI_E_X1              = 0xA6,
    DMI_SLOT_TYPE_PCI_E_X2              = 0xA7,
    DMI_SLOT_TYPE_PCI_E_X4              = 0xA8,
    DMI_SLOT_TYPE_PCI_E_X8              = 0xA9,
    DMI_SLOT_TYPE_PCI_E_X16             = 0xAA,
    DMI_SLOT_TYPE_PCI_E_G2              = 0xAB,
```

```
    DMI_SLOT_TYPE_PCI_E_G2_X1           = 0xAC,
    DMI_SLOT_TYPE_PCI_E_G2_X2           = 0xAD,
    DMI_SLOT_TYPE_PCI_E_G2_X4           = 0xAE,
    DMI_SLOT_TYPE_PCI_E_G2_X8           = 0xAF,
    DMI_SLOT_TYPE_PCI_E_G2_X16          = 0xB0,
    DMI_SLOT_TYPE_PCI_E_G3              = 0xB1,
    DMI_SLOT_TYPE_PCI_E_G3_X1           = 0xB2,
    DMI_SLOT_TYPE_PCI_E_G3_X2           = 0xB3,
    DMI_SLOT_TYPE_PCI_E_G3_X4           = 0xB4,
    DMI_SLOT_TYPE_PCI_E_G3_X8           = 0xB5,
    DMI_SLOT_TYPE_PCI_E_G3_X16          = 0xB6,
    DMI_SLOT_TYPE_PCI_E_G4              = 0xB8,
    DMI_SLOT_TYPE_PCI_E_G4_X1           = 0xB9,
    DMI_SLOT_TYPE_PCI_E_G4_X2           = 0xBA,
    DMI_SLOT_TYPE_PCI_E_G4_X4           = 0xBB,
    DMI_SLOT_TYPE_PCI_E_G4_X8           = 0xBC,
    DMI_SLOT_TYPE_PCI_E_G4_X16          = 0xBD,
    DMI_SLOT_TYPE_PCI_E_G5              = 0xBE,
    DMI_SLOT_TYPE_PCI_E_G5_X1           = 0xBF,
    DMI_SLOT_TYPE_PCI_E_G5_X2           = 0xC0,
    DMI_SLOT_TYPE_PCI_E_G5_X4           = 0xC1,
    DMI_SLOT_TYPE_PCI_E_G5_X8           = 0xC2,
    DMI_SLOT_TYPE_PCI_E_G5_X16          = 0xC3,
    DMI_SLOT_TYPE_PCI_E_G6              = 0xC4,
    DMI_SLOT_TYPE_EDSFF_E1              = 0xC5,
    DMI_SLOT_TYPE_EDSFF_E3              = 0xC6,
} dmi_slot_type_t;

const char *dmi_slot_type_name(dmi_slot_type_t value);
```

Description

The `dmi_slot_type_t` enumeration tells the bus or the form factor of a system slot, and is the type of the `type` field of `dmi_slot_t`. The specification assigns no values from `0x29` to `0x2F`, from `0x31` to `0x9F`, `0xB7`, and above `0xC6`.

PCI Express types which give a generation only are meant for slots whose physical width is their electrical width, the `bus_width` field of `dmi_slot_t` giving both. Types which also give a width are meant for slots wider than their electrical width: the type gives the physical width, and `bus_width` the electrical one.

The `dmi_slot_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SLOT_TYPE_UNSPEC`
: Unspecified.

`DMI_SLOT_TYPE_OTHER`
: Other.

`DMI_SLOT_TYPE_UNKNOWN`
: Unknown.

`DMI_SLOT_TYPE_ISA`
: ISA.

`DMI_SLOT_TYPE_MCA`
: Micro Channel (MCA).

`DMI_SLOT_TYPE_EISA`
: EISA.

`DMI_SLOT_TYPE_PCI`
: PCI.

`DMI_SLOT_TYPE_PCMCIA`
: PC Card (PCMCIA).

`DMI_SLOT_TYPE_VESA`
: VESA Local Bus (VL-VESA).

`DMI_SLOT_TYPE_PROPRIETARY`
: Proprietary.

`DMI_SLOT_TYPE_PROCESSOR_CARD`
: Processor card slot.

`DMI_SLOT_TYPE_MEMORY_CARD`
: Proprietary memory card slot.

`DMI_SLOT_TYPE_IO_RISER_CARD`
: I/O riser card slot.

`DMI_SLOT_TYPE_NUBUS`
: NuBus.

`DMI_SLOT_TYPE_PCI_66`
: PCI, 66 MHz capable.

`DMI_SLOT_TYPE_AGP`
: AGP.

`DMI_SLOT_TYPE_AGP_2X`
: AGP 2x.

`DMI_SLOT_TYPE_AGP_4X`
: AGP 4x.

`DMI_SLOT_TYPE_PCI_X`
: PCI-X.

`DMI_SLOT_TYPE_AGP_8X`

AGP 8x.

`DMI_SLOT_TYPE_M2_S1_DP`

M.2 Socket 1-DP, mechanical key A.

`DMI_SLOT_TYPE_M2_S1_SD`

M.2 Socket 1-SD, mechanical key E.

`DMI_SLOT_TYPE_M2_S2`

M.2 Socket 2, mechanical key B.

`DMI_SLOT_TYPE_M2_S3`

M.2 Socket 3, mechanical key M.

`DMI_SLOT_TYPE_MXM_I`

MXM type I.

`DMI_SLOT_TYPE_MXM_II`

MXM type II.

`DMI_SLOT_TYPE_MXM_III_STD`

MXM type III, standard connector.

`DMI_SLOT_TYPE_MXM_III_HE`

MXM type III, HE connector.

`DMI_SLOT_TYPE_MXM_IV`

MXM type IV.

`DMI_SLOT_TYPE_MXM_30_A`

MXM 3.0 type A.

`DMI_SLOT_TYPE_MXM_30_B`

MXM 3.0 type B.

`DMI_SLOT_TYPE_PCI_E_G2_SFF8639`

PCI Express Gen 2, SFF-8639 (U.2).

`DMI_SLOT_TYPE_PCI_E_G3_SFF8639`

PCI Express Gen 3, SFF-8639 (U.2).

`DMI_SLOT_TYPE_PCI_E_MINI_52PIN_BSKO`

PCI Express Mini, 52 pins (CEM 2.0), with bottom-side keep-outs. A short `length` means half-size cards only, a long one full-size cards.

`DMI_SLOT_TYPE_PCI_E_MINI_52PIN`

PCI Express Mini, 52 pins (CEM 2.0), without bottom-side keep-outs. A short `length` means half-size cards only, a long one full-size cards.

`DMI_SLOT_TYPE_PCI_E_MINI_76PIN`
PCI Express Mini, 76 pins (CEM 2.0), for Display-Mini cards.

`DMI_SLOT_TYPE_PCI_E_G4_SFF8639`
PCI Express Gen 4, SFF-8639 (U.2).

`DMI_SLOT_TYPE_PCI_E_G5_SFF8639`
PCI Express Gen 5, SFF-8639 (U.2).

`DMI_SLOT_TYPE_OCP_NIC_30_SFF`
OCP NIC 3.0, small form factor (SFF).

`DMI_SLOT_TYPE_OCP_NIC_30_LFF`
OCP NIC 3.0, large form factor (LFF).

`DMI_SLOT_TYPE_OCP_NIC_LEGACY`
OCP NIC older than 3.0.

`DMI_SLOT_TYPE_CXL_FLEXBUS_10`
CXL Flex Bus 1.0. Deprecated: the CXL bits of `dmi_slot_features_ex` describe CXL-capable slots of any PCI Express Gen 5 or later type instead.

`DMI_SLOT_TYPE_PC_98_C20`
PC-98/C20.

`DMI_SLOT_TYPE_PC_98_C24`
PC-98/C24.

`DMI_SLOT_TYPE_PC_98_E`
PC-98/E.

`DMI_SLOT_TYPE_PC_98_LOCAL_BUS`
PC-98/Local bus.

`DMI_SLOT_TYPE_PC_98_CARD`
PC-98/Card.

`DMI_SLOT_TYPE_PCI_E`
PCI Express (Gen 1), physical width equal to the electrical one, which `bus_width` gives.

`DMI_SLOT_TYPE_PCI_E_X1`
PCI Express (Gen 1), x1 physical width.

`DMI_SLOT_TYPE_PCI_E_X2`
PCI Express (Gen 1), x2 physical width.

`DMI_SLOT_TYPE_PCI_E_X4`
PCI Express (Gen 1), x4 physical width.

`DMI_SLOT_TYPE_PCI_E_X8`
PCI Express (Gen 1), x8 physical width.

`DMI_SLOT_TYPE_PCI_E_X16`
PCI Express (Gen 1), x16 physical width.

`DMI_SLOT_TYPE_PCI_E_G2`
PCI Express Gen 2, physical width equal to the electrical one, which `bus_width` gives.

`DMI_SLOT_TYPE_PCI_E_G2_X1`
PCI Express Gen 2, x1 physical width.

`DMI_SLOT_TYPE_PCI_E_G2_X2`
PCI Express Gen 2, x2 physical width.

`DMI_SLOT_TYPE_PCI_E_G2_X4`
PCI Express Gen 2, x4 physical width.

`DMI_SLOT_TYPE_PCI_E_G2_X8`
PCI Express Gen 2, x8 physical width.

`DMI_SLOT_TYPE_PCI_E_G2_X16`
PCI Express Gen 2, x16 physical width.

`DMI_SLOT_TYPE_PCI_E_G3`
PCI Express Gen 3, physical width equal to the electrical one, which `bus_width` gives.

`DMI_SLOT_TYPE_PCI_E_G3_X1`
PCI Express Gen 3, x1 physical width.

`DMI_SLOT_TYPE_PCI_E_G3_X2`
PCI Express Gen 3, x2 physical width.

`DMI_SLOT_TYPE_PCI_E_G3_X4`
PCI Express Gen 3, x4 physical width.

`DMI_SLOT_TYPE_PCI_E_G3_X8`
PCI Express Gen 3, x8 physical width.

`DMI_SLOT_TYPE_PCI_E_G3_X16`
PCI Express Gen 3, x16 physical width.

`DMI_SLOT_TYPE_PCI_E_G4`
PCI Express Gen 4, physical width equal to the electrical one, which `bus_width` gives.

`DMI_SLOT_TYPE_PCI_E_G4_X1`
PCI Express Gen 4, x1 physical width.

`DMI_SLOT_TYPE_PCI_E_G4_X2`
PCI Express Gen 4, x2 physical width.

`DMI_SLOT_TYPE_PCI_E_G4_X4`
PCI Express Gen 4, x4 physical width.

`DMI_SLOT_TYPE_PCI_E_G4_X8`
: PCI Express Gen 4, x8 physical width.

`DMI_SLOT_TYPE_PCI_E_G4_X16`
: PCI Express Gen 4, x16 physical width.

`DMI_SLOT_TYPE_PCI_E_G5`
: PCI Express Gen 5, physical width equal to the electrical one, which `bus_width` gives.

`DMI_SLOT_TYPE_PCI_E_G5_X1`
: PCI Express Gen 5, x1 physical width.

`DMI_SLOT_TYPE_PCI_E_G5_X2`
: PCI Express Gen 5, x2 physical width.

`DMI_SLOT_TYPE_PCI_E_G5_X4`
: PCI Express Gen 5, x4 physical width.

`DMI_SLOT_TYPE_PCI_E_G5_X8`
: PCI Express Gen 5, x8 physical width.

`DMI_SLOT_TYPE_PCI_E_G5_X16`
: PCI Express Gen 5, x16 physical width.

`DMI_SLOT_TYPE_PCI_E_G6`
: PCI Express Gen 6 or later; the `information` field of `dmi_slot_t` holds the generation.

`DMI_SLOT_TYPE_EDSFF_E1`
: EDSFF E1 slot (1U enterprise and datacenter form factor, E1.S or E1.L).

`DMI_SLOT_TYPE_EDSFF_E3`
: EDSFF E3 slot (3-inch enterprise and datacenter form factor, E3.S or E3.L).

Return value

`dmi_slot_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_slot`, `dmi_slot_features_ex`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_slot_usage

Synopsis

```
#include <opendmi/entity/slot.h>

typedef enum dmi_slot_usage {
    DMI_SLOT_USAGE_UNSPEC      = 0x00,
    DMI_SLOT_USAGE_OTHER       = 0x01,
    DMI_SLOT_USAGE_UNKNOWN     = 0x02,
    DMI_SLOT_USAGE_AVAILABLE   = 0x03,
    DMI_SLOT_USAGE_IN_USE      = 0x04,
    DMI_SLOT_USAGE_UNAVAILABLE = 0x05,
} dmi_slot_usage_t;

const char *dmi_slot_usage_name(dmi_slot_usage_t value);
```

Description

The `dmi_slot_usage_t` enumeration tells whether a system slot is in use, and is the type of the `usage` field of `dmi_slot_t`.

The `dmi_slot_usage_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SLOT_USAGE_UNSPEC`
: Unspecified.

`DMI_SLOT_USAGE_OTHER`
: Other.

`DMI_SLOT_USAGE_UNKNOWN`
: Unknown.

`DMI_SLOT_USAGE_AVAILABLE`
: Free.

`DMI_SLOT_USAGE_IN_USE`
: In use.

`DMI_SLOT_USAGE_UNAVAILABLE`
: Not usable, e.g. wired to a processor socket which is empty.

Return value

`dmi_slot_usage_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid

until the locale is changed.

Errors

None.

See also

`dmi_slot`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_slot_width

Synopsis

```
#include <opendmi/entity/slot.h>

typedef enum dmi_slot_width {
    DMI_SLOT_WIDTH_UNSPEC  = 0x00,
    DMI_SLOT_WIDTH_OTHER   = 0x01,
    DMI_SLOT_WIDTH_UNKNOWN = 0x02,
    DMI_SLOT_WIDTH_8_BIT   = 0x03,
    DMI_SLOT_WIDTH_16_BIT  = 0x04,
    DMI_SLOT_WIDTH_32_BIT  = 0x05,
    DMI_SLOT_WIDTH_64_BIT  = 0x06,
    DMI_SLOT_WIDTH_128_BIT = 0x07,
    DMI_SLOT_WIDTH_1X      = 0x08,
    DMI_SLOT_WIDTH_2X      = 0x09,
    DMI_SLOT_WIDTH_4X      = 0x0A,
    DMI_SLOT_WIDTH_8X      = 0x0B,
    DMI_SLOT_WIDTH_12X     = 0x0C,
    DMI_SLOT_WIDTH_16X     = 0x0D,
    DMI_SLOT_WIDTH_32X     = 0x0E,
} dmi_slot_width_t;

const char *dmi_slot_width_name(dmi_slot_width_t value);
```

Description

The `dmi_slot_width_t` enumeration tells the width of the data bus of a system slot: a number of bits for parallel buses such as PCI, a number of lanes for serial links such as PCI Express. It is the type of the `bus_width`, `base_bus_width` and `physical_width` fields of `dmi_slot_t`, and of the `bus_width` field of `dmi_slot_peer_group_t`. For PCI Express, the electrical width is the widest link the slot supports; the width of the link actually trained is not recorded in SMBIOS.

The `dmi_slot_width_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SLOT_WIDTH_UNSPEC`
: Unspecified.

`DMI_SLOT_WIDTH_OTHER`
: Other.

`DMI_SLOT_WIDTH_UNKNOWN`
: Unknown.

`DMI_SLOT_WIDTH_8_BIT`
: 8 bits.

`DMI_SLOT_WIDTH_16_BIT`
: 16 bits.

`DMI_SLOT_WIDTH_32_BIT`
: 32 bits.

`DMI_SLOT_WIDTH_64_BIT`
: 64 bits.

`DMI_SLOT_WIDTH_128_BIT`
: 128 bits.

`DMI_SLOT_WIDTH_1X`
: One lane (x1).

`DMI_SLOT_WIDTH_2X`
: Two lanes (x2).

`DMI_SLOT_WIDTH_4X`
: Four lanes (x4).

`DMI_SLOT_WIDTH_8X`
: Eight lanes (x8).

`DMI_SLOT_WIDTH_12X`
: Twelve lanes (x12).

`DMI_SLOT_WIDTH_16X`
: Sixteen lanes (x16).

`DMI_SLOT_WIDTH_32X`
: Thirty-two lanes (x32).

Return value

`dmi_slot_width_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_slot`, `dmi_slot_peer_group`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/string-property.h>

dmi_string_property

Synopsis

```
#include <opendmi/entity/string-property.h>

typedef struct dmi_string_property {
    dmi_property_t ident;
    const char *value;
    dmi_handle_t parent_handle;
    dmi_entity_t *parent;
} dmi_string_property_t;
```

Description

The `dmi_string_property_t` structure holds the decoded string property structure (type 46) of SMBIOS: a string attached to another structure under a property identifier, which lets firmware give any structure extra strings without changing its definition. A structure can have several properties, each coming from a type 46 structure of its own.

While linking, each property is attached to its parent structure, where `dmi_entity_property()` finds it by its identifier.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(string_property)`, and lives as long as the context it has been decoded in.

Fields

`ident`

Identifier of the property: `DMI_PROPERTY_ID_UEFI_DEVICE_PATH` for the text form of a UEFI device path, or a value of the ranges set aside for the DMTF (2 to 32767), for firmware vendors (32768 to 49151) and for OEMs (49152 to 65535). Zero is reserved. The field holds the value the firmware reports.

`value`

Value of the property, `nullptr` if the structure carries none. A UEFI device path is given in its text form, encoded in UTF-8.

`parent_handle`

Handle of the structure the property applies to.

`parent`

The structure the property applies to, set once the property has been attached to it while linking. `nullptr` before linking, or if the parent handle is not set, refers to no structure, or refers to another string property; linking reports those cases as errors.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(string_property), true);
dmi_string_property_t *property = dmi_entity_info(entity,
                                                  DMI_TYPE(string_property));

if ((property != nullptr) && (property->value != nullptr))
    printf("Property %u of 0x%04X: %s\n", (unsigned)property->ident,
           (unsigned)property->parent_handle, property->value);
```

See also

`dmi_entity_info`, `dmi_entity_link`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.5 or newer.

<opendmi/entity/system-boot.h>

dmi_boot_status

Synopsis

```
#include <opendmi/entity/system-boot.h>

typedef enum dmi_boot_status {
    DMI_BOOT_STATUS_NO_ERRORS_DETECTED         = 0x00,
    DMI_BOOT_STATUS_NO_BOOTABLE_MEDIA          = 0x01,
    DMI_BOOT_STATUS_OS_FAILED_TO_LOAD          = 0x02,
    DMI_BOOT_STATUS_FW_DETECTED_HW_FAILURE     = 0x03,
    DMI_BOOT_STATUS_OS_DETECTED_HW_FAILURE     = 0x04,
    DMI_BOOT_STATUS_USER_REQUESTED_BOOT        = 0x05,
    DMI_BOOT_STATUS_SYSTEM_SECURITY_VIOLATION  = 0x06,
    DMI_BOOT_STATUS_PREVIOUSLY_REQUESTED_IMAGE = 0x07,
    DMI_BOOT_STATUS_SYSTEM_WDT_EXPIRED         = 0x08,
    __DMI_BOOT_STATUS_RESERVED_START           = 0x09,
    __DMI_BOOT_STATUS_RESERVED_END             = 0x7F,
    __DMI_BOOT_STATUS_VENDOR_SPECIFIC_START    = 0x80,
    __DMI_BOOT_STATUS_VENDOR_SPECIFIC_END      = 0xBF,
    __DMI_BOOT_STATUS_PRODUCT_SPECIFIC_START   = 0xC0,
    __DMI_BOOT_STATUS_PRODUCT_SPECIFIC_END     = 0xFF
} dmi_boot_status_t;

const char *dmi_boot_status_name(dmi_boot_status_t value);
```

Description

The `dmi_boot_status_t` enumeration tells why the system has booted, as the system boot information structure (type 32) of SMBIOS reports it, and is the type of the `status` field of `dmi_system_boot_t`.

Only the codes 0x00 to 0x08 are defined. The codes 0x09 to 0x7F are reserved, the codes 0x80 to 0xBF are specific to the vendor named by the manufacturer string of the system information structure (type 1), and the codes 0xC0 to 0xFF are specific to the product named by its manufacturer and product name strings. The hidden enumerators starting with `_DMI_BOOT_STATUS` bound these ranges.

The `dmi_boot_status_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_BOOT_STATUS_NO_ERRORS_DETECTED`
: No errors detected.

`DMI_BOOT_STATUS_NO_BOOTABLE_MEDIA`

No bootable media.

`DMI_BOOT_STATUS_OS_FAILED_TO_LOAD`

The normal operating system failed to load.

`DMI_BOOT_STATUS_FW_DETECTED_HW_FAILURE`

The firmware detected a hardware failure, of a known kind or not.

`DMI_BOOT_STATUS_OS_DETECTED_HW_FAILURE`

The operating system detected a hardware failure.

`DMI_BOOT_STATUS_USER_REQUESTED_BOOT`

The user requested the boot, usually with a key.

`DMI_BOOT_STATUS_SYSTEM_SECURITY_VIOLATION`

System security violation.

`DMI_BOOT_STATUS_PREVIOUSLY_REQUESTED_IMAGE`

A boot image requested earlier, for example by software of the operating system; additional data may follow the code.

`DMI_BOOT_STATUS_SYSTEM_WDT_EXPIRED`

The system watchdog timer expired, and rebooted the system.

`__DMI_BOOT_STATUS_RESERVED_START`

First reserved code.

`__DMI_BOOT_STATUS_RESERVED_END`

Last reserved code.

`__DMI_BOOT_STATUS_VENDOR_SPECIFIC_START`

First vendor/OEM-specific code.

`__DMI_BOOT_STATUS_VENDOR_SPECIFIC_END`

Last vendor/OEM-specific code.

`__DMI_BOOT_STATUS_PRODUCT_SPECIFIC_START`

First product-specific code.

`__DMI_BOOT_STATUS_PRODUCT_SPECIFIC_END`

Last product-specific code.

Return value

`dmi_boot_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_boot`, `dmi_system`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

dmi_system_boot

Synopsis

```
#include <opendmi/entity/system-boot.h>

typedef struct dmi_system_boot {
    dmi_boot_status_t status;
    dmi_binary_t      status_data;
    bool              has_status_data;
} dmi_system_boot_t;
```

Description

The `dmi_system_boot_t` structure holds the decoded system boot information structure (type 32) of SMBIOS: the reason of the last boot of the system, which the firmware passes to the boot image, for example of a PXE boot, or to management software. A boot image may use it, for example, to run diagnostics after a hardware failure.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system_boot)`, and lives as long as the context it has been decoded in.

Fields

`status`
: Boot status code, see `dmi_boot_status`.

`status_data`
: Bytes following the status code up to the end of the structure, as stored. Empty if there are none.

`has_status_data`
: `true` if the status code is vendor-specific or product-specific (0x80 and above), whose additional data the vendor defines. The command line tool only shows `status_data` then.

Examples

```
dmi_registry_t    *registry = dmi_get_registry(context);
dmi_entity_t      *entity   = dmi_registry_lookup_first(
        registry, DMI_TYPE(system_boot), true);
dmi_system_boot_t *boot     = dmi_entity_info(entity, DMI_TYPE(system_boot));
```

```
if ((boot != nullptr) && (boot->status != DMI_BOOT_STATUS_NO_ERRORS_DETECTED))
    printf("Boot status: 0x%02X\n", (unsigned)boot->status);
```

See also

`dmi_entity_info`, `dmi_boot_status`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.3 or newer.

<opendmi/entity/system-config.h>

dmi_system_config_opts

Synopsis

```
#include <opendmi/entity/system-config.h>

typedef struct dmi_system_config_opts {
    size_t option_count;
    const char **options;
} dmi_system_config_opts_t;
```

Description

The `dmi_system_config_opts_t` structure holds the decoded system configuration options structure (type 12) of SMBIOS: free-form strings telling how to set the jumpers and the switches of the baseboard, e.g. which jumper position clears the CMOS.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system_config_options)` and lives as long as the context it has been decoded in.

Fields

`option_count`
: Number of options, as the structure declares it.

`options`
: Option strings, in order, `option_count` of them. `nullptr` when `option_count` is 0. An element is `nullptr` when the structure does not carry the string.

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                         DMI_TYPE(system_config_options),
                                         true);
dmi_system_config_opts_t *config   = dmi_entity_info(entity,
                                         DMI_TYPE(system_config_options));

if (config != nullptr) {
    for (size_t i = 0; i < config->option_count; i++) {
        if (config->options[i] != nullptr)
            printf("%s\n", config->options[i]);
    }
}
```

See also

`dmi_entity_info`, `dmi_entity_string`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

<opendmi/entity/system-event-log.h>

dmi_event_log_data_format

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef enum dmi_event_log_data_format {
    DMI_EVENT_LOG_DATA_FORMAT_NONE                 = 0x00,
    DMI_EVENT_LOG_DATA_FORMAT_HANDLE               = 0x01,
    DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_EVENT       = 0x02,
    DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_HANDLE      = 0x03,
    DMI_EVENT_LOG_DATA_FORMAT_POST_RESULTS         = 0x04,
    DMI_EVENT_LOG_DATA_FORMAT_SYSTEM_MGMT          = 0x05,
    DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_SYSTEM_MGMT = 0x06,
} dmi_event_log_data_format_t;

const char *dmi_event_log_data_format_name(dmi_event_log_data_format_t value);
```

Description

The `dmi_event_log_data_format_t` enumeration tells how the first bytes of the variable data of an event of the system event log are laid out, and is the type of the `data_format` field of `dmi_system_log_type_descriptor_t`. Values 0x07 to 0x7F are left for future use by the specification, and values 0x80 to 0xFF are specific to the system or the OEM. The enumeration also has private members, starting with two underscores, which mark the bounds of these ranges.

The `dmi_event_log_data_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_EVENT_LOG_DATA_FORMAT_NONE`

No standard format: the variable data, if any, is OEM-specific.

`DMI_EVENT_LOG_DATA_FORMAT_HANDLE`

A word with the handle of the SMBIOS structure of the failed hardware.

`DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_EVENT`

A double word with a multiple-event counter.

`DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_HANDLE`

A word with the handle of the SMBIOS structure of the failed hardware, followed by a double word with a multiple-event counter.

`DMI_EVENT_LOG_DATA_FORMAT_POST_RESULTS`

Two double words with the bitmap of the errors found by the power-on self-test.

`DMI_EVENT_LOG_DATA_FORMAT_SYSTEM_MGMT`

A double word identifying a system management condition.

`DMI_EVENT_LOG_DATA_FORMAT_MULTIPLE_SYSTEM_MGMT`

A double word identifying a system management condition, followed by a double word with a multiple-event counter.

Return value

`dmi_event_log_data_format_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_log_type_descriptor`, `dmi_system_event_log`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_event_log_type

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef enum dmi_event_log_type {
    DMI_EVENT_LOG_TYPE_SINGLE_BIT_ECC        = 0x01,
    DMI_EVENT_LOG_TYPE_MULTI_BIT_ECC         = 0x02,
    DMI_EVENT_LOG_TYPE_PARITY                = 0x03,
    DMI_EVENT_LOG_TYPE_BUS_TIMEOUT           = 0x04,
    DMI_EVENT_LOG_TYPE_IO_CHANNEL_CHECK      = 0x05,
    DMI_EVENT_LOG_TYPE_SOFTWARE_NMI          = 0x06,
    DMI_EVENT_LOG_TYPE_POST_MEMORY_RESIZE    = 0x07,
    DMI_EVENT_LOG_TYPE_POST_ERROR            = 0x08,
    DMI_EVENT_LOG_TYPE_PCI_PARITY            = 0x09,
    DMI_EVENT_LOG_TYPE_PCI_SYSTEM            = 0x0A,
    DMI_EVENT_LOG_TYPE_CPU_FAILURE           = 0x0B,
    DMI_EVENT_LOG_TYPE_EISA_FAILSAFE_TIMEOUT = 0x0C,
    DMI_EVENT_LOG_TYPE_MEMORY_LOG_DISABLED   = 0x0D,
    DMI_EVENT_LOG_TYPE_LOGGING_DISABLED      = 0x0E,
    DMI_EVENT_LOG_TYPE_LIMIT_EXCEEDED        = 0x10,
```

```
    DMI_EVENT_LOG_TYPE_TIMER_EXPIRED         = 0x11,
    DMI_EVENT_LOG_TYPE_CONFIG_INFO           = 0x12,
    DMI_EVENT_LOG_TYPE_HARD_DISK_INFO        = 0x13,
    DMI_EVENT_LOG_TYPE_RECONFIGURED          = 0x14,
    DMI_EVENT_LOG_TYPE_CPU_COMPLEX_ERROR     = 0x15,
    DMI_EVENT_LOG_TYPE_LOG_CLEARED           = 0x16,
    DMI_EVENT_LOG_TYPE_SYSTEM_BOOT           = 0x17,
    DMI_EVENT_LOG_TYPE_END_OF_LOG            = 0xFF,
} dmi_event_log_type_t;

const char *dmi_event_log_type_name(dmi_event_log_type_t value);
```

Description

The `dmi_event_log_type_t` enumeration tells the type of an event of the system event log, and is the type of the `type` field of `dmi_system_log_type_descriptor_t`. Values 0x00 and 0x0F are reserved, values 0x18 to 0x7F are left for future use by the specification, and values 0x80 to 0xFE are specific to the system or the OEM. The enumeration also has private members, starting with two underscores, which mark the bounds of these ranges.

The `dmi_event_log_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_EVENT_LOG_TYPE_SINGLE_BIT_ECC`
: Single-bit ECC memory error.

`DMI_EVENT_LOG_TYPE_MULTI_BIT_ECC`
: Multi-bit ECC memory error.

`DMI_EVENT_LOG_TYPE_PARITY`
: Parity memory error.

`DMI_EVENT_LOG_TYPE_BUS_TIMEOUT`
: Bus time-out.

`DMI_EVENT_LOG_TYPE_IO_CHANNEL_CHECK`
: I/O channel check.

`DMI_EVENT_LOG_TYPE_SOFTWARE_NMI`
: Software non-maskable interrupt.

`DMI_EVENT_LOG_TYPE_POST_MEMORY_RESIZE`
: Memory size changed during the power-on self-test.

`DMI_EVENT_LOG_TYPE_POST_ERROR`
: Error during the power-on self-test.

`DMI_EVENT_LOG_TYPE_PCI_PARITY`
: PCI parity error.

`DMI_EVENT_LOG_TYPE_PCI_SYSTEM`
: PCI system error.

`DMI_EVENT_LOG_TYPE_CPU_FAILURE`
: Processor failure.

`DMI_EVENT_LOG_TYPE_EISA_FAILSAFE_TIMEOUT`
: EISA fail-safe timer time-out.

`DMI_EVENT_LOG_TYPE_MEMORY_LOG_DISABLED`
: Logging of correctable memory errors disabled.

`DMI_EVENT_LOG_TYPE_LOGGING_DISABLED`
: Logging disabled for one type of event, after too many events of that type in a short time.

`DMI_EVENT_LOG_TYPE_LIMIT_EXCEEDED`
: A limit of the system, such as a voltage or temperature threshold, has been exceeded.

`DMI_EVENT_LOG_TYPE_TIMER_EXPIRED`
: An asynchronous hardware timer has expired and reset the system.

`DMI_EVENT_LOG_TYPE_CONFIG_INFO`
: Information about the system configuration.

`DMI_EVENT_LOG_TYPE_HARD_DISK_INFO`
: Information about a hard disk.

`DMI_EVENT_LOG_TYPE_RECONFIGURED`
: The system has been reconfigured.

`DMI_EVENT_LOG_TYPE_CPU_COMPLEX_ERROR`
: Uncorrectable error of the processor complex.

`DMI_EVENT_LOG_TYPE_LOG_CLEARED`
: The log area has been reset or cleared.

`DMI_EVENT_LOG_TYPE_SYSTEM_BOOT`
: System boot. When the firmware records it, it is the first event written on every boot.

`DMI_EVENT_LOG_TYPE_END_OF_LOG`
: End of the log: a record of this type follows the last event.

Return value

`dmi_event_log_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_log_type_descriptor`, `dmi_system_event_log`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_system_event_log

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef struct dmi_system_event_log {
    uint16_t                          area_length;
    uint16_t                          header_offset;
    uint16_t                          data_offset;
    dmi_system_log_access_method_t    access_method;
    dmi_system_log_status_t           status;
    uint32_t                          access_address;
    dmi_system_log_io_ports_t         access_ports;
    uint16_t                          access_gpnv_handle;
    uint32_t                          change_token;
    dmi_system_log_header_format_t    header_format;
    size_t                            descriptor_count;
    dmi_system_log_type_descriptor_t *descriptors;
} dmi_system_event_log_t;
```

Description

The `dmi_system_event_log_t` structure holds the decoded system event log structure (type 15) of SMBIOS. Its presence tells that the system keeps an event log: a fixed-length area of non-volatile storage, which starts with an optional header and goes on with event records of variable length. The structure says where the area lies, how software reaches it, and which kinds of events the log records. The library does not read the log area itself.

Software can notice new events by reading the structure again from time to time and comparing `change_token` with the value it has seen before. When the table has been read from a copy of the firmware data, `status` and `change_token` may be out of date.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system_event_log)`, and lives as long as the context it has been decoded in.

Fields

`area_length`

Length of the log area, in bytes, from the first byte of the header to the last byte of the data.

`header_offset`

Offset (or index) of the log header within the non-volatile storage, counted from the access method address. For the access method with a single 8-bit index port, the high byte is 0.

`data_offset`

Offset (or index) of the first byte of the log data, counted the same way. The data follows the header directly, so the length of the header is `data_offset - header_offset`.

`access_method`

Method used to reach the log area, see `dmi_system_log_access_method`. It also tells how `access_address` is to be read.

`status`

Whether the log area is valid and whether it is full, see `dmi_system_log_status`.

`access_address`

Access method address, as stored in the structure. For `DMI_SYSTEM_LOG_ACCESS_METHOD_MMAP_32BIT_ADDR`, the 32-bit physical address of the log area; the other access methods are better read through `access_ports` and `access_gpnv_handle`.

`access_ports`

Index and data I/O ports of the indexed I/O access methods, see `dmi_system_log_io_ports`: the low word of `access_address` is the index port, the high word the data port. Filled in whatever the access method is, but only meaningful for the indexed I/O ones.

`access_gpnv_handle`

Handle of the general-purpose non-volatile data area holding the log, which is the low word of `access_address`. Filled in whatever the access method is, but only meaningful for `DMI_SYSTEM_LOG_ACCESS_METHOD_DATA_FUNCTIONS`.

`change_token`

Token which the firmware changes every time the log is updated, 0 if the firmware does not implement it.

`header_format`

Format of the log header, see `dmi_system_log_header_format`. Present since SMBIOS 2.1; for older structures holds 0, which reads as `DMI_SYSTEM_LOG_HEADER_FORMAT_NO_HEADER`.

`descriptor_count`

Number of elements of `descriptors`: the descriptors the structure holds completely, 0 if it holds none. Present since SMBIOS 2.1; 0 otherwise.

`descriptors`

Array of the event log types the log supports, each with the format of the variable data of its events, see `dmi_system_log_type_descriptor`; `nullptr` if there are none. Descriptors longer than two bytes are allowed for future extensions, and their extra bytes are skipped. Present since SMBIOS 2.1.

Examples

```
dmi_registry_t         *registry = dmi_get_registry(context);
dmi_entity_t           *entity   = dmi_registry_lookup_first(registry,
                                       DMI_TYPE(system_event_log), false);
dmi_system_event_log_t *log_info = dmi_entity_info(entity,
                                       DMI_TYPE(system_event_log));

if (log_info != nullptr) {
    printf("Log area: %u bytes%s\n", (unsigned)log_info->area_length,
           log_info->status.is_log_area_full ? ", full" : "");

    for (size_t i = 0; i < log_info->descriptor_count; i++)
        printf("Supported event type: 0x%02x\n",
               (unsigned)log_info->descriptors[i].type);
}
```

See also

`dmi_entity_info`, `dmi_system_log_access_method`, `dmi_system_log_status`, `dmi_system_log_io_ports`, `dmi_system_log_header_format`, `dmi_system_log_type_descriptor`, `dmi_event_log_type`, `dmi_event_log_data_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_system_log_access_method

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef enum dmi_system_log_access_method {
    DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_8BIT_1_1 = 0x00,
    DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_8BIT_2_1 = 0x01,
    DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_16BIT    = 0x02,
    DMI_SYSTEM_LOG_ACCESS_METHOD_MMAP_32BIT_ADDR     = 0x03,
    DMI_SYSTEM_LOG_ACCESS_METHOD_DATA_FUNCTIONS      = 0x04,
    DMI_SYSTEM_LOG_ACCESS_METHOD_OEM_START           = 0x80,
    DMI_SYSTEM_LOG_ACCESS_METHOD_OEM_END             = 0xFF,
} dmi_system_log_access_method_t;

const char *dmi_system_log_access_method_name(
        dmi_system_log_access_method_t value);
```

Description

The `dmi_system_log_access_method_t` enumeration tells how software reaches the area of the system event log, and how the access method address is to be read. It is the type of the `access_method` field of `dmi_system_event_log_t`. Values 0x05 to 0x7F are reserved by the specification, and values 0x80 to 0xFF are specific to the firmware vendor or the OEM.

The `dmi_system_log_access_method_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_8BIT_1_1`

Indexed I/O with one 8-bit index port and one 8-bit data port, given by the `access_ports` field of the event log. Up to 256 bytes of the log can be reached.

`DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_8BIT_2_1`

Indexed I/O with two 8-bit index ports, which take the low and the high byte of the offset, and one 8-bit data port.

`DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_16BIT`

Indexed I/O with one 16-bit index port and one 8-bit data port.

`DMI_SYSTEM_LOG_ACCESS_METHOD_MMAP_32BIT_ADDR`

Memory-mapped log area, at the 32-bit physical address held by the `access_address` field of the event log.

`DMI_SYSTEM_LOG_ACCESS_METHOD_DATA_FUNCTIONS`

Log reached through the general-purpose non-volatile data functions of the firmware, with the handle held by the `access_gpnv_handle` field of the event log.

`DMI_SYSTEM_LOG_ACCESS_METHOD_OEM_START`

First value of the firmware vendor or OEM-specific range.

`DMI_SYSTEM_LOG_ACCESS_METHOD_OEM_END`

Last value of the firmware vendor or OEM-specific range.

Return value

`dmi_system_log_access_method_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_event_log`, `dmi_system_log_io_ports`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_system_log_header_format

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef enum dmi_system_log_header_format {
    DMI_SYSTEM_LOG_HEADER_FORMAT_NO_HEADER = 0x00,
    DMI_SYSTEM_LOG_HEADER_FORMAT_TYPE_1    = 0x01,
    DMI_SYSTEM_LOG_HEADER_FORMAT_OEM_START = 0x80,
    DMI_SYSTEM_LOG_HEADER_FORMAT_OEM_END   = 0xFF,
} dmi_system_log_header_format_t;

const char *dmi_system_log_header_format_name(
        dmi_system_log_header_format_t value);
```

Description

The `dmi_system_log_header_format_t` enumeration tells the format of the header at the start of the system event log area, and is the type of the `header_format` field of `dmi_system_event_log_t`. Values 0x02 to 0x7F are reserved by the specification, and values 0x80 to 0xFF are specific to the firmware vendor or the OEM.

The `dmi_system_log_header_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_SYSTEM_LOG_HEADER_FORMAT_NO_HEADER`

The log has no header: its data starts right at the start of the area.

`DMI_SYSTEM_LOG_HEADER_FORMAT_TYPE_1`

Type 1 log header, which holds the settings of the multiple-event counters and the location in CMOS memory of the flag which makes the firmware clear the log on the next boot.

`DMI_SYSTEM_LOG_HEADER_FORMAT_OEM_START`

First value of the firmware vendor or OEM-specific range.

`DMI_SYSTEM_LOG_HEADER_FORMAT_OEM_END`

Last value of the firmware vendor or OEM-specific range.

Return value

`dmi_system_log_header_format_name` returns the name of the value, or `nullptr` if the value has no

name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_event_log`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

dmi_system_log_io_ports

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef struct dmi_system_log_io_ports {
    uint16_t index_port;
    uint16_t data_port;
} dmi_system_log_io_ports_t;
```

Description

The `dmi_system_log_io_ports_t` structure holds the I/O port addresses used to reach the system event log with one of the indexed I/O access methods, and is the type of the `access_ports` field of `dmi_system_event_log_t`. Software writes the offset of a byte of the log area to the index port, then reads or writes that byte through the data port.

The decoder fills the structure from the access method address whatever the access method is; it is only meaningful when the `access_method` field of the event log is one of the `DMI_SYSTEM_LOG_ACCESS_METHOD_INDEXED_IO_*` values.

Fields

`index_port`
: I/O address of the index port, the low word of the access method address. With two 8-bit index ports, this is the port of the low byte of the offset, and the port of the high byte follows it.

`data_port`
: I/O address of the data port, the high word of the access method address.

See also

`dmi_system_event_log`, `dmi_system_log_access_method`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_system_log_status

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef union dmi_system_log_status {
    dmi_byte_t __value;

    struct {
        dmi_byte_t is_log_area_valid : 1;
        dmi_byte_t is_log_area_full  : 1;
        dmi_byte_t __reserved        : 6;
    };
} dmi_system_log_status_t;
```

Description

The `dmi_system_log_status_t` union holds the status byte of the system event log, and is the type of the `status` field of `dmi_system_event_log_t`. The byte can be read as a whole through `__value`, or bit by bit through the named members.

The firmware may not keep the status up to date in the table, so it tells the state of the log at the time the table was built.

Fields

`__value`
: Raw value of the status byte.

`is_log_area_valid`
: Set if the log area holds a valid log.

`is_log_area_full`
: Set if the log area is full.

`__reserved`
: Reserved bits, set to 0.

See also

`dmi_system_event_log`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_system_log_type_descriptor

Synopsis

```
#include <opendmi/entity/system-event-log.h>

typedef struct dmi_system_log_type_descriptor {
    dmi_event_log_type_t        type;
    dmi_event_log_data_format_t data_format;
} dmi_system_log_type_descriptor_t;
```

Description

The `dmi_system_log_type_descriptor_t` structure describes one kind of event the system event log supports, and is the type of the elements of the `descriptors` array of `dmi_system_event_log_t`. The presence of a descriptor tells that the log records events of its type, and how the first bytes of their variable data are laid out; a record may carry more variable data than its format covers.

Fields

`type`
: Type of the events, see `dmi_event_log_type`.

`data_format`
: Format of the start of the variable data of the events of that type, see `dmi_event_log_data_format`.

See also

`dmi_system_event_log`, `dmi_event_log_type`, `dmi_event_log_data_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/system-reset.h>

dmi_boot_option

Synopsis

```
#include <opendmi/entity/system-reset.h>

typedef enum dmi_boot_option {
    DMI_BOOT_OPTION_RESERVED         = 0x00,
    DMI_BOOT_OPTION_OPERATING_SYSTEM = 0x01,
    DMI_BOOT_OPTION_SYSTEM_UTILS     = 0x02,
    DMI_BOOT_OPTION_AVOID_REBOOT     = 0x03,
} dmi_boot_option_t;

const char *dmi_boot_option_name(dmi_boot_option_t value);
```

Description

The `dmi_boot_option_t` enumeration tells what a system does after an automatic system reset. It is the type of the `boot_on_watchdog` and `boot_on_limit` fields of `dmi_system_reset_t`.

The `dmi_boot_option_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_BOOT_OPTION_RESERVED`

Reserved by the specification, and not to be used.

`DMI_BOOT_OPTION_OPERATING_SYSTEM`

Boot the operating system.

`DMI_BOOT_OPTION_SYSTEM_UTILS`

Boot the system utilities.

`DMI_BOOT_OPTION_AVOID_REBOOT`

Do not reboot.

Return value

`dmi_boot_option_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_system_reset`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

dmi_system_reset

Synopsis

```
#include <opendmi/entity/system-reset.h>

typedef struct dmi_system_reset {
    bool is_enabled;
    dmi_boot_option_t boot_on_watchdog;
    dmi_boot_option_t boot_on_limit;
    bool has_watchdog;
    unsigned short reset_count;
    unsigned short reset_limit;
    unsigned short timer_interval;
    unsigned short timeout;
} dmi_system_reset_t;
```

Description

The `dmi_system_reset_t` structure holds the decoded system reset structure (type 23) of SMBIOS, which describes the automatic system reset function. If the system has a watchdog timer, and the timer is not reset within its interval, the system resets itself and boots as `boot_on_watchdog` says. Once the resets reach their limit, the system does what `boot_on_limit` says instead.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system_reset)`, and lives as long as the context it has been decoded in.

Fields

`is_enabled`
: `true` if the user has enabled the system reset.

`boot_on_watchdog`
: What the system does after a watchdog reset, see `dmi_boot_option`.

`boot_on_limit`
: What the system does when the reset limit is reached, see `dmi_boot_option`.

`has_watchdog`
: `true` if the system has a watchdog timer.

`reset_count`
: Number of automatic system resets since the last intentional reset, or `USHRT_MAX` if unknown.

`reset_limit`
: Number of consecutive times the system reset is attempted, or `USHRT_MAX` if unknown.

`timer_interval`
: Interval of the watchdog timer, in minutes: if the timer is not reset within it, the system reset timeout starts. `USHRT_MAX` if unknown.

`timeout`
: Number of minutes before the reboot starts, after a power cycle, a local or remote system reset, or an automatic system reset. `USHRT_MAX` if unknown.

Examples

```
dmi_registry_t     *registry = dmi_get_registry(context);
dmi_entity_t       *entity   = dmi_registry_lookup_first(registry,
                                    DMI_TYPE(system_reset), false);
dmi_system_reset_t *reset    = dmi_entity_info(entity,
                                    DMI_TYPE(system_reset));

if ((reset != nullptr) && reset->is_enabled)
    printf("on watchdog reset: %s\n",
           dmi_boot_option_name(reset->boot_on_watchdog));
```

See also

`dmi_entity_info`, `dmi_boot_option`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.2 or newer.

<opendmi/entity/system.h>

dmi_system

Synopsis

```
#include <opendmi/entity/system.h>

typedef struct dmi_system {
    const char *vendor;
    const char *product;
    const char *version;
    const char *serial_number;
    dmi_uuid_t uuid;
    dmi_system_wakeup_type_t wakeup_type;
    const char *sku_number;
    const char *family;
} dmi_system_t;
```

Description

The `dmi_system_t` structure holds the decoded system information structure (type 1) of SMBIOS: the identity of the system as a whole, i.e. its maker, product name, serial number and UUID, and the event which powered it up. A table carries exactly one such structure, which is required since SMBIOS 2.3.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(system)`, and lives as long as the context it has been decoded in.

Fields

`vendor`
: Name of the manufacturer of the system; `nullptr` if the structure does not carry it.

`product`
: Product name of the system; `nullptr` if the structure does not carry it.

`version`
: Version of the product; `nullptr` if the structure does not carry it.

`serial_number`
: Serial number of the system; `nullptr` if the structure does not carry it.

`uuid`
: Universally unique identifier of the system, converted from the byte order of SMBIOS (the first three fields little-endian) to the one of RFC 4122, see `dmi_uuid_decode`. All bytes 0 mean that the system has no UUID; all bytes `0xFF`, that it has none yet but one can be set. Present since SMBIOS 2.1; all zeros in older data.

`wakeup_type`

Event which powered the system up, see `dmi_system_wakeup_type`. Present since SMBIOS 2.1; `DMI_SYSTEM_WAKEUP_TYPE_RESERVED` in older data.

`sku_number`

Stock keeping unit of the system: the identifier of the configuration it was sold in, also known as product ID or order number, in a format the vendor chooses; `nullptr` if the structure does not carry it. Present since SMBIOS 2.4.

`family`

Family of products the system belongs to: models which share their design and branding but differ in configuration and price; `nullptr` if the structure does not carry it. Present since SMBIOS 2.4.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                                     DMI_TYPE(system), false);
dmi_system_t   *system   = dmi_entity_info(entity, DMI_TYPE(system));

if ((system != nullptr) && (system->product != nullptr))
    printf("%s (%s)\n", system->product,
           dmi_system_wakeup_type_name(system->wakeup_type));
```

See also

`dmi_entity_info`, `dmi_system_wakeup_type`, `dmi_uuid_decode`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.0 or newer.

dmi_system_wakeup_type

Synopsis

```
#include <opendmi/entity/system.h>

typedef enum dmi_system_wakeup_type {
    DMI_SYSTEM_WAKEUP_TYPE_RESERVED      = 0x00,
    DMI_SYSTEM_WAKEUP_TYPE_OTHER         = 0x01,
    DMI_SYSTEM_WAKEUP_TYPE_UNKNOWN       = 0x02,
    DMI_SYSTEM_WAKEUP_TYPE_APM_TIMER     = 0x03,
    DMI_SYSTEM_WAKEUP_TYPE_MODEM_RING    = 0x04,
    DMI_SYSTEM_WAKEUP_TYPE_LAN_REMOTE    = 0x05,
    DMI_SYSTEM_WAKEUP_TYPE_POWER_SWITCH  = 0x06,
    DMI_SYSTEM_WAKEUP_TYPE_PCI_PME       = 0x07,
```

```
    DMI_SYSTEM_WAKEUP_TYPE_POWER_RESTORE = 0x08,
} dmi_system_wakeup_type_t;

const char *dmi_system_wakeup_type_name(dmi_system_wakeup_type_t value);
```

Description

The `dmi_system_wakeup_type_t` enumeration tells which event powered a system up, and is the type of the `wakeup_type` field of `dmi_system_t`.

The `dmi_system_wakeup_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_SYSTEM_WAKEUP_TYPE_RESERVED`
: Reserved; also the value of data older than SMBIOS 2.1, which does not carry the field.

`DMI_SYSTEM_WAKEUP_TYPE_OTHER`
: Other.

`DMI_SYSTEM_WAKEUP_TYPE_UNKNOWN`
: Unknown.

`DMI_SYSTEM_WAKEUP_TYPE_APM_TIMER`
: APM timer.

`DMI_SYSTEM_WAKEUP_TYPE_MODEM_RING`
: Ring of a modem.

`DMI_SYSTEM_WAKEUP_TYPE_LAN_REMOTE`
: Remote wake-up over the local network.

`DMI_SYSTEM_WAKEUP_TYPE_POWER_SWITCH`
: Power switch.

`DMI_SYSTEM_WAKEUP_TYPE_PCI_PME`
: PCI power management event (PME#).

`DMI_SYSTEM_WAKEUP_TYPE_POWER_RESTORE`
: Return of AC power.

Return value

`dmi_system_wakeup_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is in English, or translated to the locale set with `dmi_set_locale` when the library is built with localization support and has a translation of it. The string must not be modified or freed; a translated one stays valid until the locale is changed.

Errors

None.

See also

`dmi_system`

Standards

DMTF SMBIOS Specification (DSP0134), version 2.1 or newer.

<opendmi/entity/tpm-device.h>

dmi_tpm_device

Synopsis

```
#include <opendmi/entity/tpm-device.h>

typedef struct dmi_tpm_device {
    dmi_binary_t                       vendor_raw;
    char                              *vendor;
    dmi_version_t                      specification_version;
    uint64_t                           firmware_version;
    dmi_tpm_firmware_version_format_t firmware_version_format;
    dmi_version_t                      firmware_revision;
    dmi_tpm_firmware_version_t         firmware_version_2;
    const char                        *description;
    dmi_tpm_device_chars_t             characteristics;
    uint32_t                           oem_defined;
} dmi_tpm_device_t;
```

Description

The `dmi_tpm_device_t` structure holds the decoded TPM device structure (type 43) of SMBIOS: it describes a Trusted Platform Module (TPM) of the system, with its vendor, the version of the TPM specification it implements and the version of its firmware.

The firmware version is kept as stored in `firmware_version`, and is also parsed according to the major version of the TPM specification; `firmware_version_format` tells which field holds the parsed version.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(tpm_device)`, and lives as long as the context it has been decoded in.

Fields

`vendor_raw`

Vendor identifier as the structure holds it: four bytes of text, which are not terminated, and which some firmware stores in reverse order.

`vendor`

Vendor identifier, as registered by the Trusted Computing Group (TCG): up to four characters, or `nullptr` if the identifier holds no printable character. Some firmware stores the four bytes in reverse order; the decoder puts them back in order, and cuts the identifier at the first character which is not printable. The string belongs to the structure, and is freed along with it.

`specification_version`

Version of the TPM specification the device supports, such as 1.2 or 2.0; use `dmi_version_major` and `dmi_version_minor` to read it.

`firmware_version`

Vendor-specific firmware version, as stored: the first double word of the structure (firmware version 1) in the upper 32 bits, the second one (firmware version 2) in the lower 32 bits.

`firmware_version_format`

Format of the firmware version, chosen by the major version of the TPM specification, see `dmi_tpm_firmware_version_format`. It tells whether `firmware_revision` or `firmware_version_2` holds the parsed version.

`firmware_revision`

Firmware revision of a TPM 1.x device, made of the revMajor and revMinor bytes of its TCPA_VERSION structure. Only meaningful when `firmware_version_format` is `DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_1`, 0 otherwise.

`firmware_version_2`

Firmware version of a TPM 2.0 device, see `dmi_tpm_firmware_version`. Only meaningful when `firmware_version_format` is `DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_2`, zeroed otherwise.

`description`

Descriptive information about the TPM device, `nullptr` if the structure has none.

`characteristics`

Characteristics of the TPM device, see `dmi_tpm_device_chars`.

`oem_defined`

Information specific to the OEM or the firmware vendor.

Examples

```
dmi_registry_t   *registry = dmi_get_registry(context);
dmi_entity_t     *entity   = dmi_registry_lookup_first(registry,
                                 DMI_TYPE(tpm_device), false);
dmi_tpm_device_t *tpm      = dmi_entity_info(entity, DMI_TYPE(tpm_device));

if (tpm != nullptr) {
    printf("TPM %u.%u by %s\n", dmi_version_major(tpm->specification_version),
           dmi_version_minor(tpm->specification_version),
           tpm->vendor != nullptr ? tpm->vendor : "unknown vendor");

    if (tpm->firmware_version_format ==
        DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_2)
        printf("Firmware %u.%u\n", (unsigned)tpm->firmware_version_2.major,
               (unsigned)tpm->firmware_version_2.minor);
}
```

See also

`dmi_entity_info`, `dmi_tpm_firmware_version_format`, `dmi_tpm_firmware_version`, `dmi_tpm_device_chars`, `dmi_version_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.1 or newer.

dmi_tpm_device_chars

Synopsis

```
#include <opendmi/entity/tpm-device.h>

typedef union dmi_tpm_device_chars {
    uint64_t __value;

    struct {
        uint64_t reserved_1                 : 2;
        uint64_t is_unsupported             : 1;
        uint64_t is_update_configurable     : 1;
        uint64_t is_software_configurable   : 1;
        uint64_t is_proprietary_configurable : 1;
        uint64_t reserved_2                 : 58;
    };
} dmi_tpm_device_chars_t;
```

Description

The `dmi_tpm_device_chars_t` union holds the characteristics of a TPM device, and is the type of the `characteristics` field of `dmi_tpm_device_t`. The value can be read as a whole through `__value`, or bit by bit through the named members. The configurable characteristics tell how the TPM family of the device, such as TPM 1.2 or TPM 2.0, can be switched.

Fields

`__value`
: Raw value of the characteristics.

`reserved_1`
: Reserved bits.

`is_unsupported`
: Set if the device does not report its characteristics.

`is_update_configurable`
: Set if the TPM family can be changed by a firmware update.

`is_software_configurable`
: Set if the TPM family can be changed through platform software, such as the firmware setup.

`is_proprietary_configurable`

Set if the TPM family can be changed through a mechanism of the OEM.

`reserved_2`

Reserved bits.

See also

`dmi_tpm_device`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.1 or newer.

dmi_tpm_firmware_version

Synopsis

```
#include <opendmi/entity/tpm-device.h>

typedef struct dmi_tpm_firmware_version {
    uint16_t major;
    uint16_t minor;
    uint32_t vendor_specific;
} dmi_tpm_firmware_version_t;
```

Description

The `dmi_tpm_firmware_version_t` structure holds the firmware version of a TPM 2.0 device, parsed from the TPM_PT_FIRMWARE_VERSION_1 and TPM_PT_FIRMWARE_VERSION_2 properties the structure carries. It is the type of the `firmware_version_2` field of `dmi_tpm_device_t`, which is only meaningful when the `firmware_version_format` field of the device is `DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_2`.

Fields

`major`

Major version number, the high word of TPM_PT_FIRMWARE_VERSION_1.

`minor`

Minor version number, the low word of TPM_PT_FIRMWARE_VERSION_1.

`vendor_specific`

Vendor-specific part of the version, the value of TPM_PT_FIRMWARE_VERSION_2.

See also

`dmi_tpm_device`, `dmi_tpm_firmware_version_format`

Standards

DMTF SMBIOS Specification (DSP0134), version 3.1 or newer.

dmi_tpm_firmware_version_format

Synopsis

```
#include <opendmi/entity/tpm-device.h>

typedef enum dmi_tpm_firmware_version_format {
    DMI_TPM_FIRMWARE_VERSION_FORMAT_RAW,
    DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_1,
    DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_2
} dmi_tpm_firmware_version_format_t;
```

Description

The `dmi_tpm_firmware_version_format_t` enumeration tells how the firmware version of a TPM device has been parsed, which depends on the major version of the TPM specification the device supports. It is the type of the `firmware_version_format` field of `dmi_tpm_device_t`. The values are the library's own, and do not appear in the SMBIOS data.

Values

`DMI_TPM_FIRMWARE_VERSION_FORMAT_RAW`

Unknown TPM version: the firmware version is only available as stored, in the `firmware_version` field.

`DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_1`

TPM 1.x: the `firmware_revision` field holds the revision taken from the TCPA_VERSION structure.

`DMI_TPM_FIRMWARE_VERSION_FORMAT_TPM_2`

TPM 2.0: the `firmware_version_2` field holds the version taken from the TPM_PT_FIRMWARE_VERSION_1 and TPM_PT_FIRMWARE_VERSION_2 properties.

See also

`dmi_tpm_device`, `dmi_tpm_firmware_version`

Standards

Not part of the SMBIOS specification: the values are defined by OpenDMI to describe the TPM device structure (type 43) of the DMTF SMBIOS Specification (DSP0134), version 3.1 or newer.

<opendmi/entity/intel/asf.h>

dmi_intel_asf

Synopsis

```
#include <opendmi/entity/intel/asf.h>

typedef struct dmi_intel_asf {
    uint8_t version;
    const char *name;
    const char *identifier;
    uint8_t parameter;
    uint8_t extra[8];
    bool has_extra;
} dmi_intel_asf_t;
```

Description

The `dmi_intel_asf_t` structure holds the decoded Intel ASF information, which names the Alert Standard Format (ASF) support of the platform. The meaning of the fields other than the strings is not established.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_asf_spec`, of type `DMI_TYPE(intel_asf)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled, which is done for any platform with Intel processors when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structures are told by the version, `1`, and by the number of the first string, `1`, at offset 0x04, since other vendors give type 129 to structures of their own, e.g. Gigabyte. The strings are the ones of the Intel reference code, e.g. `Intel_ASF` and `Intel_ASF_001`, or of the vendor of the firmware, e.g. `Dell_ASF_001` or `Insyde_ASF_001`.

Fields

`version`

Version of the structure, presumably, 1 in all known data.

`name`

Name of the ASF support, e.g. `Intel_ASF` or `Insyde_ASF_001`.

`identifier`

Identifier of the ASF support, e.g. `Intel_ASF_001` or `Dell_ASF_002`.

`parameter`

Value whose meaning is not established, 1 in most known data and 0 on ThinkPad T61p.

`extra`

Bytes which follow in the structures of 16 bytes, whose meaning is not established, e.g. on Acer Aspire 3680. Not shown when `has_extra` is not set.

`has_extra`

Whether the structure holds `extra`, which the structures of 8 bytes leave out.

See also

`dmi_entity_info`, `dmi_intel_amt`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, and reverse engineered from the data corpus.

<opendmi/entity/intel/amt.h>

dmi_intel_amt

Synopsis

```
#include <opendmi/entity/intel/amt.h>

typedef struct dmi_intel_amt {
    dmi_binary_t signature;
    bool is_supported;
    bool is_enabled;
    bool is_ider_enabled;
    bool is_sol_enabled;
    bool is_network_enabled;
    uint8_t extended_data;
    dmi_intel_amt_oem_caps_1_t oem_capabilities_1;
    dmi_intel_amt_terminal_t terminal;
    dmi_intel_amt_oem_caps_3_t oem_capabilities_3;
    dmi_intel_amt_oem_caps_4_t oem_capabilities_4;
    bool is_kvm_enabled;
} dmi_intel_amt_t;
```

Description

The `dmi_intel_amt_t` structure holds the decoded Intel Active Management Technology (AMT) information, which tells whether the platform supports AMT, and which of its features are enabled.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_amt_spec`, of type `DMI_TYPE(intel_amt)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled, which is done for any platform with Intel processors when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in. The structures are told by the `$AMT` signature at offset 0x04.

The fields are the ones of the AMT information table of the Intel AMT implementation and reference guide. The data of the corpus agrees with them, e.g. the features are enabled on the platforms whose Management Engine firmware is of the corporate SKU, see `dmi_intel_me_sku`, while the flags are not consistent with each other on every platform: some give the network as enabled while AMT is not supported. The guide gives the values from 2 on as not applicable, which are taken as set.

Fields

`signature`
: Signature of the structure, `$AMT`.

`is_supported`
: Whether the platform supports AMT.

`is_enabled`
: Whether AMT is enabled.

`is_ider_enabled`
: Whether storage redirection is enabled: IDE redirection (IDE-R), or USB redirection (USB-R) from AMT 11.0 onwards.

`is_sol_enabled`
: Whether Serial over LAN (SOL) is enabled.

`is_network_enabled`
: Whether the network interface of AMT is enabled.

`extended_data`
: Extended data, which Intel leaves to the platform vendor, `0xA5` in all known data.

`oem_capabilities_1`
: Features of AMT the platform supports, see `dmi_intel_amt_oem_caps_1`.

`terminal`
: Terminal emulation of Serial over LAN, from the low nibble of the second byte of the OEM capabilities, see `dmi_intel_amt_terminal`.

`oem_capabilities_3`
: Security capabilities of the platform, see `dmi_intel_amt_oem_caps_3`.

`oem_capabilities_4`
: Targets of the remote boot the platform supports, see `dmi_intel_amt_oem_caps_4`.

`is_kvm_enabled`
: Whether keyboard, video and mouse (KVM) redirection is enabled.

See also

`dmi_entity_info`, `dmi_intel_amt_oem_caps_1`, `dmi_intel_amt_oem_caps_3`, `dmi_intel_amt_oem_caps_4`, `dmi_intel_amt_terminal`, `dmi_intel_vpro`, `dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the AMT information table of the Intel AMT implementation and reference guide.

dmi_intel_amt_oem_caps_1

Synopsis

```
#include <opendmi/entity/intel/amt.h>

typedef union dmi_intel_amt_oem_caps_1 {
    dmi_byte_t __value;
```

```
    struct {
        dmi_byte_t is_storage_redirection_supported : 1;
        dmi_byte_t is_sol_supported                 : 1;
        dmi_byte_t is_bios_reflash_supported        : 1;
        dmi_byte_t is_bios_setup_supported          : 1;
        dmi_byte_t is_bios_pause_supported          : 1;
        dmi_byte_t is_floppy_boot_blockable         : 1;
        dmi_byte_t is_cd_boot_blockable             : 1;
        dmi_byte_t __reserved                       : 1;
    };
} dmi_intel_amt_oem_caps_1_t;
```

Description

The `dmi_intel_amt_oem_caps_1_t` union holds the first byte of the OEM capabilities of Intel AMT information: the features of AMT the platform supports. It is the type of the `oem_capabilities_1` field of `dmi_intel_amt_t`.

The union is packed into a single byte: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the byte the structure carries them in, `is_storage_redirection_supported` being bit 0.

`is_storage_redirection_supported`
: Set if storage redirection, IDE-R or USB-R, is supported.

`is_sol_supported`
: Set if Serial over LAN is supported.

`is_bios_reflash_supported`
: Set if the BIOS can be reflashed remotely.

`is_bios_setup_supported`
: Set if the setup of the BIOS can be entered remotely.

`is_bios_pause_supported`
: Set if the BIOS can be paused remotely.

`is_floppy_boot_blockable`
: Set if booting from the floppy disk can be prevented.

`is_cd_boot_blockable`
: Set if booting from the CD can be prevented.

`__reserved`
: Reserved bit, which most firmware sets nevertheless.

See also

`dmi_intel_amt`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the AMT information table of the Intel AMT implementation and reference guide.

dmi_intel_amt_oem_caps_3

Synopsis

```
#include <opendmi/entity/intel/amt.h>

typedef union dmi_intel_amt_oem_caps_3 {
    dmi_byte_t __value;
    struct {
        dmi_byte_t __reserved                : 6;
        dmi_byte_t is_secure_erase_supported : 1;
        dmi_byte_t is_secure_boot_supported  : 1;
    };
} dmi_intel_amt_oem_caps_3_t;
```

Description

The `dmi_intel_amt_oem_caps_3_t` union holds the third byte of the OEM capabilities of Intel AMT information: the security capabilities of the platform. It is the type of the `oem_capabilities_3` field of `dmi_intel_amt_t`.

The union is packed into a single byte: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the byte the structure carries them in.

`__reserved`
: Reserved bits, zero.

`is_secure_erase_supported`
: Set if Remote Secure Erase is supported.

`is_secure_boot_supported`
: Set if the BIOS supports Secure Boot.

See also

`dmi_intel_amt`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the AMT information table of the Intel AMT implementation and reference guide.

dmi_intel_amt_oem_caps_4

Synopsis

```
#include <opendmi/entity/intel/amt.h>

typedef union dmi_intel_amt_oem_caps_4 {
    dmi_byte_t __value;
    struct {
        dmi_byte_t is_thunderbolt_dock_supported : 1;
        dmi_byte_t is_https_boot_supported       : 1;
        dmi_byte_t is_pba_boot_supported         : 1;
        dmi_byte_t is_winre_boot_supported       : 1;
        dmi_byte_t is_nonsecure_boot_supported   : 1;
        dmi_byte_t is_wifi_coexistence_supported : 1;
        dmi_byte_t __reserved                    : 2;
    };
} dmi_intel_amt_oem_caps_4_t;
```

Description

The `dmi_intel_amt_oem_caps_4_t` union holds the fourth byte of the OEM capabilities of Intel AMT information: the targets AMT can boot the platform to remotely. It is the type of the `oem_capabilities_4` field of `dmi_intel_amt_t`.

The union is packed into a single byte: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the byte the structure carries them in, `is_thunderbolt_dock_supported` being bit 0.

`is_thunderbolt_dock_supported`
: Set if a Thunderbolt dock is supported.

`is_https_boot_supported`
: Set if booting over HTTPS is supported.

`is_pba_boot_supported`
: Set if booting to the local pre-boot authentication (PBA) is supported.

`is_winre_boot_supported`
: Set if booting to the Windows recovery environment (WinRE) is supported.

`is_nonsecure_boot_supported`
: Set if booting without Secure Boot is supported.

`is_wifi_coexistence_supported`
: Set if the coexistence with Wi-Fi is supported.

`__reserved`
: Reserved bits, zero.

See also

`dmi_intel_amt`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the AMT information table of the Intel AMT implementation and reference guide.

dmi_intel_amt_terminal

Synopsis

```
#include <opendmi/entity/intel/amt.h>

typedef enum dmi_intel_amt_terminal {
    DMI_INTEL_AMT_TERMINAL_UNSPEC     = 0x00,
    DMI_INTEL_AMT_TERMINAL_VT52       = 0x01,
    DMI_INTEL_AMT_TERMINAL_VT100_PLUS = 0x02,
    DMI_INTEL_AMT_TERMINAL_VT_UTF8    = 0x03,
    DMI_INTEL_AMT_TERMINAL_PC_ANSI    = 0x04
} dmi_intel_amt_terminal_t;

const char *dmi_intel_amt_terminal_name(dmi_intel_amt_terminal_t value);
```

Description

The `dmi_intel_amt_terminal_t` enumeration tells the terminal emulation of Serial over LAN, as the low nibble of the second byte of the OEM capabilities of Intel AMT information tells it, and is the type of the `terminal` field of `dmi_intel_amt_t`. Other values are reserved, and are kept as they are.

The `dmi_intel_amt_terminal_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_AMT_TERMINAL_UNSPEC`
: Terminal emulation is not given.

`DMI_INTEL_AMT_TERMINAL_VT52`
: VT52.

`DMI_INTEL_AMT_TERMINAL_VT100_PLUS`
: VT100+, which most platforms give.

`DMI_INTEL_AMT_TERMINAL_VT_UTF8`
: VT-UTF8.

`DMI_INTEL_AMT_TERMINAL_PC_ANSI`
: PC-ANSI.

Return value

`dmi_intel_amt_terminal_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_amt`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the AMT information table of the Intel AMT implementation and reference guide.

<opendmi/entity/intel/fvi.h>

dmi_intel_fvi

Synopsis

```
#include <opendmi/entity/intel/fvi.h>

typedef struct dmi_intel_fvi {
    size_t item_count;
    dmi_intel_fvi_item_t *items;
} dmi_intel_fvi_t;
```

Description

The `dmi_intel_fvi_t` structure holds the decoded Intel firmware version information: the versions of the firmware components of a platform, as the Intel reference code knows them, e.g. the reference code modules, the microcode, the Management Engine firmware and the option ROMs.

Each module of the reference code adds a structure of its own, e.g. for the CPU, the Management Engine, the PCH or the System Agent, whose first item is the version of the reference code module itself, named `Reference Code - <module>`. The structures of a platform are often listed by a group association (type 14) named `Firmware Version Info`.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_fvi_spec`, of type `DMI_TYPE(intel_fvi)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled, which is done for any platform when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structures are found on Intel platforms whatever the vendor of the firmware is. The Intel reference code gives them type 221, but vendors who give that type to structures of their own place them elsewhere or not at all: Dell places them at type 205, or at type 237 on some platforms, e.g. Precision Tower 3620, and HP servers carry none. The type is checked against `DMI_TYPE(intel_fvi)` wherever the structure is placed, see `dmi_entity_info`.

Fields

`item_count`

Number of elements of `items`. Smaller than the number the structure declares when the data ends before the last item.

`items`

Versions of the firmware components, in the order of the structure, see `dmi_intel_fvi_item`. May be `nullptr` when `item_count` is 0.

Examples

```
dmi_registry_iter_t iter;
```

```
dmi_entity_t *entity;

dmi_registry_iter_initialize(&iter, dmi_get_registry(context), nullptr);

while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
    if (entity->spec != &dmi_intel_fvi_spec)
        continue;

    const dmi_intel_fvi_t *info = dmi_entity_info(entity, DMI_TYPE(intel_fvi));

    for (size_t i = 0; i < info->item_count; i++) {
        const dmi_intel_fvi_item_t *item = &info->items[i];

        printf("%s: %u.%u.%u.%u\n", item->component,
               item->major, item->minor, item->revision, item->build);
    }
}
```

See also

`dmi_entity_info`, `dmi_intel_fvi_item`, `dmi_get_platform`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, whose firmware version information module gives the structures type 221 by default.

dmi_intel_fvi_item

Synopsis

```
#include <opendmi/entity/intel/fvi.h>

typedef struct dmi_intel_fvi_item {
    const char *component;
    const char *version_string;
    uint8_t major;
    uint8_t minor;
    uint8_t revision;
    uint16_t build;
} dmi_intel_fvi_item_t;
```

Description

The `dmi_intel_fvi_item_t` structure describes the version of a firmware component, and is the type of the elements of the `items` array of `dmi_intel_fvi_t`.

Each part of the version is unspecified on its own. Components which carry a single number set that part only, e.g. the revision identifiers of the PCH and the System Agent (`PCH-CRID Original Value`) set the build number, and the HSIO versions set the major version. Components which carry a state rather than a version, e.g. `PCH-CRID Status`, leave the version unspecified, and give the state as the version string. Some components give their version as a string only, e.g. `FSP Binary Version` on some platforms.

The parts are written in decimal or in hexadecimal depending on the component: the versions of the Management Engine firmware are usually shown in decimal, e.g. 16.1.32.2473, while the versions of the reference code modules are shown in hexadecimal, e.g. 7.0.5C.50.

The revision of the microcode, `uCode Version`, is the revision followed by the low byte of the build number, `(revision << 8) | build`: e.g. revision 4 and build 55 stand for 0x437 of Alder Lake-P, and revision 0x41 and build 14 for 0x410E of Raptor Lake-P. The older platforms give it in the build number alone, e.g. build 234 for 0xEA.

Fields

`component`

Name of the component, e.g. `Reference Code - CPU`, `uCode Version` or `ME Firmware Version`.

`version_string`

Version of the component written as a string, or its state, e.g. `Consumer SKU` for the Management Engine firmware or `Disabled`. `nullptr` if the component has none.

`major`

Major version. `UINT8_MAX` means unspecified.

`minor`

Minor version. `UINT8_MAX` means unspecified.

`revision`

Revision. `UINT8_MAX` means unspecified.

`build`

Build number. `UINT16_MAX` means unspecified. Some items whose other parts are all unspecified give it as `0x00FF`, e.g. `UNDI Version` and `MEBx version` on some platforms, which is presumably a byte of all bits set, and is kept as it is.

See also

`dmi_intel_fvi`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, whose firmware version information module gives the structures type 221 by default.

<opendmi/entity/intel/mei.h>

dmi_intel_mei

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef struct dmi_intel_mei {
    uint8_t version;
    size_t device_count;
    dmi_intel_mei_device_t *devices;
    dmi_intel_me_state_t state;
    dmi_intel_me_mode_t mode;
    dmi_intel_me_error_t error_code;
    bool is_init_complete;
    bool is_manufacturing;
    dmi_intel_me_sku_t sku;
} dmi_intel_mei_t;
```

Description

The `dmi_intel_mei_t` structure holds the decoded Intel Management Engine interface information: the firmware status registers of the Host Embedded Controller Interfaces (HECI) of the Management Engine, as the firmware of the host has read them at boot.

The state of the Management Engine firmware is told from the first and the third registers of the first interface, which the host talks to the firmware through, and whose layout is the same in all generations of the Management Engine. The rest of the registers are kept as they are, since their layouts differ between generations, e.g. the sixth one tells the state of Intel Boot Guard on the generations which have it.

The structure is told by its version, 1, and by the name of the first interface, `MEI1`, which every known structure holds, so that the structures vendors give the same types are not taken for it.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_mei_spec`, of type `DMI_TYPE(intel_mei)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled, which is done for any platform with Intel processors when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The Intel reference code gives the structures type 219, but vendors who give that type to structures of their own place them elsewhere or not at all: Dell places them at type 203, or at type 235 on some platforms, e.g. Precision Tower 3620, and HP servers carry none. Most tables hold a group association (type 14) named `$MEI`, which is not a reliable way to find the structure: in about half of the tables of the data corpus the group gives handle 0, and in some it refers to other structures.

Fields

`version`

Version of the structure, 1 in all known data.

`device_count`

Number of elements of `devices`. Smaller than the number the structure declares when the data ends before the last interface.

`devices`

Interfaces, see `dmi_intel_mei_device`. May be `nullptr` when `device_count` is 0.

`state`

Working state of the firmware, see `dmi_intel_me_state`.

`mode`

Operation mode of the firmware, see `dmi_intel_me_mode`.

`error_code`

Error code of the firmware, see `dmi_intel_me_error`.

`is_init_complete`

Whether the firmware has completed its initialization.

`is_manufacturing`

Whether the firmware is in the manufacturing mode, in which the platform has not been locked down yet.

`sku`

SKU of the firmware, see `dmi_intel_me_sku`.

Examples

```
dmi_entity_t *entity = dmi_registry_lookup_first(dmi_get_registry(context),
                                                 DMI_TYPE(intel_mei), true);
const dmi_intel_mei_t *info = dmi_entity_info(entity, DMI_TYPE(intel_mei));

if ((info != nullptr) && (info->sku != DMI_INTEL_ME_SKU_UNSPEC))
    printf("ME firmware: %s, %s\n", dmi_intel_me_sku_name(info->sku),
           dmi_intel_me_state_name(info->state));
```

See also

`dmi_entity_info`, `dmi_intel_mei_device`, `dmi_intel_me_state`, `dmi_intel_me_error`, `dmi_intel_me_mode`, `dmi_intel_me_sku`, `dmi_intel_fvi`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine, as coreboot describes them.

dmi_intel_mei_device

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef struct dmi_intel_mei_device {
    const char *name;
    uint32_t hfsts[6];
    bool is_present;
    bool is_reported;
} dmi_intel_mei_device_t;
```

Description

The `dmi_intel_mei_device_t` structure describes a Host Embedded Controller Interface (HECI) of the Intel Management Engine, and is the type of the elements of the `devices` array of `dmi_intel_mei_t`.

The interfaces are the PCI functions 0:16.0, 0:16.1, 0:16.4 and 0:16.5, named `MEI1` to `MEI4`. The firmware status registers are the ones of the PCI configuration space of the function, at offsets 0x40, 0x48, 0x60, 0x64, 0x68 and 0x6C. An absent function reads with all bits set, and the registers of an interface which is not reported are all zeroes, e.g. the ones of `MEI4` on most platforms.

Fields

`name`

Name of the interface, e.g. `MEI1`.

`hfsts`

Firmware status registers HFSTS1 to HFSTS6. Their meaning depends on the interface, and on the generation of the Management Engine for all but the first and the third registers of the first interface: the first one tells the working state, the operation mode and the error code, and the third one the SKU. Not shown if the interface is not reported.

`is_present`

Whether the function is present, i.e. its registers do not have all bits set. Not set, and not shown, if the interface is not reported.

`is_reported`

Whether the interface is reported, i.e. its registers are not all zeroes.

See also

`dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine,

as coreboot describes them.

dmi_intel_me_state

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef enum dmi_intel_me_state {
    DMI_INTEL_ME_STATE_RESET       = 0x00,
    DMI_INTEL_ME_STATE_INIT        = 0x01,
    DMI_INTEL_ME_STATE_RECOVERY    = 0x02,
    DMI_INTEL_ME_STATE_TEST        = 0x03,
    DMI_INTEL_ME_STATE_M3_NO_UMA   = 0x04,
    DMI_INTEL_ME_STATE_NORMAL      = 0x05,
    DMI_INTEL_ME_STATE_WAIT        = 0x06,
    DMI_INTEL_ME_STATE_TRANSITION  = 0x07,
    DMI_INTEL_ME_STATE_INVALID_CPU = 0x08,
    DMI_INTEL_ME_STATE_UNSPEC      = 0xFF
} dmi_intel_me_state_t;

const char *dmi_intel_me_state_name(dmi_intel_me_state_t value);
```

Description

The `dmi_intel_me_state_t` enumeration tells the working state of the Intel Management Engine firmware, as bits 0 to 3 of the first firmware status register (HFSTS1) of the first interface tell it, and is the type of the `state` field of `dmi_intel_mei_t`. Other values may be found in the data, and are kept as they are.

The `dmi_intel_me_state_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_ME_STATE_RESET`
: Firmware is being reset.

`DMI_INTEL_ME_STATE_INIT`
: Firmware is initializing.

`DMI_INTEL_ME_STATE_RECOVERY`
: Firmware runs in the recovery mode, e.g. after its image has been found corrupted.

`DMI_INTEL_ME_STATE_TEST`
: Firmware runs its tests.

`DMI_INTEL_ME_STATE_M3_NO_UMA`
: Firmware runs in the M3 power state, without the memory of the host. coreboot names the value

disabled for the 6 series chipsets.

`DMI_INTEL_ME_STATE_NORMAL`

Firmware runs normally.

`DMI_INTEL_ME_STATE_WAIT`

Firmware waits for the platform to be disabled.

`DMI_INTEL_ME_STATE_TRANSITION`

Firmware changes its operation state.

`DMI_INTEL_ME_STATE_INVALID_CPU`

Processor does not match the firmware.

`DMI_INTEL_ME_STATE_UNSPEC`

State is unknown, since the first interface is absent or not reported.

Return value

`dmi_intel_me_state_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine, as coreboot describes them.

dmi_intel_me_error

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef enum dmi_intel_me_error {
    DMI_INTEL_ME_ERROR_NONE          = 0x00,
    DMI_INTEL_ME_ERROR_UNCATEGORIZED = 0x01,
    DMI_INTEL_ME_ERROR_IMAGE         = 0x03,
    DMI_INTEL_ME_ERROR_DEBUG         = 0x04,
    DMI_INTEL_ME_ERROR_UNSPEC        = 0xFF
```

```
} dmi_intel_me_error_t;

const char *dmi_intel_me_error_name(dmi_intel_me_error_t value);
```

Description

The `dmi_intel_me_error_t` enumeration tells the error code of the Intel Management Engine firmware, as bits 12 to 15 of the first firmware status register (HFSTS1) of the first interface tell it, and is the type of the `error_code` field of `dmi_intel_mei_t`. The values are the ones coreboot names. Other values may be found in the data, and are kept as they are.

The `dmi_intel_me_error_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_ME_ERROR_NONE`
: No error.

`DMI_INTEL_ME_ERROR_UNCATEGORIZED`
: Failure of no particular category.

`DMI_INTEL_ME_ERROR_IMAGE`
: Failure of the image of the firmware, e.g. a corrupted one.

`DMI_INTEL_ME_ERROR_DEBUG`
: Debug failure.

`DMI_INTEL_ME_ERROR_UNSPEC`
: Error code is unknown, since the first interface is absent or not reported.

Return value

`dmi_intel_me_error_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine, as coreboot describes them.

dmi_intel_me_mode

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef enum dmi_intel_me_mode {
    DMI_INTEL_ME_MODE_NORMAL           = 0x00,
    DMI_INTEL_ME_MODE_DEBUG            = 0x02,
    DMI_INTEL_ME_MODE_DISABLED         = 0x03,
    DMI_INTEL_ME_MODE_OVERRIDE_JUMPER  = 0x04,
    DMI_INTEL_ME_MODE_OVERRIDE_MESSAGE = 0x05,
    DMI_INTEL_ME_MODE_UNSPEC           = 0xFF
} dmi_intel_me_mode_t;

const char *dmi_intel_me_mode_name(dmi_intel_me_mode_t value);
```

Description

The `dmi_intel_me_mode_t` enumeration tells the operation mode of the Intel Management Engine firmware, as bits 16 to 19 of the first firmware status register (HFSTS1) of the first interface tell it, and is the type of the `mode` field of `dmi_intel_mei_t`. Other values may be found in the data, and are kept as they are.

The `dmi_intel_me_mode_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_ME_MODE_NORMAL`
: Firmware runs normally.

`DMI_INTEL_ME_MODE_DEBUG`
: Firmware runs in the debug mode.

`DMI_INTEL_ME_MODE_DISABLED`
: Firmware has been disabled temporarily, e.g. by the firmware of the host.

`DMI_INTEL_ME_MODE_OVERRIDE_JUMPER`
: Security of the firmware is overridden by a jumper, which allows it to be rewritten.

`DMI_INTEL_ME_MODE_OVERRIDE_MESSAGE`
: Security of the firmware is overridden by a message from the host, which allows it to be rewritten.

`DMI_INTEL_ME_MODE_UNSPEC`
: Mode is unknown, since the first interface is absent.

Return value

`dmi_intel_me_mode_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine, as coreboot describes them.

dmi_intel_me_sku

Synopsis

```
#include <opendmi/entity/intel/mei.h>

typedef enum dmi_intel_me_sku {
    DMI_INTEL_ME_SKU_CONSUMER  = 0x02,
    DMI_INTEL_ME_SKU_CORPORATE = 0x03,
    DMI_INTEL_ME_SKU_LITE      = 0x05,
    DMI_INTEL_ME_SKU_UNSPEC    = 0xFF
} dmi_intel_me_sku_t;

const char *dmi_intel_me_sku_name(dmi_intel_me_sku_t value);
```

Description

The `dmi_intel_me_sku_t` enumeration tells the SKU of the Intel Management Engine firmware, as bits 4 to 6 of the third firmware status register (HFSTS3) of the first interface tell it, and is the type of the `sku` field of `dmi_intel_mei_t`. The SKU matches the version string of the `ME Firmware Version` component of Intel firmware version information, see `dmi_intel_fvi_item`. Other values may be found in the data, and are kept as they are.

The `dmi_intel_me_sku_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_ME_SKU_CONSUMER`
: Consumer firmware.

`DMI_INTEL_ME_SKU_CORPORATE`

Corporate firmware, which the platforms of Intel vPro have, with Intel Active Management Technology.

`DMI_INTEL_ME_SKU_LITE`

Lite firmware, e.g. of the Chromebooks.

`DMI_INTEL_ME_SKU_UNSPEC`

SKU is unknown, since the first interface is absent, or since the register holds zero for it, as the firmware of Management Engine 11.0 may.

Return value

`dmi_intel_me_sku_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code, which gives the structures type 219 by default. The layouts of the firmware status registers are the ones of the Intel Management Engine, as coreboot describes them.

<opendmi/entity/intel/platform.h>

dmi_intel_platform

Synopsis

```
#include <opendmi/entity/intel/platform.h>

typedef struct dmi_intel_platform {
    const char *gop_version;
    const char *microcode_version;
    const char *mrc_version;
    const char *sec_version;
    const char *ulpmc_version;
    const char *pmc_version;
    const char *punit_version;
    const char *soc_version;
    const char *board_version;
    const char *fab_version;
    const char *cpu_flavor;
    const char *bios_version;
    const char *pmic_version;
    const char *touch_version;
    const char *secure_boot;
    const char *boot_mode;
    const char *speedstep_mode;
    const char *turbo_mode;
    const char *max_cstate;
    const char *gfx_turbo;
    const char *idle_reserve;
    const char *rc6;
} dmi_intel_platform_t;
```

Description

The `dmi_intel_platform_t` structure holds the decoded Intel platform information, which gives the versions of the firmware components and some settings of an Intel Bay Trail platform, all of them as strings, as the firmware writes them: the numbers in hexadecimal, and the settings as their values, e.g. `0` or `1`.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_platform_spec`, of type `DMI_TYPE(intel_platform)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled, which is done for any platform with Intel processors when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structures are told by their length of 26 bytes, since Intel server boards give type 148 to structures of

other layouts.

Fields

`gop_version`
Version of the graphics output protocol (GOP) driver, e.g. `7.2.1013`.

`microcode_version`
Revision of the microcode in hexadecimal, e.g. `832`.

`mrc_version`
Version of the memory reference code (MRC), e.g. `1.02`.

`sec_version`
Version of the firmware of the security engine, the Trusted Execution Engine (TXE), e.g. `1.2.0.1149`.

`ulpmc_version`
Version of the firmware of the ultra low power microcontroller (ULPMC), or `Non ULPMC!!` if there is none.

`pmc_version`
Version of the firmware of the power management controller (PMC), e.g. `0x4_45`.

`punit_version`
Version of the firmware of the P-unit, e.g. `0x27`.

`soc_version`
Stepping of the system on a chip, e.g. `0F (C0 Stepping)`.

`board_version`
Board, e.g. `BAY LAKE CR (6)`.

`fab_version`
Fab of the board, e.g. `0`.

`cpu_flavor`
Flavor of the processor, e.g. `VLV-QC Notebook (3)`.

`bios_version`
Version of the firmware, e.g. `E2CN13WW`.

`pmic_version`
Vendor and revision of the power management IC (PMIC), e.g. `41.01`.

`touch_version`
Version of the firmware of the touch screen, `NA` if there is none.

`secure_boot`
Secure Boot setting, `0` or `1`.

`boot_mode`
: Boot mode of the firmware, as a number.

`speedstep_mode`
: Enhanced Intel SpeedStep setting.

`turbo_mode`
: Turbo Boost setting.

`max_cstate`
: Deepest C-state of the processor, e.g. `3`.

`gfx_turbo`
: Turbo setting of the graphics.

`idle_reserve`
: S0ix setting.

`rc6`
: RC6 power saving setting of the graphics.

See also

`dmi_entity_info`, `dmi_intel_fvi`

Standards

Not part of the SMBIOS specification. Defined by the Intel Bay Trail reference code, as the OEM type 0x94 of `Vlv2TbltDevicePkg` of EDK II.

<opendmi/entity/intel/svt.h>

dmi_intel_svt

Synopsis

```
#include <opendmi/entity/intel/svt.h>

typedef struct dmi_intel_svt {
    uint8_t version;
    uint16_t parameter;
    size_t milestone_count;
    dmi_intel_svt_milestone_t *milestones;
} dmi_intel_svt_t;
```

Description

The `dmi_intel_svt_t` structure holds the decoded Intel Silicon View Technology (SVT) milestones: the milestones of the boot the firmware reports to the debug and validation tools of Intel.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_svt_spec` or `dmi_intel_svt_aligned_spec`, of type `DMI_TYPE(intel_svt)`, declared in `<opendmi/module/intel.h>`, once the Intel extension module (`dmi_intel_module`) is enabled. It lives as long as the context it has been decoded in.

The Intel reference code gives the structures type 222, but vendors who give that type to structures of their own place them elsewhere or not at all: Dell places them at type 206, and HP servers carry none. A group association (type 14) named `Intel® Silicon View Technology` often refers to the structure.

Some firmware does not pack the structure, e.g. the one of Dell XPS 13 9365: the parameter is aligned to two bytes, after a byte of zero which follows the version, and the milestones are followed by a byte which makes the length of the structure even. Such structures are decoded by `dmi_intel_svt_aligned_spec` into the same structure.

Both layouts are told by the version and the parameter, 1 and `0x0099`, which every known structure holds, so that the structures vendors give type 222 are not taken for the milestones on the systems whose firmware names no vendor.

Fields

`version`

Version of the structure, 1 in all known data.

`parameter`

Value whose meaning is not established, `0x0099` in all known data.

`milestone_count`

Number of elements of `milestones`.

`milestones`
: Milestones, see `dmi_intel_svt_milestone`. May be `nullptr` when `milestone_count` is 0.

See also

`dmi_entity_info`, `dmi_intel_svt_milestone`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code.

dmi_intel_svt_milestone

Synopsis

```
#include <opendmi/entity/intel/svt.h>

typedef struct dmi_intel_svt_milestone {
    uint8_t code;
    const char *name;
} dmi_intel_svt_milestone_t;
```

Description

The `dmi_intel_svt_milestone_t` structure describes a milestone of the boot, and is the type of the elements of the `milestones` array of `dmi_intel_svt_t`. All known data lists the same milestones: `0x10` for `Memory Init Complete`, `0x20` for `End of DXE Phase` and `0x30` for `BIOS Boot Complete`.

Fields

`code`
: Code of the milestone.

`name`
: Name of the milestone.

See also

`dmi_intel_svt`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code.

<opendmi/entity/intel/vpro.h>

dmi_intel_vpro

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef struct dmi_intel_vpro {
    dmi_intel_vpro_cpu_caps_t cpu_capabilities;
    dmi_intel_vpro_version_t mebx_version;
    uint8_t lpc_devfn;
    uint8_t lpc_bus;
    uint16_t lpc_device_id;
    dmi_intel_vpro_me_caps_t me_capabilities;
    dmi_intel_vpro_version_t me_version;
    dmi_intel_vpro_tpm_caps_t tpm_capabilities;
    uint8_t gbe_devfn;
    uint8_t gbe_bus;
    uint16_t gbe_device_id;
    uint8_t wlan_devfn;
    uint8_t wlan_bus;
    uint16_t wlan_device_id;
    dmi_intel_vpro_bios_caps_t bios_capabilities;
    dmi_binary_t signature;
    uint8_t tcg_major;
    uint8_t tcg_minor;
    uint8_t va_version;
    bool has_wlan;
    bool has_mch_capabilities;
    uint8_t mch_devfn;
    uint8_t mch_bus;
    uint16_t mch_device_id;
    dmi_intel_vpro_mch_caps_t mch_capabilities;
} dmi_intel_vpro_t;
```

Description

The `dmi_intel_vpro_t` structure holds the decoded Intel vPro information, which Intel calls the vPro verification table. It describes the parts of the platform Intel vPro technology relies on: the capabilities of the processor, of the Management Engine firmware, of the TPM and of the BIOS, the versions of the firmware and of its BIOS extension (MEBx), and the PCI functions of the chipset and of the network controller the Management Engine uses. The structure is found on Intel platforms whether they are of vPro or not.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_intel_vpro_spec`, of type `DMI_TYPE(intel_vpro)`, declared in `<opendmi/module/intel.h>`, once the Intel extension

module (`dmi_intel_module`) is enabled, which is done for any platform with Intel processors when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structures are told by the `vPro` signature at offset 0x38, since some vendors give type 131 to structures of their own in the same table, e.g. Lenovo ThinkVantage Technologies enablement, see `dmi_lenovo_tvt`. The structures of type 131 without the signature are not decoded by this specification.

PCI functions are given as their bus, and their device and function numbers, `(device << 3) | function`, e.g. `0xF8` for 0:1F.0 and `0xFE` for 0:1F.6.

The layout Intel documents holds the capabilities of the memory controller hub where the newer firmware puts the version of the BIOS extension, e.g. the one of HP EliteBook 6930p. Such structures are told by the major version of zero next to a minor version which is not, since the hub is found at 0:0.0 and has a device ID, while a version of the BIOS extension is either given in full or not at all. The fields of the hub are taken from the words of the version then, and are shown in place of it.

Fields

`cpu_capabilities`
: Capabilities of the processor, see `dmi_intel_vpro_cpu_caps`.

`mebx_version`
: Version of the Management Engine BIOS extension, see `dmi_intel_vpro_version`. All zeroes if not reported. Not shown when `has_mch_capabilities` is set.

`lpc_devfn`, **`lpc_bus`**, **`lpc_device_id`**
: PCI device and function of the LPC bridge of the chipset, its bus, and its device ID, which tells the chipset, e.g. `0xA3C8` for Intel B460.

`me_capabilities`
: Capabilities of the Management Engine firmware, see `dmi_intel_vpro_me_caps`.

`me_version`
: Version of the Management Engine firmware, which matches the one of Intel firmware version information, see `dmi_intel_fvi_item`. All zeroes if not reported.

`tpm_capabilities`
: Capabilities of the Trusted Platform Module, see `dmi_intel_vpro_tpm_caps`.

`gbe_devfn`, **`gbe_bus`**, **`gbe_device_id`**
: PCI device and function of the wired network controller the Management Engine uses, its bus, and its device ID, `0xFFFF` or zero if the controller is absent.

`wlan_devfn`, **`wlan_bus`**, **`wlan_device_id`**
: PCI device and function of the wireless network controller, its bus, and its device ID, e.g. `0x088F` for Intel Centrino Advanced-N 6235. The firmware of HP laptops gives the host bridge in its place, e.g. `0x0A04` for Intel Haswell-ULT, while most firmware gives no controller, and holds `0xFF00` or `0x00FF` for the device ID. Not shown when `has_wlan` is not set.

`bios_capabilities`
: Capabilities of the BIOS, see `dmi_intel_vpro_bios_caps`.

`signature`

Signature of the structure, `vPro`.

`tcg_major`, **`tcg_minor`**

Version of the TCG specification the TPM is designed to, taken from `tpm_capabilities`. Zero if not given.

`va_version`

Highest version of the Virtual Appliance the BIOS supports, taken from `bios_capabilities`: 0 for 2.6, 1 for 3.0.

`has_wlan`

Whether the structure gives the wireless network controller, or the host bridge in its place, which it does unless the device ID is zero, `0x00FF`, `0xFF00` or `0xFFFF`.

`has_mch_capabilities`

Whether the structure is of the older layout, which holds the capabilities of the memory controller hub in place of the version of the BIOS extension.

`mch_devfn`, **`mch_bus`**, **`mch_device_id`**

PCI device and function of the memory controller hub, its bus, and its device ID, e.g. `0x2A40` for Intel GM45. Older layout only.

`mch_capabilities`

Capabilities of the memory controller hub, see `dmi_intel_vpro_mch_caps`. Older layout only.

See also

`dmi_entity_info`, `dmi_intel_vpro_version`, `dmi_intel_vpro_cpu_caps`, `dmi_intel_vpro_mch_caps`, `dmi_intel_vpro_me_caps`, `dmi_intel_vpro_tpm_caps`, `dmi_intel_vpro_bios_caps`, `dmi_intel_mei`, `dmi_intel_fvi`, `dmi_lenovo_tvt`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT implementation and reference guide.

dmi_intel_vpro_version

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef struct dmi_intel_vpro_version {
    uint16_t major;
    uint16_t minor;
    uint16_t hotfix;
    uint16_t build;
} dmi_intel_vpro_version_t;
```

Description

The `dmi_intel_vpro_version_t` structure holds the version of a firmware component, as Intel vPro information gives it, see `dmi_intel_vpro`. The version of the Management Engine firmware is written as `major.minor.hotfix.build`, e.g. 14.5.12.1111.

The structure holds the parts of the version in the order of their significance, whatever the order the data holds them in: the version of the BIOS extension is laid out from the major part down, while the one of the firmware is laid out as the firmware reports it, minor, major, build and hotfix.

Fields

`major`
Major version.

`minor`
Minor version.

`hotfix`
Hotfix number.

`build`
Build number.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by the Intel reference code.

dmi_intel_vpro_cpu_caps

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef union dmi_intel_vpro_cpu_caps {
    dmi_dword_t __value;
    struct {
        dmi_dword_t is_vmx_enabled : 1;
        dmi_dword_t is_smx_enabled : 1;
        dmi_dword_t is_txt_capable : 1;
        dmi_dword_t is_txt_enabled : 1;
        dmi_dword_t is_vtx_capable : 1;
        dmi_dword_t is_vtx_enabled : 1;
        dmi_dword_t __reserved     : 26;
    };
```

```
} dmi_intel_vpro_cpu_caps_t;
```

Description

The `dmi_intel_vpro_cpu_caps_t` union holds the capabilities of the processor, as Intel vPro information gives them. It is the type of the `cpu_capabilities` field of `dmi_intel_vpro_t`.

The union is packed into a double word: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the double word the structure carries them in, `is_vmx_enabled` being bit 0.

`is_vmx_enabled`
: Set if the virtual machine extensions (VMX) are enabled.

`is_smx_enabled`
: Set if the safer mode extensions (SMX) are enabled.

`is_txt_capable`
: Set if the processor supports Trusted Execution Technology (TXT).

`is_txt_enabled`
: Set if Trusted Execution Technology is enabled.

`is_vtx_capable`
: Set if the processor supports Virtualization Technology (VT-x).

`is_vtx_enabled`
: Set if Virtualization Technology is enabled.

`__reserved`
: Reserved bits, zero.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT implementation and reference guide.

dmi_intel_vpro_mch_caps

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef union dmi_intel_vpro_mch_caps {
    dmi_dword_t __value;
    struct {
        dmi_dword_t is_vtd_capable : 1;
        dmi_dword_t is_vtd_enabled : 1;
        dmi_dword_t is_txt_capable : 1;
        dmi_dword_t is_txt_enabled : 1;
        dmi_dword_t __reserved     : 28;
    };
} dmi_intel_vpro_mch_caps_t;
```

Description

The `dmi_intel_vpro_mch_caps_t` union holds the capabilities of the memory controller hub, which the older layout of Intel vPro information holds in place of the version of the BIOS extension. It is the type of the `mch_capabilities` field of `dmi_intel_vpro_t`.

The union is packed into a double word: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the double word at offset 0x0C of the structure, `is_vtd_capable` being bit 0.

`is_vtd_capable`
: Set if the chipset supports Virtualization Technology for Directed I/O (VT-d).

`is_vtd_enabled`
: Set if VT-d is enabled.

`is_txt_capable`
: Set if the chipset supports Trusted Execution Technology.

`is_txt_enabled`
: Set if Trusted Execution Technology is enabled.

`__reserved`
: Reserved bits, zero.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT

implementation and reference guide.

dmi_intel_vpro_me_caps

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef union dmi_intel_vpro_me_caps {
    dmi_dword_t __value;
    struct {
        dmi_dword_t is_me_enabled    : 1;
        dmi_dword_t is_qst_supported : 1;
        dmi_dword_t is_asf_supported : 1;
        dmi_dword_t is_amt_supported : 1;
        dmi_dword_t __reserved_1     : 1;
        dmi_dword_t is_sbt_supported : 1;
        dmi_dword_t is_l3_supported  : 1;
        dmi_dword_t __reserved_2     : 25;
    };
} dmi_intel_vpro_me_caps_t;
```

Description

The `dmi_intel_vpro_me_caps_t` union holds the capabilities of the Management Engine firmware, as Intel vPro information gives them. It is the type of the `me_capabilities` field of `dmi_intel_vpro_t`.

The union is packed into a double word: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the double word the structure carries them in, `is_me_enabled` being bit 0.

`is_me_enabled`
: Set if the Management Engine is enabled.

`is_qst_supported`
: Set if the firmware supports Quiet System Technology (QST).

`is_asf_supported`
: Set if the firmware supports Alert Standard Format (ASF).

`is_amt_supported`
: Set if the firmware supports Active Management Technology (AMT).

`__reserved_1`
: Reserved bit, zero.

`is_sbt_supported`
: Set if the firmware supports Small Business Technology (SBT).

`is_l3_supported`
: Set if the firmware supports the level III manageability.

`__reserved_2`
: Bits Intel reserves. The newer firmware sets some of them, e.g. bits 13 to 15, whose meaning is not established.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT implementation and reference guide.

dmi_intel_vpro_tpm_caps

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef union dmi_intel_vpro_tpm_caps {
    dmi_dword_t __value;
    struct {
        dmi_dword_t is_tpm_present : 1;
        dmi_dword_t is_tpm_enabled : 1;
        dmi_dword_t __reserved     : 14;
        dmi_dword_t tcg_major      : 8;
        dmi_dword_t tcg_minor      : 8;
    };
} dmi_intel_vpro_tpm_caps_t;
```

Description

The `dmi_intel_vpro_tpm_caps_t` union holds the capabilities of the Trusted Platform Module, as Intel vPro information gives them. It is the type of the `tpm_capabilities` field of `dmi_intel_vpro_t`.

The union is packed into a double word: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the double word the structure carries them in, `is_tpm_present` being bit 0.

`is_tpm_present`
: Set if a TPM is on board.

`is_tpm_enabled`
: Set if the TPM is enabled.

`__reserved`
: Reserved bits, zero.

`tcg_major`, `tcg_minor`
: Version of the TCG specification the TPM is designed to, e.g. 1 and 2 for TPM 1.2. Zero if not given. The structure holds them in the `tcg_major` and `tcg_minor` fields of `dmi_intel_vpro_t` as well.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT implementation and reference guide.

dmi_intel_vpro_bios_caps

Synopsis

```
#include <opendmi/entity/intel/vpro.h>

typedef union dmi_intel_vpro_bios_caps {
    dmi_dword_t __value;
    struct {
        dmi_dword_t is_vtx_configurable  : 1;
        dmi_dword_t is_vtd_configurable  : 1;
        dmi_dword_t is_txt_configurable  : 1;
        dmi_dword_t is_tpm_configurable  : 1;
        dmi_dword_t is_me_configurable   : 1;
        dmi_dword_t is_va_supported      : 1;
        dmi_dword_t is_spi_data_reserved : 1;
        dmi_dword_t va_version           : 3;
        dmi_dword_t __reserved           : 22;
    };
} dmi_intel_vpro_bios_caps_t;
```

Description

The `dmi_intel_vpro_bios_caps_t` union holds the capabilities of the BIOS, as Intel vPro information gives them. It is the type of the `bios_capabilities` field of `dmi_intel_vpro_t`.

The union is packed into a double word: the bits can be read either one by one, as bit fields, or all together, as `__value`.

Fields

`__value`
: All the bits, as the double word the structure carries them in, `is_vtx_configurable` being bit 0.

`is_vtx_configurable`, **`is_vtd_configurable`**, **`is_txt_configurable`**, **`is_tpm_configurable`**, **`is_me_configurable`**
: Set if VT-x, VT-d, Trusted Execution Technology, the TPM or the Management Engine can be set in the setup of the BIOS.

`is_va_supported`
: Set if the BIOS supports the extensions of the Virtual Appliance, the ACPI operation region.

`is_spi_data_reserved`
: Set if the platform data area of the SPI flash is reserved.

`va_version`
: Highest version of the Virtual Appliance the BIOS supports: 0 for 2.6, 1 for 3.0, the other values being reserved. The structure holds it in the `va_version` field of `dmi_intel_vpro_t` as well.

`__reserved`
: Reserved bits, zero.

See also

`dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Intel, as the vPro verification table of the Intel AMT implementation and reference guide.

<opendmi/entity/lenovo/oem.h>

dmi_lenovo_oem

Synopsis

```
#include <opendmi/entity/lenovo/oem.h>

typedef struct dmi_lenovo_oem {
    dmi_binary_t signature;
    uint8_t offset;
    uint8_t number;
    uint8_t revision;
    dmi_binary_t data;
} dmi_lenovo_oem_t;
```

Description

The `dmi_lenovo_oem_t` structure holds a decoded Lenovo OEM structure whose layout is not known, with its data as the structure holds it.

Lenovo firmware carries OEM data in structures of type 135, signed `TP`, and of type 140, signed `LENOVO`. Each structure holds one OEM structure, told by its number and revision, which follow the signature and the offset of the number. The structures holding the OEM structures of known layouts are told by their signature together with the number and the revision, and the rest by the signature alone.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_mobile_oem_spec`, of type `DMI_TYPE(lenovo_mobile_oem)`, and by `dmi_lenovo_oem_spec`, of type `DMI_TYPE(lenovo_oem)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

Fields

`signature`

Signature, `TP` for type 135 and `LENOVO` for type 140.

`offset`

Offset of the OEM structure within the structure, which is the offset of `number`.

`number`

Number of the OEM structure.

`revision`

Revision of the format of the OEM structure.

`data`

Data of the OEM structure, which runs to the end of the structure.

See also

`dmi_entity_info`, `dmi_lenovo_device_presence`, `dmi_lenovo_bay_io`, `dmi_lenovo_ecp`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, as the service manuals of Lenovo laptops and dmidecode describe it.

dmi_lenovo_device_presence

Synopsis

```
#include <opendmi/entity/lenovo/oem.h>

typedef struct dmi_lenovo_device_presence {
    dmi_binary_t signature;
    uint8_t offset;
    uint8_t number;
    uint8_t revision;
    uint8_t devices;
    bool is_fingerprint_reader;
} dmi_lenovo_device_presence_t;
```

Description

The `dmi_lenovo_device_presence_t` structure holds the decoded Lenovo device presence detection, OEM structure 3 of revision 1 of type 135, which tells which devices of a ThinkPad are present, of which only the fingerprint reader is known.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_device_presence_spec`, of type `DMI_TYPE(lenovo_device_presence)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

Fields

`signature`
: Signature, `TP`.

`offset`
: Offset of the OEM structure, `0x07`.

`number`
: Number of the OEM structure, `3`.

`revision`
: Revision of the format of the OEM structure, `1`.

`devices`
: Device presence bits, of which only bit 0 is known.

`is_fingerprint_reader`
: Whether a fingerprint reader is present, as bit 0 of the device presence bits tells.

See also

`dmi_entity_info`, `dmi_lenovo_oem`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, as the service manuals of Lenovo laptops and dmidecode describe it.

dmi_lenovo_bay_io

Synopsis

```
#include <opendmi/entity/lenovo/oem.h>

typedef struct dmi_lenovo_bay_io {
    dmi_binary_t signature;
    uint8_t offset;
    uint8_t number;
    dmi_binary_t bay_signature;
    uint8_t version;
    dmi_binary_t data;
} dmi_lenovo_bay_io_t;
```

Description

The `dmi_lenovo_bay_io_t` structure holds the decoded Lenovo bay I/O, OEM structure 2 of type 135, which describes the bays of a laptop. The OEM structure carries a signature of its own, `BAY I/O `, in place of the revision, followed by its version, and the layout of the rest is not known.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_bay_io_spec`, of type `DMI_TYPE(lenovo_bay_io)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

Fields

`signature`
: Signature, `TP`.

`offset`
: Offset of the OEM structure, `0x07`.

`number`
: Number of the OEM structure, `2`.

`bay_signature`
: Signature of the OEM structure, `BAY I/O `.

`version`
: Version of the OEM structure.

`data`
: Data of the OEM structure, which runs to the end of the structure.

See also

`dmi_entity_info`, `dmi_lenovo_oem`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, as the service manuals of Lenovo laptops and dmidecode describe it.

dmi_lenovo_ecp

Synopsis

```
#include <opendmi/entity/lenovo/oem.h>

typedef struct dmi_lenovo_ecp {
    dmi_binary_t signature;
    uint8_t offset;
    uint8_t number;
    uint8_t revision;
    const char *version;
    const char *release_date;
} dmi_lenovo_ecp_t;
```

Description

The `dmi_lenovo_ecp_t` structure holds the decoded Lenovo ThinkPad embedded controller program, OEM structure 7 of revision 1 of type 140, which tells the version of the program of the embedded controller, which the README files of the firmware updates of Lenovo name.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_ecp_spec`, of type `DMI_TYPE(lenovo_ecp)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

Fields

`signature`
: Signature, `LENOVO`.

`offset`
: Offset of the OEM structure, `0x0B`.

`number`
: Number of the OEM structure, `7`.

`revision`
: Revision of the format of the OEM structure, `1`.

`version`
: Version ID of the program, e.g. `N20HT28W`.

`release_date`
: Release date of the program, as the firmware writes it, e.g. `10/08/2020`.

See also

`dmi_entity_info`, `dmi_lenovo_oem`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, as the service manuals of Lenovo laptops and dmidecode describe it.

<opendmi/entity/lenovo/records.h>

dmi_lenovo_date

Synopsis

```
#include <opendmi/entity/lenovo/records.h>

typedef struct dmi_lenovo_date {
    uint32_t raw_date;
    dmi_date_t date;
} dmi_lenovo_date_t;
```

Description

The `dmi_lenovo_date_t` structure holds the decoded Lenovo date record (type 134) of a ThinkPad. The date falls within the years the model has been made in, and so is probably its manufacture date.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_date_spec`, of type `DMI_TYPE(lenovo_date)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in. The structures of type 134 holding TPM information instead are told by their first string, see `dmi_lenovo_tpm_info`.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`raw_date`
: Date, as the structure holds it: day, month and year in binary-coded decimal, the year low byte first.

`date`
: Date.

See also

`dmi_entity_info`, `dmi_lenovo_tpm_info`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, and reverse engineered from the data corpus.

dmi_lenovo_mtm

Synopsis

```
#include <opendmi/entity/lenovo/records.h>

typedef struct dmi_lenovo_mtm {
    const char *brand;
    const char *mtm;
    dmi_binary_t data;
} dmi_lenovo_mtm_t;
```

Description

The `dmi_lenovo_mtm_t` structure holds the decoded Lenovo machine type model (type 200) of a consumer or small business laptop of Lenovo, e.g. an IdeaPad, a Yoga, a Legion or a ThinkBook, which tells its brand, `IdeaPad` for all of them, and its full machine type model, e.g. `20VE00U9RU`, of which the system information (type 1) holds only the machine type, e.g. `20VE`.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_mtm_spec`, of type `DMI_TYPE(lenovo_mtm)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position. The structures are 16 bytes long. ThinkPads carry structures of 7 bytes at the same type, which refer to three strings and whose meaning is not known, and which are left undecoded.

Fields

`brand`
: Brand, e.g. `IdeaPad`, which ThinkBooks hold too.

`mtm`
: Machine type model.

`data`
: Bytes past the strings, whose meaning is not established.

See also

`dmi_entity_info`, `dmi_system`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, and reverse engineered from the data corpus.

dmi_lenovo_tpm_info

Synopsis

```
#include <opendmi/entity/lenovo/records.h>

typedef struct dmi_lenovo_tpm_info {
    uint8_t unknown_1;
    dmi_binary_t vendor_raw;
    char *vendor;
    uint8_t unknown_2[2];
    uint16_t unknown_3;
    uint8_t unknown_4;
    const char *signature;
    const char *description;
} dmi_lenovo_tpm_info_t;
```

Description

The `dmi_lenovo_tpm_info_t` structure holds the decoded Lenovo TPM information (type 134) of a Lenovo laptop, which tells the vendor of its TPM by the TCG vendor ID, e.g. `ATML` for Atmel or `STM ` for STMicroelectronics. The structures are told from the date records of the same type, see `dmi_lenovo_date`, by their first string, `TPM INFO`.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_tpm_info_spec`, of type `DMI_TYPE(lenovo_tpm_info)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`unknown_1`

Value whose meaning is not established, `0` in all known data.

`vendor_raw`

TCG vendor ID of the TPM, as the structure holds it.

`vendor`

TCG vendor ID of the TPM, e.g. `ATML`, or `nullptr` if it is not printable. The string belongs to the structure, and is freed along with it.

`unknown_2`

Values whose meaning is not established, `1` and `1` in all known data.

`unknown_3`

Value whose meaning is not established, `0` in all known data.

`unknown_4`

Value whose meaning is not established, `0`, `2` or `3` in the known data.

`signature`

Signature, `TPM INFO`.

`description`

Description of the rest, `System Reserved`.

See also

`dmi_entity_info`, `dmi_lenovo_date`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, and reverse engineered from the data corpus.

<opendmi/entity/lenovo/tvt.h>

dmi_lenovo_tvt

Synopsis

```
#include <opendmi/entity/lenovo/tvt.h>

typedef struct dmi_lenovo_tvt {
    uint8_t version;
    dmi_binary_t features;
    bool is_diagnostics;
    const char *signature;
} dmi_lenovo_tvt_t;
```

Description

The `dmi_lenovo_tvt_t` structure holds the decoded Lenovo ThinkVantage Technologies enablement, which tells the technologies the platform enables by 128 bits, of which only the last one is known.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_lenovo_tvt_spec`, of type `DMI_TYPE(lenovo_tvt)`, declared in `<opendmi/module/lenovo.h>`, once the Lenovo extension module (`dmi_lenovo_module`) is enabled, which is done for IBM and Lenovo platforms when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structures are told by their length, 0x16 bytes, and by their first string, `TVT-Enablement`, since Intel vPro information takes type 131 in the same table, see `dmi_intel_vpro`.

Fields

`version`
: Version of the structure.

`features`
: Bits of the technologies, 16 bytes, from the least significant bit of the first byte up.

`is_diagnostics`
: Whether the diagnostics (PC-Doctor) are available, as bit 127 tells.

`signature`
: Signature of the structure, `TVT-Enablement`.

See also

`dmi_entity_info`, `dmi_intel_vpro`

Standards

Not part of the SMBIOS specification. Defined by Lenovo firmware, as dmidecode describes it.

<opendmi/entity/acer/devices.h>

dmi_acer_devices

Synopsis

```
#include <opendmi/entity/acer/devices.h>

typedef struct dmi_acer_devices {
    size_t device_count;
    dmi_acer_device_entry_t *devices;
} dmi_acer_devices_t;
```

Description

The `dmi_acer_devices_t` structure holds the decoded Acer device list (type 171), which lists the devices of a laptop by their PCI or USB vendor and device IDs, e.g. the graphics, the webcam, the audio and the wireless network adapter.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_acer_devices_spec`, of type `DMI_TYPE(acer_devices)`, declared in `<opendmi/module/acer.h>`, once the Acer extension module (`dmi_acer_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`device_count`
: Number of the devices.

`devices`
: Devices, see `dmi_acer_device_entry`. May be `nullptr` when `device_count` is 0.

See also

`dmi_entity_info`, `dmi_acer_device_entry`

Standards

Not part of the SMBIOS specification. Defined by Acer firmware, and reverse engineered from the data corpus.

dmi_acer_device_entry

Synopsis

```
#include <opendmi/entity/acer/devices.h>

typedef struct dmi_acer_device_entry {
    dmi_acer_device_kind_t kind;
    uint16_t vendor_id;
    uint16_t device_id;
} dmi_acer_device_entry_t;
```

Description

The `dmi_acer_device_entry_t` structure describes a device of an Acer laptop, as an element of the `devices` field of `dmi_acer_devices_t`.

Fields

`kind`
Kind of the device, whose meaning is not established, see `dmi_acer_device_kind`.

`vendor_id`
PCI or USB vendor ID of the device, zero if the device is absent.

`device_id`
PCI or USB device ID of the device, zero if the device is absent.

See also

`dmi_acer_devices`, `dmi_acer_device_kind`

Standards

Not part of the SMBIOS specification. Defined by Acer firmware, and reverse engineered from the data corpus.

dmi_acer_device_kind

Synopsis

```
#include <opendmi/entity/acer/devices.h>

typedef enum dmi_acer_device_kind {
    DMI_ACER_DEVICE_KIND_UNKNOWN_1  = 0x01,
    DMI_ACER_DEVICE_KIND_UNKNOWN_2  = 0x02,
    DMI_ACER_DEVICE_KIND_UNKNOWN_3  = 0x03,
    DMI_ACER_DEVICE_KIND_CAMERA     = 0x04,
    DMI_ACER_DEVICE_KIND_AUDIO      = 0x05,
    DMI_ACER_DEVICE_KIND_WLAN       = 0x07,
```

```
    DMI_ACER_DEVICE_KIND_BLUETOOTH  = 0x08,
    DMI_ACER_DEVICE_KIND_UNKNOWN_13 = 0x0D,
    DMI_ACER_DEVICE_KIND_UNKNOWN_17 = 0x11,
    DMI_ACER_DEVICE_KIND_UNKNOWN_19 = 0x13,
    DMI_ACER_DEVICE_KIND_UNKNOWN_21 = 0x15,
    DMI_ACER_DEVICE_KIND_UNKNOWN_22 = 0x16,
    DMI_ACER_DEVICE_KIND_UNKNOWN_25 = 0x19
} dmi_acer_device_kind_t;

const char *dmi_acer_device_kind_name(dmi_acer_device_kind_t value);
```

Description

The `dmi_acer_device_kind_t` enumeration tells the kind of a device of an Acer laptop, and is the type of the `kind` field of `dmi_acer_device_entry_t`.

The kinds are reverse engineered from the PCI and USB IDs of the devices of the data corpus. The ones named are given to the devices of a single class on all the laptops of the corpus. The other kinds found in it are named by their values: kinds `1` to `3` are given to devices of different classes, e.g. `1` to the graphics on one laptop and to the card reader on another, kind `25` to the host bridge on a single laptop, and the other ones to absent devices only. Other values may be found in the data, and are kept as they are rather than taken as invalid ones.

The `dmi_acer_device_kind_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_ACER_DEVICE_KIND_UNKNOWN_1`, **`DMI_ACER_DEVICE_KIND_UNKNOWN_2`**, **`DMI_ACER_DEVICE_KIND_UNKNOWN_3`**

Kinds of devices of different classes, e.g. graphics, card readers, Ethernet adapters and NVMe drives.

`DMI_ACER_DEVICE_KIND_CAMERA`

Webcam, a USB one on all the laptops of the corpus.

`DMI_ACER_DEVICE_KIND_AUDIO`

Audio, the HD Audio controller of the chipset.

`DMI_ACER_DEVICE_KIND_WLAN`

Wireless network adapter.

`DMI_ACER_DEVICE_KIND_BLUETOOTH`

Bluetooth adapter, a USB one.

`DMI_ACER_DEVICE_KIND_UNKNOWN_25`

Kind given to the host bridge of the chipset on a single laptop.

`DMI_ACER_DEVICE_KIND_UNKNOWN_13`, **`DMI_ACER_DEVICE_KIND_UNKNOWN_17`**, **`DMI_ACER_DEVICE_KIND_UNKNOWN_19`**, **`DMI_ACER_DEVICE_KIND_UNKNOWN_21`**, **`DMI_ACER_DEVICE_KIND_UNKNOWN_22`**

Kinds given to absent devices only.

Return value

`dmi_acer_device_kind_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_acer_device_entry`

Standards

Not part of the SMBIOS specification. Defined by Acer firmware, and reverse engineered from the data corpus.

<opendmi/entity/acer/hotkeys.h>

dmi_acer_hotkeys

Synopsis

```
#include <opendmi/entity/acer/hotkeys.h>

typedef struct dmi_acer_hotkeys {
    uint16_t comm_functions;
    uint16_t app_functions;
    uint16_t media_functions;
    uint16_t display_functions;
    uint16_t other_functions;
    uint8_t comm_key;
    size_t hotkey_count;
    dmi_acer_hotkey_entry_t *hotkeys;
} dmi_acer_hotkeys_t;
```

Description

The `dmi_acer_hotkeys_t` structure holds the decoded Acer hotkey functions (type 170), which tell the functions of the hotkeys of a laptop the Acer WMI driver reads. Some laptops of other brands, e.g. Fujitsu-Siemens, Medion, Lenovo and eMachines, are said to carry it too.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_acer_hotkeys_spec`, of type `DMI_TYPE(acer_hotkeys)`, declared in `<opendmi/module/acer.h>`, once the Acer extension module (`dmi_acer_module`) is enabled. It lives as long as the context it has been decoded in. The module is enabled for the platforms whose firmware or system vendor is Acer, since Acer laptops carry the firmware of other vendors, e.g. Insyde.

The hotkeys follow the function bitmaps up to the end of the structure. Their key numbers tell the button group by the high bits, and the functions of the keys of a group add up to its bitmap. The first key is the one of the communication function, the only one the Acer WMI driver reads.

Fields

`comm_functions`

Functions of the communication button, whose bits are the values of `dmi_acer_comm_function_t`: bit 0 WiFi, bit 6 3G, bit 7 WiMAX, bit 11 Bluetooth and bit 14 radio button.

`app_functions`

Functions of the application button.

`media_functions`

Functions of the media button.

`display_functions`

Functions of the display button.

`other_functions`

Functions of the other buttons.

`comm_key`

Number of the key of the communication function, at offset `0x0E`, where the structures holding hotkeys begin the first of them. Structures of `0x0F` bytes end with it and hold no hotkeys, the way the Acer WMI driver reads them, and are decoded by `dmi_acer_hotkeys_basic_spec` into the same structure.

`hotkey_count`

Number of the hotkeys.

`hotkeys`

Hotkeys, see `dmi_acer_hotkey_entry`. May be `nullptr` when `hotkey_count` is 0.

See also

`dmi_entity_info`, `dmi_acer_hotkey_entry`

Standards

Not part of the SMBIOS specification. Defined by Acer firmware, as the Acer WMI driver of Linux and dmidecode describe it. The hotkeys past the function bitmaps are reverse engineered from the data corpus.

dmi_acer_hotkey_entry

Synopsis

```
#include <opendmi/entity/acer/hotkeys.h>

typedef struct dmi_acer_hotkey_entry {
    uint8_t key;
    uint8_t kind;
    uint16_t function;
} dmi_acer_hotkey_entry_t;
```

Description

The `dmi_acer_hotkey_entry_t` structure describes a hotkey, as an element of the `hotkeys` field of `dmi_acer_hotkeys_t`.

Fields

`key`

Number of the key, whose high bits tell the button group: `0x0_` communication, `0x4_` media, `0x6_` display and `0x8_` others.

`kind`

Value whose meaning is not established, `1` or `2` in the known data, the same for all the hotkeys of a structure.

`function`

Function of the key, as bits of the function bitmap of its button group: the key of the communication function has all the bits of the communication bitmap, e.g. those of WiFi and Bluetooth.

See also

`dmi_acer_hotkeys`

Standards

Not part of the SMBIOS specification. Defined by Acer firmware, as the Acer WMI driver of Linux and dmidecode describe it. The hotkeys past the function bitmaps are reverse engineered from the data corpus.

<opendmi/entity/ami/firewire-guid.h>

dmi_ami_firewire_guid

Synopsis

```
#include <opendmi/entity/ami/firewire-guid.h>

typedef struct dmi_ami_firewire_guid {
    dmi_binary_t signature;
    dmi_binary_t data;
    uint8_t unknown;
    const char *name;
} dmi_ami_firewire_guid_t;
```

Description

The `dmi_ami_firewire_guid_t` structure holds the decoded AMI FireWire GUID, which keeps the data holding the GUID of the IEEE 1394 (FireWire) controller of the board, under the name `V1394GUID`.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_ami_firewire_guid_spec`, of type `DMI_TYPE(ami_firewire_guid)`, declared in `<opendmi/module/ami.h>`, once the AMI extension module (`dmi_ami_module`) is enabled, which is done for the platforms whose firmware vendor is AMI when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The layout is reverse engineered from a single structure of the data corpus, so the layout of the data is not established, and the data is kept as a whole. The data identifies the board: it is replaced when the table is anonymized, see `dmi_anonymize`.

Fields

`signature`

Signature of the structure, `FE DC BA 98 76 54 32 10`, which the structure is told by, along with its length of 54 bytes.

`data`

Data holding the GUID of the controller, 40 bytes, which begin like the bus information block of the configuration ROM of IEEE 1394.

`unknown`

Value whose meaning is not established, 0 in all known data.

`name`

Name of the data, `V1394GUID`.

See also

`dmi_entity_info`, `dmi_anonymize`

Standards

Not part of the SMBIOS specification. Defined by AMI firmware, and reverse engineered from the data corpus.

<opendmi/entity/apple/firmware-volume.h>

dmi_apple_firmware_volume

Synopsis

```
#include <opendmi/entity/apple/firmware-volume.h>

typedef struct dmi_apple_firmware_volume {
    uint8_t region_count;
    uint32_t features;
    uint32_t features_mask;
    uint8_t region_types[DMI_APPLE_FLASH_REGION_COUNT];
    dmi_apple_flash_region_t regions[DMI_APPLE_FLASH_REGION_COUNT];
    uint32_t extended_features;
    uint32_t extended_features_mask;
} dmi_apple_firmware_volume_t;
```

Description

The `dmi_apple_firmware_volume_t` structure holds the decoded Apple firmware volume information (type 128), which tells the features of the firmware, which macOS and its boot loader read, and the layout of its flash memory.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_firmware_volume_spec`, of type `DMI_TYPE(apple_firmware_volume)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The features are 64 bits wide: the structure holds their low 32 bits and, on newer firmware, their high 32 bits as the extended features, which the older firmware does not tell.

Fields

`region_count`
: Number of the regions of the flash memory.

`features`
: Features of the firmware, a set of `dmi_apple_firmware_feature_t` bits, see `dmi_apple_firmware_feature`.

`features_mask`
: Mask of the features the firmware tells.

`region_types`

Types of the regions, as the data holds them, before the regions.

`regions`

Regions of the flash memory, whose types are taken from `region_types`, see `dmi_apple_flash_region`. The entries past `region_count` are not used.

`extended_features`

Extended features of the firmware, the high 32 bits of the features, a set of `dmi_apple_extended_feature_t` bits, see `dmi_apple_extended_feature`. Zero when the structure is too short to hold them.

`extended_features_mask`

Mask of the extended features the firmware tells. Zero when the structure is too short to hold it.

See also

`dmi_apple_flash_region`, `dmi_apple_firmware_feature`, `dmi_apple_extended_feature`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_flash_region

Synopsis

```
#include <opendmi/entity/apple/firmware-volume.h>

typedef struct dmi_apple_flash_region {
    dmi_apple_region_type_t type;
    uint32_t start_address;
    uint32_t end_address;
} dmi_apple_flash_region_t;
```

Description

The `dmi_apple_flash_region_t` structure describes a region of the flash memory of the firmware, and is the type of the elements of the `regions` array of `dmi_apple_firmware_volume_t`.

Fields

`type`

Type of the region, see `dmi_apple_region_type`.

`start_address`
: Start address of the region.

`end_address`
: End address of the region.

See also

`dmi_apple_firmware_volume`, `dmi_apple_region_type`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_firmware_feature

Synopsis

```
#include <opendmi/entity/apple/firmware-volume.h>

typedef enum dmi_apple_firmware_feature {
    DMI_APPLE_FIRMWARE_FEATURE_CSM_LEGACY_MODE              = 0,
    DMI_APPLE_FIRMWARE_FEATURE_CD_DRIVE_BOOT                = 1,
    DMI_APPLE_FIRMWARE_FEATURE_TARGET_DISK_MODE             = 2,
    DMI_APPLE_FIRMWARE_FEATURE_NET_BOOT                     = 4,
    DMI_APPLE_FIRMWARE_FEATURE_SLING_SHOT                   = 5,
    DMI_APPLE_FIRMWARE_FEATURE_WIRELESS                     = 8,
    DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY_1            = 10,
    DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY_2            = 11,
    DMI_APPLE_FIRMWARE_FEATURE_FIRMWARE_LOCKED              = 12,
    DMI_APPLE_FIRMWARE_FEATURE_HIGH_SPEED_USB               = 14,
    DMI_APPLE_FIRMWARE_FEATURE_NO_USB_SUBSTITUTE_WORKAROUND = 17,
    DMI_APPLE_FIRMWARE_FEATURE_APFS                         = 19,
    DMI_APPLE_FIRMWARE_FEATURE_APFS_EXTRA                   = 20,
    DMI_APPLE_FIRMWARE_FEATURE_TRBX                         = 22,
    DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY              = 24,
    DMI_APPLE_FIRMWARE_FEATURE_EXTENDED_FEATURES            = 25,
    DMI_APPLE_FIRMWARE_FEATURE_NO_MBA_S4_WORKAROUND         = 28,
    DMI_APPLE_FIRMWARE_FEATURE_UEFI_WINDOWS_BOOT            = 29,
    DMI_APPLE_FIRMWARE_FEATURE_NO_BOOTSCRIPT_WORKAROUND     = 31
} dmi_apple_firmware_feature_t;

const char *dmi_apple_firmware_feature_name(dmi_apple_firmware_feature_t
value);
```

Description

The `dmi_apple_firmware_feature_t` enumeration names the bits of the `features` field of `dmi_apple_firmware_volume_t`, by their numbers. The bits of no known meaning are named after their numbers, e.g. `Unknown bit 3`, the way OpenCore names them.

The `dmi_apple_firmware_feature_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_APPLE_FIRMWARE_FEATURE_CSM_LEGACY_MODE`

Supports booting legacy operating systems through the Compatibility Support Module (CSM).

`DMI_APPLE_FIRMWARE_FEATURE_CD_DRIVE_BOOT`

Supports booting from optical drives.

`DMI_APPLE_FIRMWARE_FEATURE_TARGET_DISK_MODE`

Supports target disk mode.

`DMI_APPLE_FIRMWARE_FEATURE_NET_BOOT`

Supports network boot.

`DMI_APPLE_FIRMWARE_FEATURE_SLING_SHOT`

Supports the recovery embedded into the firmware (SlingShot).

`DMI_APPLE_FIRMWARE_FEATURE_WIRELESS`

Supports wireless networks in the firmware.

`DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY_1`

Bit 0 of the platform security policy.

`DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY_2`

Bit 1 of the platform security policy.

`DMI_APPLE_FIRMWARE_FEATURE_FIRMWARE_LOCKED`

Firmware is locked.

`DMI_APPLE_FIRMWARE_FEATURE_HIGH_SPEED_USB`

Supports USB 2.0 in the firmware.

`DMI_APPLE_FIRMWARE_FEATURE_NO_USB_SUBSTITUTE_WORKAROUND`

USB ports are not substituted with others by macOS.

`DMI_APPLE_FIRMWARE_FEATURE_APFS`

Supports APFS.

`DMI_APPLE_FIRMWARE_FEATURE_APFS_EXTRA`

Supports APFS, unused by the known Macs.

`DMI_APPLE_FIRMWARE_FEATURE_TRBX`

Supports Apple tamper resistant boot X.

`DMI_APPLE_FIRMWARE_FEATURE_SECURITY_POLICY`
: Supports the platform security policy.

`DMI_APPLE_FIRMWARE_FEATURE_EXTENDED_FEATURES`
: Supports the extended features, see `dmi_apple_extended_feature`.

`DMI_APPLE_FIRMWARE_FEATURE_NO_MBA_S4_WORKAROUND`
: Memory map is not adjusted by the boot loader on hibernation wake.

`DMI_APPLE_FIRMWARE_FEATURE_UEFI_WINDOWS_BOOT`
: Supports UEFI Windows boot.

`DMI_APPLE_FIRMWARE_FEATURE_NO_BOOTSCRIPT_WORKAROUND`
: Memory map is not adjusted by the boot loader for the boot script.

Return value

`dmi_apple_firmware_feature_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_apple_firmware_volume`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_extended_feature

Synopsis

```
#include <opendmi/entity/apple/firmware-volume.h>

typedef enum dmi_apple_extended_feature {
    DMI_APPLE_EXTENDED_FEATURE_LARGE_BASESYSTEM = 3
} dmi_apple_extended_feature_t;

const char *dmi_apple_extended_feature_name(dmi_apple_extended_feature_t
value);
```

Description

The `dmi_apple_extended_feature_t` enumeration names the bits of the `extended_features` field of `dmi_apple_firmware_volume_t`, which are the bits 32 to 63 of the features, by their numbers within the field. The bits of no known meaning are named after the numbers of the features, e.g. `Unknown bit 32`.

The `dmi_apple_extended_feature_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_APPLE_EXTENDED_FEATURE_LARGE_BASESYSTEM`

Supports the large BaseSystem, which macOS 12 requires, bit 35 of the features.

Return value

`dmi_apple_extended_feature_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_apple_firmware_volume`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_region_type

Synopsis

```
#include <opendmi/entity/apple/firmware-volume.h>

typedef enum dmi_apple_region_type {
    DMI_APPLE_REGION_TYPE_UNUSED    = 0,
    DMI_APPLE_REGION_TYPE_RECOVERY  = 1,
    DMI_APPLE_REGION_TYPE_MAIN      = 2,
    DMI_APPLE_REGION_TYPE_NVRAM     = 3,
    DMI_APPLE_REGION_TYPE_CONFIG    = 4,
    DMI_APPLE_REGION_TYPE_DIAGVAULT = 5
} dmi_apple_region_type_t;

const char *dmi_apple_region_type_name(dmi_apple_region_type_t value);
```

Description

The `dmi_apple_region_type_t` enumeration tells the type of a region of the flash memory of the firmware, and is the type of the `type` field of `dmi_apple_flash_region_t`.

The `dmi_apple_region_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_APPLE_REGION_TYPE_UNUSED`
: Entry is not used.

`DMI_APPLE_REGION_TYPE_RECOVERY`
: Recovery firmware.

`DMI_APPLE_REGION_TYPE_MAIN`
: Main firmware.

`DMI_APPLE_REGION_TYPE_NVRAM`
: Non-volatile variables.

`DMI_APPLE_REGION_TYPE_CONFIG`
: Configuration.

`DMI_APPLE_REGION_TYPE_DIAGVAULT`
: Diagnostics vault.

Return value

`dmi_apple_region_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_apple_flash_region`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

<opendmi/entity/apple/memory-spd-data.h>

dmi_apple_memory_spd_data

Synopsis

```
#include <opendmi/entity/apple/memory-spd-data.h>

typedef struct dmi_apple_memory_spd_data {
    dmi_handle_t handle;
    uint16_t offset;
    uint16_t size;
    dmi_binary_t data;
} dmi_apple_memory_spd_data_t;
```

Description

The `dmi_apple_memory_spd_data_t` structure holds the decoded Apple memory SPD data (type 130): the serial presence detect (SPD) data of a memory module, or a part of it. The firmware adds a structure for each memory module.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_memory_spd_data_spec`, of type `DMI_TYPE(apple_memory_spd_data)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The Intel reference code gives type 130 to the Intel Active Management Technology information too, which is told apart by its signature.

Fields

`handle`

Handle of the memory device (type 17) of the module.

`offset`

Offset of the data within the SPD data of the module.

`size`

Length of the data, in bytes.

`data`

SPD data.

See also

`dmi_memory_device`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

<opendmi/entity/apple/platform-feature.h>

dmi_apple_platform_feature

Synopsis

```
#include <opendmi/entity/apple/platform-feature.h>

typedef struct dmi_apple_platform_feature {
    uint64_t features;
} dmi_apple_platform_feature_t;
```

Description

The `dmi_apple_platform_feature_t` structure holds the decoded Apple platform feature information (type 133), which tells the features of the platform, which macOS reads. Older Macs carry no such structure.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_platform_feature_spec`, of type `DMI_TYPE(apple_platform_feature)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

Fields

`features`
: Features of the platform, a set of `dmi_apple_platform_feature_bit_t` bits, see `dmi_apple_platform_feature_bit`.

See also

`dmi_apple_platform_feature_bit`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_platform_feature_bit

Synopsis

```
#include <opendmi/entity/apple/platform-feature.h>
```

```
typedef enum dmi_apple_platform_feature_bit {
    DMI_APPLE_PLATFORM_FEATURE_SOLDERED_MEMORY = 1,
    DMI_APPLE_PLATFORM_FEATURE_HEADLESS_GPU    = 2,
    DMI_APPLE_PLATFORM_FEATURE_HOST_PM         = 3,
    DMI_APPLE_PLATFORM_FEATURE_POWER_CHIME     = 4
} dmi_apple_platform_feature_bit_t;

const char
*dmi_apple_platform_feature_bit_name(dmi_apple_platform_feature_bit_t value);
```

Description

The `dmi_apple_platform_feature_bit_t` enumeration names the bits of the `features` field of `dmi_apple_platform_feature_t`, by their numbers. Bit 0 is of no known meaning, and is named `Unknown bit 0`, the way OpenCore names it.

The `dmi_apple_platform_feature_bit_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_APPLE_PLATFORM_FEATURE_SOLDERED_MEMORY`
: System memory is soldered, and cannot be replaced.

`DMI_APPLE_PLATFORM_FEATURE_HEADLESS_GPU`
: Headless GPU, of no display connectors, is present.

`DMI_APPLE_PLATFORM_FEATURE_HOST_PM`
: Supports host power management.

`DMI_APPLE_PLATFORM_FEATURE_POWER_CHIME`
: Supports the chime at boot.

Return value

`dmi_apple_platform_feature_bit_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_apple_platform_feature`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h`

header of OpenCore describes it.

<opendmi/entity/apple/processor-bus-speed.h>

dmi_apple_processor_bus_speed

Synopsis

```
#include <opendmi/entity/apple/processor-bus-speed.h>

typedef struct dmi_apple_processor_bus_speed {
    uint16_t speed;
} dmi_apple_processor_bus_speed_t;
```

Description

The `dmi_apple_processor_bus_speed_t` structure holds the decoded Apple processor bus speed information (type 132), which tells the speed of the bus of the processor, e.g. of its front-side bus or QuickPath Interconnect.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_processor_bus_speed_spec`, of type `DMI_TYPE(apple_processor_bus_speed)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

Fields

`speed`

Speed of the bus, in megatransfers per second, e.g. `4800` for a QuickPath Interconnect of 4.8 GT/s. OpenCore sets it to four times the external clock of the processor.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

<opendmi/entity/apple/processor-type.h>

dmi_apple_processor_type

Synopsis

```
#include <opendmi/entity/apple/processor-type.h>

typedef struct dmi_apple_processor_type {
    uint8_t generation;
    dmi_apple_processor_class_t clazz;
} dmi_apple_processor_type_t;
```

Description

The `dmi_apple_processor_type_t` structure holds the decoded Apple processor type information (type 131), which tells the type of the processor, by which macOS names it. The type is a word whose high byte is the class of the processor and whose low byte is its generation within the class, e.g. `0x0701` for the first generation of Intel Core i7. Newer firmware follows the type with two reserved bytes.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_processor_type_spec`, of type `DMI_TYPE(apple_processor_type)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The Intel reference code gives type 131 to the Intel vPro information too, which is told apart by its signature.

Fields

`generation`
: Generation of the processor within its class, the minor type.

`clazz`
: Class of the processor, the major type, see `dmi_apple_processor_class`. It is exported as `class`, the member being named otherwise since `class` is a keyword of C++.

See also

`dmi_apple_processor_class`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

dmi_apple_processor_class

Synopsis

```
#include <opendmi/entity/apple/processor-type.h>

typedef enum dmi_apple_processor_class {
    DMI_APPLE_PROCESSOR_CLASS_UNKNOWN      = 0x00,
    DMI_APPLE_PROCESSOR_CLASS_CORE         = 0x02,
    DMI_APPLE_PROCESSOR_CLASS_CORE_2       = 0x03,
    DMI_APPLE_PROCESSOR_CLASS_XEON_PENRYN  = 0x04,
    DMI_APPLE_PROCESSOR_CLASS_XEON_NEHALEM = 0x05,
    DMI_APPLE_PROCESSOR_CLASS_CORE_I5      = 0x06,
    DMI_APPLE_PROCESSOR_CLASS_CORE_I7      = 0x07,
    DMI_APPLE_PROCESSOR_CLASS_CORE_I3      = 0x09,
    DMI_APPLE_PROCESSOR_CLASS_XEON_E5      = 0x0A,
    DMI_APPLE_PROCESSOR_CLASS_CORE_M       = 0x0B,
    DMI_APPLE_PROCESSOR_CLASS_CORE_M3      = 0x0C,
    DMI_APPLE_PROCESSOR_CLASS_CORE_M5      = 0x0D,
    DMI_APPLE_PROCESSOR_CLASS_CORE_M7      = 0x0E,
    DMI_APPLE_PROCESSOR_CLASS_XEON_W       = 0x0F,
    DMI_APPLE_PROCESSOR_CLASS_CORE_I9      = 0x10
} dmi_apple_processor_class_t;

const char *dmi_apple_processor_class_name(dmi_apple_processor_class_t value);
```

Description

The `dmi_apple_processor_class_t` enumeration tells the class of a processor, the major type of an Apple processor type, and is the type of the `clazz` field of `dmi_apple_processor_type_t`. Other values may be found in the data, and are kept as they are.

The `dmi_apple_processor_class_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_APPLE_PROCESSOR_CLASS_UNKNOWN`

Unknown.

`DMI_APPLE_PROCESSOR_CLASS_CORE`

Intel Core.

`DMI_APPLE_PROCESSOR_CLASS_CORE_2`

Intel Core 2.

`DMI_APPLE_PROCESSOR_CLASS_XEON_PENRYN`

Intel Xeon of the Penryn microarchitecture.

`DMI_APPLE_PROCESSOR_CLASS_XEON_NEHALEM`
: Intel Xeon of the Nehalem microarchitecture.

`DMI_APPLE_PROCESSOR_CLASS_CORE_I5`
: Intel Core i5.

`DMI_APPLE_PROCESSOR_CLASS_CORE_I7`
: Intel Core i7.

`DMI_APPLE_PROCESSOR_CLASS_CORE_I3`
: Intel Core i3.

`DMI_APPLE_PROCESSOR_CLASS_XEON_E5`
: Intel Xeon E5.

`DMI_APPLE_PROCESSOR_CLASS_CORE_M`
: Intel Core M.

`DMI_APPLE_PROCESSOR_CLASS_CORE_M3`
: Intel Core m3.

`DMI_APPLE_PROCESSOR_CLASS_CORE_M5`
: Intel Core m5.

`DMI_APPLE_PROCESSOR_CLASS_CORE_M7`
: Intel Core m7.

`DMI_APPLE_PROCESSOR_CLASS_XEON_W`
: Intel Xeon W.

`DMI_APPLE_PROCESSOR_CLASS_CORE_I9`
: Intel Core i9.

Return value

`dmi_apple_processor_class_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_apple_processor_type`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

<opendmi/entity/apple/smc-version.h>

dmi_apple_smc_version

Synopsis

```
#include <opendmi/entity/apple/smc-version.h>

typedef struct dmi_apple_smc_version {
    dmi_binary_t version_raw;
    char *version;
} dmi_apple_smc_version_t;
```

Description

The `dmi_apple_smc_version_t` structure holds the decoded Apple SMC version information (type 134), which tells the version of the firmware of the System Management Controller (SMC). Newer Macs carry no such structure.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_apple_smc_version_spec`, of type `DMI_TYPE(apple_smc_version)`, declared in `<opendmi/module/apple.h>`, once the Apple extension module (`dmi_apple_module`) is enabled, which is done for the platforms of Apple firmware when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

Fields

`version_raw`

Version of the SMC firmware, as the structure holds it, of `DMI_APPLE_SMC_VERSION_SIZE` (16) bytes.

`version`

Version of the SMC firmware as text, e.g. `2.15f7`, which the bytes of zero end. `nullptr` if the structure holds no text. The string belongs to the structure, and is freed along with it.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Macs, and laid out as the `AppleSmBios.h` header of OpenCore describes it.

<opendmi/entity/cisco/pci-adapter.h>

dmi_cisco_pci_adapter

Synopsis

```
#include <opendmi/entity/cisco/pci-adapter.h>

typedef struct dmi_cisco_pci_adapter {
    uint8_t version;
    uint16_t raw_vendor_id;
    uint16_t raw_device_id;
    uint8_t prog_if;
    uint8_t subclass;
    uint8_t pci_class;
    uint8_t revision;
    const char *model;
    const char *slot;
    uint16_t raw_subsystem_vendor_id;
    uint16_t raw_subsystem_id;
    const char *firmware_version;
    dmi_pci_vendor_id_t vendor_id;
    uint16_t device_id;
    dmi_pci_vendor_id_t subsystem_vendor_id;
    uint16_t subsystem_id;
} dmi_cisco_pci_adapter_t;
```

Description

The `dmi_cisco_pci_adapter_t` structure holds the decoded Cisco PCI adapter information, which describes an adapter of a Cisco UCS server: its PCI identifiers and class, its model, the slot it is installed in, and the version of its firmware. There is a structure for each adapter, including the ones on board.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_cisco_pci_adapter_spec`, of type `DMI_TYPE(cisco_pci_adapter)`, declared in `<opendmi/module/cisco.h>`, once the Cisco extension module (`dmi_cisco_module`) is enabled, which is done for the platforms whose firmware vendor is Cisco when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

The structure holds the identifiers in the big-endian byte order, unlike the other fields of SMBIOS. The `raw_` fields keep them as they are read, in the little-endian byte order, so that the structure is written back as it was, and the other identifier fields give them in the order of the host. The identifiers of the data corpus match the PCI ID database and the models the structures name, e.g. `1137:0042` of subsystem `1137:0217` for Cisco UCS VIC 1455.

Fields

`version`

Version of the structure, presumably, 3 in all known data.

`raw_vendor_id`, **`raw_device_id`**, **`raw_subsystem_vendor_id`**, **`raw_subsystem_id`**

Identifiers as they are read, with their bytes swapped.

`prog_if`, **`subclass`**, **`pci_class`**

Class code of the function: the programming interface, the subclass and the class, e.g. `0x01` and `0x04` for a RAID controller.

`revision`

Revision ID of the function.

`model`

Model of the adapter, e.g. `Cisco UCS VIC 1455`.

`slot`

Slot the adapter is installed in, e.g. `SlotID:1`, `SlotID:MLOM` for the modular LAN on motherboard or `SlotID:MRAID` for the modular RAID controller.

`firmware_version`

Version of the firmware of the adapter, `N/A` if it has none.

`vendor_id`, **`device_id`**

PCI vendor ID and device ID of the adapter.

`subsystem_vendor_id`, **`subsystem_id`**

PCI subsystem vendor ID and subsystem ID of the adapter.

See also

`dmi_entity_info`, `dmi_cisco_slot_buses`, `dmi_pci_vendor_id`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Cisco UCS servers, and reverse engineered from the data corpus.

<opendmi/entity/cisco/slot-buses.h>

dmi_cisco_slot_buses

Synopsis

```
#include <opendmi/entity/cisco/slot-buses.h>

typedef struct dmi_cisco_slot_buses {
    const char *slot;
    size_t bus_count;
    uint8_t *buses;
} dmi_cisco_slot_buses_t;
```

Description

The `dmi_cisco_slot_buses_t` structure holds the decoded Cisco PCI slot buses, which list the PCI buses behind a slot of a Cisco UCS server, presumably: the numbers match the adapters the PCI adapter information gives for the same slot, see `dmi_cisco_pci_adapter`, e.g. six of them for Cisco UCS VIC 1455, which has a PCI switch.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_cisco_slot_buses_spec`, of type `DMI_TYPE(cisco_slot_buses)`, declared in `<opendmi/module/cisco.h>`, once the Cisco extension module (`dmi_cisco_module`) is enabled, which is done for the platforms whose firmware vendor is Cisco when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`. It lives as long as the context it has been decoded in.

Fields

`slot`
: Slot, as the PCI adapter information names it, e.g. `SlotID:1`, `SlotID:MLOM` or `SlotID:MRAID`.

`bus_count`
: Number of elements of `buses`.

`buses`
: PCI bus numbers. May be `nullptr` when `bus_count` is 0.

See also

`dmi_entity_info`, `dmi_cisco_pci_adapter`

Standards

Not part of the SMBIOS specification. Defined by the firmware of Cisco UCS servers, and reverse engineered from the data corpus.

<opendmi/entity/dell/bios-flags.h>

dmi_dell_bios_flags

Synopsis

```
#include <opendmi/entity/dell/bios-flags.h>

typedef struct dmi_dell_bios_flags {
    uint16_t flags;
    bool is_acpi_wmi;
} dmi_dell_bios_flags_t;
```

Description

The `dmi_dell_bios_flags_t` structure holds the decoded Dell BIOS flags (type 177), which tell the features of the firmware the Dell drivers rely on, of which only the support of ACPI WMI is known. The Dell SMBIOS WMI driver of Linux reads the first word of the flags of the structures of 12 bytes and of handle `0xB100`; the rest of the flags is `0` in all known data.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_bios_flags_spec`, of type `DMI_TYPE(dell_bios_flags)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

Fields

`flags`

Flags, of which only bit 1 is known.

`is_acpi_wmi`

Whether the firmware supports ACPI WMI, which the Dell WMI drivers use, as bit 1 of the flags tells.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, as the Dell WMI driver of Linux and dmidecode describe it.

<opendmi/entity/dell/calling-iface.h>

dmi_dell_calling_iface

Synopsis

```
#include <opendmi/entity/dell/calling-iface.h>

typedef struct dmi_dell_calling_iface {
    uint16_t command_io_address;
    uint8_t command_io_code;
    uint32_t supported_commands;
    size_t token_count;
    dmi_dell_calling_iface_token_t *tokens;
} dmi_dell_calling_iface_t;
```

Description

The `dmi_dell_calling_iface_t` structure holds the decoded Dell calling interface structure (type 218): the I/O port through which commands are issued to the firmware, and the tokens, settings of the firmware kept in non-volatile storage, which these commands read and write.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_calling_iface)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`command_io_address`

Address of the I/O port written to issue a command to the firmware.

`command_io_code`

Value written to the command I/O port to issue a command.

`supported_commands`

Bit mask of the commands the firmware supports, as the structure carries it. The meaning of the bits is not publicly documented.

`token_count`

Number of elements of `tokens`.

`tokens`

Tokens, in the order of the structure. Unused records (token identifier `0x0000`) and the end-of-table marker (`0xFFFF`) are left out. May be `nullptr` when `token_count` is 0.

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                         DMI_TYPE(dell_calling_iface), true);
dmi_dell_calling_iface_t *iface    = dmi_entity_info(entity,
                                         DMI_TYPE(dell_calling_iface));

if (iface != nullptr) {
    for (size_t i = 0; i < iface->token_count; i++)
        printf("Token 0x%04x: 0x%04x\n", iface->tokens[i].id,
               iface->tokens[i].value);
}
```

See also

`dmi_entity_info`, `dmi_dell_calling_iface_token`, `dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Calling Interface (Type 218) chapter of the reference manual.

dmi_dell_calling_iface_token

Synopsis

```
#include <opendmi/entity/dell/calling-iface.h>

typedef struct dmi_dell_calling_iface_token {
    uint16_t id;
    uint16_t location;
    uint16_t value;
} dmi_dell_calling_iface_token_t;
```

Description

The `dmi_dell_calling_iface_token_t` structure describes one token of the Dell calling interface structure (type 218), and is the type of the elements of the `tokens` array of `dmi_dell_calling_iface_t`. A token is a setting of the firmware kept in non-volatile storage, which the commands of the calling interface read and write.

Fields

`id`

Token identifier. Never `0x0000` or `0xFFFF`: the records with these identifiers are unused ones and the end-of-table marker, which are not decoded as tokens.

`location`
: Location of the token in non-volatile storage, 0 for string tokens.

`value`
: Value of the token, or length of the string for string tokens.

See also

`dmi_dell_calling_iface`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Calling Interface (Type 218) chapter of the reference manual.

<opendmi/entity/dell/common.h>

dmi_dell_check_type

Synopsis

```
#include <opendmi/entity/dell/common.h>

typedef enum dmi_dell_check_type {
    DMI_DELL_CHECK_TYPE_WORD_CHECKSUM   = 0x00,
    DMI_DELL_CHECK_TYPE_BYTE_CHECKSUM   = 0x01,
    DMI_DELL_CHECK_TYPE_WORD_CRC        = 0x02,
    DMI_DELL_CHECK_TYPE_WORD_CHECKSUM_N = 0x03,
} dmi_dell_check_type_t;

const char *dmi_dell_check_type_name(dmi_dell_check_type_t value);
```

Description

The `dmi_dell_check_type_t` enumeration tells how the checksum of a range of CMOS locations is computed. It is the type of the `check_type` field of `dmi_dell_indexed_io_t`, `dmi_dell_protected_area_1_t` and `dmi_dell_protected_area_2_t`, and of the `range_check_type` field of `dmi_dell_protected_area_2_t`.

The `dmi_dell_check_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_DELL_CHECK_TYPE_WORD_CHECKSUM`

Running sum of the checked locations, kept in a 16-bit word.

`DMI_DELL_CHECK_TYPE_BYTE_CHECKSUM`

Running sum of the checked locations, kept in a byte.

`DMI_DELL_CHECK_TYPE_WORD_CRC`

CRC of the checked locations, kept in a 16-bit word.

`DMI_DELL_CHECK_TYPE_WORD_CHECKSUM_N`

Running sum of the checked locations, negated (`~sum + 1`), kept in a 16-bit word.

Return value

`dmi_dell_check_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_dell_indexed_io`, `dmi_dell_protected_area_1`, `dmi_dell_protected_area_2`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Indexed IO (Type 212) chapter of the reference manual.

dmi_dell_enable_state

Synopsis

```
#include <opendmi/entity/dell/common.h>

typedef enum dmi_dell_enable_state {
    DMI_DELL_ENABLE_STATE_UNSPEC   = 0x00,
    DMI_DELL_ENABLE_STATE_OTHER    = 0x01,
    DMI_DELL_ENABLE_STATE_UNKNOWN  = 0x02,
    DMI_DELL_ENABLE_STATE_ENABLED  = 0x03,
    DMI_DELL_ENABLE_STATE_DISABLED = 0x04,
} dmi_dell_enable_state_t;

const char *dmi_dell_enable_state_name(dmi_dell_enable_state_t value);
```

Description

The `dmi_dell_enable_state_t` enumeration tells whether a feature of a Dell port is enabled, and is the type of the `state` and `speed_limit_state` fields of `dmi_dell_infrared_port_t`.

The `dmi_dell_enable_state_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_ENABLE_STATE_UNSPEC`
: Unspecified. The firmware does not define the value 0.

`DMI_DELL_ENABLE_STATE_OTHER`
: Other state.

`DMI_DELL_ENABLE_STATE_UNKNOWN`
: Unknown state.

`DMI_DELL_ENABLE_STATE_ENABLED`
: Enabled.

`DMI_DELL_ENABLE_STATE_DISABLED`
: Disabled.

Return value

`dmi_dell_enable_state_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_infrared_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Infrared Port (Type 211) chapter of the reference manual.

dmi_dell_port_security

Synopsis

```
#include <opendmi/entity/dell/common.h>

typedef enum dmi_dell_port_security {
    DMI_DELL_PORT_SECURITY_UNSPEC            = 0x00,
    DMI_DELL_PORT_SECURITY_OTHER             = 0x01,
    DMI_DELL_PORT_SECURITY_UNKNOWN           = 0x02,
    DMI_DELL_PORT_SECURITY_NONE              = 0x03,
    DMI_DELL_PORT_SECURITY_EXT_IFACE_LOCKED  = 0x04,
    DMI_DELL_PORT_SECURITY_EXT_IFACE_ENABLED = 0x05,
    DMI_DELL_PORT_SECURITY_BOOT_BYPASS       = 0x06,
} dmi_dell_port_security_t;

const char *dmi_dell_port_security_name(dmi_dell_port_security_t value);
```

Description

The `dmi_dell_port_security_t` enumeration tells which security setting of the firmware applies to a Dell port, and is the type of the `security` field of `dmi_dell_parallel_port_t` and `dmi_dell_serial_port_t`.

The `dmi_dell_port_security_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_PORT_SECURITY_UNSPEC`
: Unspecified. The firmware does not define the value 0.

`DMI_DELL_PORT_SECURITY_OTHER`
: Other setting.

`DMI_DELL_PORT_SECURITY_UNKNOWN`
: Unknown setting.

`DMI_DELL_PORT_SECURITY_NONE`
: No security setting applies to the port.

`DMI_DELL_PORT_SECURITY_EXT_IFACE_LOCKED`
: External interface of the port is locked out.

`DMI_DELL_PORT_SECURITY_EXT_IFACE_ENABLED`
: External interface of the port is enabled.

`DMI_DELL_PORT_SECURITY_BOOT_BYPASS`
: Boot-bypass.

Return value

`dmi_dell_port_security_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_parallel_port`, `dmi_dell_serial_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Parallel Port (Type 209) and Dell Serial Port (Type 210) chapters of the reference manual.

dmi_dell_value_format

Synopsis

```
#include <opendmi/entity/dell/common.h>
```

```
typedef enum dmi_dell_value_format {
    DMI_DELL_VALUE_FORMAT_SCAN_CODE    = 0x00,
    DMI_DELL_VALUE_FORMAT_ASCII        = 0x01,
    DMI_DELL_VALUE_FORMAT_SCAN_CODE_NS = 0x02,
    DMI_DELL_VALUE_FORMAT_ASCII_NS     = 0x03,
} dmi_dell_value_format_t;

const char *dmi_dell_value_format_name(dmi_dell_value_format_t value);
```

Description

The `dmi_dell_value_format_t` enumeration tells how a protected value, such as a password, is encoded in CMOS, and is the type of the `value_format` field of `dmi_dell_protected_area_1_t` and `dmi_dell_protected_area_2_t`.

The `dmi_dell_value_format_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_DELL_VALUE_FORMAT_SCAN_CODE`
: Alphanumeric value, stored as keyboard scan codes.

`DMI_DELL_VALUE_FORMAT_ASCII`
: Alphanumeric value, stored as ASCII characters.

`DMI_DELL_VALUE_FORMAT_SCAN_CODE_NS`
: Alphanumeric value, stored as keyboard scan codes, NS variant.

`DMI_DELL_VALUE_FORMAT_ASCII_NS`
: Alphanumeric value, stored as ASCII characters, NS variant.

Return value

`dmi_dell_value_format_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_dell_protected_area_1`, `dmi_dell_protected_area_2`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Protected Area Type 2 (Type

214) chapter of the reference manual.

<opendmi/entity/dell/device-bay.h>

dmi_dell_device_bay

Synopsis

```
#include <opendmi/entity/dell/device-bay.h>

typedef struct dmi_dell_device_bay {
    uint8_t unknown_1;
    const char *name;
    const char *supported_devices;
    const char *installed_device;
    uint8_t unknown_2;
    const char *unknown_string_1;
    const char *unknown_string_2;
    bool has_unknown_strings;
} dmi_dell_device_bay_t;
```

Description

The `dmi_dell_device_bay_t` structure holds the decoded Dell device bay (type 219), which describes a bay of a laptop or of its docking station that takes one of several devices, e.g. an optical drive or a second battery. The structures are found on Latitude laptops, e.g. Latitude D500, D610 and E6230, and newer systems which have no bay carry one of blank strings, e.g. Precision M3800 and G15 5515.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_device_bay_spec`, of type `DMI_TYPE(dell_device_bay)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

Some Dell platforms give type 219 to the Intel Management Engine interface information, see `dmi_intel_mei`, which is told by its signature, and the specification decodes the other structures of the type.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`unknown_1`

Value whose meaning is not established, `3` on the systems which have a bay, and `0` or `2` on the ones which have none.

`name`

Name of the bay, e.g. `System Device Bay` or `Dock DBay`.

`supported_devices`

Devices the bay takes, separated by commas, e.g. `Floppy, Battery, CD-ROM, CD-RW, Hard Disk, DVD`.

`installed_device`

Device the bay holds, e.g. `Battery`, or `EMPTY` for none, padded with spaces by the firmware.

`unknown_2`

Value whose meaning is not established, `0xFF` in most known data, and `0` on Precision M3800.

`unknown_string_1`, **`unknown_string_2`**

Strings whose meaning is not established, which the structures of 11 bytes refer to, e.g. blank ones on Precision M3800 or `00h` on G15 5515. Not shown when `has_unknown_strings` is not set.

`has_unknown_strings`

Whether the structure refers to `unknown_string_1` and `unknown_string_2`, which the structures of 9 bytes leave out.

See also

`dmi_entity_info`, `dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/dell/device-names.h>

dmi_dell_device_names

Synopsis

```
#include <opendmi/entity/dell/device-names.h>

typedef struct dmi_dell_device_names {
    uint8_t unknown;
    size_t device_count;
    dmi_dell_device_name_entry_t *devices;
} dmi_dell_device_names_t;
```

Description

The `dmi_dell_device_names_t` structure holds the decoded Dell device names (type 225), which give the devices of a kind, e.g. the processors or the memory devices, the names the management controller (iDRAC) of a PowerEdge server knows them by, its fully qualified device descriptors (FQDD). The structures are found on PowerEdge servers of the 13th and 14th generations, e.g. PowerEdge R630 and R740xd, one for the processors and one for the memory devices.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_device_names_spec`, of type `DMI_TYPE(dell_device_names)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`unknown`
: Value whose meaning is not established, `1` in all known data.

`device_count`
: Number of elements of `devices`.

`devices`
: Names of the devices, see `dmi_dell_device_name_entry`. May be `nullptr` when `device_count` is 0.

See also

`dmi_dell_device_name_entry`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

dmi_dell_device_name_entry

Synopsis

```
#include <opendmi/entity/dell/device-names.h>

typedef struct dmi_dell_device_name_entry {
    const char *fqdd;
    dmi_handle_t handle;
    uint8_t unknown;
} dmi_dell_device_name_entry_t;
```

Description

The `dmi_dell_device_name_entry_t` structure gives a device the name the management controller (iDRAC) knows it by, and is the type of the elements of the `devices` array of `dmi_dell_device_names_t`.

Fields

`fqdd`
: Fully qualified device descriptor (FQDD) of the device, e.g. `CPU.Socket.1` or `DIMM.Socket.A1`.

`handle`
: Handle of the structure describing the device, of a processor (type 4) or of a memory device (type 17) in all known data, which the `link.wrong-type` lint rule checks it against.

`unknown`
: Value whose meaning is not established, `0` in all known data.

See also

`dmi_dell_device_names`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/dell/hotkeys.h>

dmi_dell_hotkeys

Synopsis

```
#include <opendmi/entity/dell/hotkeys.h>

typedef struct dmi_dell_hotkeys {
    size_t hotkey_count;
    dmi_dell_hotkey_entry_t *hotkeys;
} dmi_dell_hotkeys_t;
```

Description

The `dmi_dell_hotkeys_t` structure holds the decoded Dell hotkeys (type 178), which map the scan codes of the hotkeys of a laptop to their functions, which the Dell WMI driver reads.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_hotkeys_spec`, of type `DMI_TYPE(dell_hotkeys)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

Mappings run to the end of the structure, which carries no number of them of its own.

Fields

`hotkey_count`
: Number of the hotkeys.

`hotkeys`
: Hotkeys, see `dmi_dell_hotkey_entry`. May be `nullptr` when `hotkey_count` is 0.

See also

`dmi_entity_info`, `dmi_dell_hotkey_entry`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, as the Dell WMI driver of Linux and dmidecode describe it.

dmi_dell_hotkey_entry

Synopsis

```
#include <opendmi/entity/dell/hotkeys.h>
```

```
typedef struct dmi_dell_hotkey_entry {
    uint16_t scancode;
    uint16_t keycode;
} dmi_dell_hotkey_entry_t;
```

Description

The `dmi_dell_hotkey_entry_t` structure describes the mapping of a hotkey, as an element of the `hotkeys` field of `dmi_dell_hotkeys_t`. Key code `0xFFFF` is found for the keys of no known function, which the Dell WMI driver of Linux takes for an unknown key, while key code `0x00FF` stands for a function of its own, the third programmable key.

Fields

`scancode`

Scan code of the key, as the firmware reports it in the WMI events of the hotkeys.

`keycode`

Code of the function of the key, as the Dell WMI driver of Linux numbers the functions.

See also

`dmi_dell_hotkeys`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, as the Dell WMI driver of Linux and dmidecode describe it.

<opendmi/entity/dell/indexed-io.h>

dmi_dell_indexed_io

Synopsis

```
#include <opendmi/entity/dell/indexed-io.h>

typedef struct dmi_dell_indexed_io {
    uint16_t index_port;
    uint16_t data_port;
    dmi_dell_check_type_t check_type;
    uint8_t check_start;
    uint8_t check_end;
    uint8_t check_index;
    size_t token_count;
    dmi_dell_indexed_io_token_t *tokens;
} dmi_dell_indexed_io_t;
```

Description

The `dmi_dell_indexed_io_t` structure holds the decoded Dell indexed IO structure (type 212): the pair of I/O ports through which the locations of CMOS are selected and accessed, the checksum guarding a range of CMOS, and the tokens, settings of the firmware kept in CMOS.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_indexed_io)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`index_port`
: Address of the I/O port written to select a CMOS location.

`data_port`
: Address of the I/O port through which the selected CMOS location is read and written.

`check_type`
: How the checksum of the checked range is computed.

`check_start`
: CMOS index of the first location of the checked range.

`check_end`
: CMOS index of the last location of the checked range.

`check_index`
: CMOS index of the checksum. When `check_start`, `check_end` and `check_index` are all 0, the CMOS area has no checksum.

`token_count`
: Number of elements of `tokens`.

`tokens`
: Tokens, in the order of the structure. Unused records (token identifier `0x0000`) and the end-of-table marker (`0xFFFF`) are left out. May be `nullptr` when `token_count` is 0.

Examples

```
dmi_registry_t        *registry = dmi_get_registry(context);
dmi_entity_t          *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(dell_indexed_io), true);
dmi_dell_indexed_io_t *io       = dmi_entity_info(entity,
                                      DMI_TYPE(dell_indexed_io));

if (io != nullptr) {
    printf("Ports 0x%04x/0x%04x, %zu tokens\n", io->index_port,
           io->data_port, io->token_count);
}
```

See also

`dmi_entity_info`, `dmi_dell_indexed_io_token`, `dmi_dell_check_type`, `dmi_dell_calling_iface`, `dmi_dell_protected_area_1`, `dmi_dell_protected_area_2`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Indexed IO (Type 212) chapter of the reference manual.

dmi_dell_indexed_io_token

Synopsis

```
#include <opendmi/entity/dell/indexed-io.h>

typedef struct dmi_dell_indexed_io_token {
    uint16_t id;
    uint8_t location;
    uint8_t and_mask;
    bool is_string;
    uint8_t or_value;
    uint8_t string_length;
} dmi_dell_indexed_io_token_t;
```

Description

The `dmi_dell_indexed_io_token_t` structure describes one token of the Dell indexed IO structure (type 212), and is the type of the elements of the `tokens` array of `dmi_dell_indexed_io_t`. A token is a setting of the firmware kept in CMOS: either a boolean one, activated by masking the bits of a location and setting some of them, or a string starting at a location.

Fields

`id`

Token identifier. Never `0x0000` or `0xFFFF`: the records with these identifiers are unused ones and the end-of-table marker, which are not decoded as tokens.

`location`

CMOS index of the value of the token.

`and_mask`

Mask applied with a bitwise AND to the value at the location before `or_value` is set. 0 for string tokens.

`is_string`

Whether the token is a string token, which the structure tells by an `and_mask` of 0.

`or_value`

For boolean tokens, bits set with a bitwise OR in the value at the location when the token is activated. 0 for string tokens.

`string_length`

For string tokens, length of the string. 0 for boolean tokens.

See also

`dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Indexed IO (Type 212) chapter of the reference manual.

<opendmi/entity/dell/infrared-port.h>

dmi_dell_infrared_port

Synopsis

```
#include <opendmi/entity/dell/infrared-port.h>

typedef struct dmi_dell_infrared_port {
    const char *location;
    dmi_dell_enable_state_t state;
    dmi_dell_enable_state_t speed_limit_state;
    uint32_t speed_limit;
    const char *physical_port;
    const char *virtual_com_port;
    const char *virtual_lpt_port;
    dmi_dell_infrared_proto_t protocol;
} dmi_dell_infrared_port_t;
```

Description

The `dmi_dell_infrared_port_t` structure holds the decoded Dell infrared port structure (type 211): where an infrared port is, whether it and its speed limit are enabled, its protocol, and the legacy ports it is seen as.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_infrared_port)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

The structures are told by their length of 13 bytes, since structures of other lengths are found at the same type, e.g. the one of 17 bytes of Studio 1555, which holds text in its formatted area. Those are not decoded.

Fields

`location`

Location of the port. `nullptr` when the structure carries no string.

`state`

Whether the port is enabled.

`speed_limit_state`

Whether the speed limit is enabled.

`speed_limit`

Speed limit, in bits per second. The structure carries it in hundreds of bits per second, which is assumed rather than known: no source gives the unit, and no data confirms it.

`physical_port`

Name of the physical port. `nullptr` when the structure carries no string.

`virtual_com_port`

Name of the virtual COM port the infrared port is seen as. `nullptr` when the structure carries no string.

`virtual_lpt_port`

Name of the virtual LPT port the infrared port is seen as. `nullptr` when the structure carries no string.

`protocol`

Infrared protocol of the port.

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                         DMI_TYPE(dell_infrared_port), true);
dmi_dell_infrared_port_t *port     = dmi_entity_info(entity,
                                         DMI_TYPE(dell_infrared_port));

if (port != nullptr) {
    const char *proto = dmi_dell_infrared_proto_name(port->protocol);

    printf("Infrared port: %s, limit %u bit/s\n",
           proto != nullptr ? proto : "?", (unsigned)port->speed_limit);
}
```

See also

`dmi_entity_info`, `dmi_dell_enable_state`, `dmi_dell_infrared_proto`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Infrared Port (Type 211) chapter of the reference manual.

dmi_dell_infrared_proto

Synopsis

```
#include <opendmi/entity/dell/infrared-port.h>

typedef enum dmi_dell_infrared_proto {
    DMI_DELL_INFRARED_PROTO_UNSPEC  = 0x00,
    DMI_DELL_INFRARED_PROTO_OTHER   = 0x01,
    DMI_DELL_INFRARED_PROTO_UNKNOWN = 0x02,
```

```
    DMI_DELL_INFRARED_PROTO_SIR     = 0x03,
    DMI_DELL_INFRARED_PROTO_FIR     = 0x04,
    DMI_DELL_INFRARED_PROTO_MIR     = 0x05,
} dmi_dell_infrared_proto_t;

const char *dmi_dell_infrared_proto_name(dmi_dell_infrared_proto_t value);
```

Description

The `dmi_dell_infrared_proto_t` enumeration tells the protocol of a Dell infrared port, and is the type of the `protocol` field of `dmi_dell_infrared_port_t`.

The `dmi_dell_infrared_proto_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_INFRARED_PROTO_UNSPEC`

Unspecified. The firmware does not define the value 0.

`DMI_DELL_INFRARED_PROTO_OTHER`

Other protocol.

`DMI_DELL_INFRARED_PROTO_UNKNOWN`

Unknown protocol.

`DMI_DELL_INFRARED_PROTO_SIR`

Standard IR (SIR).

`DMI_DELL_INFRARED_PROTO_FIR`

Fast IR (FIR).

`DMI_DELL_INFRARED_PROTO_MIR`

Medium speed IR (MIR).

Return value

`dmi_dell_infrared_proto_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_infrared_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Infrared Port (Type 211) chapter of the reference manual.

<opendmi/entity/dell/memory-ids.h>

dmi_dell_memory_ids

Synopsis

```
#include <opendmi/entity/dell/memory-ids.h>

typedef struct dmi_dell_memory_ids {
    uint8_t id_length;
    size_t module_count;
    dmi_dell_memory_id_entry_t *modules;
} dmi_dell_memory_ids_t;
```

Description

The `dmi_dell_memory_ids_t` structure holds the decoded Dell memory module identifiers (type 223), which tell the manufacturer and the serial number of the memory modules, as their serial presence detect (SPD) data holds them. The memory devices (type 17) of the older systems carry no fields for them, e.g. the ones of PowerEdge 1300, 400SC and SC420 and OptiPlex GX110.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_memory_ids_spec`, of type `DMI_TYPE(dell_memory_ids)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The specification decodes the structures whose identifiers are 12 bytes long only, since the ones of other lengths may be laid out otherwise, and leaves the others undecoded.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`id_length`

Length of the identifiers of a module, which follow the handle of its memory device, `12` in all known data.

`module_count`

Number of elements of `modules`.

`modules`

Identifiers of the memory modules, one for each memory device, see `dmi_dell_memory_id_entry`. May be `nullptr` when `module_count` is 0.

See also

`dmi_dell_memory_id_entry`, `dmi_memory_device`, `dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

dmi_dell_memory_id_entry

Synopsis

```
#include <opendmi/entity/dell/memory-ids.h>

typedef struct dmi_dell_memory_id_entry {
    dmi_handle_t handle;
    dmi_binary_t manufacturer;
    dmi_binary_t serial_number;
} dmi_dell_memory_id_entry_t;
```

Description

The `dmi_dell_memory_id_entry_t` structure holds the identifiers of a memory module, as its serial presence detect (SPD) data holds them, and is the type of the elements of the `modules` array of `dmi_dell_memory_ids_t`. Sockets which hold no module usually carry bytes of `0xFF`, though some firmware repeats the identifiers of another module there, e.g. the one of PowerEdge 1300.

The serial number is marked as private, so that `dmi_anonymize` replaces it.

Fields

`handle`
: Handle of the memory device (type 17) of the socket.

`manufacturer`
: Identifier of the manufacturer of the module, of 8 bytes, normally its JEDEC identifier, whose leading bytes of `0x7F` are continuation codes, e.g. `CE` for Samsung, `2C` for Micron, or `7F 98` for Kingston.

`serial_number`
: Serial number of the module, of 4 bytes, in the order the SPD data holds them.

See also

`dmi_dell_memory_ids`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/dell/parallel-port.h>

dmi_dell_parallel_port

Synopsis

```
#include <opendmi/entity/dell/parallel-port.h>

typedef struct dmi_dell_parallel_port {
    uint16_t base_address;
    uint8_t irq_number;
    dmi_dell_parallel_port_connector_type_t connector;
    dmi_dell_parallel_port_connector_pinout_t pinout;
    dmi_dell_parallel_port_caps_t capabilities;
    dmi_dell_port_security_t security;
} dmi_dell_parallel_port_t;
```

Description

The `dmi_dell_parallel_port_t` structure holds the decoded Dell parallel port structure (type 209): the resources, the connector and the capabilities of a parallel port, and the security setting applied to it.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_parallel_port)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`base_address`

Base I/O address of the port.

`irq_number`

IRQ the port uses.

`connector`

Connector of the port.

`pinout`

Pin-out of the connector.

`capabilities`

Modes and features the port supports.

`security`

Security setting applied to the port.

Examples

```
dmi_registry_t           *registry = dmi_get_registry(context);
dmi_entity_t             *entity   = dmi_registry_lookup_first(registry,
                                         DMI_TYPE(dell_parallel_port), true);
dmi_dell_parallel_port_t *port     = dmi_entity_info(entity,
                                         DMI_TYPE(dell_parallel_port));

if (port != nullptr) {
    printf("Parallel port at 0x%04x, IRQ %u, ECP %s\n", port->base_address,
           port->irq_number, port->capabilities.is_ecp ? "yes" : "no");
}
```

See also

`dmi_entity_info`, `dmi_dell_parallel_port_connector_type`, `dmi_dell_parallel_port_connector_pinout`, `dmi_dell_parallel_port_caps`, `dmi_dell_port_security`, `dmi_dell_serial_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Parallel Port (Type 209) chapter of the reference manual.

dmi_dell_parallel_port_caps

Synopsis

```
#include <opendmi/entity/dell/parallel-port.h>

typedef union dmi_dell_parallel_port_caps {
    dmi_word_t __value;
    struct {
        dmi_word_t is_xt_at        : 1;
        dmi_word_t is_ps2          : 1;
        dmi_word_t is_ecp          : 1;
        dmi_word_t is_epp          : 1;
        dmi_word_t is_pc_98        : 1;
        dmi_word_t is_pc_98_hireso : 1;
        dmi_word_t is_pc_h98       : 1;
        dmi_word_t reserved        : 8;
        dmi_word_t has_dma_support : 1;
    };
} dmi_dell_parallel_port_caps_t;
```

Description

The `dmi_dell_parallel_port_caps_t` union holds the capabilities of a Dell parallel port, and is the type of the `capabilities` field of `dmi_dell_parallel_port_t`. It is a packed 16-bit word, whose bits are read one by one through the bit members, or all at once through `__value`. A bit is set when the port supports the mode or the feature.

Fields

`__value`
: The whole capability word, as the structure carries it.

`is_xt_at`
: XT/AT mode (bit 0).

`is_ps2`
: PS/2 mode (bit 1).

`is_ecp`
: ECP mode (bit 2).

`is_epp`
: EPP mode (bit 3).

`is_pc_98`
: PC-98 mode (bit 4).

`is_pc_98_hireso`
: PC-98 Hireso mode (bit 5).

`is_pc_h98`
: PC-H98 mode (bit 6).

`reserved`
: Reserved bits 7 to 14.

`has_dma_support`
: The port supports DMA (bit 15).

See also

`dmi_dell_parallel_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Parallel Port (Type 209) chapter of the reference manual.

dmi_dell_parallel_port_connector_pinout

Synopsis

```
#include <opendmi/entity/dell/parallel-port.h>

typedef enum dmi_dell_parallel_port_connector_pinout {
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_UNSPEC       = 0x00,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_OTHER        = 0x01,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_UNKNOWN      = 0x02,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_XT_AT        = 0x03,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PS2          = 0x04,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_IEEE_1284    = 0x05,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PROPRIETARY  = 0x06,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98        = 0xA0,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_HIRESO = 0xA1,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_H98       = 0xA2,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_NOTE   = 0xA3,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_FULL   = 0xA4,
} dmi_dell_parallel_port_connector_pinout_t;

const char *dmi_dell_parallel_port_connector_pinout_name(
        dmi_dell_parallel_port_connector_pinout_t value);
```

Description

The `dmi_dell_parallel_port_connector_pinout_t` enumeration tells the pin-out of the connector of a Dell parallel port, and is the type of the `pinout` field of `dmi_dell_parallel_port_t`.

The `dmi_dell_parallel_port_connector_pinout_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_UNSPEC`

Unspecified. The firmware does not define the value 0.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_OTHER`

Other pin-out.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_UNKNOWN`

Unknown pin-out.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_XT_AT`

XT/AT.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PS2`

PS/2.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_IEEE_1284`
: IEEE 1284.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PROPRIETARY`
: Proprietary pin-out.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98`
: PC-98.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_HIRESO`
: PC-98 Hireso.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_H98`
: PC-H98.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_NOTE`
: PC-98 Note.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_PINOUT_PC_98_FULL`
: PC-98 Full.

Return value

`dmi_dell_parallel_port_connector_pinout_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_parallel_port`, `dmi_dell_parallel_port_connector_type`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Parallel Port (Type 209) chapter of the reference manual.

dmi_dell_parallel_port_connector_type

Synopsis

```
#include <opendmi/entity/dell/parallel-port.h>

typedef enum dmi_dell_parallel_port_connector_type {
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_UNSPEC      = 0x00,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_OTHER       = 0x01,
```

```
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_UNKNOWN     = 0x02,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_25_F     = 0x03,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_25_M     = 0x04,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_CN_36       = 0x05,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_HPCN_36     = 0x06,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_PROPRIETARY = 0x07,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_CN_14       = 0xA0,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_36_F     = 0xA1,
    DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_HPCN_20     = 0xA2,
} dmi_dell_parallel_port_connector_type_t;

const char *dmi_dell_parallel_port_connector_type_name(
        dmi_dell_parallel_port_connector_type_t value);
```

Description

The `dmi_dell_parallel_port_connector_type_t` enumeration tells the connector of a Dell parallel port, and is the type of the `connector` field of `dmi_dell_parallel_port_t`.

The `dmi_dell_parallel_port_connector_type_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_UNSPEC`
: Unspecified. The firmware does not define the value 0.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_OTHER`
: Other connector.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_UNKNOWN`
: Unknown connector.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_25_F`
: DB-25 female receptacle (IEEE 1284-A).

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_25_M`
: DB-25 male plug (IEEE 1284-A).

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_CN_36`
: Centronics receptacle (IEEE 1284-B).

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_HPCN_36`
: Mini-Centronics receptacle (IEEE 1284-C).

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_PROPRIETARY`
: Proprietary connector.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_CN_14`

Centronics-14.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_DB_36_F`

DB-36 female.

`DMI_DELL_PARALLEL_PORT_CONNECTOR_TYPE_HPCN_20`

Mini-Centronics-20.

Return value

`dmi_dell_parallel_port_connector_type_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_parallel_port`, `dmi_dell_parallel_port_connector_pinout`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Parallel Port (Type 209) chapter of the reference manual.

<opendmi/entity/dell/protected-area-1.h>

dmi_dell_protected_area_1

Synopsis

```
#include <opendmi/entity/dell/protected-area-1.h>

typedef struct dmi_dell_protected_area_1 {
    uint16_t token_id;
    uint8_t value_length;
    dmi_dell_value_format_t value_format;
    uint16_t validation_key;
    uint16_t index_port;
    uint16_t data_port;
    dmi_dell_check_type_t check_type;
    uint8_t value_start;
    uint8_t check_index;
} dmi_dell_protected_area_1_t;
```

Description

The `dmi_dell_protected_area_1_t` structure holds the decoded Dell protected area type 1 structure (type 213): where a protected value, such as a password, is kept in CMOS, how it is encoded, and where its checksum is. The CMOS locations are accessed through the indexed I/O ports the structure gives.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_protected_area_1)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`token_id`

Identifier of the token of the protected value.

`value_length`

Length of the protected value.

`value_format`

How the protected value is encoded.

`validation_key`

Validation key of the protected value, as the structure carries it.

`index_port`

Address of the I/O port written to select a CMOS location.

`data_port`

Address of the I/O port through which the selected CMOS location is read and written.

`check_type`

How the checksum of the protected value is computed.

`value_start`

CMOS index of the protected value.

`check_index`

CMOS index of the checksum of the protected value.

Examples

```
dmi_registry_t              *registry = dmi_get_registry(context);
dmi_entity_t                *entity   = dmi_registry_lookup_first(registry,
                                            DMI_TYPE(dell_protected_area_1),
                                            true);
dmi_dell_protected_area_1_t *area     = dmi_entity_info(entity,
                                            DMI_TYPE(dell_protected_area_1));

if (area != nullptr) {
    printf("Token 0x%04x: %u bytes at CMOS index 0x%02x\n", area->token_id,
           area->value_length, area->value_start);
}
```

See also

`dmi_entity_info`,`dmi_dell_value_format`,`dmi_dell_check_type`,`dmi_dell_protected_area_2`, `dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Protected Area Type 1 (Type 213) chapter of the reference manual.

<opendmi/entity/dell/protected-area-2.h>

dmi_dell_protected_area_2

Synopsis

```
#include <opendmi/entity/dell/protected-area-2.h>

typedef struct dmi_dell_protected_area_2 {
    uint16_t token_id;
    uint8_t value_length;
    dmi_dell_value_format_t value_format;
    uint16_t validation_key;
    uint16_t index_port;
    uint16_t data_port;
    dmi_dell_check_type_t check_type;
    uint8_t value_start;
    uint8_t check_index;
    dmi_dell_check_type_t range_check_type;
    uint8_t range_check_start;
    uint8_t range_check_end;
    uint8_t range_check_index;
} dmi_dell_protected_area_2_t;
```

Description

The `dmi_dell_protected_area_2_t` structure holds the decoded Dell protected area type 2 structure (type 214): where a protected value, such as a password, is kept in CMOS, how it is encoded, and where its checksum is. Unlike the type 1 structure (`dmi_dell_protected_area_1`), it also describes the checksum of a range of CMOS. The CMOS locations are accessed through the indexed I/O ports the structure gives.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_protected_area_2)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`token_id`

Identifier of the token of the protected value.

`value_length`

Length of the protected value.

`value_format`

How the protected value is encoded.

`validation_key`
: Validation key of the protected value, as the structure carries it.

`index_port`
: Address of the I/O port written to select a CMOS location.

`data_port`
: Address of the I/O port through which the selected CMOS location is read and written.

`check_type`
: How the checksum of the protected value is computed.

`value_start`
: CMOS index of the protected value.

`check_index`
: CMOS index of the checksum of the protected value.

`range_check_type`
: How the checksum of the checked range is computed.

`range_check_start`
: CMOS index of the first location of the checked range.

`range_check_end`
: CMOS index of the last location of the checked range.

`range_check_index`
: CMOS index of the checksum of the checked range.

Examples

```
dmi_registry_t              *registry = dmi_get_registry(context);
dmi_entity_t                *entity   = dmi_registry_lookup_first(registry,
                                            DMI_TYPE(dell_protected_area_2),
                                            true);
dmi_dell_protected_area_2_t *area     = dmi_entity_info(entity,
                                            DMI_TYPE(dell_protected_area_2));

if (area != nullptr) {
    printf("Checked range 0x%02x-0x%02x\n", area->range_check_start,
           area->range_check_end);
}
```

See also

`dmi_entity_info`, `dmi_dell_value_format`, `dmi_dell_check_type`, `dmi_dell_protected_area_1`, `dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Protected Area Type 2 (Type 214) chapter of the reference manual.

<opendmi/entity/dell/revisions.h>

dmi_dell_revisions

Synopsis

```
#include <opendmi/entity/dell/revisions.h>

typedef struct dmi_dell_revisions {
    dmi_version_t implementation_version;
    uint16_t system_id;
    uint8_t hardware_revision;
    const char *manufacture_date;
    const char *first_poweron_date;
} dmi_dell_revisions_t;
```

Description

The `dmi_dell_revisions_t` structure holds the decoded Dell revisions and IDs structure (type 208): the revision of the Dell implementation of the firmware tables, the identifier and the hardware revision of the system, and the dates of its manufacture and first power-on.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_revisions)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`implementation_version`

Revision of the Dell implementation, as a version number with the major and the minor numbers set and a revision number of 0.

`system_id`

Dell system identifier. When the one-byte identifier holds `0xFE`, the structure carries the identifier in a 16-bit extension, which the field then holds; it stays `0xFE` when the structure is too short to carry the extension.

`hardware_revision`

Hardware revision of the system.

`manufacture_date`

Manufacture date of the system, as a string in the format of the firmware. `nullptr` when the structure is too short to carry it, or carries no string.

`first_poweron_date`

Date the system was first powered on, as a string in the format of the firmware. `nullptr` when the structure is too short to carry it, or carries no string.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(dell_revisions), true);
dmi_dell_revisions_t *rev      = dmi_entity_info(entity,
                                      DMI_TYPE(dell_revisions));

if (rev != nullptr) {
    printf("Dell implementation %u.%u, system ID 0x%04x\n",
           dmi_version_major(rev->implementation_version),
           dmi_version_minor(rev->implementation_version), rev->system_id);
}
```

See also

`dmi_entity_info`, `dmi_version_format`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Revisions and IDs (Type 208) chapter of the reference manual.

<opendmi/entity/dell/serial-port.h>

dmi_dell_serial_port

Synopsis

```
#include <opendmi/entity/dell/serial-port.h>

typedef struct dmi_dell_serial_port {
    uint16_t base_address;
    uint8_t irq_number;
    dmi_dell_serial_port_connector_type_t connector;
    dmi_dell_serial_port_caps_t capabilities;
    uint32_t maximum_speed;
    dmi_dell_port_security_t security;
} dmi_dell_serial_port_t;
```

Description

The `dmi_dell_serial_port_t` structure holds the decoded Dell serial port structure (type 210): the resources, the connector and the capabilities of a serial port, the highest speed the BIOS supports on it, and the security setting applied to it.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(dell_serial_port)`, declared in `<opendmi/module/dell.h>`, and lives as long as the context it has been decoded in.

Fields

`base_address`
Base I/O address of the port.

`irq_number`
IRQ the port uses.

`connector`
Connector of the port.

`capabilities`
Serial controller the port is compatible with.

`maximum_speed`
Highest speed the BIOS supports on the port, in bits per second. The structure carries it in hundreds of bits per second.

`security`
Security setting applied to the port.

Examples

```
dmi_registry_t         *registry = dmi_get_registry(context);
dmi_entity_t           *entity   = dmi_registry_lookup_first(registry,
                                       DMI_TYPE(dell_serial_port), true);
dmi_dell_serial_port_t *port     = dmi_entity_info(entity,
                                       DMI_TYPE(dell_serial_port));

if (port != nullptr) {
    const char *caps = dmi_dell_serial_port_caps_name(port->capabilities);

    printf("Serial port at 0x%04x: %s, up to %u bit/s\n", port->base_address,
           caps != nullptr ? caps : "?", (unsigned)port->maximum_speed);
}
```

See also

`dmi_entity_info`, `dmi_dell_serial_port_connector_type`, `dmi_dell_serial_port_caps`, `dmi_dell_port_security`, `dmi_dell_parallel_port`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Serial Port (Type 210) chapter of the reference manual.

dmi_dell_serial_port_caps

Synopsis

```
#include <opendmi/entity/dell/serial-port.h>

typedef enum dmi_dell_serial_port_caps {
    DMI_DELL_SERIAL_PORT_CAPS_UNSPEC    = 0x00,
    DMI_DELL_SERIAL_PORT_CAPS_OTHER     = 0x01,
    DMI_DELL_SERIAL_PORT_CAPS_UNKNOWN   = 0x02,
    DMI_DELL_SERIAL_PORT_CAPS_XT_AT     = 0x03,
    DMI_DELL_SERIAL_PORT_CAPS_16450     = 0x04,
    DMI_DELL_SERIAL_PORT_CAPS_16550     = 0x05,
    DMI_DELL_SERIAL_PORT_CAPS_16550A    = 0x06,
    DMI_DELL_SERIAL_PORT_CAPS_8251      = 0xA0,
    DMI_DELL_SERIAL_PORT_CAPS_8251_FIFO = 0xA1,
} dmi_dell_serial_port_caps_t;

const char *dmi_dell_serial_port_caps_name(dmi_dell_serial_port_caps_t value);
```

Description

The `dmi_dell_serial_port_caps_t` enumeration tells which serial controller a Dell serial port is compatible with, and is the type of the `capabilities` field of `dmi_dell_serial_port_t`. Unlike the capabilities of a parallel port, it is a single value, not a set of bits.

The `dmi_dell_serial_port_caps_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_SERIAL_PORT_CAPS_UNSPEC`
: Unspecified. The firmware does not define the value 0.

`DMI_DELL_SERIAL_PORT_CAPS_OTHER`
: Other.

`DMI_DELL_SERIAL_PORT_CAPS_UNKNOWN`
: Unknown.

`DMI_DELL_SERIAL_PORT_CAPS_XT_AT`
: XT/AT compatible.

`DMI_DELL_SERIAL_PORT_CAPS_16450`
: 16450 compatible.

`DMI_DELL_SERIAL_PORT_CAPS_16550`
: 16550 compatible.

`DMI_DELL_SERIAL_PORT_CAPS_16550A`
: 16550A compatible.

`DMI_DELL_SERIAL_PORT_CAPS_8251`
: 8251 compatible.

`DMI_DELL_SERIAL_PORT_CAPS_8251_FIFO`
: 8251 FIFO compatible.

Return value

`dmi_dell_serial_port_caps_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_serial_port`, `dmi_dell_serial_port_connector_type`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Serial Port (Type 210) chapter of the reference manual.

dmi_dell_serial_port_connector_type

Synopsis

```
#include <opendmi/entity/dell/serial-port.h>

typedef enum dmi_dell_serial_port_connector_type {
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_UNSPEC           = 0x00,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_OTHER            = 0x01,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_UNKNOWN          = 0x02,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_9_M           = 0x03,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_9_F           = 0x04,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_25_M          = 0x05,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_25_F          = 0x06,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_RJ_11            = 0x07,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_RJ_45            = 0x08,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_PROPRIETARY      = 0x09,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_CIRCULAR_DIN_8_M = 0x0A,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_CIRCULAR_DIN_8_F = 0x0B,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_IEEE_1284_C_14   = 0xA0,
    DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_IEEE_1284_C_26   = 0xA1,
} dmi_dell_serial_port_connector_type_t;

const char *dmi_dell_serial_port_connector_type_name(
        dmi_dell_serial_port_connector_type_t value);
```

Description

The `dmi_dell_serial_port_connector_type_t` enumeration tells the connector of a Dell serial port, and is the type of the `connector` field of `dmi_dell_serial_port_t`.

The `dmi_dell_serial_port_connector_type_name` function returns the name of a value, as the command line tool shows it. The name is in English, or in the language of the locale the library is set to when it has a translation of the name.

Values

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_UNSPEC`
: Unspecified. The firmware does not define the value 0.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_OTHER`
: Other connector.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_UNKNOWN`
: Unknown connector.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_9_M`
: DB-9 male.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_9_F`
: DB-9 female.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_25_M`
: DB-25 male.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_DB_25_F`
: DB-25 female.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_RJ_11`
: RJ-11.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_RJ_45`
: RJ-45.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_PROPRIETARY`
: Proprietary connector.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_CIRCULAR_DIN_8_M`
: Circular DIN-8 male.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_CIRCULAR_DIN_8_F`
: Circular DIN-8 female.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_IEEE_1284_C_14`
: Mini-Centronics Type-14.

`DMI_DELL_SERIAL_PORT_CONNECTOR_TYPE_IEEE_1284_C_26`
: Mini-Centronics Type-26.

Return value

`dmi_dell_serial_port_connector_type_name` returns the name of the value, a constant string which must not be modified or freed, or `nullptr` if the value has no name, e.g. a value out of the range of the enumeration.

Errors

None.

See also

`dmi_dell_serial_port`, `dmi_dell_serial_port_caps`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, see the Dell Serial Port (Type 210) chapter of the reference manual.

<opendmi/entity/dell/system-id.h>

dmi_dell_system_id

Synopsis

```
#include <opendmi/entity/dell/system-id.h>

typedef struct dmi_dell_system_id {
    const char *identifier;
    const char *system_id;
} dmi_dell_system_id_t;
```

Description

The `dmi_dell_system_id_t` structure holds the decoded Dell system ID record (type 255), which holds the system ID of the platform in hexadecimal digits, as the revisions and IDs (type 208) and the OEM strings (type 11, key `1`) hold it too, along with an identifier whose meaning is not established. The identifier is replaced when the table is anonymized, see `dmi_anonymize`, since it may identify the system.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_system_id_spec`, of type `DMI_TYPE(dell_system_id)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`identifier`

Identifier whose meaning is not established, beginning with `_SID` in most known data, e.g. `dellDelldellDELL` in the rest. The two bytes past the strings, which most structures carry, are `0` in all known data, and are not decoded.

`system_id`

System ID, in hexadecimal digits.

See also

`dmi_entity_info`, `dmi_dell_revisions`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/dell/token-refs.h>

dmi_dell_token_refs_1

Synopsis

```
#include <opendmi/entity/dell/token-refs.h>

typedef struct dmi_dell_token_refs_1 {
    size_t token_count;
    uint16_t *tokens;
} dmi_dell_token_refs_1_t;
```

Description

The `dmi_dell_token_refs_1_t` structure holds the decoded Dell token references of type 220, which list tokens, most of which the calling interface (type 218) or the indexed I/O access (type 212) of the same table defines, grouped by the device they belong to. The meaning of the groups is not established.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_token_refs_1_spec`, of type `DMI_TYPE(dell_token_refs_1)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`token_count`

Number of the elements of `tokens`, as many as the structure holds: 8 or 9 in the known data.

`tokens`

Tokens, zero for none, an array of `token_count` elements.

See also

`dmi_entity_info`, `dmi_dell_token_refs_2`, `dmi_dell_calling_iface`, `dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

dmi_dell_token_refs_2

Synopsis

```
#include <opendmi/entity/dell/token-refs.h>

typedef struct dmi_dell_token_refs_2 {
    uint8_t unknown_1;
    uint16_t unknown_2;
    uint16_t tokens[6];
} dmi_dell_token_refs_2_t;
```

Description

The `dmi_dell_token_refs_2_t` structure holds the decoded Dell token references of type 221, which list tokens, most of which the calling interface (type 218) or the indexed I/O access (type 212) of the same table defines, following two values whose meaning is not established. Dell gives type 221 to this structure, and the Intel firmware version information is found at type 205 or 237 on most Dell platforms, see `dmi_intel_fvi`. Some of them, e.g. XPS 13 9350, keep it at type 221, where the structure of Dell is told from it by its length of 19 bytes and by carrying no strings.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_token_refs_2_spec`, of type `DMI_TYPE(dell_token_refs_2)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`unknown_1`
: Value whose meaning is not established, `0`, `2` or `3` in the known data.

`unknown_2`
: Value whose meaning is not established, `0` or `1` in the known data.

`tokens`
: Tokens, zero for none.

See also

`dmi_entity_info`, `dmi_dell_token_refs_1`, `dmi_dell_calling_iface`, `dmi_dell_indexed_io`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/dell/video-rom.h>

dmi_dell_video_rom

Synopsis

```
#include <opendmi/entity/dell/video-rom.h>

typedef struct dmi_dell_video_rom {
    const char *vendor;
    const char *version;
    uint8_t unknown_1;
    uint16_t unknown_2;
} dmi_dell_video_rom_t;
```

Description

The `dmi_dell_video_rom_t` structure holds the decoded Dell video BIOS information (type 216), which tells the vendor and the version of the video BIOS.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_dell_video_rom_spec`, of type `DMI_TYPE(dell_video_rom)`, declared in `<opendmi/module/dell.h>`, once the Dell extension module (`dmi_dell_module`) is enabled. It lives as long as the context it has been decoded in.

The layout is reverse engineered from the dumps of the data corpus, since no public source describes the structure, and the members whose meaning is not established are named after their position.

Fields

`vendor`

Vendor of the video BIOS, as the firmware writes it, which may be quoted.

`version`

Version of the video BIOS, as the firmware writes it, which may be quoted.

`unknown_1`

Value whose meaning is not established, `1`, or `0` on the PowerEdge servers of the 13th and 14th generations, in the known data.

`unknown_2`

Value whose meaning is not established, `0`, `0x0088`, `0x0110` or `0x03F0` in the known data.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by Dell firmware, and reverse engineered from the data corpus.

<opendmi/entity/hpe/backplane.h>

dmi_hpe_backplane

Synopsis

```
#include <opendmi/entity/hpe/backplane.h>

typedef struct dmi_hpe_backplane {
    uint8_t i2c_address;
    uint16_t box_number;
    uint16_t nvram_id;
    uint64_t wwid;
    uint8_t bay_count;
    uint8_t a0_bay_count;
    uint8_t a2_bay_count;
    const char *name;
    bool has_legacy_details;
} dmi_hpe_backplane_t;
```

Description

The `dmi_hpe_backplane_t` structure holds the decoded HP/HPE HDD backplane FRU information, which describes a drive backplane: where its FRU is, and the SAS expander and the drive bays it has.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_backplane_spec`, of type `DMI_TYPE(hpe_backplane)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers up to Gen10 Plus. It lives as long as the context it has been decoded in.

Fields the data ends before are zero, and the name `nullptr`.

Fields

`i2c_address`

I2C address of the FRU of the backplane, shifted left by one bit.

`box_number`

Box number of the backplane.

`nvram_id`

NVRAM ID of the backplane.

`wwid`

WWID of the SAS expander, zero if unknown.

`bay_count`

Number of the SAS drive bays.

`a0_bay_count`
: Number of the SAS drive bays behind port 0xA0, deprecated from Gen10 Plus onwards.

`a2_bay_count`
: Number of the SAS drive bays behind port 0xA2, deprecated from Gen10 Plus onwards.

`name`
: Name of the backplane, deprecated from Gen10 Plus onwards.

`has_legacy_details`
: Whether `a0_bay_count`, `a2_bay_count` and `name` are shown, which they are not from Gen10 Plus onwards, as told by the generation of the platform.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/common.h>

dmi_hpe_flag

Synopsis

```
#include <opendmi/entity/hpe/common.h>

typedef enum dmi_hpe_flag {
    DMI_HPE_FLAG_NO     = 0x00,
    DMI_HPE_FLAG_YES    = 0x01,
    DMI_HPE_FLAG_UNSPEC = 0xFF
} dmi_hpe_flag_t;

const char *dmi_hpe_flag_name(dmi_hpe_flag_t value);
```

Description

The `dmi_hpe_flag_t` enumeration tells the value of a flag which the structure may leave undefined, e.g. because the module or the firmware does not support it. It is the type of the flags of `dmi_hpe_dimm_attrs_t` and of `dmi_hpe_inventory_t`.

The `dmi_hpe_flag_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_FLAG_NO`

No.

`DMI_HPE_FLAG_YES`

Yes.

`DMI_HPE_FLAG_UNSPEC`

Not defined.

Return value

`dmi_hpe_flag_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_dimm_attrs`, `dmi_hpe_inventory`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_encryption

Synopsis

```
#include <opendmi/entity/hpe/common.h>

typedef enum dmi_hpe_encryption {
    DMI_HPE_ENCRYPTION_NONE        = 0x00,
    DMI_HPE_ENCRYPTION_ENCRYPTED   = 0x01,
    DMI_HPE_ENCRYPTION_UNKNOWN     = 0x02,
    DMI_HPE_ENCRYPTION_UNSUPPORTED = 0x03
} dmi_hpe_encryption_t;

const char *dmi_hpe_encryption_name(dmi_hpe_encryption_t value);
```

Description

The `dmi_hpe_encryption_t` enumeration tells the encryption status of a memory module or a drive, and is the type of the `encryption` field of `dmi_hpe_dimm_attrs_t` and of `dmi_hpe_drive_t`.

The `dmi_hpe_encryption_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_ENCRYPTION_NONE`
: Not encrypted.

`DMI_HPE_ENCRYPTION_ENCRYPTED`
: Encrypted.

`DMI_HPE_ENCRYPTION_UNKNOWN`
: Unknown.

`DMI_HPE_ENCRYPTION_UNSUPPORTED`
: Not supported.

Return value

`dmi_hpe_encryption_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_dimm_attrs`, `dmi_hpe_drive`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/cru.h>

dmi_hpe_cru

Synopsis

```
#include <opendmi/entity/hpe/cru.h>

typedef struct dmi_hpe_cru {
    dmi_binary_t signature_raw;
    char *signature;
    uint64_t address;
    uint32_t length;
    uint32_t offset;
    uint64_t entry_point;
    bool is_cru;
} dmi_hpe_cru_t;
```

Description

The `dmi_hpe_cru_t` structure holds the decoded HP/HPE 64-bit CRU information, which tells where the 64-bit entry point of the Compaq ROM Utility (CRU) services of the firmware is, which the watchdog timer driver calls.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_cru_spec`, of type `DMI_TYPE(hpe_cru)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The structure is decoded up to Gen8 only, which is the last generation known to have it.

Fields

`signature_raw`
: Signature, as the structure holds it.

`signature`
: Signature, `$CRU` for the CRU services, or `nullptr` if it is not printable. Some servers carry records of other signatures too, e.g. `$SHF`, whose meaning is not known. The string belongs to the structure, and is freed along with it.

`address`
: Physical address of the service area.

`length`
: Length of the service area.

`offset`
: Offset of the entry point within the service area.

`entry_point`
: Physical address of the entry point.

`is_cru`
: Whether the record describes the CRU services, as its signature `$CRU` tells. The address, the length, the offset and the entry point are shown for these records only.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/device-correlation.h>

dmi_hpe_device_correlation

Synopsis

```
#include <opendmi/entity/hpe/device-correlation.h>

typedef struct dmi_hpe_device_correlation {
    dmi_handle_t device_handle;
    dmi_handle_t smbus_handle;
    uint16_t pci_vendor_id;
    uint16_t pci_device_id;
    uint16_t pci_subvendor_id;
    uint16_t pci_subdevice_id;
    uint8_t pci_class;
    uint8_t pci_subclass;
    dmi_handle_t parent_handle;
    bool is_peer_bifurcated;
    bool is_upstream;
    dmi_hpe_device_type_t device_type;
    dmi_hpe_device_location_t device_location;
    uint8_t instance;
    uint8_t sub_instance;
    uint8_t bay;
    uint8_t enclosure;
    const char *uefi_device_path;
    const char *uefi_device_name;
    const char *device_name;
    const char *uefi_location;
    dmi_handle_t physical_handle;
    const char *part_number;
    const char *serial_number;
    uint16_t segment;
    uint8_t bus;
    uint8_t devfn;
    bool has_pci_location;
} dmi_hpe_device_correlation_t;
```

Description

The `dmi_hpe_device_correlation_t` structure holds the decoded HP/HPE device correlation record, which correlates a device with the system slot (type 9) or the onboard device (type 41) it is found at, and with the UEFI boot entries naming it.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_device_correlation_spec`, of type `DMI_TYPE(hpe_device_correlation)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen9 onwards. It lives as long as the context it has been decoded in.

The structure has grown with the generations of the servers: strings the data ends before are `nullptr`, `physical_handle` is `DMI_HANDLE_INVALID`, and the PCI location is zero.

Fields

`device_handle`

Handle of the system slot (type 9) or the onboard device (type 41) structure. Set to `0xFFFE` when not applicable.

`smbus_handle`

Handle of the SMBus segment record (type 228). Set to `0xFFFE` when not applicable.

`pci_vendor_id`

PCI vendor ID of the device. Set to `0xFFFF` when the device is not present.

`pci_device_id`

PCI device ID of the device. Set to `0xFFFF` when the device is not present.

`pci_subvendor_id`

PCI subsystem vendor ID of the device. Set to `0xFFFF` when the device is not present.

`pci_subdevice_id`

PCI subsystem ID of the device. Set to `0xFFFF` when the device is not present.

`pci_class`

PCI class code of the device. Set to `0xFF` when the device is not present.

`pci_subclass`

PCI subclass code of the device. Set to `0xFF` when the device is not present.

`parent_handle`

Handle of the parent device. Set to `0xFFFE` when not applicable. The structure it refers to is not documented, and is presumably the device correlation record of the parent device.

`is_peer_bifurcated`

Whether the device is a peer bifurcated one, whose physical slot `physical_handle` tells.

`is_upstream`

Whether the device is an upstream one.

`device_type`

Type of the device, UEFI only, see `dmi_hpe_device_type`.

`device_location`

Location of the device, see `dmi_hpe_device_location`.

`instance`
Instance of the device.

`sub_instance`
Sub-instance of the device: the port of a NIC, or the bay of an NVMe drive.

`bay`
Bay of the device, zero if unknown. Set to `0xFF` when not shown.

`enclosure`
Enclosure of the device, zero if unknown. Set to `0xFF` when not shown.

`uefi_device_path`
UEFI device path.

`uefi_device_name`
UEFI device structured name.

`device_name`
UEFI device name.

`uefi_location`
UEFI location.

`physical_handle`
Handle of the system slot structure (type 9) of the real physical slot, valid only if `is_peer_bifurcated` is set, and neither shown nor checked otherwise.

`part_number`
Part number of the PCI device.

`serial_number`
Serial number of the PCI device.

`segment`
PCI segment group, zero for a single group topology.

`bus`
PCI bus.

`devfn`
PCI device and function, `(device << 3) | function`.

`has_pci_location`
Whether the structure holds `segment`, `bus` and `devfn`, which the structures shorter than 40 bytes leave out. The fields are not shown then.

See also

`dmi_entity_info`, `dmi_hpe_device_type`, `dmi_hpe_device_location`, `dmi_slot`, `dmi_onboard_device_ex`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_device_type

Synopsis

```
#include <opendmi/entity/hpe/device-correlation.h>

typedef enum dmi_hpe_device_type {
    DMI_HPE_DEVICE_TYPE_UNKNOWN         = 0x00,
    DMI_HPE_DEVICE_TYPE_FLEXIBLE_LOM    = 0x03,
    DMI_HPE_DEVICE_TYPE_EMBEDDED_LOM    = 0x04,
    DMI_HPE_DEVICE_TYPE_SLOT_NIC        = 0x05,
    DMI_HPE_DEVICE_TYPE_STORAGE         = 0x06,
    DMI_HPE_DEVICE_TYPE_SMART_ARRAY     = 0x07,
    DMI_HPE_DEVICE_TYPE_USB_DISK        = 0x08,
    DMI_HPE_DEVICE_TYPE_OTHER_PCI       = 0x09,
    DMI_HPE_DEVICE_TYPE_RAM_DISK        = 0x0A,
    DMI_HPE_DEVICE_TYPE_FIRMWARE_VOLUME = 0x0B,
    DMI_HPE_DEVICE_TYPE_UEFI_SHELL      = 0x0C,
    DMI_HPE_DEVICE_TYPE_USB_BOOT        = 0x0D,
    DMI_HPE_DEVICE_TYPE_DYNAMIC_ARRAY   = 0x0E,
    DMI_HPE_DEVICE_TYPE_FILE            = 0x0F,
    DMI_HPE_DEVICE_TYPE_NVME            = 0x10,
    DMI_HPE_DEVICE_TYPE_NVDIMM          = 0x11,
    DMI_HPE_DEVICE_TYPE_EMBEDDED_GPU    = 0x12
} dmi_hpe_device_type_t;

const char *dmi_hpe_device_type_name(dmi_hpe_device_type_t value);
```

Description

The `dmi_hpe_device_type_t` enumeration tells the type of a device, as UEFI boot entries name it, and is the type of the `device_type` field of `dmi_hpe_device_correlation_t`. Other values may be found in the data, and are kept as they are.

The `dmi_hpe_device_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_DEVICE_TYPE_UNKNOWN`
: Unknown.

`DMI_HPE_DEVICE_TYPE_FLEXIBLE_LOM`
: Flexible LOM.

`DMI_HPE_DEVICE_TYPE_EMBEDDED_LOM`
: Embedded LOM.

`DMI_HPE_DEVICE_TYPE_SLOT_NIC`
: NIC in a slot.

`DMI_HPE_DEVICE_TYPE_STORAGE`
: Storage controller.

`DMI_HPE_DEVICE_TYPE_SMART_ARRAY`
: Smart Array storage controller.

`DMI_HPE_DEVICE_TYPE_USB_DISK`
: USB hard disk.

`DMI_HPE_DEVICE_TYPE_OTHER_PCI`
: Other PCI device.

`DMI_HPE_DEVICE_TYPE_RAM_DISK`
: RAM disk.

`DMI_HPE_DEVICE_TYPE_FIRMWARE_VOLUME`
: Firmware volume.

`DMI_HPE_DEVICE_TYPE_UEFI_SHELL`
: UEFI shell.

`DMI_HPE_DEVICE_TYPE_USB_BOOT`
: Generic UEFI USB boot entry.

`DMI_HPE_DEVICE_TYPE_DYNAMIC_ARRAY`
: Dynamic Smart Array controller.

`DMI_HPE_DEVICE_TYPE_FILE`
: File.

`DMI_HPE_DEVICE_TYPE_NVME`
: NVMe hard drive.

`DMI_HPE_DEVICE_TYPE_NVDIMM`
: NVDIMM.

`DMI_HPE_DEVICE_TYPE_EMBEDDED_GPU`
: Embedded GPU.

Return value

`dmi_hpe_device_type_name` returns the name of the value, translated to the language of the current

locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_device_correlation`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_device_location

Synopsis

```
#include <opendmi/entity/hpe/device-correlation.h>

typedef enum dmi_hpe_device_location {
    DMI_HPE_DEVICE_LOCATION_UNKNOWN          = 0x00,
    DMI_HPE_DEVICE_LOCATION_EMBEDDED         = 0x01,
    DMI_HPE_DEVICE_LOCATION_ILO_MEDIA        = 0x02,
    DMI_HPE_DEVICE_LOCATION_FRONT_USB        = 0x03,
    DMI_HPE_DEVICE_LOCATION_REAR_USB         = 0x04,
    DMI_HPE_DEVICE_LOCATION_INTERNAL_USB     = 0x05,
    DMI_HPE_DEVICE_LOCATION_INTERNAL_SD      = 0x06,
    DMI_HPE_DEVICE_LOCATION_VIRTUAL_USB      = 0x07,
    DMI_HPE_DEVICE_LOCATION_EMBEDDED_SATA    = 0x08,
    DMI_HPE_DEVICE_LOCATION_EMBEDDED_ARRAY   = 0x09,
    DMI_HPE_DEVICE_LOCATION_PCI_SLOT         = 0x0A,
    DMI_HPE_DEVICE_LOCATION_RAM              = 0x0B,
    DMI_HPE_DEVICE_LOCATION_USB              = 0x0C,
    DMI_HPE_DEVICE_LOCATION_DYNAMIC_ARRAY    = 0x0D,
    DMI_HPE_DEVICE_LOCATION_URL              = 0x0E,
    DMI_HPE_DEVICE_LOCATION_NVME_BAY         = 0x0F,
    DMI_HPE_DEVICE_LOCATION_NVDIMM_PROCESSOR = 0x10,
    DMI_HPE_DEVICE_LOCATION_NVDIMM_BOARD     = 0x11,
    DMI_HPE_DEVICE_LOCATION_NVME_RISER       = 0x12,
    DMI_HPE_DEVICE_LOCATION_NVDIMM_NAMESPACE = 0x13,
    DMI_HPE_DEVICE_LOCATION_VROC_SATA        = 0x14,
    DMI_HPE_DEVICE_LOCATION_VROC_NVME        = 0x15
} dmi_hpe_device_location_t;
```

```
const char *dmi_hpe_device_location_name(dmi_hpe_device_location_t value);
```

Description

The `dmi_hpe_device_location_t` enumeration tells the location of a device, and is the type of the `device_location` field of `dmi_hpe_device_correlation_t`. Other values may be found in the data, and are kept as they are.

The `dmi_hpe_device_location_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_DEVICE_LOCATION_UNKNOWN`
: Unknown.

`DMI_HPE_DEVICE_LOCATION_EMBEDDED`
: Embedded.

`DMI_HPE_DEVICE_LOCATION_ILO_MEDIA`
: iLO virtual media.

`DMI_HPE_DEVICE_LOCATION_FRONT_USB`
: Front USB port.

`DMI_HPE_DEVICE_LOCATION_REAR_USB`
: Rear USB port.

`DMI_HPE_DEVICE_LOCATION_INTERNAL_USB`
: Internal USB.

`DMI_HPE_DEVICE_LOCATION_INTERNAL_SD`
: Internal SD card.

`DMI_HPE_DEVICE_LOCATION_VIRTUAL_USB`
: Internal virtual USB (embedded NAND).

`DMI_HPE_DEVICE_LOCATION_EMBEDDED_SATA`
: Embedded SATA port.

`DMI_HPE_DEVICE_LOCATION_EMBEDDED_ARRAY`
: Embedded Smart Array.

`DMI_HPE_DEVICE_LOCATION_PCI_SLOT`
: PCI slot.

`DMI_HPE_DEVICE_LOCATION_RAM`
: RAM memory.

`DMI_HPE_DEVICE_LOCATION_USB`
: USB.

`DMI_HPE_DEVICE_LOCATION_DYNAMIC_ARRAY`
: Dynamic Smart Array controller.

`DMI_HPE_DEVICE_LOCATION_URL`
: URL.

`DMI_HPE_DEVICE_LOCATION_NVME_BAY`
: NVMe drive bay.

`DMI_HPE_DEVICE_LOCATION_NVDIMM_PROCESSOR`
: NVDIMM processor.

`DMI_HPE_DEVICE_LOCATION_NVDIMM_BOARD`
: NVDIMM board.

`DMI_HPE_DEVICE_LOCATION_NVME_RISER`
: NVMe riser.

`DMI_HPE_DEVICE_LOCATION_NVDIMM_NAMESPACE`
: NVDIMM namespace.

`DMI_HPE_DEVICE_LOCATION_VROC_SATA`
: VROC SATA.

`DMI_HPE_DEVICE_LOCATION_VROC_NVME`
: VROC NVMe.

Return value

`dmi_hpe_device_location_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_device_correlation`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/dimm-attrs.h>

dmi_hpe_dimm_attrs

Synopsis

```
#include <opendmi/entity/hpe/dimm-attrs.h>

typedef struct dmi_hpe_dimm_attrs {
    dmi_handle_t device_handle;
    uint32_t attributes;
    dmi_hpe_flag_t smart_memory;
    dmi_hpe_flag_t load_reduced;
    dmi_hpe_flag_t standard_memory;
    uint16_t minimum_voltage;
    uint16_t configured_voltage;
    bool is_config_error;
    bool is_training_error;
    dmi_hpe_encryption_t encryption;
} dmi_hpe_dimm_attrs_t;
```

Description

The `dmi_hpe_dimm_attrs_t` structure holds the decoded HP/HPE DIMM attributes record, which tells what the memory device structure (type 17) of a DIMM socket does not: whether the module is an HPE one, and the voltages it runs at. There is a record for each socket.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_dimm_attrs_spec`, of type `DMI_TYPE(hpe_dimm_attrs)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen9 onwards. It lives as long as the context it has been decoded in.

Fields

`device_handle`

Handle of the memory device structure (type 17).

`attributes`

Attributes of the module, as the structure holds them.

`smart_memory`

Whether the module is HPE SmartMemory, as bits 0 and 1 of the attributes tell, see `dmi_hpe_flag`.

`load_reduced`

Whether the module is a load reduced one (LRDIMM), as bits 2 and 3 of the attributes tell, see `dmi_hpe_flag`.

`standard_memory`
: Whether the module is HPE standard memory, as bits 4 and 5 of the attributes tell, see `dmi_hpe_flag`.

`minimum_voltage`
: Minimum operating voltage in millivolts, zero if unknown.

`configured_voltage`
: Configured operating voltage in millivolts, zero if unknown.

`is_config_error`
: Whether the module is mapped out because of a configuration error.

`is_training_error`
: Whether the module is mapped out because of a training error.

`encryption`
: Encryption status of the module, see `dmi_hpe_encryption`. Set to `UINT8_MAX` when the structure holds none.

See also

`dmi_entity_info`, `dmi_hpe_flag`, `dmi_hpe_encryption`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/dimm-config.h>

dmi_hpe_dimm_config

Synopsis

```
#include <opendmi/entity/hpe/dimm-config.h>

typedef struct dmi_hpe_dimm_config {
    dmi_handle_t device_handle;
    uint8_t region_id;
    bool is_volatile;
    bool is_byte_accessible;
    bool is_block_io;
    uint64_t raw_size;
    uint64_t size;
    uint8_t passphrase_state;
    bool is_passphrase_enabled;
    uint16_t interleave_set;
    uint8_t interleave_dimm_count;
    dmi_hpe_interleave_health_t interleave_health;
} dmi_hpe_dimm_config_t;
```

Description

The `dmi_hpe_dimm_config_t` structure holds the decoded HP/HPE DIMM current configuration record, which describes a memory region configured on a module of a memory device (type 17), e.g. of a persistent memory one, and the interleave set it belongs to. There is a record for each region.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_dimm_config_spec`, of type `DMI_TYPE(hpe_dimm_config)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`device_handle`

Handle of the memory device structure (type 17).

`region_id`

ID of the region, unique on the module.

`is_volatile`

Whether the region is volatile, its data lost on a power cycle.

`is_byte_accessible`

Whether the region is byte accessible persistent memory, its data kept through a reset.

`is_block_io`
: Whether the region is block I/O persistent memory.

`raw_size`
: Size of the region in mebibytes, as the structure holds it.

`size`
: Size of the region in bytes.

`passphrase_state`
: State of the passphrase, as the structure holds it.

`is_passphrase_enabled`
: Whether a passphrase is required, byte accessible persistent regions only: any state other than zero.

`interleave_set`
: 1-based index of the interleave set of the region, unique per set. Regions of the same index are interleaved.

`interleave_dimm_count`
: Number of the DIMMs of the interleave set.

`interleave_health`
: Health of the interleave set, see `dmi_hpe_interleave_health`, or `DMI_HPE_INTERLEAVE_HEALTH_ABSENT` if the structure is too short to hold it.

See also

`dmi_entity_info`, `dmi_hpe_interleave_health`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_interleave_health

Synopsis

```
#include <opendmi/entity/hpe/dimm-config.h>

typedef enum dmi_hpe_interleave_health {
    DMI_HPE_INTERLEAVE_HEALTH_HEALTHY      = 0x00,
    DMI_HPE_INTERLEAVE_HEALTH_DIMM_MISSING = 0x01,
    DMI_HPE_INTERLEAVE_HEALTH_INACTIVE     = 0x02,
    DMI_HPE_INTERLEAVE_HEALTH_SPA_MISSING  = 0x03,
    DMI_HPE_INTERLEAVE_HEALTH_NEW_GOAL     = 0x04,
    DMI_HPE_INTERLEAVE_HEALTH_LOCKED       = 0x05,
    DMI_HPE_INTERLEAVE_HEALTH_ABSENT       = 0x100
```

```
} dmi_hpe_interleave_health_t;

const char *dmi_hpe_interleave_health_name(dmi_hpe_interleave_health_t value);
```

Description

The `dmi_hpe_interleave_health_t` enumeration tells the health of an interleave set of memory regions, and is the type of the `interleave_health` field of `dmi_hpe_dimm_config_t`. Values 0x06 to 0xFF are reserved.

The `dmi_hpe_interleave_health_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_INTERLEAVE_HEALTH_HEALTHY`
: Healthy.

`DMI_HPE_INTERLEAVE_HEALTH_DIMM_MISSING`
: DIMM missing.

`DMI_HPE_INTERLEAVE_HEALTH_INACTIVE`
: Configuration inactive.

`DMI_HPE_INTERLEAVE_HEALTH_SPA_MISSING`
: SPA missing.

`DMI_HPE_INTERLEAVE_HEALTH_NEW_GOAL`
: New goal.

`DMI_HPE_INTERLEAVE_HEALTH_LOCKED`
: Locked.

`DMI_HPE_INTERLEAVE_HEALTH_ABSENT`
: The structure is too short to hold the health. It is no value the field may hold, and is shown as unspecified.

Return value

`dmi_hpe_interleave_health_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. Values 0x06 to 0xFF are named "Reserved". The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_dimm_config`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/dimm-location.h>

dmi_hpe_dimm_location

Synopsis

```
#include <opendmi/entity/hpe/dimm-location.h>

typedef struct dmi_hpe_dimm_location {
    dmi_handle_t device_handle;
    uint8_t board;
    uint8_t dimm;
    uint8_t processor;
    uint8_t logical_dimm;
    const char *uefi_device_path;
    const char *uefi_device_name;
    const char *device_name;
    uint8_t controller;
    uint8_t channel;
    uint8_t ie_dimm;
    uint8_t ie_pldm_id;
    uint16_t vendor_id;
    uint16_t device_id;
    uint16_t controller_vendor_id;
    uint16_t controller_device_id;
    uint8_t interleave;
    const char *part_number;
    uint8_t channel_index;
    bool is_system_board;
    bool has_ie;
    bool has_channel_index;
} dmi_hpe_dimm_location_t;
```

Description

The `dmi_hpe_dimm_location_t` structure holds the decoded HP/HPE DIMM location record, which tells where the DIMM socket of a memory device (type 17) is: the memory board, the processor and the channel it belongs to. There is a record for each socket, including the ones of the memory boards which are not installed.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_dimm_location_spec`, of type `DMI_TYPE(hpe_dimm_location)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The structure has grown with the generations of the servers: fields the data ends before are set to the values they have when not specified, and strings to `nullptr`.

Fields

`device_handle`

Handle of the memory device structure (type 17).

`board`

1-based number of the memory board. Set to `UINT8_MAX` for the sockets of the system board, and not shown then.

`dimm`

1-based number of the DIMM.

`processor`

1-based number of the processor. Set to `UINT8_MAX` when not shown.

`logical_dimm`

1-based logical number of the DIMM, as ACPI numbers it, zero if unspecified.

`uefi_device_path`

UEFI device path.

`uefi_device_name`

UEFI device structured name.

`device_name`

Device name.

`controller`

1-based number of the memory controller, zero if unknown.

`channel`

1-based number of the memory channel, as the silk screen names it, zero if unknown.

`ie_dimm`

0-based number of the DIMM, as the Innovation Engine reports it. Set to `UINT8_MAX` when not supported. Reserved from Gen12 onwards, and not shown then.

`ie_pldm_id`

Sensor ID of the Innovation Engine. Set to `UINT8_MAX` when not supported. Reserved from Gen12 onwards, and not shown then.

`vendor_id`

Manufacturer ID code of the module, as the SPD gives it.

`device_id`

Product ID code of the module, NVDIMMs only.

`controller_vendor_id`

Manufacturer ID code of the controller, NVDIMMs only, as the SPD gives it: the number of continuation codes in the low byte, whose bit 7 is a parity bit, and the JEP106 code in the high byte.

`controller_device_id`
: Product ID code of the controller, NVDIMMs only.

`interleave`
: 1-based interleave set within the processor, zero if unknown.

`part_number`
: HPE part number, from the OEM area of the SPD.

`channel_index`
: 0-based index of the DIMM within its channel. Set to `UINT8_MAX` when the structure holds none.

`is_system_board`
: Whether the socket is one of the system board, which `board` of `UINT8_MAX` tells.

`has_ie`
: Whether `ie_dimm` and `ie_pldm_id` are shown, which they are up to Gen11.

`has_channel_index`
: Whether the structure holds `channel_index`, which the structures shorter than 28 bytes leave out.

See also

`dmi_entity_info`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/dimm-vendor.h>

dmi_hpe_dimm_vendor

Synopsis

```
#include <opendmi/entity/hpe/dimm-vendor.h>

typedef struct dmi_hpe_dimm_vendor {
    dmi_handle_t device_handle;
    const char *manufacturer;
    const char *part_number;
    const char *serial_number;
    uint8_t manufacture_year;
    uint8_t manufacture_week;
} dmi_hpe_dimm_vendor_t;
```

Description

The `dmi_hpe_dimm_vendor_t` structure holds the decoded HP/HPE DIMM vendor information, which tells the actual manufacturer of the module of a memory device (type 17), whose standard fields hold the ones of HPE.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_dimm_vendor_spec`, of type `DMI_TYPE(hpe_dimm_vendor)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen9 onwards. It lives as long as the context it has been decoded in.

Fields

`device_handle`

Handle of the memory device structure (type 17).

`manufacturer`

Manufacturer of the module.

`part_number`

Part number of the module, as its manufacturer gives it.

`serial_number`

Serial number of the module, as its manufacturer gives it.

`manufacture_year`

Last two digits of the year the module was manufactured in, from 2000, zero if unknown.

`manufacture_week`

Week of the year the module was manufactured in, zero if unknown.

See also

`dmi_entity_info`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/drive.h>

dmi_hpe_drive

Synopsis

```
#include <opendmi/entity/hpe/drive.h>

typedef struct dmi_hpe_drive {
    dmi_handle_t correlation_handle;
    dmi_hpe_drive_type_t drive_type;
    uint64_t unique_id;
    uint32_t legacy_capacity;
    uint64_t power_on_hours;
    uint8_t power;
    dmi_hpe_drive_form_factor_t form_factor;
    dmi_hpe_drive_health_t health;
    const char *serial_number;
    const char *model_number;
    const char *firmware_revision;
    const char *location;
    dmi_hpe_encryption_t encryption;
    uint64_t capacity;
    uint32_t block_size;
    uint16_t rotation_speed;
    uint16_t negotiated_speed;
    uint16_t capable_speed;
} dmi_hpe_drive_t;
```

Description

The `dmi_hpe_drive_t` structure holds the decoded HP/HPE hard drive inventory record, which describes an NVMe or SATA drive attached to the system directly, which the firmware has found at boot. Drives behind a storage controller, e.g. a Smart Array one, and the ones plugged in later are not described.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_drive_spec`, of type `DMI_TYPE(hpe_drive)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen10 onwards. It lives as long as the context it has been decoded in.

The older structures hold the capacity in megabytes only, and `capacity` is then zero.

Fields

`correlation_handle`
: Handle of the device correlation record (type 203).

`drive_type`

Type of the drive, see `dmi_hpe_drive_type`.

`unique_id`

Unique ID of the drive: WWID of SATA drives, and IEEE extended unique identifier of NVMe ones.

`legacy_capacity`

Capacity of the drive in megabytes.

`power_on_hours`

Number of the hours the drive has been powered on for.

`power`

Power of the drive in watts, zero if unknown.

`form_factor`

Form factor of the drive, see `dmi_hpe_drive_form_factor`.

`health`

Health status of the drive, see `dmi_hpe_drive_health`.

`serial_number`

Serial number of the drive.

`model_number`

Model number of the drive.

`firmware_revision`

Firmware revision of the drive.

`location`

Location of the drive.

`encryption`

Encryption status of the drive, as the firmware tells it, see `dmi_hpe_encryption`. Set to `UINT8_MAX` when the structure holds none.

`capacity`

Capacity of the drive in bytes, zero if the structure holds none.

`block_size`

Size of a logical block in bytes, zero if the structure holds none.

`rotation_speed`

Nominal rotation speed in RPM: zero if not reported, and one for solid state drives.

`negotiated_speed`

Negotiated speed of the bus in Gbit/s, zero if unknown.

`capable_speed`

Fastest speed of the bus the drive is capable of in Gbit/s, zero if unknown.

See also

`dmi_entity_info`, `dmi_hpe_drive_type`, `dmi_hpe_drive_form_factor`, `dmi_hpe_drive_health`, `dmi_hpe_encryption`, `dmi_hpe_device_correlation`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_drive_type

Synopsis

```
#include <opendmi/entity/hpe/drive.h>

typedef enum dmi_hpe_drive_type {
    DMI_HPE_DRIVE_TYPE_UNDETERMINED = 0x00,
    DMI_HPE_DRIVE_TYPE_NVME_SSD     = 0x01,
    DMI_HPE_DRIVE_TYPE_SATA         = 0x02,
    DMI_HPE_DRIVE_TYPE_SAS          = 0x03,
    DMI_HPE_DRIVE_TYPE_SATA_SSD     = 0x04,
    DMI_HPE_DRIVE_TYPE_NVME_VROC    = 0x05
} dmi_hpe_drive_type_t;

const char *dmi_hpe_drive_type_name(dmi_hpe_drive_type_t value);
```

Description

The `dmi_hpe_drive_type_t` enumeration tells the type of a drive, and is the type of the `drive_type` field of `dmi_hpe_drive_t`.

The `dmi_hpe_drive_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_DRIVE_TYPE_UNDETERMINED`
: Undetermined.

`DMI_HPE_DRIVE_TYPE_NVME_SSD`
: NVMe SSD.

`DMI_HPE_DRIVE_TYPE_SATA`
: SATA.

`DMI_HPE_DRIVE_TYPE_SAS`
: SAS.

`DMI_HPE_DRIVE_TYPE_SATA_SSD`
: SATA SSD.

`DMI_HPE_DRIVE_TYPE_NVME_VROC`
: NVMe managed by VROC/VMD.

Return value

`dmi_hpe_drive_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_drive`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_drive_form_factor

Synopsis

```
#include <opendmi/entity/hpe/drive.h>

typedef enum dmi_hpe_drive_form_factor {
    DMI_HPE_DRIVE_FORM_FACTOR_3_5        = 0x02,
    DMI_HPE_DRIVE_FORM_FACTOR_2_5        = 0x03,
    DMI_HPE_DRIVE_FORM_FACTOR_1_8        = 0x04,
    DMI_HPE_DRIVE_FORM_FACTOR_SMALLER    = 0x05,
    DMI_HPE_DRIVE_FORM_FACTOR_MSATA      = 0x06,
    DMI_HPE_DRIVE_FORM_FACTOR_M2         = 0x07,
    DMI_HPE_DRIVE_FORM_FACTOR_MICRO_SSD  = 0x08,
    DMI_HPE_DRIVE_FORM_FACTOR_CFAST      = 0x09,
    DMI_HPE_DRIVE_FORM_FACTOR_EDSFF      = 0x20,
    DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_1U_S = 0x21,
    DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_1U_L = 0x22,
    DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_E3_S = 0x23,
    DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_E3_L = 0x24
} dmi_hpe_drive_form_factor_t;

const char *dmi_hpe_drive_form_factor_name(dmi_hpe_drive_form_factor_t value);
```

Description

The `dmi_hpe_drive_form_factor_t` enumeration tells the form factor of a drive, and is the type of the `form_factor` field of `dmi_hpe_drive_t`.

The `dmi_hpe_drive_form_factor_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_DRIVE_FORM_FACTOR_3_5`
: 3.5-inch form factor.

`DMI_HPE_DRIVE_FORM_FACTOR_2_5`
: 2.5-inch form factor.

`DMI_HPE_DRIVE_FORM_FACTOR_1_8`
: 1.8-inch form factor.

`DMI_HPE_DRIVE_FORM_FACTOR_SMALLER`
: Smaller than 1.8-inch form factor.

`DMI_HPE_DRIVE_FORM_FACTOR_MSATA`
: mSATA.

`DMI_HPE_DRIVE_FORM_FACTOR_M2`
: M.2.

`DMI_HPE_DRIVE_FORM_FACTOR_MICRO_SSD`
: MicroSSD.

`DMI_HPE_DRIVE_FORM_FACTOR_CFAST`
: CFast.

`DMI_HPE_DRIVE_FORM_FACTOR_EDSFF`
: EDSFF of unknown form factor.

`DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_1U_S`
: EDSFF 1U short.

`DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_1U_L`
: EDSFF 1U long.

`DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_E3_S`
: EDSFF E3 short.

`DMI_HPE_DRIVE_FORM_FACTOR_EDSFF_E3_L`
: EDSFF E3 long.

Return value

`dmi_hpe_drive_form_factor_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_drive`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_drive_health

Synopsis

```
#include <opendmi/entity/hpe/drive.h>

typedef enum dmi_hpe_drive_health {
    DMI_HPE_DRIVE_HEALTH_OK       = 0x00,
    DMI_HPE_DRIVE_HEALTH_WARNING  = 0x01,
    DMI_HPE_DRIVE_HEALTH_CRITICAL = 0x02,
    DMI_HPE_DRIVE_HEALTH_UNKNOWN  = 0xFF
} dmi_hpe_drive_health_t;

const char *dmi_hpe_drive_health_name(dmi_hpe_drive_health_t value);
```

Description

The `dmi_hpe_drive_health_t` enumeration tells the health status of a drive, and is the type of the `health` field of `dmi_hpe_drive_t`.

The `dmi_hpe_drive_health_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_DRIVE_HEALTH_OK`
: OK.

`DMI_HPE_DRIVE_HEALTH_WARNING`
: Warning.

`DMI_HPE_DRIVE_HEALTH_CRITICAL`
: Critical.

`DMI_HPE_DRIVE_HEALTH_UNKNOWN`
: Unknown.

Return value

`dmi_hpe_drive_health_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_drive`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/extension-board.h>

dmi_hpe_riser

Synopsis

```
#include <opendmi/entity/hpe/extension-board.h>

typedef struct dmi_hpe_riser {
    dmi_hpe_board_type_t board_type;
    dmi_hpe_riser_position_t position;
    uint8_t riser_id;
    uint8_t cpld_version;
    bool is_cpld_b_release;
    const char *name;
} dmi_hpe_riser_t;
```

Description

The `dmi_hpe_riser_t` structure holds the decoded HP/HPE extension board inventory record of a PCIe riser, which ML, DL and Alletra servers have from Gen11 onwards. It is returned by `dmi_entity_info` for the structures of board type `DMI_HPE_BOARD_TYPE_RISER`.

The structures are decoded by `dmi_hpe_riser_spec` and `dmi_hpe_mhs_riser_spec`, of type `DMI_TYPE(hpe_riser)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. The board type at offset `0x04` selects the specification, and the structures of other board types are not decoded. The structure lives as long as the context it has been decoded in.

Fields

`board_type`
: Type of the board, `DMI_HPE_BOARD_TYPE_RISER`, see `dmi_hpe_board_type`.

`position`
: Position of the riser, see `dmi_hpe_riser_position`.

`riser_id`
: ID of the riser.

`cpld_version`
: Version of the CPLD of the riser, bits 0 to 6 of its field, zero if the riser has no CPLD.

`is_cpld_b_release`
: Whether the version of the CPLD is a `B.` release, bit 7 of its field.

`name`

Name of the riser.

See also

`dmi_entity_info`, `dmi_hpe_mhs_riser`, `dmi_hpe_board_type`, `dmi_hpe_riser_position`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_mhs_riser

Synopsis

```
#include <opendmi/entity/hpe/extension-board.h>

typedef struct dmi_hpe_mhs_riser {
    dmi_hpe_board_type_t board_type;
    uint8_t riser_id;
    uint8_t firmware_major;
    uint8_t firmware_minor;
    bool is_downgradable;
    const char *name;
    uint8_t slot_total;
    size_t slot_count;
    uint8_t *slot_ids;
} dmi_hpe_mhs_riser_t;
```

Description

The `dmi_hpe_mhs_riser_t` structure holds the decoded HP/HPE extension board inventory record of a PCIe riser of an MHS platform. It is returned by `dmi_entity_info` for the structures of board type `DMI_HPE_BOARD_TYPE_MHS_RISER`.

The structures are decoded by `dmi_hpe_riser_spec` and `dmi_hpe_mhs_riser_spec`, of type `DMI_TYPE(hpe_mhs_riser)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. The board type at offset `0x04` selects the specification, and the structures of other board types are not decoded. The structure lives as long as the context it has been decoded in.

Fields

`board_type`

Type of the board, `DMI_HPE_BOARD_TYPE_MHS_RISER`, see `dmi_hpe_board_type`.

`riser_id`
: ID of the riser.

`firmware_major`
: Major version of the firmware of the riser, zero if the riser has no firmware.

`firmware_minor`
: Minor version of the firmware of the riser.

`is_downgradable`
: Whether the firmware of the riser may be downgraded.

`name`
: Name of the riser.

`slot_total`
: Number of the slots the structure declares.

`slot_count`
: Number of elements of `slot_ids`, which is smaller than `slot_total` when the data ends before the last slot.

`slot_ids`
: IDs of the slots of the riser. May be `nullptr` when `slot_count` is 0.

See also

`dmi_entity_info`, `dmi_hpe_riser`, `dmi_hpe_board_type`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_board_type

Synopsis

```
#include <opendmi/entity/hpe/extension-board.h>

typedef enum dmi_hpe_board_type {
    DMI_HPE_BOARD_TYPE_RISER     = 0x00,
    DMI_HPE_BOARD_TYPE_MHS_RISER = 0x01
} dmi_hpe_board_type_t;

const char *dmi_hpe_board_type_name(dmi_hpe_board_type_t value);
```

Description

The `dmi_hpe_board_type_t` enumeration tells the type of an extension board, which selects the layout of the rest of the structure, and is the type of the `board_type` field of `dmi_hpe_riser_t` and of `dmi_hpe_mhs_riser_t`.

The `dmi_hpe_board_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_BOARD_TYPE_RISER`
: PCIe riser.

`DMI_HPE_BOARD_TYPE_MHS_RISER`
: PCIe riser of an MHS platform.

Return value

`dmi_hpe_board_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_riser`, `dmi_hpe_mhs_riser`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_riser_position

Synopsis

```
#include <opendmi/entity/hpe/extension-board.h>

typedef enum dmi_hpe_riser_position {
    DMI_HPE_RISER_POSITION_PRIMARY    = 0x01,
    DMI_HPE_RISER_POSITION_SECONDARY  = 0x02,
    DMI_HPE_RISER_POSITION_TERTIARY   = 0x03,
    DMI_HPE_RISER_POSITION_QUATERNARY = 0x04,
    DMI_HPE_RISER_POSITION_FRONT      = 0x0A
} dmi_hpe_riser_position_t;
```

```
const char *dmi_hpe_riser_position_name(dmi_hpe_riser_position_t value);
```

Description

The `dmi_hpe_riser_position_t` enumeration tells the position of a PCIe riser, and is the type of the `position` field of `dmi_hpe_riser_t`.

The `dmi_hpe_riser_position_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_RISER_POSITION_PRIMARY`
: Primary.

`DMI_HPE_RISER_POSITION_SECONDARY`
: Secondary.

`DMI_HPE_RISER_POSITION_TERTIARY`
: Tertiary.

`DMI_HPE_RISER_POSITION_QUATERNARY`
: Quaternary.

`DMI_HPE_RISER_POSITION_FRONT`
: Front.

Return value

`dmi_hpe_riser_position_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_riser`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/inventory.h>

dmi_hpe_inventory

Synopsis

```
#include <opendmi/entity/hpe/inventory.h>

typedef struct dmi_hpe_inventory {
    dmi_handle_t correlation_handle;
    uint32_t package_version;
    const char *version_string;
    uint64_t image_size;
    uint64_t attrs_defined;
    uint64_t attrs_set;
    dmi_hpe_flag_t is_updatable;
    dmi_hpe_flag_t is_reset_required;
    dmi_hpe_flag_t is_auth_required;
    dmi_hpe_flag_t is_in_use;
    dmi_hpe_flag_t is_uefi_image;
    uint32_t lowest_version;
} dmi_hpe_inventory_t;
```

Description

The `dmi_hpe_inventory_t` structure holds the decoded HP/HPE firmware inventory record, which tells the version of the firmware of a device which reports it through its UEFI driver.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_inventory_spec`, of type `DMI_TYPE(hpe_inventory)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The attributes of the firmware image are defined by the bits of `attrs_defined`, and set by the ones of `attrs_set`; a flag is `DMI_HPE_FLAG_UNSPEC` when the record does not define it, see `dmi_hpe_flag`.

Fields

`correlation_handle`

Handle of the device correlation record (type 203).

`package_version`

Version of the package of all the firmware of the device.

`version_string`

Version of the firmware, as a string.

`image_size`
: Size of the firmware image in bytes, zero if not available.

`attrs_defined`
: Mask of the attributes the record defines: bit 0 updatable, bit 1 reset required, bit 2 authentication required, bit 3 in use, bit 4 UEFI image.

`attrs_set`
: Mask of the defined attributes which are set.

`is_updatable`
: Whether the firmware is updatable.

`is_reset_required`
: Whether a reset is required after an update.

`is_auth_required`
: Whether an update requires authentication.

`is_in_use`
: Whether the firmware image is in use.

`is_uefi_image`
: Whether the firmware image is a UEFI one.

`lowest_version`
: Lowest version the firmware may be downgraded to, zero if not available.

See also

`dmi_entity_info`, `dmi_hpe_flag`, `dmi_hpe_device_correlation`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/microcode.h>

dmi_hpe_microcode

Synopsis

```
#include <opendmi/entity/hpe/microcode.h>

typedef struct dmi_hpe_microcode {
    size_t patch_count;
    dmi_hpe_microcode_patch_t *patches;
} dmi_hpe_microcode_t;
```

Description

The `dmi_hpe_microcode_t` structure holds the decoded HP/HPE CPU microcode patch support information, which lists the CPU microcode patches the firmware carries.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_microcode_spec`, of type `DMI_TYPE(hpe_microcode)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Patches run to the end of the structure, which carries no number of them of its own.

Fields

`patch_count`
: Number of patches.

`patches`
: Patches, see `dmi_hpe_microcode_patch`. May be `nullptr` when `patch_count` is 0.

See also

`dmi_entity_info`, `dmi_hpe_microcode_patch`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_microcode_patch

Synopsis

```
#include <opendmi/entity/hpe/microcode.h>
```

```
typedef struct dmi_hpe_microcode_patch {
    uint32_t patch_id;
    uint32_t raw_date;
    dmi_date_t date;
    uint32_t cpuid;
    uint32_t signature;
} dmi_hpe_microcode_patch_t;
```

Description

The `dmi_hpe_microcode_patch_t` structure describes a CPU microcode patch the firmware carries, as an element of the `patches` field of `dmi_hpe_microcode_t`.

Fields

`patch_id`
: Patch ID, which is the microcode revision.

`raw_date`
: Release date of the patch, as the microcode header gives it: the month, the day and the year in binary-coded decimal.

`date`
: Release date of the patch.

`cpuid`
: Processor signature (CPUID leaf 1 EAX) the patch applies to, as the structure holds it. AMD platforms leave out the byte holding the base family, bits 8 to 15, and move the bytes above it down by one.

`signature`
: Processor signature the patch applies to. On the platforms whose processors are of AMD, the byte left out of `cpuid` is put back with the base family of 15, the way dmidecode does, which is valid for the families of 15 onwards only. On other platforms, the same as `cpuid`.

See also

`dmi_hpe_microcode`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/nic.h>

dmi_hpe_nic_info

Synopsis

```
#include <opendmi/entity/hpe/nic.h>

typedef struct dmi_hpe_nic_info {
    size_t port_count;
    dmi_hpe_nic_port_t *ports;
} dmi_hpe_nic_info_t;
```

Description

The `dmi_hpe_nic_info_t` structure holds the decoded HP/HPE BIOS PXE NIC PCI and MAC information, and BIOS iSCSI NIC PCI and MAC information, which list the network ports the firmware boots from over PXE or iSCSI.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_pxe_nic_spec`, of type `DMI_TYPE(hpe_pxe_nic)`, and by `dmi_hpe_iscsi_nic_spec`, of type `DMI_TYPE(hpe_iscsi_nic)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Ports run to the end of the structure, which carries no number of them of its own. The iSCSI ports are listed up to G7 only: type 221 is deprecated from Gen8 onwards, and structures of that type are not decoded on the newer servers, the Intel firmware version information included.

Fields

`port_count`

Number of ports.

`ports`

Ports, see `dmi_hpe_nic_port`. May be `nullptr` when `port_count` is 0.

See also

`dmi_entity_info`, `dmi_hpe_nic_port`, `dmi_intel_fvi`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_nic_port

Synopsis

```
#include <opendmi/entity/hpe/nic.h>

typedef struct dmi_hpe_nic_port {
    uint8_t devfn;
    uint8_t bus;
    dmi_binary_t mac_address;
    dmi_hpe_nic_state_t state;
} dmi_hpe_nic_port_t;
```

Description

The `dmi_hpe_nic_port_t` structure describes a network port the firmware boots from, as an element of the `ports` field of `dmi_hpe_nic_info_t`.

Fields

`devfn`
: PCI device and function, `(device << 3) | function`.

`bus`
: PCI bus.

`mac_address`
: MAC address.

`state`
: State of the port, see `dmi_hpe_nic_state`: disabled if the bus and the device are zero, and not installed if they have all bits set.

See also

`dmi_hpe_nic_info`, `dmi_hpe_nic_state`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_nic_state

Synopsis

```
#include <opendmi/entity/hpe/nic.h>
```

```
typedef enum dmi_hpe_nic_state {
    DMI_HPE_NIC_STATE_INSTALLED     = 0x00,
    DMI_HPE_NIC_STATE_DISABLED      = 0x01,
    DMI_HPE_NIC_STATE_NOT_INSTALLED = 0x02
} dmi_hpe_nic_state_t;

const char *dmi_hpe_nic_state_name(dmi_hpe_nic_state_t value);
```

Description

The `dmi_hpe_nic_state_t` enumeration tells the state of a network port the firmware boots from, as the PCI bus and device of the port tell it, and is the type of the `state` field of `dmi_hpe_nic_port_t`.

The `dmi_hpe_nic_state_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_NIC_STATE_INSTALLED`
: Installed.

`DMI_HPE_NIC_STATE_DISABLED`
: Disabled.

`DMI_HPE_NIC_STATE_NOT_INSTALLED`
: Not installed.

Return value

`dmi_hpe_nic_state_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_nic_port`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/nic-mac.h>

dmi_hpe_nic_mac

Synopsis

```
#include <opendmi/entity/hpe/nic-mac.h>

typedef struct dmi_hpe_nic_mac {
    uint16_t segment;
    uint8_t bus;
    uint8_t devfn;
    dmi_binary_t mac_address;
    dmi_hpe_nic_state_t state;
    uint8_t port;
    const char *uefi_device_path;
} dmi_hpe_nic_mac_t;
```

Description

The `dmi_hpe_nic_mac_t` structure holds the decoded HP/HPE NIC PCI and MAC information, which describes a network port the firmware boots from over PXE, one per structure.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_nic_mac_spec`, of type `DMI_TYPE(hpe_nic)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`segment`
PCI segment group, zero for a single group topology.

`bus`
PCI bus.

`devfn`
PCI device and function, `(device << 3) | function`.

`mac_address`
MAC address.

`state`
State of the port, see `dmi_hpe_nic_state`: disabled if the bus and the device are zero, and not installed if they have all bits set.

`port`
Number of the port. Set to `UINT8_MAX` when the structure holds none.

`uefi_device_path`
: UEFI device path of the port.

See also

`dmi_entity_info`, `dmi_hpe_nic_state`, `dmi_hpe_nic_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

`<opendmi/entity/hpe/physical-attrs.h>`

dmi_hpe_physical_attrs

Synopsis

```
#include <opendmi/entity/hpe/physical-attrs.h>

typedef struct dmi_hpe_physical_attrs {
    dmi_uuid_t uuid;
    dmi_binary_t identifier_raw;
    char *identifier;
    const char *serial_number;
} dmi_hpe_physical_attrs_t;
```

Description

The `dmi_hpe_physical_attrs_t` structure holds the decoded HP/HPE physical attribute information, which keeps the physical serial number and UUID of the server, whose standard fields hold virtual ones, e.g. on Synergy, where a workload moves from one server to another along with its identity.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_physical_attrs_legacy_spec` up to G7, and by `dmi_hpe_physical_attrs_spec` from Gen8 onwards, of type `DMI_TYPE(hpe_physical_attrs)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Up to G7, the structure holds 16 characters instead of the UUID: the product number and the serial number, which `identifier` points to, and `uuid` is zero. From Gen8 onwards, `identifier` is `nullptr`. The structures of unknown generation are not decoded.

Fields

`uuid`

Physical UUID, from Gen8 onwards.

`identifier_raw`

Product number and serial number, 16 characters, up to G7, as the structure holds them.

`identifier`

Product number and serial number, e.g. `484184GB894484YN`, or `nullptr` if they are not printable. The string belongs to the structure, and is freed along with it.

`serial_number`

Physical serial number.

See also

`dmi_entity_info`, `dmi_system`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/power-supply.h>

dmi_hpe_power_supply

Synopsis

```
#include <opendmi/entity/hpe/power-supply.h>

typedef struct dmi_hpe_power_supply {
    dmi_handle_t power_supply_handle;
    const char *manufacturer;
    const char *revision;
    dmi_hpe_fru_access_t fru_access;
    uint8_t i2c_bus;
    uint8_t i2c_address;
} dmi_hpe_power_supply_t;
```

Description

The `dmi_hpe_power_supply_t` structure holds the decoded HP/HPE power supply information, which supplements the system power supply information (type 39) with the actual manufacturer of the power supply and the way its FRU is accessed.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_power_supply_spec`, of type `DMI_TYPE(hpe_power_supply)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`power_supply_handle`
: Handle of the system power supply structure (type 39).

`manufacturer`
: Actual manufacturer.

`revision`
: Revision.

`fru_access`
: Method the FRU is accessed by, see `dmi_hpe_fru_access`.

`i2c_bus`
: I2C bus, or I2C segment with iLO. Set to `UINT8_MAX` when not available.

`i2c_address`
: I2C address, shifted left by one bit. Set to `UINT8_MAX` when not available.

See also

`dmi_entity_info`, `dmi_hpe_fru_access`, `dmi_power_supply`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_fru_access

Synopsis

```
#include <opendmi/entity/hpe/power-supply.h>

typedef enum dmi_hpe_fru_access {
    DMI_HPE_FRU_ACCESS_NONE    = 0x00,
    DMI_HPE_FRU_ACCESS_IPMI    = 0x01,
    DMI_HPE_FRU_ACCESS_ILO     = 0x02,
    DMI_HPE_FRU_ACCESS_CHASSIS = 0x03
} dmi_hpe_fru_access_t;

const char *dmi_hpe_fru_access_name(dmi_hpe_fru_access_t value);
```

Description

The `dmi_hpe_fru_access_t` enumeration tells the method the FRU of a power supply is accessed by, and is the type of the `fru_access` field of `dmi_hpe_power_supply_t`.

The `dmi_hpe_fru_access_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_FRU_ACCESS_NONE`
: Not available.

`DMI_HPE_FRU_ACCESS_IPMI`
: IPMI I2C.

`DMI_HPE_FRU_ACCESS_ILO`
: iLO.

`DMI_HPE_FRU_ACCESS_CHASSIS`
: Chassis manager.

Return value

`dmi_hpe_fru_access_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not

be modified or freed.

Errors

None.

See also

`dmi_hpe_power_supply`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/processor.h>

dmi_hpe_processor

Synopsis

```
#include <opendmi/entity/hpe/processor.h>

typedef struct dmi_hpe_processor {
    dmi_handle_t processor_handle;
    uint8_t apic_id;
    bool is_bsp;
    bool is_x2apic;
    bool is_thermal_margining;
    uint8_t slot;
    uint8_t socket;
    uint16_t maximum_power;
    uint32_t x2apic_id;
    uint64_t uuid;
    uint16_t interconnect_speed;
    uint8_t qdf_raw[6];
    char *qdf;
} dmi_hpe_processor_t;
```

Description

The `dmi_hpe_processor_t` structure holds the decoded HP/HPE processor specific information, which supplements the processor information (type 4) of each processor socket or slot with the numbers the server tells its processors by.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_processor_spec`, of type `DMI_TYPE(hpe_processor)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The structure has grown with the generations of the servers: fields the data ends before are set to the values they have when not specified.

Fields

`processor_handle`

Handle of the processor information structure (type 4).

`apic_id`

Local APIC ID of the processor.

`is_bsp`

Whether the processor is the bootstrap processor.

`is_x2apic`
: Whether the processor runs in the x2APIC mode.

`is_thermal_margining`
: Whether advanced thermal margining is enabled.

`slot`
: Physical slot, as the silk screen names it. Set to `UINT8_MAX` when the processor is not in a slot.

`socket`
: Physical socket, as the silk screen names it. Set to `UINT8_MAX` when the processor is not in a socket.

`maximum_power`
: Rated maximum power of the processor in watts, zero if unknown.

`x2apic_id`
: x2APIC ID of the processor, valid in the x2APIC mode, and not shown when `is_x2apic` is not set. Set to `UINT32_MAX` when the structure holds none.

`uuid`
: Unique identifier of the processor, zero if unknown.

`interconnect_speed`
: Speed of the interconnect in MT/s, zero if unknown.

`qdf_raw`
: QDF or S-Spec number of Intel processors, six characters padded with spaces, as the structure holds them.

`qdf`
: QDF or S-Spec number of Intel processors, `nullptr` if the structure holds none. The string belongs to the structure, and is freed along with it.

See also

`dmi_entity_info`, `dmi_processor`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/proliant-info.h>

dmi_hpe_proliant_info

Synopsis

```
#include <opendmi/entity/hpe/proliant-info.h>

typedef struct dmi_hpe_proliant_info {
    uint32_t power_features;
    uint32_t omega_features;
    uint32_t misc_features;
    bool is_icru;
    bool is_uefi;
    bool has_omega_features;
    bool has_misc_features;
} dmi_hpe_proliant_info_t;
```

Description

The `dmi_hpe_proliant_info_t` structure holds the decoded HP/HPE ProLiant information, which gives the feature flags of the server the watchdog timer driver reads.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_proliant_info_spec`, of type `DMI_TYPE(hpe_proliant_info)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

HP and HPE servers give type 219 to this structure, and the Intel Management Engine interface information is not decoded on them, see `dmi_intel_mei`. Features the data ends before are zero.

Fields

`power_features`

Power features, whose meaning is not documented.

`omega_features`

Omega features, whose meaning is not documented.

`misc_features`

Miscellaneous features.

`is_icru`

Whether the integrated CRU (iCRU) is supported, as bit 0 of the miscellaneous features tells.

`is_uefi`

Whether the firmware is UEFI one, as bits 10 and 12 of the miscellaneous features tell, the way dmidecode reads them. The `hpwdt` driver of Linux reads bits 3 and 10 instead; the meaning of bits 3

and 12 is not established.

`has_omega_features`
: Whether the structure holds the omega features, which the structures of 8 bytes leave out. The field is not shown then.

`has_misc_features`
: Whether the structure holds the miscellaneous features, which the structures shorter than 20 bytes leave out. The field, `is_icru` and `is_uefi` are not shown then.

See also

`dmi_entity_info`, `dmi_intel_mei`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/rack-locator.h>

dmi_hpe_rack_locator

Synopsis

```
#include <opendmi/entity/hpe/rack-locator.h>

typedef struct dmi_hpe_rack_locator {
    const char *rack_name;
    const char *enclosure_name;
    const char *enclosure_model;
    const char *server_bay;
    uint8_t bay_count;
    uint8_t filled_bay_count;
    const char *enclosure_serial;
} dmi_hpe_rack_locator_t;
```

Description

The `dmi_hpe_rack_locator_t` structure holds the decoded HP/HPE system/rack locator, which tells where a blade server is: the rack, the enclosure and the bay.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_rack_locator_spec`, of type `DMI_TYPE(hpe_rack_locator)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`rack_name`
: Name of the rack.

`enclosure_name`
: Name of the enclosure.

`enclosure_model`
: Model of the enclosure.

`server_bay`
: Bay of the server.

`bay_count`
: Number of the bays of the enclosure.

`filled_bay_count`
: Number of the filled bays of the enclosure.

`enclosure_serial`
: Serial number of the enclosure.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/reserved-memory.h>

dmi_hpe_reserved_memory

Synopsis

```
#include <opendmi/entity/hpe/reserved-memory.h>

typedef struct dmi_hpe_reserved_memory {
    size_t entry_count;
    dmi_hpe_reserved_memory_entry_t *entries;
} dmi_hpe_reserved_memory_t;
```

Description

The `dmi_hpe_reserved_memory_t` structure holds the decoded HP/HPE reserved memory location, which tells where the memory regions the firmware reserves at boot are, e.g. the one the storage controller and iLO exchange the temperatures of the drives in.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_reserved_memory_spec`, of type `DMI_TYPE(hpe_reserved_memory)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Regions run to the end of the structure, which carries no number of them of its own.

Fields

`entry_count`
: Number of regions.

`entries`
: Regions, see `dmi_hpe_reserved_memory_entry`. May be `nullptr` when `entry_count` is 0.

See also

`dmi_entity_info`, `dmi_hpe_reserved_memory_entry`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_reserved_memory_entry

Synopsis

```
#include <opendmi/entity/hpe/reserved-memory.h>

typedef struct dmi_hpe_reserved_memory_entry {
    dmi_binary_t signature_raw;
    char *signature;
    uint64_t address;
    uint32_t raw_size;
    bool is_kilobytes;
    uint64_t size;
} dmi_hpe_reserved_memory_entry_t;
```

Description

The `dmi_hpe_reserved_memory_entry_t` structure describes a memory region the firmware reserves, as an element of the `entries` field of `dmi_hpe_reserved_memory_t`.

Fields

`signature_raw`
: Signature, as the structure holds it.

`signature`
: Signature, which tells what the region is for, e.g. `$HDD`, or `nullptr` if it is not printable. The string belongs to the structure, and is freed along with it.

`address`
: Physical address.

`raw_size`
: Size in the units `is_kilobytes` tells.

`is_kilobytes`
: Whether the size is in kilobytes rather than in bytes.

`size`
: Size in bytes.

See also

`dmi_hpe_reserved_memory`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/rom-info.h>

dmi_hpe_rom_info

Synopsis

```
#include <opendmi/entity/hpe/rom-info.h>

typedef struct dmi_hpe_rom_info {
    bool is_redundant_rom;
    const char *redundant_rom_version;
    const char *bootblock_version;
    const char *oem_rom_filename;
    const char *oem_rom_date;
    const char *unknown_string;
    bool has_redundant_rom_version;
    bool has_oem_rom;
    bool has_unknown_string;
} dmi_hpe_rom_info_t;
```

Description

The `dmi_hpe_rom_info_t` structure holds the decoded HP/HPE other ROM information: the redundant system ROM and the OEM ROM image of the server.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_rom_info_spec`, of type `DMI_TYPE(hpe_rom_info)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields of the OEM ROM image are absent from the structures of the older servers, and are then `nullptr`. Fields which the structure leaves out, or which do not apply, are not shown, as the `has_` fields tell.

Fields

`is_redundant_rom`

Whether a redundant ROM is installed.

`redundant_rom_version`

Version of the redundant ROM, which is its release date. Gen12 onwards reserve the field.

`bootblock_version`

Version string up to Gen8, whose meaning is not documented. Its dates are earlier than the ones of the ROM, which suggests the boot block. Gen9 onwards reserve the byte.

`oem_rom_filename`

File name of the OEM ROM image. Firmware which has no image leaves the string out or fills it with blanks, which the decoded string is trimmed of.

`oem_rom_date`
: Build date of the OEM ROM image.

`unknown_string`
: String whose meaning is not established, e.g. `2.9` on a G6 server.

`has_redundant_rom_version`
: Whether `redundant_rom_version` is shown, which it is when a redundant ROM is installed, up to Gen11.

`has_oem_rom`
: Whether `oem_rom_filename` and `oem_rom_date` are shown, which they are when the raw file name is neither empty nor begins with two blanks.

`has_unknown_string`
: Whether the structure holds `unknown_string`, which the structures shorter than 10 bytes leave out.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/super-io.h>

dmi_hpe_super_io

Synopsis

```
#include <opendmi/entity/hpe/super-io.h>

typedef struct dmi_hpe_super_io {
    bool is_serial_a_enabled;
    bool is_serial_b_enabled;
    bool is_parallel_enabled;
    bool is_floppy_enabled;
    bool is_virtual_serial_enabled;
} dmi_hpe_super_io_t;
```

Description

The `dmi_hpe_super_io_t` structure holds the decoded HP/HPE Super I/O enable/disable indicator, which tells which ports of the Super I/O controller are enabled.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_super_io_spec`, of type `DMI_TYPE(hpe_super_io)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`is_serial_a_enabled`
: Whether serial port A is enabled.

`is_serial_b_enabled`
: Whether serial port B is enabled.

`is_parallel_enabled`
: Whether the parallel port is enabled.

`is_floppy_enabled`
: Whether the floppy disk port is enabled.

`is_virtual_serial_enabled`
: Whether the virtual serial port is enabled.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/system-id.h>

dmi_hpe_system_id

Synopsis

```
#include <opendmi/entity/hpe/system-id.h>

typedef struct dmi_hpe_system_id {
    const char *system_id;
    uint8_t platform_id[2];
    dmi_uuid_t guid;
    bool has_platform_id;
} dmi_hpe_system_id_t;
```

Description

The `dmi_hpe_system_id_t` structure holds the decoded HP/HPE server system ID, which replaces the EISA ID of the older systems, and the platform ID iLO tells the platform by.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_system_id_spec`, of type `DMI_TYPE(hpe_system_id)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The platform ID and the GUID are absent from the structures of the older servers, and are then zero.

Fields

`system_id`
: Server system ID, e.g. `$0E110761`.

`platform_id`
: Bytes of the platform ID, as the CPLD holds them, low byte first. The order the ID is written in is not documented.

`guid`
: GUID, reserved from Gen11 onwards.

`has_platform_id`
: Whether the structure holds the platform ID, which the structures of 5 bytes leave out. The field is not shown then.

See also

`dmi_entity_info`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/tcontrol.h>

dmi_hpe_tcontrol

Synopsis

```
#include <opendmi/entity/hpe/tcontrol.h>

typedef struct dmi_hpe_tcontrol {
    dmi_handle_t processor_handle;
    uint8_t tcontrol;
} dmi_hpe_tcontrol_t;
```

Description

The `dmi_hpe_tcontrol_t` structure holds the decoded HP/HPE processor TControl information, which gives the Tcontrol value of a processor: the temperature the fans begin to spin up at, which Intel programs into each processor.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_tcontrol_spec`, of type `DMI_TYPE(hpe_tcontrol)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

Fields

`processor_handle`
: Handle of the processor information structure (type 4).

`tcontrol`
: Tcontrol value. Set to zero when not available.

See also

`dmi_entity_info`, `dmi_processor`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/trusted-module.h>

dmi_hpe_trusted_module

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef struct dmi_hpe_trusted_module {
    dmi_hpe_tm_presence_t presence;
    bool is_orom_measuring;
    bool is_hidden;
    dmi_hpe_tm_disable_reason_t disable_reason;
    dmi_hpe_tm_error_t error_condition;
    dmi_hpe_tm_type_t type;
    bool is_standard_algorithm;
    bool is_chinese_algorithm;
    dmi_hpe_tm_mounting_t mounting;
    dmi_hpe_tm_fips_t fips;
    dmi_handle_t version_handle;
    dmi_hpe_tm_chip_t chip;
    bool has_extended_status;
    bool has_chip;
} dmi_hpe_trusted_module_t;
```

Description

The `dmi_hpe_trusted_module_t` structure holds the decoded HP/HPE trusted module (TPM or TCM) status, which tells whether a trusted module is present and enabled, and what kind of module it is.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_trusted_module_spec`, of type `DMI_TYPE(hpe_trusted_module)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers. It lives as long as the context it has been decoded in.

The structures of the older servers hold the presence only, and the other fields are then zero, which the enumerations name as not specified.

Fields

`presence`

Presence of the module, see `dmi_hpe_tm_presence`.

`is_orom_measuring`

Whether option ROMs are measured.

`is_hidden`

Whether the module is hidden.

`disable_reason`

Reason the module is disabled for, see `dmi_hpe_tm_disable_reason`. It is shown for a disabled module only.

`error_condition`

Error condition the module is disabled for, see `dmi_hpe_tm_error`. It is shown for a module disabled for an error only.

`type`

Type of the module, see `dmi_hpe_tm_type`.

`is_standard_algorithm`

Whether the standard algorithms are supported.

`is_chinese_algorithm`

Whether the Chinese algorithms are supported.

`mounting`

How the module is mounted, see `dmi_hpe_tm_mounting`.

`fips`

FIPS certification of the module, see `dmi_hpe_tm_fips`.

`version_handle`

Handle of the version indicator (type 216) of the firmware of the module. Set to `DMI_HANDLE_INVALID` when the structure holds none.

`chip`

Chip of the module, see `dmi_hpe_tm_chip`, told by the low byte of the identifier word: the high byte is reserved.

`has_extended_status`

Whether the structure holds the fields from `disable_reason` to `version_handle`, which the structures of 5 bytes leave out. The fields are not shown then.

`has_chip`

Whether the structure holds the chip identifier, which the structures shorter than 12 bytes leave out. The field is not shown then.

See also

`dmi_entity_info`, `dmi_hpe_tm_presence`, `dmi_hpe_tm_disable_reason`, `dmi_hpe_tm_type`, `dmi_hpe_tm_mounting`, `dmi_hpe_tm_fips`, `dmi_hpe_tm_chip`, `dmi_hpe_tm_error`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_presence

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_presence {
    DMI_HPE_TM_PRESENCE_ABSENT   = 0x00,
    DMI_HPE_TM_PRESENCE_ENABLED  = 0x01,
    DMI_HPE_TM_PRESENCE_DISABLED = 0x02
} dmi_hpe_tm_presence_t;

const char *dmi_hpe_tm_presence_name(dmi_hpe_tm_presence_t value);
```

Description

The `dmi_hpe_tm_presence_t` enumeration tells whether a trusted module is present and enabled, and is the type of the `presence` field of `dmi_hpe_trusted_module_t`.

The `dmi_hpe_tm_presence_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_PRESENCE_ABSENT`
: Not present.

`DMI_HPE_TM_PRESENCE_ENABLED`
: Present and enabled.

`DMI_HPE_TM_PRESENCE_DISABLED`
: Present and disabled.

Return value

`dmi_hpe_tm_presence_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_disable_reason

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_disable_reason {
    DMI_HPE_TM_DISABLE_REASON_UNSPEC = 0x00,
    DMI_HPE_TM_DISABLE_REASON_USER   = 0x01,
    DMI_HPE_TM_DISABLE_REASON_ERROR  = 0x02
} dmi_hpe_tm_disable_reason_t;

const char *dmi_hpe_tm_disable_reason_name(dmi_hpe_tm_disable_reason_t value);
```

Description

The `dmi_hpe_tm_disable_reason_t` enumeration tells the reason a trusted module is disabled for, and is the type of the `disable_reason` field of `dmi_hpe_trusted_module_t`.

The `dmi_hpe_tm_disable_reason_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_DISABLE_REASON_UNSPEC`
: Not specified.

`DMI_HPE_TM_DISABLE_REASON_USER`
: Disabled by the user.

`DMI_HPE_TM_DISABLE_REASON_ERROR`
: Error condition.

Return value

`dmi_hpe_tm_disable_reason_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_type

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_type {
    DMI_HPE_TM_TYPE_UNSPEC  = 0x00,
    DMI_HPE_TM_TYPE_TPM_1_2 = 0x01,
    DMI_HPE_TM_TYPE_TPM_2_0 = 0x02,
    DMI_HPE_TM_TYPE_PTT     = 0x03
} dmi_hpe_tm_type_t;

const char *dmi_hpe_tm_type_name(dmi_hpe_tm_type_t value);
```

Description

The `dmi_hpe_tm_type_t` enumeration tells the type of a trusted module, and is the type of the `type` field of `dmi_hpe_trusted_module_t`.

The `dmi_hpe_tm_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_TYPE_UNSPEC`
: Not specified.

`DMI_HPE_TM_TYPE_TPM_1_2`
: TPM 1.2.

`DMI_HPE_TM_TYPE_TPM_2_0`
: TPM 2.0.

`DMI_HPE_TM_TYPE_PTT`
: Intel PTT firmware TPM.

Return value

`dmi_hpe_tm_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be

modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_mounting

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_mounting {
    DMI_HPE_TM_MOUNTING_UNSPEC   = 0x00,
    DMI_HPE_TM_MOUNTING_OPTIONAL = 0x01,
    DMI_HPE_TM_MOUNTING_STANDARD = 0x02,
    DMI_HPE_TM_MOUNTING_SOLDERED = 0x03
} dmi_hpe_tm_mounting_t;

const char *dmi_hpe_tm_mounting_name(dmi_hpe_tm_mounting_t value);
```

Description

The `dmi_hpe_tm_mounting_t` enumeration tells how a trusted module is mounted, and is the type of the `mounting` field of `dmi_hpe_trusted_module_t`.

The `dmi_hpe_tm_mounting_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_MOUNTING_UNSPEC`
: Not specified.

`DMI_HPE_TM_MOUNTING_OPTIONAL`
: Pluggable and optional.

`DMI_HPE_TM_MOUNTING_STANDARD`
: Pluggable but standard.

`DMI_HPE_TM_MOUNTING_SOLDERED`
: Soldered down on the system board.

Return value

`dmi_hpe_tm_mounting_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_fips

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_fips {
    DMI_HPE_TM_FIPS_UNSPEC        = 0x00,
    DMI_HPE_TM_FIPS_NOT_CERTIFIED = 0x01,
    DMI_HPE_TM_FIPS_CERTIFIED     = 0x02
} dmi_hpe_tm_fips_t;

const char *dmi_hpe_tm_fips_name(dmi_hpe_tm_fips_t value);
```

Description

The `dmi_hpe_tm_fips_t` enumeration tells whether a trusted module is FIPS certified, and is the type of the `fips` field of `dmi_hpe_trusted_module_t`.

The `dmi_hpe_tm_fips_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_FIPS_UNSPEC`
: Not specified.

`DMI_HPE_TM_FIPS_NOT_CERTIFIED`
: Not FIPS certified.

`DMI_HPE_TM_FIPS_CERTIFIED`
: FIPS certified.

Return value

`dmi_hpe_tm_fips_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_chip

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>

typedef enum dmi_hpe_tm_chip {
    DMI_HPE_TM_CHIP_NONE           = 0x00,
    DMI_HPE_TM_CHIP_STM_GEN10      = 0x01,
    DMI_HPE_TM_CHIP_PTT            = 0x02,
    DMI_HPE_TM_CHIP_NATIONZ        = 0x03,
    DMI_HPE_TM_CHIP_STM_GEN10_PLUS = 0x04,
    DMI_HPE_TM_CHIP_STM_GEN11      = 0x05,
    DMI_HPE_TM_CHIP_STM_GEN12      = 0x06
} dmi_hpe_tm_chip_t;

const char *dmi_hpe_tm_chip_name(dmi_hpe_tm_chip_t value);
```

Description

The `dmi_hpe_tm_chip_t` enumeration tells the chip of a trusted module, and is the type of the `chip` field of `dmi_hpe_trusted_module_t`. Other values may be found in the data, and are kept as they are.

The `dmi_hpe_tm_chip_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_CHIP_NONE`
: None.

`DMI_HPE_TM_CHIP_STM_GEN10`
: STMicro TPM of Gen10.

`DMI_HPE_TM_CHIP_PTT`
: Intel firmware TPM (PTT).

`DMI_HPE_TM_CHIP_NATIONZ`
: Nationz TPM.

`DMI_HPE_TM_CHIP_STM_GEN10_PLUS`
: STMicro TPM of Gen10 Plus.

`DMI_HPE_TM_CHIP_STM_GEN11`
: STMicro TPM of Gen11.

`DMI_HPE_TM_CHIP_STM_GEN12`
: STMicro TPM of Gen12.

Return value

`dmi_hpe_tm_chip_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_tm_error

Synopsis

```
#include <opendmi/entity/hpe/trusted-module.h>
```

```
typedef enum dmi_hpe_tm_error {
    DMI_HPE_TM_ERROR_UNSPEC    = 0x00,
    DMI_HPE_TM_ERROR_SELF_TEST = 0x01
} dmi_hpe_tm_error_t;

const char *dmi_hpe_tm_error_name(
        dmi_hpe_tm_error_t value);
```

Description

The `dmi_hpe_tm_error_t` enumeration tells the error condition a trusted module is disabled for, and is the type of the `error_condition` field of `dmi_hpe_trusted_module_t`. The field is shown for a module disabled for an error only.

The `dmi_hpe_tm_error_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_TM_ERROR_UNSPEC`

Not specified.

`DMI_HPE_TM_ERROR_SELF_TEST`

Self-test failure.

Return value

`dmi_hpe_tm_error_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_trusted_module`, `dmi_hpe_tm_disable_reason`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/hpe/usb-device.h>

dmi_hpe_usb_device

Synopsis

```
#include <opendmi/entity/hpe/usb-device.h>

typedef struct dmi_hpe_usb_device {
    dmi_handle_t port_handle;
    uint16_t vendor_id;
    bool is_sd_card_present;
    uint8_t usb_class;
    uint8_t usb_subclass;
    uint8_t usb_protocol;
    uint16_t product_id;
    uint32_t raw_capacity;
    uint64_t capacity;
    const char *uefi_device_path;
    const char *uefi_device_name;
    const char *device_name;
    const char *location;
} dmi_hpe_usb_device_t;
```

Description

The `dmi_hpe_usb_device_t` structure holds the decoded HP/HPE USB device correlation record, which describes a USB device the firmware has found at boot on a USB port, and the UEFI boot entries naming it. Devices plugged in later are not described.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_usb_device_spec`, of type `DMI_TYPE(hpe_usb_device)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers with UEFI firmware from Gen9 onwards. It lives as long as the context it has been decoded in.

Fields

`port_handle`

Handle of the USB port connector correlation record (type 238).

`vendor_id`

USB vendor ID of the device.

`is_sd_card_present`

Whether an embedded SD card is present.

`usb_class`
: USB class code of the device.

`usb_subclass`
: USB subclass code of the device, named by `dmi_hpe_usb_storage_subclass` for the devices of the mass storage class (0x08).

`usb_protocol`
: USB protocol code of the device, named by `dmi_hpe_usb_storage_proto` for the devices of the mass storage class (0x08), and by `dmi_hpe_usb_hub_proto` for the hubs (class 0x09).

`product_id`
: USB product ID of the device.

`raw_capacity`
: Capacity of the device in megabytes, zero if not applicable.

`capacity`
: Capacity of the device in bytes, zero if not applicable.

`uefi_device_path`
: UEFI device path.

`uefi_device_name`
: UEFI device structured name.

`device_name`
: Device name.

`location`
: Location of the device.

See also

`dmi_entity_info`, `dmi_hpe_usb_port`, `dmi_hpe_usb_storage_subclass`, `dmi_hpe_usb_storage_proto`, `dmi_hpe_usb_hub_proto`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_usb_storage_subclass

Synopsis

```
#include <opendmi/entity/hpe/usb-device.h>

typedef enum dmi_hpe_usb_storage_subclass {
    DMI_HPE_USB_STORAGE_SUBCLASS_UNREPORTED = 0x00,
```

```
    DMI_HPE_USB_STORAGE_SUBCLASS_RBC        = 0x01,
    DMI_HPE_USB_STORAGE_SUBCLASS_ATAPI      = 0x02,
    DMI_HPE_USB_STORAGE_SUBCLASS_QIC_157    = 0x03,
    DMI_HPE_USB_STORAGE_SUBCLASS_UFI        = 0x04,
    DMI_HPE_USB_STORAGE_SUBCLASS_SFF_8070I  = 0x05,
    DMI_HPE_USB_STORAGE_SUBCLASS_SCSI       = 0x06,
    DMI_HPE_USB_STORAGE_SUBCLASS_LSD_FS     = 0x07,
    DMI_HPE_USB_STORAGE_SUBCLASS_IEEE_1667  = 0x08,
    DMI_HPE_USB_STORAGE_SUBCLASS_VENDOR     = 0xFF
} dmi_hpe_usb_storage_subclass_t;

const char *dmi_hpe_usb_storage_subclass_name(
        dmi_hpe_usb_storage_subclass_t value);
```

Description

The `dmi_hpe_usb_storage_subclass_t` enumeration tells the command set a USB mass storage device uses, and names the `usb_subclass` field of `dmi_hpe_usb_device_t` for the devices of the mass storage class (0x08). Values 0x09 to 0xFE are reserved.

The `dmi_hpe_usb_storage_subclass_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_STORAGE_SUBCLASS_UNREPORTED`
SCSI command set not reported.

`DMI_HPE_USB_STORAGE_SUBCLASS_RBC`
Reduced block commands (RBC).

`DMI_HPE_USB_STORAGE_SUBCLASS_ATAPI`
ATAPI, as used by MMC-5.

`DMI_HPE_USB_STORAGE_SUBCLASS_QIC_157`
QIC-157, obsolete.

`DMI_HPE_USB_STORAGE_SUBCLASS_UFI`
USB floppy interface (UFI).

`DMI_HPE_USB_STORAGE_SUBCLASS_SFF_8070I`
SFF-8070i, obsolete.

`DMI_HPE_USB_STORAGE_SUBCLASS_SCSI`
SCSI transparent command set.

`DMI_HPE_USB_STORAGE_SUBCLASS_LSD_FS`
LSD FS, the lockable storage devices feature specification.

`DMI_HPE_USB_STORAGE_SUBCLASS_IEEE_1667`
: IEEE 1667.

`DMI_HPE_USB_STORAGE_SUBCLASS_VENDOR`
: Vendor-specific.

Return value

`dmi_hpe_usb_storage_subclass_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. Values 0x09 to 0xFE are named "Reserved". The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_device`, `dmi_hpe_usb_storage_proto`

Standards

The codes are the ones the USB Implementers Forum defines for the class, as dmidecode names them. The structure is not part of the SMBIOS specification: it is defined by HP and HPE for their servers, and its layout is the one dmidecode documents.

dmi_hpe_usb_storage_proto

Synopsis

```
#include <opendmi/entity/hpe/usb-device.h>

typedef enum dmi_hpe_usb_storage_proto {
    DMI_HPE_USB_STORAGE_PROTO_CBI_INT  = 0x00,
    DMI_HPE_USB_STORAGE_PROTO_CBI      = 0x01,
    DMI_HPE_USB_STORAGE_PROTO_OBSOLETE = 0x02,
    DMI_HPE_USB_STORAGE_PROTO_BOT      = 0x50,
    DMI_HPE_USB_STORAGE_PROTO_UAS      = 0x62,
    DMI_HPE_USB_STORAGE_PROTO_VENDOR   = 0xFF
} dmi_hpe_usb_storage_proto_t;

const char *dmi_hpe_usb_storage_proto_name(
        dmi_hpe_usb_storage_proto_t value);
```

Description

The `dmi_hpe_usb_storage_proto_t` enumeration tells the transport protocol a USB mass storage device uses, and names the `usb_protocol` field of `dmi_hpe_usb_device_t` for the devices of the mass

storage class (0x08).

The `dmi_hpe_usb_storage_proto_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_STORAGE_PROTO_CBI_INT`
: Control/bulk/interrupt (CBI) transport with command completion interrupt.

`DMI_HPE_USB_STORAGE_PROTO_CBI`
: Control/bulk/interrupt (CBI) transport without command completion interrupt.

`DMI_HPE_USB_STORAGE_PROTO_OBSOLETE`
: Obsolete.

`DMI_HPE_USB_STORAGE_PROTO_BOT`
: Bulk-only transport.

`DMI_HPE_USB_STORAGE_PROTO_UAS`
: USB attached SCSI (UAS).

`DMI_HPE_USB_STORAGE_PROTO_VENDOR`
: Vendor-specific.

Return value

`dmi_hpe_usb_storage_proto_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_device`, `dmi_hpe_usb_storage_subclass`

Standards

The codes are the ones the USB Implementers Forum defines for the class, as dmidecode names them. The structure is not part of the SMBIOS specification: it is defined by HP and HPE for their servers, and its layout is the one dmidecode documents.

dmi_hpe_usb_hub_proto

Synopsis

```
#include <opendmi/entity/hpe/usb-device.h>
```

```
typedef enum dmi_hpe_usb_hub_proto {
    DMI_HPE_USB_HUB_PROTO_FULL_SPEED = 0x00,
    DMI_HPE_USB_HUB_PROTO_SINGLE_TT  = 0x01,
    DMI_HPE_USB_HUB_PROTO_MULTI_TT   = 0x02
} dmi_hpe_usb_hub_proto_t;

const char *dmi_hpe_usb_hub_proto_name(
        dmi_hpe_usb_hub_proto_t value);
```

Description

The `dmi_hpe_usb_hub_proto_t` enumeration tells the speed and the transaction translators of a USB hub, and names the `usb_protocol` field of `dmi_hpe_usb_device_t` for the devices of the hub class (0x09).

The `dmi_hpe_usb_hub_proto_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_HUB_PROTO_FULL_SPEED`
: Full-speed hub.

`DMI_HPE_USB_HUB_PROTO_SINGLE_TT`
: Hi-speed hub with a single transaction translator.

`DMI_HPE_USB_HUB_PROTO_MULTI_TT`
: Hi-speed hub with multiple transaction translators.

Return value

`dmi_hpe_usb_hub_proto_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_device`

Standards

The codes are the ones the USB Implementers Forum defines for the class, as dmidecode names them. The structure is not part of the SMBIOS specification: it is defined by HP and HPE for their servers, and its layout is the one dmidecode documents.

<opendmi/entity/hpe/usb-port.h>

dmi_hpe_usb_port

Synopsis

```
#include <opendmi/entity/hpe/usb-port.h>

typedef struct dmi_hpe_usb_port {
    dmi_handle_t port_handle;
    uint8_t bus;
    uint8_t devfn;
    dmi_hpe_usb_location_t location;
    dmi_hpe_usb_sharing_t sharing;
    uint8_t instance;
    uint8_t hub_instance;
    dmi_hpe_usb_speed_t speed;
    const char *uefi_device_path;
    uint16_t segment;
} dmi_hpe_usb_port_t;
```

Description

The `dmi_hpe_usb_port_t` structure holds the decoded HP/HPE USB port connector correlation record, which correlates a USB port connector (type 8) with its USB controller, its location and the UEFI device path of its endpoint.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_usb_port_spec`, of type `DMI_TYPE(hpe_usb_port)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen9 onwards. It lives as long as the context it has been decoded in.

Fields

`port_handle`

Handle of the port connector structure (type 8).

`bus`

PCI bus of the USB controller of the port.

`devfn`

PCI device and function of the USB controller of the port, `(device << 3) | function`.

`location`

Location of the port, see `dmi_hpe_usb_location`.

`sharing`

Sharing of the port with the management controller, see `dmi_hpe_usb_sharing`.

`instance`
: Instance of the port among the ports of its type.

`hub_instance`
: Instance of the internal hub the port is behind. Set to `0xFE` when not applicable.

`speed`
: Speed the firmware configures the port for, see `dmi_hpe_usb_speed`.

`uefi_device_path`
: UEFI device path of the USB endpoint.

`segment`
: PCI segment group of the USB controller.

See also

`dmi_entity_info`, `dmi_hpe_usb_location`, `dmi_hpe_usb_sharing`, `dmi_hpe_usb_speed`, `dmi_hpe_usb_device`, `dmi_port_connector`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_usb_location

Synopsis

```
#include <opendmi/entity/hpe/usb-port.h>

typedef enum dmi_hpe_usb_location {
    DMI_HPE_USB_LOCATION_INTERNAL = 0x00,
    DMI_HPE_USB_LOCATION_FRONT    = 0x01,
    DMI_HPE_USB_LOCATION_REAR     = 0x02,
    DMI_HPE_USB_LOCATION_SD_CARD  = 0x03,
    DMI_HPE_USB_LOCATION_ILO      = 0x04,
    DMI_HPE_USB_LOCATION_NAND_HUB = 0x05,
    DMI_HPE_USB_LOCATION_DEBUG    = 0x07,
    DMI_HPE_USB_LOCATION_OCP      = 0x09
} dmi_hpe_usb_location_t;

const char *dmi_hpe_usb_location_name(dmi_hpe_usb_location_t value);
```

Description

The `dmi_hpe_usb_location_t` enumeration tells the location of a USB port, and is the type of the `location` field of `dmi_hpe_usb_port_t`.

The `dmi_hpe_usb_location_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_LOCATION_INTERNAL`
: Internal.

`DMI_HPE_USB_LOCATION_FRONT`
: Front of the server.

`DMI_HPE_USB_LOCATION_REAR`
: Rear of the server.

`DMI_HPE_USB_LOCATION_SD_CARD`
: Embedded internal SD card.

`DMI_HPE_USB_LOCATION_ILO`
: iLO USB.

`DMI_HPE_USB_LOCATION_NAND_HUB`
: USB hub of the NAND controller.

`DMI_HPE_USB_LOCATION_DEBUG`
: Debug port.

`DMI_HPE_USB_LOCATION_OCP`
: OCP USB.

Return value

`dmi_hpe_usb_location_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_port`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_usb_sharing

Synopsis

```
#include <opendmi/entity/hpe/usb-port.h>

typedef enum dmi_hpe_usb_sharing {
    DMI_HPE_USB_SHARING_NONE      = 0x00,
    DMI_HPE_USB_SHARING_SWITCH    = 0x01,
    DMI_HPE_USB_SHARING_AUTOMATIC = 0x02
} dmi_hpe_usb_sharing_t;

const char *dmi_hpe_usb_sharing_name(dmi_hpe_usb_sharing_t value);
```

Description

The `dmi_hpe_usb_sharing_t` enumeration tells whether a USB port is shared with the management controller, and how, and is the type of the `sharing` field of `dmi_hpe_usb_port_t`.

The `dmi_hpe_usb_sharing_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_SHARING_NONE`
: Not shared.

`DMI_HPE_USB_SHARING_SWITCH`
: Shared with a physical switch.

`DMI_HPE_USB_SHARING_AUTOMATIC`
: Shared with automatic control.

Return value

`dmi_hpe_usb_sharing_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_port`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_usb_speed

Synopsis

```
#include <opendmi/entity/hpe/usb-port.h>

typedef enum dmi_hpe_usb_speed {
    DMI_HPE_USB_SPEED_FULL  = 0x01,
    DMI_HPE_USB_SPEED_HIGH  = 0x02,
    DMI_HPE_USB_SPEED_SUPER = 0x03
} dmi_hpe_usb_speed_t;

const char *dmi_hpe_usb_speed_name(dmi_hpe_usb_speed_t value);
```

Description

The `dmi_hpe_usb_speed_t` enumeration tells the speed the firmware configures a USB port for, and is the type of the `speed` field of `dmi_hpe_usb_port_t`.

The `dmi_hpe_usb_speed_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_USB_SPEED_FULL`
: USB 1.1 full speed.

`DMI_HPE_USB_SPEED_HIGH`
: USB 2.0 high speed.

`DMI_HPE_USB_SPEED_SUPER`
: USB 3.0 super speed.

Return value

`dmi_hpe_usb_speed_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_usb_port`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one

dmidecode documents.

<opendmi/entity/hpe/version.h>

dmi_hpe_version

Synopsis

```
#include <opendmi/entity/hpe/version.h>

typedef struct dmi_hpe_version {
    dmi_hpe_firmware_type_t firmware_type;
    const char *firmware_name;
    const char *version_string;
    uint8_t data_format;
    uint8_t version_data[12];
    char *version;
    uint16_t unique_id;
} dmi_hpe_version_t;
```

Description

The `dmi_hpe_version_t` structure holds the decoded HP/HPE version indicator record, which tells the version of a firmware component of the server, e.g. of a CPLD or of a power management controller. There is a record for each component.

The structure is returned by `dmi_entity_info` for the structures decoded by `dmi_hpe_version_spec`, of type `DMI_TYPE(hpe_version)`, declared in `<opendmi/module/hpe.h>`, once the HPE extension module (`dmi_hpe_module`) is enabled, which it is for HP and HPE servers from Gen8 onwards. It lives as long as the context it has been decoded in.

The version data is formatted into `version` the way HPE tools show it. Format 5 is laid out differently on Gen9 and on Gen10 and Gen10 Plus servers, and is not formatted on the others.

Fields

`firmware_type`

Firmware component, see `dmi_hpe_firmware_type`.

`firmware_name`

Name of the firmware.

`version_string`

Version of the firmware, as a string.

`data_format`

Format of `version_data`, zero if there is none.

`version_data`

Version of the firmware, in the format `data_format` tells.

`version`

Version of the firmware formatted from `version_data`, or `nullptr` if there is none or its format is not known. The string belongs to the structure, and is freed along with it.

`unique_id`

Unique ID of the firmware flash, zero if none.

See also

`dmi_entity_info`, `dmi_hpe_firmware_type`, `dmi_hpe_trusted_module`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

dmi_hpe_firmware_type

Synopsis

```
#include <opendmi/entity/hpe/version.h>

typedef enum dmi_hpe_firmware_type {
    DMI_HPE_FIRMWARE_TYPE_SYSTEM_ROM          = 0x01,
    DMI_HPE_FIRMWARE_TYPE_REDUNDANT_ROM       = 0x02,
    DMI_HPE_FIRMWARE_TYPE_BOOTBLOCK           = 0x03,
    DMI_HPE_FIRMWARE_TYPE_PMC                 = 0x04,
    DMI_HPE_FIRMWARE_TYPE_PMC_BOOTLOADER      = 0x05,
    DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS          = 0x06,
    DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS_LOADER   = 0x07,
    DMI_HPE_FIRMWARE_TYPE_CPLD                = 0x08,
    DMI_HPE_FIRMWARE_TYPE_SPS                 = 0x09,
    DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS_CPLD     = 0x0A,
    DMI_HPE_FIRMWARE_TYPE_CSM                 = 0x0B,
    DMI_HPE_FIRMWARE_TYPE_APML                = 0x0C,
    DMI_HPE_FIRMWARE_TYPE_BATTERY             = 0x0D,
    DMI_HPE_FIRMWARE_TYPE_TRUSTED_MODULE      = 0x0E,
    DMI_HPE_FIRMWARE_TYPE_NVME_BACKPLANE      = 0x0F,
    DMI_HPE_FIRMWARE_TYPE_PROVISIONING        = 0x10,
    DMI_HPE_FIRMWARE_TYPE_SPI_DESCRIPTOR      = 0x11,
    DMI_HPE_FIRMWARE_TYPE_IE                  = 0x12,
    DMI_HPE_FIRMWARE_TYPE_UMB_BACKPLANE       = 0x13,
    DMI_HPE_FIRMWARE_TYPE_DIAGNOSTICS         = 0x14,
    DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS_ABSTRACT = 0x20,
    DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS          = 0x21,
    DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS_CPLD     = 0x22,
    DMI_HPE_FIRMWARE_TYPE_EL_CARTRIDGE        = 0x23,
    DMI_HPE_FIRMWARE_TYPE_VIDEO               = 0x30,
```

```
    DMI_HPE_FIRMWARE_TYPE_RISER_CPLD          = 0x31,
    DMI_HPE_FIRMWARE_TYPE_CARD_CPLD           = 0x32,
    DMI_HPE_FIRMWARE_TYPE_VROC_NVME           = 0x33,
    DMI_HPE_FIRMWARE_TYPE_VROC_SATA           = 0x34,
    DMI_HPE_FIRMWARE_TYPE_INTEL_SPS           = 0x35,
    DMI_HPE_FIRMWARE_TYPE_SECONDARY_CPLD      = 0x36,
    DMI_HPE_FIRMWARE_TYPE_MEZZANINE_CPLD      = 0x37,
    DMI_HPE_FIRMWARE_TYPE_ARCTIC_SOUND        = 0x38,
    DMI_HPE_FIRMWARE_TYPE_AMPERE_SCP          = 0x39,
    DMI_HPE_FIRMWARE_TYPE_INTEL_CFR           = 0x3A,
    DMI_HPE_FIRMWARE_TYPE_OCP                 = 0x3B,
    DMI_HPE_FIRMWARE_TYPE_DC_SCM_CPLD         = 0x3C,
    DMI_HPE_FIRMWARE_TYPE_POWER_BOARD_CPLD    = 0x3D,
    DMI_HPE_FIRMWARE_TYPE_SWITCH_BOARD_CPLD   = 0x3E,
    DMI_HPE_FIRMWARE_TYPE_SIDEBAND_BOARD_CPLD = 0x3F,
    DMI_HPE_FIRMWARE_TYPE_RISER_MCU           = 0x40,
    DMI_HPE_FIRMWARE_TYPE_SWITCH_BOARD        = 0x41,
    DMI_HPE_FIRMWARE_TYPE_POWER_SUPPLY        = 0x42,
    DMI_HPE_FIRMWARE_TYPE_BMC                 = 0x43
} dmi_hpe_firmware_type_t;

const char *dmi_hpe_firmware_type_name(dmi_hpe_firmware_type_t value);
```

Description

The `dmi_hpe_firmware_type_t` enumeration tells the firmware component a version indicator describes, and is the type of the `firmware_type` field of `dmi_hpe_version_t`. Other values may be found in the data, and are kept as they are.

The `dmi_hpe_firmware_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_HPE_FIRMWARE_TYPE_SYSTEM_ROM`

System ROM.

`DMI_HPE_FIRMWARE_TYPE_REDUNDANT_ROM`

Redundant system ROM.

`DMI_HPE_FIRMWARE_TYPE_BOOTBLOCK`

System ROM bootblock.

`DMI_HPE_FIRMWARE_TYPE_PMC`

Power management controller firmware.

`DMI_HPE_FIRMWARE_TYPE_PMC_BOOTLOADER`

Power management controller firmware bootloader.

`DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS`
: SL chassis firmware.

`DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS_LOADER`
: SL chassis firmware bootloader.

`DMI_HPE_FIRMWARE_TYPE_CPLD`
: Hardware PAL/CPLD.

`DMI_HPE_FIRMWARE_TYPE_SPS`
: SPS firmware (ME firmware).

`DMI_HPE_FIRMWARE_TYPE_SL_CHASSIS_CPLD`
: SL chassis PAL/CPLD.

`DMI_HPE_FIRMWARE_TYPE_CSM`
: Compatibility support module (CSM).

`DMI_HPE_FIRMWARE_TYPE_APML`
: APML.

`DMI_HPE_FIRMWARE_TYPE_BATTERY`
: Smart storage battery (Megacell) firmware.

`DMI_HPE_FIRMWARE_TYPE_TRUSTED_MODULE`
: Trusted module (TPM or TCM) firmware.

`DMI_HPE_FIRMWARE_TYPE_NVME_BACKPLANE`
: NVMe backplane firmware.

`DMI_HPE_FIRMWARE_TYPE_PROVISIONING`
: Intelligent Provisioning.

`DMI_HPE_FIRMWARE_TYPE_SPI_DESCRIPTOR`
: SPI descriptor version.

`DMI_HPE_FIRMWARE_TYPE_IE`
: Innovation Engine firmware (IE firmware).

`DMI_HPE_FIRMWARE_TYPE_UMB_BACKPLANE`
: UMB backplane firmware.

`DMI_HPE_FIRMWARE_TYPE_DIAGNOSTICS`
: Embedded diagnostics.

`DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS_ABSTRACT`
: EL chassis abstraction revision.

`DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS`
: EL chassis firmware revision.

`DMI_HPE_FIRMWARE_TYPE_EL_CHASSIS_CPLD`
: EL chassis PAL/CPLD.

`DMI_HPE_FIRMWARE_TYPE_EL_CARTRIDGE`
: EL cartridge abstraction revision.

`DMI_HPE_FIRMWARE_TYPE_VIDEO`
: Embedded video controller.

`DMI_HPE_FIRMWARE_TYPE_RISER_CPLD`
: PCIe riser programmable logic device.

`DMI_HPE_FIRMWARE_TYPE_CARD_CPLD`
: PCIe card CPLD.

`DMI_HPE_FIRMWARE_TYPE_VROC_NVME`
: Intel NVMe VROC.

`DMI_HPE_FIRMWARE_TYPE_VROC_SATA`
: Intel SATA VROC.

`DMI_HPE_FIRMWARE_TYPE_INTEL_SPS`
: Intel SPS firmware.

`DMI_HPE_FIRMWARE_TYPE_SECONDARY_CPLD`
: Secondary system programmable logic device.

`DMI_HPE_FIRMWARE_TYPE_MEZZANINE_CPLD`
: CPU mezzanine board CPLD.

`DMI_HPE_FIRMWARE_TYPE_ARCTIC_SOUND`
: Intel Arctic Sound-M accelerator firmware.

`DMI_HPE_FIRMWARE_TYPE_AMPERE_SCP`
: Ampere system control processor (SCP).

`DMI_HPE_FIRMWARE_TYPE_INTEL_CFR`
: Intel CFR information.

`DMI_HPE_FIRMWARE_TYPE_OCP`
: OCP card.

`DMI_HPE_FIRMWARE_TYPE_DC_SCM_CPLD`
: DC-SCM CPLD.

`DMI_HPE_FIRMWARE_TYPE_POWER_BOARD_CPLD`
: Power distribution board CPLD.

`DMI_HPE_FIRMWARE_TYPE_SWITCH_BOARD_CPLD`
: PCIe switch board CPLD.

`DMI_HPE_FIRMWARE_TYPE_SIDEBAND_BOARD_CPLD`
: Sideband board CPLD.

`DMI_HPE_FIRMWARE_TYPE_RISER_MCU`
: PCIe riser MCU firmware.

`DMI_HPE_FIRMWARE_TYPE_SWITCH_BOARD`
: PCIe switch board firmware.

`DMI_HPE_FIRMWARE_TYPE_POWER_SUPPLY`
: Power supply firmware.

`DMI_HPE_FIRMWARE_TYPE_BMC`
: BMC firmware.

Return value

`dmi_hpe_firmware_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_hpe_version`

Standards

Not part of the SMBIOS specification. Defined by HP and HPE for their servers; the layout is the one dmidecode documents.

<opendmi/entity/intel-rsd/cabled-pcie.h>

dmi_intel_rsd_cabled_pcie

Synopsis

```
#include <opendmi/entity/intel-rsd/cabled-pcie.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_cabled_pcie {
    dmi_pci_slot_t pci_slot_id;
    unsigned link_width;
    size_t port_count;
    dmi_intel_rsd_cabled_pcie_port_t *ports;
} dmi_intel_rsd_cabled_pcie_t;
```

Description

The `dmi_intel_rsd_cabled_pcie_t` structure holds the decoded cabled PCIe port information structure (type 199) of Intel Rack Scale Design (RSD) firmware: one cabled PCIe port on the fixed side of the system, such as a port of a PCIe cable adapter on the board or on an adapter card in a PCIe slot, together with the groups of lanes its cables carry. Firmware provides one structure for each Cable Management Interface (CMI) controller of such a port, and none if the platform has no cabled PCIe ports.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_cabled_pcie)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`pci_slot_id`

Identifier of the PCIe slot the port is connected to. It is the slot ID of the matching system slot structure (type 9), and the BMC uses the same number to address the slot, for example to read the EEPROM of a cable over IPMI.

`link_width`

Link width of the port, in lanes: 4 for a x4 port, 8 for a x8 port, and so on.

`port_count`

Number of elements of `ports`: the cable indices the structure holds completely. The specification lays out four of them at most, and the `intel-rsd-cabled-pcie.cable-count` lint rule reports a structure which declares more.

`ports`

Cable indices of the port, an array of `port_count` elements (see

`dmi_intel_rsd_cabled_pcie_port`), `nullptr` if there are none.

Examples

```
dmi_registry_t              *registry = dmi_get_registry(context);
dmi_entity_t                *entity   = dmi_registry_lookup_first(registry,
                                            DMI_TYPE(intel_rsd_cabled_pcie),
                                            true);
dmi_intel_rsd_cabled_pcie_t *port     = dmi_entity_info(entity,
                                            DMI_TYPE(intel_rsd_cabled_pcie));

if (port != nullptr) {
    for (size_t i = 0; i < port->port_count; i++)
        printf("Slot %u, cable %u: lanes %u to %u\n",
               (unsigned)port->pci_slot_id, port->ports[i].index,
               port->ports[i].start_lane, port->ports[i].start_lane + 3);
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_cabled_pcie_port`, `dmi_slot`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_cabled_pcie_port

Synopsis

```
#include <opendmi/entity/intel-rsd/cabled-pcie.h>

typedef struct dmi_intel_rsd_cabled_pcie_port {
    unsigned index;
    unsigned start_lane;
} dmi_intel_rsd_cabled_pcie_port_t;
```

Description

The `dmi_intel_rsd_cabled_pcie_port_t` structure describes one cable index of a cabled PCIe port of Intel Rack Scale Design (RSD) firmware: which group of four lanes of the port the cable attached at that index carries.

The elements are held in the `ports` array of `dmi_intel_rsd_cabled_pcie_t`, and live as long as the context the port has been decoded in.

Fields

`index`

Zero-based number of the cable within the slot given by the `pci_slot_id` field of the port. The BMC uses the same number, for example to read the EEPROM of the cable over IPMI.

`start_lane`

First lane of the group of four lanes the cable carries: 0, 4, 8 or 12. The `intel-rsd-cabled-pcie.start-lane` lint rule reports any other value.

See also

`dmi_intel_rsd_cabled_pcie`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/fpga.h>

dmi_intel_rsd_fpga

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_fpga {
    unsigned short index;
    dmi_intel_rsd_fpga_type_t type;
    dmi_intel_rsd_fpga_status_t status;
    unsigned short socket;
    const char *vendor;
    const char *family;
    const char *model;
    const char *bit_stream_version;
    unsigned short hps_core_count;
    dmi_intel_rsd_fpga_hps_isa_t hps_isa;
    dmi_intel_rsd_fpga_hssi_config_t hssi_config;
    unsigned short hssi_port_count;
    unsigned short hssi_port_speed;
    const char *hssi_side_band_config;
    unsigned short reconfig_slots;
    dmi_pci_slot_t pci_slot_id;
    uint8_t pci_bus_number;
    uint8_t pci_device_number;
    uint8_t pci_function_number;
    uint32_t tdp;
    dmi_intel_rsd_fpga_memory_tech_t memory_tech;
    uint32_t memory_capacity;
    uint16_t memory_speed;
} dmi_intel_rsd_fpga_t;
```

Description

The `dmi_intel_rsd_fpga_t` structure holds the decoded FPGA information structure (type 198) of Intel Rack Scale Design (RSD) firmware: one field-programmable gate array (FPGA) of the system, with its identity, its hard processor subsystem (HPS), its high-speed serial interfaces (HSSI), the place it is reached at on PCIe, and the memory attached to it. There is one structure for each FPGA, and none on platforms without FPGAs.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_fpga)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`index`

One-based number of the FPGA on the platform. The `value.range` lint rule reports a structure which holds 0.

`type`

How the FPGA is attached: integrated into a processor package, or discrete (see `dmi_intel_rsd_fpga_type`).

`status`

Whether the FPGA is enabled (see `dmi_intel_rsd_fpga_status`).

`socket`

Processor socket the FPGA is integrated into. Meaningful for integrated FPGAs only, and not shown for the other ones.

`vendor`

Vendor of the FPGA, as a name or, for a discrete FPGA, as its PCI vendor identifier. `nullptr` if the structure does not carry it.

`family`

Family of the FPGA, `nullptr` if the structure does not carry it.

`model`

Model of the FPGA, `nullptr` if the structure does not carry it.

`bit_stream_version`

Version of the bit stream of the FPGA, `nullptr` if the structure does not carry it.

`hps_core_count`

Number of cores of the hard processor subsystem, if the FPGA has one.

`hps_isa`

Instruction set architecture of the cores of the hard processor subsystem (see `dmi_intel_rsd_fpga_hps_isa`).

`hssi_config`

Configuration the high-speed serial interfaces support: networking or PCIe (see `dmi_intel_rsd_fpga_hssi_config`).

`hssi_port_count`

Number of ports of that configuration: 2 for 2x10G or 2x40G networking, 2 for two x16 PCIe links, and so on. Like `hssi_port_speed` and `hssi_side_band_config`, it is not shown when the configuration is not available (`DMI_INTEL_RSD_FPGA_HSSI_CONFIG_UNAVAILABLE`), which the specification says to ignore them for, or is one the specification does not define.

`hssi_port_speed`

Speed of each port in Gbps for networking, or its number of lanes for PCIe: 10 for 2x10G, 40 for 2x40G, 16 for x16 PCIe, and so on.

`hssi_side_band_config`

Side-band mechanism the high-speed serial interfaces are configured through, such as "SPI", "I2C-0" or "SMBus". `nullptr` if the structure does not carry it.

`reconfig_slots`

Number of partial reconfiguration (PR) slots of the FPGA.

`pci_slot_id`

Physical number of the slot connected to the PCIe port the FPGA is reached through.

`pci_bus_number`

PCIe bus number the FPGA is reached at.

`pci_device_number`

PCIe device number the FPGA is reached at.

`pci_function_number`

PCIe function number the FPGA is reached at.

`tdp`

Thermal design power of the FPGA, in milliwatts.

`memory_tech`

Technology of the memory attached to the FPGA, if any (see `dmi_intel_rsd_fpga_memory_tech`).

`memory_capacity`

Capacity of the on-package memory, in mebibytes. The specification gives it in MB, which means 2^20 bytes, the way it does for the memory sizes of SMBIOS.

`memory_speed`

Speed of the memory, in MHz.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
                                      DMI_TYPE(intel_rsd_fpga), true);
dmi_intel_rsd_fpga_t *fpga     = dmi_entity_info(entity,
                                                 DMI_TYPE(intel_rsd_fpga));

if ((fpga != nullptr) && (fpga->status == DMI_INTEL_RSD_FPGA_STATUS_ENABLED))
    printf("FPGA %u: %s, %u mW\n", fpga->index,
           (fpga->model != nullptr) ? fpga->model : "unknown model",
           (unsigned)fpga->tdp);
```

See also

`dmi_entity_info`, `dmi_intel_rsd_fpga_type`, `dmi_intel_rsd_fpga_status`, `dmi_intel_rsd_fpga_hps_isa`, `dmi_intel_rsd_fpga_hssi_config`,

`dmi_intel_rsd_fpga_memory_tech`, `dmi_slot`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_fpga_hps_isa

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>

typedef enum dmi_intel_rsd_fpga_hps_isa {
    DMI_INTEL_RSD_FPGA_HPS_ISA_X86     = 0x00,
    DMI_INTEL_RSD_FPGA_HPS_ISA_X86_64  = 0x01,
    DMI_INTEL_RSD_FPGA_HPS_ISA_IA_64   = 0x02,
    DMI_INTEL_RSD_FPGA_HPS_ISA_ARM_A32 = 0x03,
    DMI_INTEL_RSD_FPGA_HPS_ISA_ARM_A64 = 0x04,
    DMI_INTEL_RSD_FPGA_HPS_ISA_MIPS32  = 0x05,
    DMI_INTEL_RSD_FPGA_HPS_ISA_MIPS64  = 0x06,
    DMI_INTEL_RSD_FPGA_HPS_ISA_OEM     = 0x07
} dmi_intel_rsd_fpga_hps_isa_t;

const char *dmi_intel_rsd_fpga_hps_isa_name(
        dmi_intel_rsd_fpga_hps_isa_t value);
```

Description

The `dmi_intel_rsd_fpga_hps_isa_t` enumeration tells the instruction set architecture of the cores of the hard processor subsystem (HPS) of an FPGA, and is the type of the `hps_isa` field of `dmi_intel_rsd_fpga_t`.

The `dmi_intel_rsd_fpga_hps_isa_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_FPGA_HPS_ISA_X86`
: 32-bit x86.

`DMI_INTEL_RSD_FPGA_HPS_ISA_X86_64`
: x86-64.

`DMI_INTEL_RSD_FPGA_HPS_ISA_IA_64`
: IA-64 (Itanium).

`DMI_INTEL_RSD_FPGA_HPS_ISA_ARM_A32`
: 32-bit Arm (A32).

`DMI_INTEL_RSD_FPGA_HPS_ISA_ARM_A64`
: 64-bit Arm (A64).

`DMI_INTEL_RSD_FPGA_HPS_ISA_MIPS32`
: 32-bit MIPS.

`DMI_INTEL_RSD_FPGA_HPS_ISA_MIPS64`
: 64-bit MIPS.

`DMI_INTEL_RSD_FPGA_HPS_ISA_OEM`
: Architecture specific to the OEM.

Return value

`dmi_intel_rsd_fpga_hps_isa_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_fpga`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_fpga_hssi_config

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>

typedef enum dmi_intel_rsd_fpga_hssi_config {
    DMI_INTEL_RSD_FPGA_HSSI_CONFIG_NETWORKING  = 0x00,
    DMI_INTEL_RSD_FPGA_HSSI_CONFIG_PCIE        = 0x01,
    DMI_INTEL_RSD_FPGA_HSSI_CONFIG_UNAVAILABLE = 0xFF,
} dmi_intel_rsd_fpga_hssi_config_t;

const char *dmi_intel_rsd_fpga_hssi_config_name(
        dmi_intel_rsd_fpga_hssi_config_t value);
```

Description

The `dmi_intel_rsd_fpga_hssi_config_t` enumeration tells what the high-speed serial interfaces (HSSI) of an FPGA are configured for, and is the type of the `hssi_config` field of `dmi_intel_rsd_fpga_t`. The `hssi_port_count` and `hssi_port_speed` fields of the structure are read according to it.

The `dmi_intel_rsd_fpga_hssi_config_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_FPGA_HSSI_CONFIG_NETWORKING`
: Networking ports.

`DMI_INTEL_RSD_FPGA_HSSI_CONFIG_PCIE`
: PCIe links.

`DMI_INTEL_RSD_FPGA_HSSI_CONFIG_UNAVAILABLE`
: The firmware does not report the configuration.

Return value

`dmi_intel_rsd_fpga_hssi_config_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_fpga`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_fpga_memory_tech

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>

typedef enum dmi_intel_rsd_fpga_memory_tech {
    DMI_INTEL_RSD_FPGA_MEMORY_TECH_NONE  = 0x00,
    DMI_INTEL_RSD_FPGA_MEMORY_TECH_EDRAM = 0x01,
    DMI_INTEL_RSD_FPGA_MEMORY_TECH_HBM   = 0x02,
    DMI_INTEL_RSD_FPGA_MEMORY_TECH_HBM2  = 0x03
```

```
} dmi_intel_rsd_fpga_memory_tech_t;

const char *dmi_intel_rsd_fpga_memory_tech_name(
        dmi_intel_rsd_fpga_memory_tech_t value);
```

Description

The `dmi_intel_rsd_fpga_memory_tech_t` enumeration tells the technology of the memory attached to an FPGA, and is the type of the `memory_tech` field of `dmi_intel_rsd_fpga_t`.

The `dmi_intel_rsd_fpga_memory_tech_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_FPGA_MEMORY_TECH_NONE`
: No memory is attached.

`DMI_INTEL_RSD_FPGA_MEMORY_TECH_EDRAM`
: Embedded DRAM (eDRAM).

`DMI_INTEL_RSD_FPGA_MEMORY_TECH_HBM`
: High Bandwidth Memory (HBM).

`DMI_INTEL_RSD_FPGA_MEMORY_TECH_HBM2`
: High Bandwidth Memory of the second generation (HBM2).

Return value

`dmi_intel_rsd_fpga_memory_tech_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_fpga`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_fpga_status

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>

typedef enum dmi_intel_rsd_fpga_status {
    DMI_INTEL_RSD_FPGA_STATUS_DISABLED = 0x00,
    DMI_INTEL_RSD_FPGA_STATUS_ENABLED  = 0x01,
} dmi_intel_rsd_fpga_status_t;

const char *dmi_intel_rsd_fpga_status_name(dmi_intel_rsd_fpga_status_t value);
```

Description

The `dmi_intel_rsd_fpga_status_t` enumeration tells whether an FPGA is in use, and is the type of the `status` field of `dmi_intel_rsd_fpga_t`.

The `dmi_intel_rsd_fpga_status_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_FPGA_STATUS_DISABLED`
: The FPGA is disabled.

`DMI_INTEL_RSD_FPGA_STATUS_ENABLED`
: The FPGA is enabled.

Return value

`dmi_intel_rsd_fpga_status_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_fpga`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_fpga_type

Synopsis

```
#include <opendmi/entity/intel-rsd/fpga.h>

typedef enum dmi_intel_rsd_fpga_type {
    DMI_INTEL_RSD_FPGA_TYPE_INTEGRATED   = 0x00,
    DMI_INTEL_RSD_FPGA_TYPE_DISCRETE     = 0x01,
    DMI_INTEL_RSD_FPGA_TYPE_DISCRETE_SOC = 0x02
} dmi_intel_rsd_fpga_type_t;

const char *dmi_intel_rsd_fpga_type_name(dmi_intel_rsd_fpga_type_t value);
```

Description

The `dmi_intel_rsd_fpga_type_t` enumeration tells how an FPGA is attached to the system, and is the type of the `type` field of `dmi_intel_rsd_fpga_t`. The field holds the value the firmware reports, which may be none of the ones listed.

The `dmi_intel_rsd_fpga_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_FPGA_TYPE_INTEGRATED`
: FPGA integrated into a processor package.

`DMI_INTEL_RSD_FPGA_TYPE_DISCRETE`
: Discrete FPGA device.

`DMI_INTEL_RSD_FPGA_TYPE_DISCRETE_SOC`
: Discrete FPGA with a system on chip, the hard processor subsystem (HPS).

Return value

`dmi_intel_rsd_fpga_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_fpga`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/memory-device.h>

dmi_intel_rsd_memory_device

Synopsis

```
#include <opendmi/entity/intel-rsd/memory-device.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_memory_device {
    dmi_handle_t device_handle;
    dmi_intel_rsd_memory_type_t memory_type;
    dmi_intel_rsd_memory_media_t memory_media;
    const char *firmware_revision;
    const char *firmware_api_version;
    uint32_t maximum_tdp;
    uint8_t smbus_address;
} dmi_intel_rsd_memory_device_t;
```

Description

The `dmi_intel_rsd_memory_device_t` structure holds the decoded memory device extended information structure (type 197) of Intel Rack Scale Design (RSD) firmware: details of a memory device which its memory device structure (type 17) does not carry, such as whether the module is persistent memory, the version of its firmware, its power budget and the SMBus address it is reached at. Each structure extends the memory device structure its `device_handle` refers to.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_memory_device)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`device_handle`

Handle of the memory device structure (type 17) the structure extends. The handle is not resolved to an entity, and the `link.wrong-type` lint rule reports one which refers to a structure of another type.

`memory_type`

Kind of the module: a conventional DIMM or one of the NVDIMM kinds (see `dmi_intel_rsd_memory_type`).

`memory_media`

Media the module keeps its data in (see `dmi_intel_rsd_memory_media`).

`firmware_revision`

Revision of the firmware of the device, `nullptr` if the structure does not carry it.

`firmware_api_version`

Version of the interface the firmware of the device exposes, `nullptr` if the structure does not carry it.

`maximum_tdp`

Maximum thermal design power of the device, in milliwatts.

`smbus_address`

SMBus address of the device, through which its SPD and, for persistent memory, its firmware interface are reached. The BMC hands it to the Intel Intelligent Power Node Manager to query the memory subsystem.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(intel_rsd_memory_device), true);

dmi_intel_rsd_memory_device_t *device =
        dmi_entity_info(entity, DMI_TYPE(intel_rsd_memory_device));

if (device != nullptr) {
    const char *type = dmi_intel_rsd_memory_type_name(device->memory_type);

    printf("Device 0x%04X: %s, up to %u mW\n",
           (unsigned)device->device_handle,
           (type != nullptr) ? type : "unknown type",
           (unsigned)device->maximum_tdp);
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_memory_type`, `dmi_intel_rsd_memory_media`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_memory_media

Synopsis

```
#include <opendmi/entity/intel-rsd/memory-device.h>
```

```
typedef enum dmi_intel_rsd_memory_media {
    DMI_INTEL_RSD_MEMORY_MEDIA_DRAM        = 0x00,
    DMI_INTEL_RSD_MEMORY_MEDIA_NAND        = 0x01,
    DMI_INTEL_RSD_MEMORY_MEDIA_PROPRIETARY = 0x03
} dmi_intel_rsd_memory_media_t;

const char *dmi_intel_rsd_memory_media_name(
        dmi_intel_rsd_memory_media_t value);
```

Description

The `dmi_intel_rsd_memory_media_t` enumeration tells the media a memory module keeps its data in, and is the type of the `memory_media` field of `dmi_intel_rsd_memory_device_t`. The specification defines no value 0x02.

The `dmi_intel_rsd_memory_media_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_MEMORY_MEDIA_DRAM`

DRAM.

`DMI_INTEL_RSD_MEMORY_MEDIA_NAND`

NAND flash.

`DMI_INTEL_RSD_MEMORY_MEDIA_PROPRIETARY`

Media of a proprietary technology.

Return value

`dmi_intel_rsd_memory_media_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_memory_device`, `dmi_intel_rsd_memory_type`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_memory_type

Synopsis

```
#include <opendmi/entity/intel-rsd/memory-device.h>

typedef enum dmi_intel_rsd_memory_type {
    DMI_INTEL_RSD_MEMORY_TYPE_DIMM     = 0x00,
    DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_N = 0x01,
    DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_F = 0x02,
    DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_P = 0x03
} dmi_intel_rsd_memory_type_t;

const char *dmi_intel_rsd_memory_type_name(dmi_intel_rsd_memory_type_t value);
```

Description

The `dmi_intel_rsd_memory_type_t` enumeration tells the kind of a memory module, volatile or persistent, and is the type of the `memory_type` field of `dmi_intel_rsd_memory_device_t`.

The `dmi_intel_rsd_memory_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_MEMORY_TYPE_DIMM`
: Conventional volatile DIMM.

`DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_N`
: NVDIMM-N: persistent memory accessed by bytes.

`DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_F`
: NVDIMM-F: persistent memory accessed by blocks.

`DMI_INTEL_RSD_MEMORY_TYPE_NVDIMM_P`
: NVDIMM-P persistent memory.

Return value

`dmi_intel_rsd_memory_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_memory_device`, `dmi_intel_rsd_memory_media`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/network-card.h>

dmi_intel_rsd_network_card

Synopsis

```
#include <opendmi/entity/intel-rsd/network-card.h>
#include <opendmi/module/intel-rsd.h>

#define DMI_INTEL_RSD_MAC_ADDR_LENGTH 32

typedef struct dmi_intel_rsd_network_card {
    dmi_pci_class_t pci_class;
    dmi_pci_slot_t pci_slot_id;
    dmi_pci_vendor_id_t vendor_id;
    dmi_pci_device_id_t device_id;
    dmi_pci_vendor_id_t sub_vendor_id;
    dmi_pci_device_id_t sub_device_id;
    uint32_t maximum_speed;
    uint32_t current_speed;
    uint16_t port_index;
    dmi_binary_t mac_address;
    const char *firmware_version;
} dmi_intel_rsd_network_card_t;
```

Description

The `dmi_intel_rsd_network_card_t` structure holds the decoded network card information structure (type 190) of Intel Rack Scale Design (RSD) firmware: one port of a network interface card, with the PCI identity of the card, the speeds of the port, its MAC address, the version of the firmware of the card, and the index the BMC selects the port by to boot from the network.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_network_card)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`pci_class`

PCI class code of the device, which tells its generic function.

`pci_slot_id`

Physical number of the slot connected to the PCIe port of the card.

`vendor_id`

PCI vendor identifier of the card.

`device_id`

PCI device identifier of the card.

`sub_vendor_id`

PCI subsystem vendor identifier, which tells the specific model.

`sub_device_id`

PCI subsystem device identifier, which tells the specific model.

`maximum_speed`

Maximum speed of the port, in Mbps: 10000 for 10 Gbps, 40000 for 40 Gbps, and so on.

`current_speed`

Current speed of the port, in Mbps.

`port_index`

Index of the port in the system. It is the device instance the firmware exposes in the boot flags of the IPMI Get/Set System Boot Options command.

`mac_address`

Field holding the MAC address of the port, as stored in the structure: `DMI_INTEL_RSD_MAC_ADDR_LENGTH` bytes. The data points into the structure and lives as long as the context.

`firmware_version`

Version of the firmware of the card, `nullptr` if the structure does not carry it.

Examples

```
dmi_registry_t               *registry = dmi_get_registry(context);
dmi_entity_t                 *entity   = dmi_registry_lookup_first(registry,
                                             DMI_TYPE(intel_rsd_network_card),
                                             true);
dmi_intel_rsd_network_card_t *card     = dmi_entity_info(entity,
                                             DMI_TYPE(intel_rsd_network_card));

if (card != nullptr)
    printf("Port %u: %04X:%04X, %u of %u Mbps\n", (unsigned)card->port_index,
           (unsigned)card->vendor_id, (unsigned)card->device_id,
           (unsigned)card->current_speed, (unsigned)card->maximum_speed);
```

See also

`dmi_entity_info`, `dmi_slot`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/pcie.h>

dmi_intel_rsd_pcie

Synopsis

```
#include <opendmi/entity/intel-rsd/pcie.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_pcie {
    dmi_pci_class_t pci_class;
    dmi_pci_slot_t pci_slot_id;
    dmi_pci_vendor_id_t vendor_id;
    dmi_pci_device_id_t device_id;
    dmi_pci_vendor_id_t sub_vendor_id;
    dmi_pci_device_id_t sub_device_id;
    uint32_t link_speed;
    uint32_t link_width;
} dmi_intel_rsd_pcie_t;
```

Description

The `dmi_intel_rsd_pcie_t` structure holds the decoded PCIe information structure (type 192) of Intel Rack Scale Design (RSD) firmware: one PCIe device of the system, with its PCI identity, the slot it is installed in, and the current state of its link.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_pcie)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`pci_class`

PCI class code of the device, which tells its generic function.

`pci_slot_id`

Physical number of the slot the device is installed in. It matches the slot ID of the system slot structure (type 9) of that slot.

`vendor_id`

PCI vendor identifier of the device.

`device_id`

PCI device identifier of the device.

`sub_vendor_id`

PCI subsystem vendor identifier, which tells the specific model.

`sub_device_id`

PCI subsystem device identifier, which tells the specific model.

`link_speed`

Current speed of the link, as the firmware reports it.

`link_width`

Current width of the link, in lanes: 1, 2, 4, 8, 16, and so on.

Examples

```
dmi_registry_t       *registry = dmi_get_registry(context);
dmi_entity_t         *entity   = dmi_registry_lookup_first(registry,
                                     DMI_TYPE(intel_rsd_pcie), true);
dmi_intel_rsd_pcie_t *device   = dmi_entity_info(entity,
                                                 DMI_TYPE(intel_rsd_pcie));

if (device != nullptr)
    printf("Slot %u: %04X:%04X, x%u\n", (unsigned)device->pci_slot_id,
           (unsigned)device->vendor_id, (unsigned)device->device_id,
           (unsigned)device->link_width);
```

See also

`dmi_entity_info`, `dmi_slot`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/phys-device-mapping.h>

dmi_intel_rsd_phys_device

Synopsis

```
#include <opendmi/entity/intel-rsd/phys-device-mapping.h>

typedef struct dmi_intel_rsd_phys_device {
    dmi_handle_t handle;
    dmi_intel_rsd_phys_device_type_t type;
    uint8_t primary_number;
    uint8_t secondary_number;
    bool is_raw;
    dmi_binary_t data;
} dmi_intel_rsd_phys_device_t;
```

Description

The `dmi_intel_rsd_phys_device_t` structure describes the physical location of one device listed by a physical device mapping structure of Intel Rack Scale Design (RSD) firmware: the structure describing the device, and the numbers which locate it. All numbers are zero-based.

The elements are held in the `devices` array of `dmi_intel_rsd_phys_device_mapping_t`, and live as long as the context the mapping has been decoded in.

Fields

`handle`

Handle of the structure describing the device: a processor (type 4), system slot (type 9) or memory device (type 17) structure, as `type` tells. The handle is not resolved to an entity, and the `link.wrong-type` lint rule reports one which refers to a structure of another type than the one `type` names.

`type`

Type of the device, the `device_type` field of the mapping structure.

`primary_number`

First location number: the socket of a processor, the processor socket a memory device belongs to, or the riser of a PCIe slot.

`secondary_number`

Second location number: the DIMM slot of a memory device within its processor, or the slot within the riser for a PCIe slot. Reserved for processors.

`is_raw`

Set if `type` is none of the known device types, including

`DMI_INTEL_RSD_PHYS_DEVICE_TYPE_INVALID`: the location numbers mean nothing then, and `data` is what the command line tool shows.

`data`

The location bytes, as stored in the structure. The data points into the structure and lives as long as the context.

See also

`dmi_intel_rsd_phys_device_mapping`, `dmi_intel_rsd_phys_device_type`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Physical Device Mapping Information (Type 200) chapter of the reference manual.

dmi_intel_rsd_phys_device_mapping

Synopsis

```
#include <opendmi/entity/intel-rsd/phys-device-mapping.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_phys_device_mapping {
    dmi_intel_rsd_phys_device_type_t device_type;
    size_t device_count;
    dmi_intel_rsd_phys_device_t *devices;
} dmi_intel_rsd_phys_device_mapping_t;
```

Description

The `dmi_intel_rsd_phys_device_mapping_t` structure holds the decoded physical device mapping information structure (type 200) of Intel Rack Scale Design (RSD) firmware: where the devices described by other structures physically are, as socket, riser and slot numbers. A structure lists devices of a single type; a table may hold several structures for the same type when the devices do not fit into one.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_phys_device_mapping)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`device_type`

Type of all the devices of the structure, which tells what their location numbers mean (see `dmi_intel_rsd_phys_device_type`).

`device_count`

Number of elements of `devices`: the entries the structure holds completely.

`devices`

Locations of the devices, an array of `device_count` elements (see `dmi_intel_rsd_phys_device`), `nullptr` if there are none.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                DMI_TYPE(intel_rsd_phys_device_mapping), true);

dmi_intel_rsd_phys_device_mapping_t *mapping =
        dmi_entity_info(entity, DMI_TYPE(intel_rsd_phys_device_mapping));

if ((mapping != nullptr) &&
    (mapping->device_type == DMI_INTEL_RSD_PHYS_DEVICE_TYPE_MEMORY)) {
    for (size_t i = 0; i < mapping->device_count; i++)
        printf("Handle 0x%04X: socket %u, DIMM slot %u\n",
               (unsigned)mapping->devices[i].handle,
               (unsigned)mapping->devices[i].primary_number,
               (unsigned)mapping->devices[i].secondary_number);
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_phys_device`, `dmi_intel_rsd_phys_device_type`, `dmi_processor`, `dmi_slot`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Physical Device Mapping Information (Type 200) chapter of the reference manual.

dmi_intel_rsd_phys_device_type

Synopsis

```
#include <opendmi/entity/intel-rsd/phys-device-mapping.h>

typedef enum dmi_intel_rsd_phys_device_type {
    DMI_INTEL_RSD_PHYS_DEVICE_TYPE_INVALID   = 0x00,
    DMI_INTEL_RSD_PHYS_DEVICE_TYPE_PROCESSOR = 0x01,
    DMI_INTEL_RSD_PHYS_DEVICE_TYPE_PCIE_SLOT = 0x02,
    DMI_INTEL_RSD_PHYS_DEVICE_TYPE_MEMORY    = 0x03
} dmi_intel_rsd_phys_device_type_t;

const char *dmi_intel_rsd_phys_device_type_name(
```

```
        dmi_intel_rsd_phys_device_type_t value);
```

Description

The `dmi_intel_rsd_phys_device_type_t` enumeration tells the type of the devices a physical device mapping structure lists, and so what their location numbers mean. It is the type of the `device_type` field of `dmi_intel_rsd_phys_device_mapping_t` and of the `type` field of `dmi_intel_rsd_phys_device_t`. Values other than the ones listed are undefined too; the location data of such devices is kept as stored.

The `dmi_intel_rsd_phys_device_type_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_PHYS_DEVICE_TYPE_INVALID`
: Undefined or invalid type.

`DMI_INTEL_RSD_PHYS_DEVICE_TYPE_PROCESSOR`
: Processors, described by processor structures (type 4), located by their socket number.

`DMI_INTEL_RSD_PHYS_DEVICE_TYPE_PCIE_SLOT`
: PCIe slots, described by system slot structures (type 9), located by riser and slot numbers.

`DMI_INTEL_RSD_PHYS_DEVICE_TYPE_MEMORY`
: Memory devices, described by memory device structures (type 17), located by processor socket and DIMM slot numbers.

Return value

`dmi_intel_rsd_phys_device_type_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_phys_device_mapping`, `dmi_intel_rsd_phys_device`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Physical Device Mapping Information (Type 200) chapter of the reference manual.

<opendmi/entity/intel-rsd/processor-cpuid.h>

dmi_intel_rsd_cpuid_leaf

Synopsis

```
#include <opendmi/entity/intel-rsd/processor-cpuid.h>

typedef struct dmi_intel_rsd_cpuid_leaf {
    uint32_t leaf;
    bool has_subleaf;
    uint32_t subleaf;
    uint32_t eax;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
} dmi_intel_rsd_cpuid_leaf_t;
```

Description

The `dmi_intel_rsd_cpuid_leaf_t` structure holds one CPUID leaf of a processor CPUID information structure of Intel Rack Scale Design (RSD) firmware: the input the CPUID instruction has been executed with, and the values of the registers it has returned. The input is not stored in the structure; the decoder fills it in from the list of leaves of the subtype.

The elements are held in the `leaves` array of `dmi_intel_rsd_processor_cpuid_t`, and live as long as the context the structure has been decoded in.

Fields

`leaf`

Leaf number, the value of EAX the instruction has been executed with.

`has_subleaf`

Set if the leaf has been read for a given subleaf.

`subleaf`

Subleaf number, the value of ECX the instruction has been executed with. Meaningful only if `has_subleaf` is set.

`eax`

Value of EAX returned by the instruction.

`ebx`

Value of EBX returned by the instruction.

`ecx`
: Value of ECX returned by the instruction.

`edx`
: Value of EDX returned by the instruction.

See also

`dmi_intel_rsd_processor_cpuid`, `dmi_intel_rsd_cpuid_subtype`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Processor CPUID Information (Type 193) chapter of the reference manual.

dmi_intel_rsd_cpuid_subtype

Synopsis

```
#include <opendmi/entity/intel-rsd/processor-cpuid.h>

typedef enum dmi_intel_rsd_cpuid_subtype {
    DMI_INTEL_RSD_CPUID_SUBTYPE_BASIC    = 0x01,
    DMI_INTEL_RSD_CPUID_SUBTYPE_EXTENDED = 0x02
} dmi_intel_rsd_cpuid_subtype_t;

const char *dmi_intel_rsd_cpuid_subtype_name(
        dmi_intel_rsd_cpuid_subtype_t value);
```

Description

The `dmi_intel_rsd_cpuid_subtype_t` enumeration tells which set of CPUID leaves a processor CPUID information structure holds, and is the type of the `subtype` field of `dmi_intel_rsd_processor_cpuid_t`. The data of structures of other subtypes is kept as stored.

The `dmi_intel_rsd_cpuid_subtype_name` function returns the printable name of a value, as the command line tool shows it. The name is translated to the locale set with `dmi_set_locale`, when the library is built with ICU4C support and has a translation of it.

Values

`DMI_INTEL_RSD_CPUID_SUBTYPE_BASIC`
: Leaves 00h to 10h.

`DMI_INTEL_RSD_CPUID_SUBTYPE_EXTENDED`
: Leaves 14h to 80000008h.

Return value

`dmi_intel_rsd_cpuid_subtype_name` returns the name of the value, or `nullptr` if the value has no name. The name is a static string, which must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_processor_cpuid`, `dmi_intel_rsd_cpuid_leaf`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Processor CPUID Information (Type 193) chapter of the reference manual.

dmi_intel_rsd_processor_cpuid

Synopsis

```
#include <opendmi/entity/intel-rsd/processor-cpuid.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_processor_cpuid {
    const char *socket_designation;
    dmi_intel_rsd_cpuid_subtype_t subtype;
    size_t leaf_count;
    dmi_intel_rsd_cpuid_leaf_t *leaves;
    bool is_raw;
    dmi_binary_t data;
} dmi_intel_rsd_processor_cpuid_t;
```

Description

The `dmi_intel_rsd_processor_cpuid_t` structure holds the decoded processor CPUID information structure (type 193) of Intel Rack Scale Design (RSD) firmware: what the CPUID instruction returns on a processor of the system, for a fixed list of leaves. Leaves 00h to 10h and leaves 14h to 80000008h are carried by structures of two different subtypes, so that a processor usually has one structure of each. The firmware builds them late during boot, so that they reflect the current configuration of the system.

The structure stores the register values only; the leaf and subleaf numbers come from the list of the subtype. Subtype 1 lists leaves 00h to 07h, 09h, 0Ah, 0Bh, and 0Dh, 0Fh and 10h with subleaf 0. Subtype 2 lists leaf 14h with subleaf 0, leaves 15h and 16h, leaf 17h with subleaves 0 to 3, leaves 80000000h to 80000004h, and 80000006h to 80000008h.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_processor_cpuid)`, once the Intel RSD extension module

(`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`socket_designation`

Label of the socket or board position of the processor, such as "Socket 0". `nullptr` if the structure does not carry it.

`subtype`

Set of leaves the structure holds (see `dmi_intel_rsd_cpuid_subtype`).

`leaf_count`

Number of elements of `leaves`: the leaves the structure holds completely. A structure cut short holds fewer leaves than its subtype lists. Zero if the subtype is unknown.

`leaves`

CPUID leaves, in the order the subtype lists them (see `dmi_intel_rsd_cpuid_leaf`). `nullptr` if the subtype is unknown.

`is_raw`

Set if the subtype is unknown: the leaves cannot be told apart then, and the data is kept in `data` instead.

`data`

Data following the subtype, as stored in the structure, if the subtype is unknown; empty otherwise. The data points into the structure and lives as long as the context.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(intel_rsd_processor_cpuid), true);

dmi_intel_rsd_processor_cpuid_t *cpuid =
        dmi_entity_info(entity, DMI_TYPE(intel_rsd_processor_cpuid));

if (cpuid != nullptr) {
    for (size_t i = 0; i < cpuid->leaf_count; i++) {
        const dmi_intel_rsd_cpuid_leaf_t *leaf = &cpuid->leaves[i];

        if (leaf->leaf == 0x01)
            printf("Processor signature: %08X\n", (unsigned)leaf->eax);
    }
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_cpuid_leaf`, `dmi_intel_rsd_cpuid_subtype`, `dmi_processor`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware, see the Intel RSD Processor CPUID Information (Type 193) chapter of the reference manual.

<opendmi/entity/intel-rsd/storage-device.h>

dmi_intel_rsd_storage_connector

Synopsis

```
#include <opendmi/entity/intel-rsd/storage-device.h>

typedef enum dmi_intel_rsd_storage_connector {
    DMI_INTEL_RSD_STORAGE_CONNECTOR_UNKNOWN = 0x00,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_SATA    = 0x01,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_SAS     = 0x02,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_PCIE    = 0x03,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_M2      = 0x04,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_USB     = 0x05,
    DMI_INTEL_RSD_STORAGE_CONNECTOR_U2      = 0x06
} dmi_intel_rsd_storage_connector_t;

const char *dmi_intel_rsd_storage_connector_name(
        dmi_intel_rsd_storage_connector_t value);
```

Description

The `dmi_intel_rsd_storage_connector_t` enumeration tells the connector a storage device is attached with, and is the type of the `connector` field of `dmi_intel_rsd_storage_device_t`.

The `dmi_intel_rsd_storage_connector_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_STORAGE_CONNECTOR_UNKNOWN`
: Unknown.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_SATA`
: SATA.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_SAS`
: SAS.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_PCIE`
: PCIe.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_M2`
: M.2.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_USB`
: USB.

`DMI_INTEL_RSD_STORAGE_CONNECTOR_U2`
: U.2.

Return value

`dmi_intel_rsd_storage_connector_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_storage_device`, `dmi_intel_rsd_storage_proto`, `dmi_intel_rsd_storage_device_type`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_storage_device

Synopsis

```
#include <opendmi/entity/intel-rsd/storage-device.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_storage_device {
    const char *port;
    unsigned index;
    dmi_intel_rsd_storage_connector_t connector;
    dmi_intel_rsd_storage_proto_t protocol;
    dmi_intel_rsd_storage_device_type_t type;
    uint32_t capacity;
    uint16_t rpm;
    const char *model;
    const char *serial_number;
    dmi_pci_class_t pci_class;
    dmi_pci_vendor_id_t vendor_id;
    dmi_pci_device_id_t device_id;
    dmi_pci_vendor_id_t sub_vendor_id;
    dmi_pci_device_id_t sub_device_id;
    const char *firmware_version;
```

```
} dmi_intel_rsd_storage_device_t;
```

Description

The `dmi_intel_rsd_storage_device_t` structure holds the decoded storage device information structure (type 194) of Intel Rack Scale Design (RSD) firmware: one storage device of the system, with the place and the way it is attached, what kind of device it is, its identity, and the PCI identifiers of a PCIe device. The BMC selects the device to boot from by its `index`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_storage_device)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`port`

Label of the port or board position the device is attached at, such as "SATA Port 0" or "PCIe Port 1". `nullptr` if the structure does not carry it.

`index`

Index of the device, the one the firmware exposes in the IPMI Get/Set System Boot Options command. The BMC uses it to select the device to boot from.

`connector`

Connector the device is attached with (see `dmi_intel_rsd_storage_connector`).

`protocol`

Protocol the device is accessed with (see `dmi_intel_rsd_storage_proto`).

`type`

Kind of the device (see `dmi_intel_rsd_storage_device_type`).

`capacity`

Capacity of the device, in gigabytes.

`rpm`

Rotation speed of a hard disk drive, in revolutions per minute. Zero for other devices.

`model`

Model of the device, `nullptr` if the structure does not carry it.

`serial_number`

Serial number of the device, `nullptr` if the structure does not carry it.

`pci_class`

PCI class code of a PCIe device, zero for other devices.

`vendor_id`

PCI vendor identifier of a PCIe device, zero for other devices.

`device_id`
PCI device identifier of a PCIe device, zero for other devices.

`sub_vendor_id`
PCI subsystem vendor identifier of a PCIe device, zero for other devices.

`sub_device_id`
PCI subsystem device identifier of a PCIe device, zero for other devices.

`firmware_version`
Version of the firmware of the device, `nullptr` if the structure does not carry it.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(intel_rsd_storage_device), true);

dmi_intel_rsd_storage_device_t *disk =
        dmi_entity_info(entity, DMI_TYPE(intel_rsd_storage_device));

if (disk != nullptr) {
    const char *type = dmi_intel_rsd_storage_device_type_name(disk->type);

    printf("%s: %s, %u GB\n",
           (disk->port != nullptr) ? disk->port : "unknown port",
           (type != nullptr) ? type : "unknown type",
           (unsigned)disk->capacity);
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_storage_connector`, `dmi_intel_rsd_storage_proto`, `dmi_intel_rsd_storage_device_type`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_storage_device_type

Synopsis

```
#include <opendmi/entity/intel-rsd/storage-device.h>

typedef enum dmi_intel_rsd_storage_device_type {
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_UNKNOWN = 0x00,
```

```
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_HDD     = 0x01,
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_SSD     = 0x02,
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_DVD     = 0x03,
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_BLURAY  = 0x04,
    DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_USB     = 0x05
} dmi_intel_rsd_storage_device_type_t;

const char *dmi_intel_rsd_storage_device_type_name(
        dmi_intel_rsd_storage_device_type_t value);
```

Description

The `dmi_intel_rsd_storage_device_type_t` enumeration tells the kind of a storage device, and is the type of the `type` field of `dmi_intel_rsd_storage_device_t`.

The `dmi_intel_rsd_storage_device_type_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_UNKNOWN`
: Unknown.

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_HDD`
: Hard disk drive.

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_SSD`
: Solid-state drive.

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_DVD`
: Optical DVD drive.

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_BLURAY`
: Optical Blu-ray drive.

`DMI_INTEL_RSD_STORAGE_DEVICE_TYPE_USB`
: USB storage device.

Return value

`dmi_intel_rsd_storage_device_type_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_storage_device`, `dmi_intel_rsd_storage_connector`, `dmi_intel_rsd_storage_proto`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_storage_proto

Synopsis

```
#include <opendmi/entity/intel-rsd/storage-device.h>

typedef enum dmi_intel_rsd_storage_proto {
    DMI_INTEL_RSD_STORAGE_PROTO_UNKNOWN = 0x00,
    DMI_INTEL_RSD_STORAGE_PROTO_IDE     = 0x01,
    DMI_INTEL_RSD_STORAGE_PROTO_AHCI    = 0x02,
    DMI_INTEL_RSD_STORAGE_PROTO_NVME    = 0x03,
    DMI_INTEL_RSD_STORAGE_PROTO_USB     = 0x04
} dmi_intel_rsd_storage_proto_t;

const char *dmi_intel_rsd_storage_proto_name(
        dmi_intel_rsd_storage_proto_t value);
```

Description

The `dmi_intel_rsd_storage_proto_t` enumeration tells the protocol a storage device is accessed with, and is the type of the `protocol` field of `dmi_intel_rsd_storage_device_t`.

The `dmi_intel_rsd_storage_proto_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_STORAGE_PROTO_UNKNOWN`
: Unknown.

`DMI_INTEL_RSD_STORAGE_PROTO_IDE`
: IDE.

`DMI_INTEL_RSD_STORAGE_PROTO_AHCI`
: AHCI.

`DMI_INTEL_RSD_STORAGE_PROTO_NVME`
: NVM Express.

`DMI_INTEL_RSD_STORAGE_PROTO_USB`

USB.

Return value

`dmi_intel_rsd_storage_proto_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_storage_device`, `dmi_intel_rsd_storage_connector`, `dmi_intel_rsd_storage_device_type`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/tpm.h>

dmi_intel_rsd_tpm

Synopsis

```
#include <opendmi/entity/intel-rsd/tpm.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_tpm {
    uint8_t config_index;
    const char *version;
    dmi_intel_rsd_tpm_status_t status;
} dmi_intel_rsd_tpm_t;
```

Description

The `dmi_intel_rsd_tpm_t` structure holds the decoded Trusted Platform Module (TPM) information structure (type 195) of Intel Rack Scale Design (RSD) firmware: one TPM configuration of the platform, with the version of the TPM and whether it is enabled. The pod manager selects the configuration to use by its index.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_tpm)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`config_index`

One-based index of the configuration, the one the pod manager (PSME) passes to the IPMI Set TPM Configuration command to select it. The `value.range` lint rule reports a structure which holds 0.

`version`

Version of the TPM, `nullptr` if the structure does not carry it.

`status`

Whether the TPM is enabled (see `dmi_intel_rsd_tpm_status`).

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_entity_t        *entity   = dmi_registry_lookup_first(registry,
                                    DMI_TYPE(intel_rsd_tpm), true);
dmi_intel_rsd_tpm_t *tpm      = dmi_entity_info(entity,
                                              DMI_TYPE(intel_rsd_tpm));
```

```
if (tpm != nullptr) {
    const char *status = dmi_intel_rsd_tpm_status_name(tpm->status);

    printf("TPM %s: %s\n",
           (tpm->version != nullptr) ? tpm->version : "unknown version",
           (status != nullptr) ? status : "unknown status");
}
```

See also

`dmi_entity_info`, `dmi_intel_rsd_tpm_status`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_tpm_status

Synopsis

```
#include <opendmi/entity/intel-rsd/tpm.h>

typedef enum dmi_intel_rsd_tpm_status {
    DMI_INTEL_RSD_TPM_STATUS_DISABLED = 0x00,
    DMI_INTEL_RSD_TPM_STATUS_ENABLED  = 0x01,
} dmi_intel_rsd_tpm_status_t;

const char *dmi_intel_rsd_tpm_status_name(dmi_intel_rsd_tpm_status_t value);
```

Description

The `dmi_intel_rsd_tpm_status_t` enumeration tells whether a Trusted Platform Module (TPM) is enabled, and is the type of the `status` field of `dmi_intel_rsd_tpm_t`.

The `dmi_intel_rsd_tpm_status_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_TPM_STATUS_DISABLED`
: The TPM is disabled.

`DMI_INTEL_RSD_TPM_STATUS_ENABLED`
: The TPM is enabled.

Return value

`dmi_intel_rsd_tpm_status_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_tpm`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/intel-rsd/txt.h>

dmi_intel_rsd_txt

Synopsis

```
#include <opendmi/entity/intel-rsd/txt.h>
#include <opendmi/module/intel-rsd.h>

typedef struct dmi_intel_rsd_txt {
    dmi_intel_rsd_txt_status_t status;
} dmi_intel_rsd_txt_t;
```

Description

The `dmi_intel_rsd_txt_t` structure holds the decoded TXT information structure (type 196) of Intel Rack Scale Design (RSD) firmware: whether Intel Trusted Execution Technology (TXT) is enabled on the platform.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(intel_rsd_txt)`, once the Intel RSD extension module (`dmi_intel_rsd_module`, declared in `<opendmi/module/intel-rsd.h>`) has been enabled in the context with `dmi_add_extension()`. It lives as long as the context it has been decoded in.

Fields

`status`
: Whether Intel TXT is enabled (see `dmi_intel_rsd_txt_status`).

Examples

```
dmi_registry_t      *registry = dmi_get_registry(context);
dmi_entity_t        *entity   = dmi_registry_lookup_first(registry,
                                    DMI_TYPE(intel_rsd_txt), true);
dmi_intel_rsd_txt_t *txt      = dmi_entity_info(entity,
                                            DMI_TYPE(intel_rsd_txt));

if ((txt != nullptr) && (txt->status == DMI_INTEL_RSD_TXT_STATUS_ENABLED))
    printf("Intel TXT is enabled\n");
```

See also

`dmi_entity_info`, `dmi_intel_rsd_txt_status`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

dmi_intel_rsd_txt_status

Synopsis

```
#include <opendmi/entity/intel-rsd/txt.h>

typedef enum dmi_intel_rsd_txt_status {
    DMI_INTEL_RSD_TXT_STATUS_DISABLED = 0x00,
    DMI_INTEL_RSD_TXT_STATUS_ENABLED  = 0x01,
} dmi_intel_rsd_txt_status_t;

const char *dmi_intel_rsd_txt_status_name(dmi_intel_rsd_txt_status_t value);
```

Description

The `dmi_intel_rsd_txt_status_t` enumeration tells whether Intel Trusted Execution Technology (TXT) is enabled, and is the type of the `status` field of `dmi_intel_rsd_txt_t`.

The `dmi_intel_rsd_txt_status_name` function returns the name of a value, as the command line tool shows it.

Values

`DMI_INTEL_RSD_TXT_STATUS_DISABLED`
: Intel TXT is disabled.

`DMI_INTEL_RSD_TXT_STATUS_ENABLED`
: Intel TXT is enabled.

Return value

`dmi_intel_rsd_txt_status_name` returns the name of the value, translated to the language of the current locale when a translation is available, or `nullptr` if the value has no name. The returned string must not be modified or freed.

Errors

None.

See also

`dmi_intel_rsd_txt`

Standards

Not part of the SMBIOS specification. Defined by Intel Rack Scale Design (RSD) firmware.

<opendmi/entity/sun/memory-array-ex.h>

dmi_sun_memory_array_ex

Synopsis

```
#include <opendmi/entity/sun/memory-array-ex.h>

typedef struct dmi_sun_memory_array_ex {
    dmi_handle_t memory_array_handle;
    dmi_handle_t component_handle;
    uint8_t bus_number;
    uint8_t device_number;
    uint8_t function_number;
} dmi_sun_memory_array_ex_t;
```

Description

The `dmi_sun_memory_array_ex_t` structure holds the decoded Sun memory array extended information structure (type 144), which adds the component holding a memory array and the PCI address of its memory controller to its physical memory array structure (type 16). It is not part of SMBIOS: Sun firmware provides it, and the library decodes it only when the Sun extension module (`dmi_sun_module`, declared in `<opendmi/module/sun.h>`) is enabled in the context with `dmi_add_extension`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(sun_memory_array_ex)`, and lives as long as the context it has been decoded in.

Fields

`memory_array_handle`

Handle of the physical memory array structure (type 16) this structure extends.

`component_handle`

Handle of the structure describing the parent component of the memory array, such as the processor holding its memory controller.

`bus_number`

PCI bus number of the memory controller.

`device_number`

PCI device number of the memory controller, 0 to 31.

`function_number`

PCI function number of the memory controller, 0 to 7.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(sun_memory_array_ex), true);
dmi_sun_memory_array_ex_t *info =
        dmi_entity_info(entity, DMI_TYPE(sun_memory_array_ex));

if (info != nullptr)
    printf("controller at %02X:%02X.%u\n", (unsigned)info->bus_number,
           (unsigned)info->device_number, (unsigned)info->function_number);
```

See also

`dmi_entity_info`, `dmi_memory_array`

Standards

Not part of the SMBIOS specification. Defined by Sun firmware, see the Sun Memory Array Extended Information (Type 144) chapter of the reference manual.

<opendmi/entity/sun/memory-device-ex.h>

dmi_sun_memory_device_ex

Synopsis

```
#include <opendmi/entity/sun/memory-device-ex.h>

typedef struct dmi_sun_memory_device_ex {
    dmi_handle_t memory_device_handle;
    uint8_t dram_channel;
    uint8_t chip_select_total;
    size_t chip_select_count;
    uint8_t *chip_selects;
} dmi_sun_memory_device_ex_t;
```

Description

The `dmi_sun_memory_device_ex_t` structure holds the decoded Sun memory device extended information structure (type 145), which adds the DRAM channel and the chip selects of a memory device to its memory device structure (type 17). It is not part of SMBIOS: Sun firmware provides it, and the library decodes it only when the Sun extension module (`dmi_sun_module`, declared in `<opendmi/module/sun.h>`) is enabled in the context with `dmi_add_extension`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(sun_memory_device_ex)`, and lives as long as the context it has been decoded in.

Fields

`memory_device_handle`
: Handle of the memory device structure (type 17) this structure extends.

`dram_channel`
: DRAM channel the memory device is on.

`chip_select_total`
: Number of chip selects the structure declares.

`chip_select_count`
: Number of elements of `chip_selects`: the chip selects the structure holds in full, which may be fewer than `chip_select_total` when the structure is truncated.

`chip_selects`
: Chip selects of the memory device, one byte each; `nullptr` if `chip_select_total` is 0.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(sun_memory_device_ex), true);
dmi_sun_memory_device_ex_t *info =
        dmi_entity_info(entity, DMI_TYPE(sun_memory_device_ex));

for (size_t i = 0; (info != nullptr) && (i < info->chip_select_count); i++)
    printf("channel %u, chip select %u\n", (unsigned)info->dram_channel,
           (unsigned)info->chip_selects[i]);
```

See also

`dmi_entity_info`, `dmi_memory_device`

Standards

Not part of the SMBIOS specification. Defined by Sun firmware, see the Sun Memory Device Extended Information (Type 145) chapter of the reference manual.

<opendmi/entity/sun/pcie-root-complex.h>

dmi_sun_pcie_root_complex

Synopsis

```
#include <opendmi/entity/sun/pcie-root-complex.h>

typedef struct dmi_sun_pcie_root_complex {
    dmi_handle_t baseboard_handle;
    uint8_t bus_number;
    uint8_t device_number;
    uint8_t function_number;
} dmi_sun_pcie_root_complex_t;
```

Description

The `dmi_sun_pcie_root_complex_t` structure holds the decoded Sun PCI Express root complex information structure (type 138), which gives the PCI address of a PCI Express root complex or root port and the board holding it. It is not part of SMBIOS: Sun firmware provides it, and the library decodes it only when the Sun extension module (`dmi_sun_module`, declared in `<opendmi/module/sun.h>`) is enabled in the context with `dmi_add_extension`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(sun_pcie_root_complex)`, and lives as long as the context it has been decoded in.

Fields

`baseboard_handle`

Handle of the baseboard information structure (type 2) of the board holding the root complex or root port.

`bus_number`

PCI bus number of the root complex or root port.

`device_number`

PCI device number of the root complex or root port, 0 to 31.

`function_number`

PCI function number of the root complex or root port, 0 to 7.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
```

```
                                DMI_TYPE(sun_pcie_root_complex), true);
dmi_sun_pcie_root_complex_t *info =
        dmi_entity_info(entity, DMI_TYPE(sun_pcie_root_complex));

if (info != nullptr)
    printf("root complex at %02X:%02X.%u\n", (unsigned)info->bus_number,
           (unsigned)info->device_number, (unsigned)info->function_number);
```

See also

`dmi_entity_info`, `dmi_baseboard`

Standards

Not part of the SMBIOS specification. Defined by Sun firmware, see the Sun PCI-Express Root Complex Information (Type 138) chapter of the reference manual.

<opendmi/entity/sun/port-ex.h>

dmi_sun_port_ex

Synopsis

```
#include <opendmi/entity/sun/port-ex.h>

typedef struct dmi_sun_port_ex {
    dmi_handle_t chassis_handle;
    dmi_handle_t port_handle;
    uint8_t device_type;
    dmi_handle_t device_handle;
    uint8_t phy;
} dmi_sun_port_ex_t;
```

Description

The `dmi_sun_port_ex_t` structure holds the decoded Sun port extended information structure (type 136), which adds the chassis holding a port and the device connected to it to its port connector information structure (type 8). It is not part of SMBIOS: Sun firmware provides it, and the library decodes it only when the Sun extension module (`dmi_sun_module`, declared in `<opendmi/module/sun.h>`) is enabled in the context with `dmi_add_extension`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(sun_port_ex)`, and lives as long as the context it has been decoded in.

Fields

`chassis_handle`
: Handle of the chassis information structure (type 3) of the chassis holding the port.

`port_handle`
: Handle of the port connector information structure (type 8) this structure extends.

`device_type`
: Type of the device connected to the port, as stored; the values are not documented.

`device_handle`
: Handle of the structure describing the device connected to the port.

`phy`
: Number of the PHY of the port.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
```

```
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                                DMI_TYPE(sun_port_ex), true);
dmi_sun_port_ex_t *info =
        dmi_entity_info(entity, DMI_TYPE(sun_port_ex));

if (info != nullptr)
    printf("port 0x%04X, PHY %u\n", (unsigned)info->port_handle,
           (unsigned)info->phy);
```

See also

`dmi_entity_info`, `dmi_port_connector`, `dmi_chassis`

Standards

Not part of the SMBIOS specification. Defined by Sun firmware, see the Sun Port Extended Information (Type 136) chapter of the reference manual.

<opendmi/entity/sun/processor-ex.h>

dmi_sun_processor_ex

Synopsis

```
#include <opendmi/entity/sun/processor-ex.h>

typedef struct dmi_sun_processor_ex {
    dmi_handle_t processor_handle;
    uint8_t fru;
    uint8_t apic_id_total;
    size_t apic_id_count;
    uint16_t *apic_ids;
} dmi_sun_processor_ex_t;
```

Description

The `dmi_sun_processor_ex_t` structure holds the decoded Sun processor extended information structure (type 132), which adds the initial APIC IDs of the strands (hardware threads) of a processor to its processor information structure (type 4). It is not part of SMBIOS: Sun firmware provides it, and the library decodes it only when the Sun extension module (`dmi_sun_module`, declared in `<opendmi/module/sun.h>`) is enabled in the context with `dmi_add_extension`.

The structure is returned by `dmi_entity_info` for the structures of type `DMI_TYPE(sun_processor_ex)`, and lives as long as the context it has been decoded in.

Fields

`processor_handle`

Handle of the processor information structure (type 4) this structure extends. The library does not resolve it into an entity.

`fru`

Field-replaceable unit (FRU) indicator of the processor, as stored.

`apic_id_total`

Number of initial APIC IDs the structure declares.

`apic_id_count`

Number of elements of `apic_ids`: the IDs the structure holds in full, which may be fewer than `apic_id_total` when the structure is truncated.

`apic_ids`

Initial APIC IDs of the strands of the processor; `nullptr` if `apic_id_total` is 0.

Examples

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *entity   = dmi_registry_lookup_first(registry,
                               DMI_TYPE(sun_processor_ex), true);
dmi_sun_processor_ex_t *info =
        dmi_entity_info(entity, DMI_TYPE(sun_processor_ex));

for (size_t i = 0; (info != nullptr) && (i < info->apic_id_count); i++)
    printf("strand %zu: APIC ID 0x%04X\n", i, (unsigned)info->apic_ids[i]);
```

See also

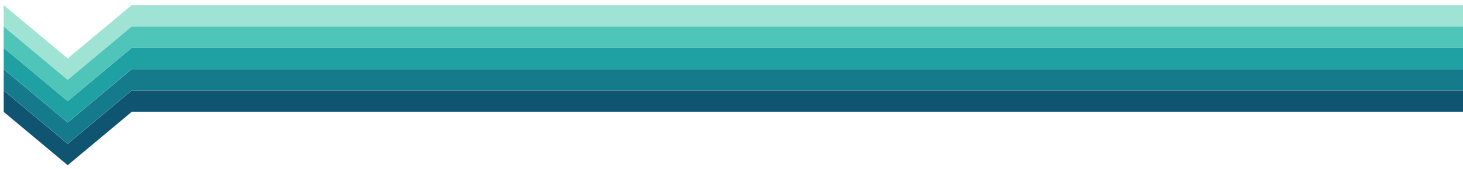
`dmi_entity_info`, `dmi_processor`

Standards

Not part of the SMBIOS specification. Defined by Sun firmware, see the Sun Processor Extended Information (Type 132) chapter of the reference manual.

PART V

Behind the scenes



Chapter 10. From bytes to structures

This part of the manual describes how the library works inside: how the bytes of a platform become the structures a program reads, and how a structure type is described to the library. It is written for the ones who extend the library, add a structure type or a module of a vendor, or want to know why the library behaves the way it does. A program which only reads the data needs none of it, see Using the library.

10.1. The context

Everything the library holds belongs to a context, which `dmi_create` makes and `dmi_destroy` disposes of: the bytes read from the platform, the structures found in them, the values decoded from those, the queue of the errors, the logger and the modules which are enabled. Nothing is shared between the contexts, and nothing outlives the one it belongs to, so a pointer the library gives out stays valid until the context is closed.

The flags a context is created with decide how the data is read:

`DMI_CONTEXT_FLAG_RELAXED`
: Structures which cannot be decoded are skipped and reported, and the rest of the table stays usable. This is the default.

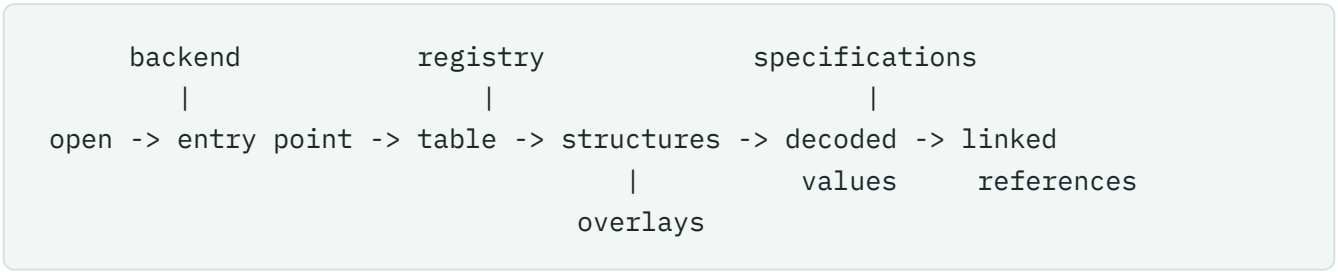
`DMI_CONTEXT_FLAG_STRICT`
: Any structure which cannot be decoded, and any reference which cannot be resolved, fails the opening of the context.

`DMI_CONTEXT_FLAG_LINK`
: References between the structures are resolved once they are decoded, see The registry.

`DMI_CONTEXT_FLAG_OVERLAY`
: Values the additional information structures carry are applied to the structures they refer to before those are decoded.

10.2. Opening the data

`dmi_open` and `dmi_load` take the same way, and differ in where the bytes come from. `dmi_open` asks the backend of the platform, which reads the tables the firmware has left in memory or the ones the kernel exposes, while `dmi_load` asks a backend which reads a dump file instead:



1. The backend opens the source of the data, e.g. a memory device, a system service or a file.
2. The backend reads the entry point into a buffer of the context, which is decoded into the version of SMBIOS and the address and the size of the table. Backends which have no access to the entry point, e.g. the one of Windows, which gets the table from a system service, tell the version by themselves.

3. Versions some firmware writes wrongly are put right: SMBIOS 2.31 and 2.33, which were meant as 2.3, and 2.51, which was meant as 2.6.
4. The backend reads the table into another buffer of the context.
5. The registry scans the table, and creates an entity for every structure it finds, which knows the type, the handle, the length and the strings of the structure, and nothing else yet.
6. With `DMI_CONTEXT_FLAG_OVERLAY`, the additional information entries are applied to the structures they refer to.
7. The vendor of the platform is told from the platform firmware information.
8. Every structure is decoded according to the specification of its type into a C structure, e.g. `dmi_system_t`, which `dmi_entity_info` returns.
9. With `DMI_CONTEXT_FLAG_LINK`, the handles the structures hold are resolved into pointers to the structures they refer to.

A step which fails raises the error telling why, e.g. `DMI_ERROR_BACKEND_OPEN_FAILED`, and then `DMI_ERROR_CONTEXT_OPEN_FAILED`, and the context is closed again. Structures which cannot be decoded or linked do not fail the opening unless the context is strict: all of them are reported to the error queue, and are left undecoded or unlinked.

The data is read once, while the context is opened. Structures are decoded from the buffers of the context rather than from copies, and the strings and the binary values of the decoded structures point into those buffers, which is why nothing the library gives out is to be freed by the program.

10.3. Where the pieces live

The library is split along the steps above:

Header	What it does
`<opendmi/backend.h>`	Backends, which read the bytes of a platform
`<opendmi/buffer.h>`	Buffers, which hold the bytes
`<opendmi/reader.h>`, `<opendmi/writer.h>`	Cursors over the bytes of a buffer
`<opendmi/decoder.h>`, `<opendmi/encoder.h>`	Reading and writing the values of one structure
`<opendmi/entity.h>`	Structures and the specifications of their types
`<opendmi/field.h>`	Layouts of the structures on the wire
`<opendmi/attribute.h>`	Values of the decoded structures, as the output shows them
`<opendmi/utils/name.h>`	Names of the enumerated values
`<opendmi/registry.h>`, `<opendmi/filter.h>`	The structures of a table, and the ways to look them up
`<opendmi/module.h>`	Structure types of the vendors
`<opendmi/lint.h>`	Checking the data against the rules of the specification

The chapters which follow take them in that order.

Chapter 11. Buffers, readers and writers

The bytes of SMBIOS data are held in buffers, and are read and written through cursors over them. Neither knows what the bytes mean: that is up to the decoders and the encoders, which the next chapter describes.

11.1. Buffers

A buffer, `dmi_buffer_t`, owns a block of memory which grows as the data is written into it. It belongs to the context it is created in, which is where the errors of its growth are raised:

```
dmi_buffer_t *buffer = dmi_buffer_create(context);

if (buffer == nullptr)
    return false;

if (!dmi_buffer_assign(buffer, data, length)) {
    dmi_buffer_destroy(buffer);
    return false;
}
```

The context holds two buffers of its own while it is open: the entry point and the table. Backends fill them rather than allocate the data themselves, and every structure refers to the table buffer and to the offset it begins at, which `dmi_entity_buffer` and `dmi_entity_offset` tell. The decoded values which are strings or bytes point into the buffer, so the buffer is never moved or freed while the context is open.

11.2. Readers

A reader, `dmi_reader_t`, reads the bytes of a range of a buffer from its beginning on. It holds no memory, so it is a plain variable which needs no destroying, and a range may be read by several readers at once. A reader of a structure covers the formatted area of it, from the header on:

```
dmi_reader_t reader;
dmi_byte_t   type;
dmi_word_t   handle;

if (!dmi_reader_initialize(&reader, dmi_entity_buffer(entity),
                           dmi_entity_offset(entity), 4))
    return false;

if (dmi_reader_get_bytes(&reader, &type, sizeof(type)) &&
    dmi_reader_skip(&reader, 1) &&
    dmi_reader_get_bytes(&reader, &handle, sizeof(handle)))
    printf("type %u, handle 0x%04X\n", type, dmi_decode(handle));
```

The bytes are read as the data holds them, which is little-endian, and are converted to the byte order of the

host with `dmi_decode`, which takes a value of any of the types of the data: `dmi_byte_t`, `dmi_word_t`, `dmi_dword_t` and `dmi_qword_t`.

Reading past the end of the range fails, and leaves the reader where it was. It is no error of the context: running out of the data is what an older structure does, and the caller decides what it means. This is how the data of any length is read safely, whatever length it declares.

Records which declare their own length are stepped over by that length rather than by what has been read of them, since a record may be longer than the fields it is known to hold:

```
while (!dmi_reader_is_done(&reader)) {
    dmi_reader_mark_t record = dmi_reader_mark(&reader);
    dmi_byte_t        kind;
    dmi_byte_t        length;

    if (!dmi_reader_get_bytes(&reader, &kind, sizeof(kind)) ||
        !dmi_reader_get_bytes(&reader, &length, sizeof(length)))
        break;

    // Fields of the record known to the program are read here

    if (!dmi_reader_skip_ex(&reader, record, length))
        break;
}
```

A mark is a value too, which is copied rather than allocated, and `dmi_reader_rewind` returns the reader to it, e.g. to read a part of the data again in another way.

11.3. Writers

A writer, `dmi_writer_t`, writes bytes into a range of a buffer the way a reader reads them, and grows the buffer as the data is written past its end. A range given no length of its own runs as far as the data is written:

```
dmi_buffer_t *buffer = dmi_buffer_create(context);
dmi_writer_t  writer;
dmi_word_t    handle = dmi_encode((dmi_word_t)0xFFFE);

if (buffer == nullptr)
    return false;

bool written = dmi_writer_initialize(&writer, buffer, 0, -1) &&
               dmi_writer_put_bytes(&writer, "\x7F\x04", 2) &&
               dmi_writer_put_bytes(&writer, &handle, sizeof(handle)) &&
               dmi_writer_skip(&writer, 2);

dmi_buffer_destroy(buffer);
```

This writes the end-of-table structure: its type, its length, a handle, and the two zero bytes which end its string set. `dmi_encode` converts the value back into the byte order of the data. Padding with `dmi_writer_skip` writes zeros past the end of the data, and steps over the bytes written already, so that padding over a part of the data which is being written again keeps it.

A writer fails the same way a reader does when the range is too short, and raises `DMI_ERROR_OUT_OF_MEMORY` on the context of the buffer when the buffer cannot grow.

Chapter 12. Decoders and encoders

A structure is read by a decoder and written by an encoder. Both work on one structure at a time, over a reader or a writer, and know what the bytes of a structure are: numbers in the byte order of the data, strings referred to by their numbers, and fields the older versions of the structure end before.

Most structure types never deal with them directly: they describe their layout instead, and the library decodes and encodes them by that description, see Describing a structure. Decoders and encoders are what that description runs on, and what the handlers of the few types which are decoded by hand are written with.

12.1. Decoders

A decoder, `dmi_decoder_t`, is set up over the data of a structure by `dmi_decoder_initialize`, positioned right after the header. When additional information has been applied to the structure, the decoder reads the copy which carries it, so the handlers never see the difference.

The library sets a decoder up for every structure it decodes, allocates the decoded structure, e.g. a `dmi_cache_t`, which `dmi_entity_info` returns, and gives the decoder to the decoding handler of the type. A handler reads the fields in the order the structure holds them:

```
typedef struct example {
    const char     *name;
    unsigned short  speed;
} example_t;

static bool example_decode(dmi_decoder_t *decoder)
{
    example_t *info = dmi_entity_info(dmi_decoder_entity(decoder),
                                      DMI_TYPE_ANY);

    if (!dmi_decoder_get_string(decoder, &info->name))
        return false;

    // Speed is added by a later version of the structure
    if (dmi_decoder_remaining(decoder) == 0)
        return dmi_decoder_stop(decoder);

    return dmi_decoder_get(decoder, dmi_word_t, &info->speed);
}
```

`dmi_decoder_get` reads a value of the width the type of the data gives, here the two bytes of a word, converts it to the byte order of the host, and stores it into a member which may be wider than the data. `dmi_decoder_get_string` reads the number of a string and gives the string itself, or `nullptr` for a string the structure does not carry.

A structure is allowed to end where a version of the specification ends it, which `dmi_decoder_stop` records: the fields after that point are left as they are, zero unless the handler says otherwise, and the structure is marked as partial. A structure which ends in the middle of the fields of a version is recorded

with `dmi_decoder_incomplete` instead, and the lint rule `entity.unknown-length` reports it.

A handler which fails leaves the structure undecoded, and `DMI_ERROR_ENTITY_DECODE_FAILED` is raised after the errors telling why.

12.2. Encoders

An encoder, `dmi_encoder_t`, writes a structure back into the bytes it is made of. `dmi_encoder_initialize` writes the header into a buffer, and `dmi_entity_encode` writes the rest of the formatted area and fills the length in:

```
dmi_buffer_t *buffer = dmi_buffer_create(context);
dmi_encoder_t encoder;

if ((buffer != nullptr) &&
    dmi_encoder_initialize(&encoder, buffer, entity,
                           DMI_ENCODE_MODE_PRESERVE, DMI_VERSION_NONE)) {
    if (dmi_entity_encode(&encoder))
        printf("%zu bytes, %zu strings\n", buffer->length,
               encoder.string_count);

    dmi_encoder_finalize(&encoder);
}

dmi_buffer_destroy(buffer);
```

The strings the fields refer to are collected by the encoder, numbered the way the fields refer to them, in its `strings` member. They follow the formatted area in the data, each ended by a zero byte, and the set of them by another one.

The decoded structure holds the values, and not every byte they came from: the bits the specification reserves, the fields no member holds, the bytes past the ones the specification knows of. The mode of the encoder says where those come from:

`DMI_ENCODE_MODE_PRESERVE`
: The bytes are taken from the data the structure has been decoded from, so that encoding a structure which has not been changed gives back the very bytes it has been decoded from. This is how a table is changed without losing what the library does not understand.

`DMI_ENCODE_MODE_CANONICAL`
: The bytes are written the way the specification says, for the version the encoder is given, which is how a structure is repaired, or written for another version of the specification.

Encoding and decoding undo each other, which the tests of the library check on every structure of the dumps it is tested with: encoded in the preserve mode, a structure gives back the bytes of the dump, and encoded in the canonical mode, decoded and encoded again, it gives the same bytes both times.

Chapter 13. Describing a structure

Every structure type the library knows is described by a specification, a `dmi_entity_spec_t`, which says everything the library needs to know of the type: its number and its names, the versions it belongs to, how its bytes are laid out, what its values mean, and the rules its data is checked against. Decoding, encoding, linking, the output of the tool in every format and the lint rules of the values all work from it, so a type described once is handled everywhere.

A structure type is made of three parts:

- The decoded structure, a C structure in a public header, e.g. `dmi_cache_t` in `<opendmi/entity/cache.h>`, which a program reads.
- The layout, a list of fields, which says how the bytes of the structure become the members of the decoded structure, and back.
- The attributes, a list of them, which say what the members mean: their names, units, limits and values.

Fields describe bytes and attributes describe values, and the two meet in the members of the decoded structure. The cache information structure is used as the example throughout this chapter.

13.1. The specification

The specification of the cache information structure, in `libopendmi/src/entity/cache.c`, starts with what the type is:

```
const dmi_entity_spec_t dmi_cache_spec =
{
    .code            = "cache",
    .name            = "Cache information",
    .type            = DMI_TYPE(cache),
    .params = {
        .minimum_version = DMI_VERSION(2, 0, 0),
        .required_from   = DMI_VERSION(2, 3, 0),
        .minimum_length  = 0x0F,
        .decoded_length  = sizeof(dmi_cache_t)
    },
    .fields     = ...,
    .attributes = ...,
    .lint_rules = ...
};
```

`code`

Name of the type in the data the tool exports and in the options which select the types, e.g. `opendmi show -t cache`. Codes are never translated.

`name`

Name of the type as the output shows it, which is translated into the locale.

`type`
: Structure type the specification decodes the structures into, see Structure types.

`params`
: Versions of the specification the type belongs to: the one which defined it, the ones from which the table is required to have a structure of it or is expected to, whether the table may have only one of it, the minimum length a structure of it has, and the size of the decoded structure, which the library allocates before decoding.

`fields`, `attributes`, `lint_rules`
: The layout, the attributes and the rules, described below.

`handlers`
: Functions which do what the fields and the attributes cannot say, described at the end of the chapter.

The specifications of the standard types are listed in the library, and the ones of the vendors are brought by their modules, see Extension modules.

13.2. Structure types

A structure type tells what the information a structure is decoded into is, which is one C type, e.g. `dmi_cache_t`. Types are objects rather than numbers: `dmi_type_t` constants, declared by `<opendmi/types.h>` for the standard types and by the headers of the modules for the vendor ones, and named by `DMI_TYPE()`, e.g. `DMI_TYPE(cache)` for `dmi_type_cache`:

```
const dmi_type_t dmi_type_cache = { .id = DMI_TYPE_ID_CACHE };
```

The number a type carries, `id`, is the one its specification gives the structures, and `DMI_TYPE_ID()` names the numbers, e.g. `DMI_TYPE_ID(CACHE)`. Numbers do not tell the types, and the objects do:

- Several specifications may decode the structures of one type, e.g. the layouts of different generations of a vendor, which all refer to it. The code handling the structures asks for the type, whatever layout the table holds.
- Several types may share a number, e.g. Intel vPro information and Lenovo ThinkVantage Technologies enablement, which are both of type 131, and asking for one of them never gives the other.
- A structure of a type may be found at another number, where a vendor places it, and is still of its type.

So everything asking for a structure of a type takes the type object: `dmi_entity_info`, `dmi_entity_data`, the lookups of the registry and the types the handles refer to. Numbers are for what the table itself holds: the number a structure is found at, which `dmi_entity_type_id` gives, the types the options of the tool select, and the structures no specification describes, which `dmi_registry_lookup_first_id` finds.

13.3. Fields

The layout of a structure is a list of fields, in the order the structure holds them, from the byte after the header on. A field names the member of the decoded structure it goes into and the bytes it occupies:

```
.fields = DMI_FIELDS({
    DMI_FIELD_STRING(dmi_cache_t, socket_designator),

    // One word holding the configuration of the cache
    DMI_FIELD_BITS(dmi_cache_t, level, 3,
                   .decode = dmi_cache_decode_level,
                   .encode = dmi_cache_encode_level),
    DMI_FIELD_BITS(dmi_cache_t, socketed, 1),
    DMI_FIELD_BITS_SKIP(1),
    DMI_FIELD_BITS(dmi_cache_t, location, 2),
    DMI_FIELD_BITS(dmi_cache_t, enabled,  1),
    DMI_FIELD_BITS(dmi_cache_t, mode,     2),
    DMI_FIELD_PAD(dmi_word_t),

    DMI_FIELD(dmi_cache_t, maximum_size, dmi_word_t,
              .decode = dmi_cache_decode_size,
              .encode = dmi_cache_encode_size),
    DMI_FIELD(dmi_cache_t, installed_size, dmi_word_t,
              .decode = dmi_cache_decode_size,
              .encode = dmi_cache_encode_size),

    DMI_FIELD(dmi_cache_t, supported_sram, dmi_word_t),
    DMI_FIELD(dmi_cache_t, current_sram,   dmi_word_t),

    DMI_FIELD_GROUP(.since = DMI_VERSION(2, 1, 0)),
    DMI_FIELD(dmi_cache_t, speed,            dmi_byte_t),
    DMI_FIELD(dmi_cache_t, error_correction, dmi_byte_t),
    DMI_FIELD(dmi_cache_t, type,             dmi_byte_t),
    DMI_FIELD(dmi_cache_t, associativity,    dmi_byte_t),

    DMI_FIELD_GROUP(.since = DMI_VERSION(3, 1, 0)),
    DMI_FIELD_EXTENDED(dmi_cache_t, maximum_size, dmi_dword_t,
                       .when_raw = 0xFFFFu,
                       .decode   = dmi_cache_decode_size_ex,
                       .encode   = dmi_cache_encode_size_ex),
    DMI_FIELD_EXTENDED(dmi_cache_t, installed_size, dmi_dword_t,
                       .when_raw = 0xFFFFu,
                       .decode   = dmi_cache_decode_size_ex,
                       .encode   = dmi_cache_encode_size_ex),
    {}
}),
```

The type of the data a field is written with, e.g. `dmi_word_t`, gives the number of the bytes it occupies, which may be fewer than the member holds: a member is made wider than its field when the value standing for "unknown" would otherwise collide with a value a device may really have. The kinds of the fields are:

Field	What it describes
`DMI_FIELD`	An integer, read into the member.
`DMI_FIELD_STRING`	The number of a string, read into a pointer to the string.
`DMI_FIELD_BCD`	A binary-coded decimal, read into the number it spells.
`DMI_FIELD_UUID`	Sixteen bytes of a UUID, read into a `dmi_uuid_t`.
`DMI_FIELD_BINARY`	Bytes taken as they are, which the member refers to in place.
`DMI_FIELD_BITS`, `DMI_FIELD_BITS_SKIP`, `DMI_FIELD_PAD`	Ranges of the bits of a unit several members share, from the least significant bit up, the bits the specification reserves between them, and the ones left over at the end of the unit.
`DMI_FIELD_ARRAY`, `DMI_FIELD_ELEMENT`	An array, preceded by the number of its elements, whose elements are described by fields of their own, or by one field for the arrays of plain values.
`DMI_FIELD_VECTOR`	A fixed number of elements in a row, held in place in an array member of the decoded structure, whose number of elements it takes. The elements are described the way the ones of an array are, e.g. the six firmware status registers of an interface of the Intel Management Engine.
`DMI_FIELD_EXTENDED`	A wider field which a later version adds for a value the plain one is too narrow for. Both name the same member, and the plain one holding the value of `when_raw` says that the real value is here.
`DMI_FIELD_SPLIT`, `DMI_FIELD_SPLIT_BINARY`	Data which feeds several members at once, whose handlers are given the whole decoded structure.
`DMI_FIELD_PRESET`	A member the data does not carry, set before anything is read, e.g. a handle standing for no reference until the data gives one.
`DMI_FIELD_SKIP`	Bytes the structure holds and nothing reads.
`DMI_FIELD_GROUP`	The beginning of the fields a version of the specification adds, which occupies no bytes.
`DMI_FIELD_OFFSET`	The offset the fields after it begin at, which occupies no bytes and is checked while decoding.

13.3.1. Versions

Every version of the specification adds fields at the end of a structure, and a structure is allowed to end at the beginning of any version, since the firmware writes the structure of the version it knows. `DMI_FIELD_GROUP` marks the fields of a version with `since`: a cache information structure of SMBIOS 2.0 ends before the speed, and one of 2.1 before the extended sizes.

The version the data has been read up to becomes the level of the structure, which the output shows, and the members of the fields the data ends before are left zero, or are given the value the field declares as `absent`. A structure which ends in the middle of the fields of a version is decoded up to where it ends, and is marked as incomplete.

Vendor structures grow the same way without versions, and a group of their fields tells whether the

structure holds it with `present`, a `bool` member the decoder sets once every field of the group has been read:

```
DMI_FIELD_GROUP(.present = dmi_member(dmi_hpe_trusted_module_t, has_chip)),
DMI_FIELD(dmi_hpe_trusted_module_t, chip, dmi_byte_t),
```

The attributes of the fields of the group use the member to tell the values from the zeros of the fields the structure is too short for, so no handler has to count the length of the structure, and encoding writes the fields of no group the structure does not hold: the canonical mode ends the structure there, and the preserve mode keeps the source bytes of the group.

A version may also give a meaning to the reserved bits of a field the structure already holds, which says nothing about its length: the version the entry point declares decides it instead. Such ranges of bits declare the version they are defined from with `from`, and a range the version widens is declared twice, up to the version with `before` and from it with `from`. The attributes byte of a memory device is read this way, since SMBIOS 3.10 widens the rank it holds to five bits and adds two flags next to it:

```
DMI_FIELD_BITS(dmi_memory_device_t, rank,        4, .before = DMI_VERSION(3,
10, 0)),
DMI_FIELD_BITS(dmi_memory_device_t, rank,        5, .from   = DMI_VERSION(3,
10, 0)),
DMI_FIELD_BITS(dmi_memory_device_t, is_disabled, 1, .from   = DMI_VERSION(3,
10, 0)),
DMI_FIELD_BITS(dmi_memory_device_t, is_unmapped, 1, .from   = DMI_VERSION(3,
10, 0)),
DMI_FIELD_PAD(dmi_byte_t),
```

Ranges of bits the version of the table does not define are left to the padding, which keeps the bits the source data has in the preserve mode of the encoder, and a range defined from a later version than the fields the data holds raises the level of the structure to it.

`DMI_FIELD_OFFSET` states where a part of the structure begins, the way the tables of the specification number the offsets. It says nothing about where the data may end, and is checked as the structure is decoded, so that a field lost from the layout, or added to it twice, is caught where it happens rather than in the values of everything after it.

13.3.2. Handlers of the fields

A field whose value is not the number the data holds has a pair of handlers: `decode` makes the value of the member from the data, and `encode` makes the data from the value, which undoes it. The level of the cache is counted from zero in the data and from one everywhere else:

```
bool dmi_cache_decode_level(
        const dmi_field_t      *field,
        const dmi_field_data_t *data,
        void                   *value)
{
    return dmi_field_set(field, value, data->number + 1);
```

```
}

bool dmi_cache_encode_level(
        const dmi_field_t *field,
        const void        *value,
        dmi_field_data_t  *data)
{
    uintmax_t level = dmi_field_get(field, value);

    data->number = (level > 0) ? level - 1 : 0;

    return true;
}
```

Handlers deal with values rather than with bytes: the engine reads the bytes, resolves the string a string field refers to, and writes them back. `dmi_field_get` and `dmi_field_set` read and write a member of whatever width the field names. A handler depends on the data of its own field alone, and whatever combines several fields is derived once all of them have been read, see Handlers of the structure. This is what lets the encoder tell whether the data it has already decodes into the value the structure holds, and keep it.

13.4. Attributes

The attributes of a structure say what its members mean, one attribute per member:

```
.attributes = DMI_ATTRIBUTES({
    DMI_ATTRIBUTE(dmi_cache_t, level, INTEGER, {
        .code    = "level",
        .name    = "Cache level",
        .minimum = dmi_value_ptr((unsigned short)1),
        .maximum = dmi_value_ptr((unsigned short)8),
    }),
    DMI_ATTRIBUTE(dmi_cache_t, installed_size, SIZE, {
        .code    = "installed-size",
        .name    = "Installed cache size"
    }),
    DMI_ATTRIBUTE(dmi_cache_t, speed, INTEGER, {
        .code    = "speed",
        .name    = "Cache speed",
        .unit    = DMI_UNIT_NANOSECOND,
        .level   = DMI_VERSION(2, 1, 0)
    }),
    DMI_ATTRIBUTE(dmi_cache_t, type, ENUM, {
        .code    = "type",
        .name    = "System cache type",
        .unspec  = dmi_value_ptr(DMI_CACHE_TYPE_UNSPEC),
        .unknown = dmi_value_ptr(DMI_CACHE_TYPE_UNKNOWN),
        .values  = &dmi_cache_type_names,
```

```
        .level   = DMI_VERSION(2, 1, 0)
    }),
    ...
    {}
}),
```

The type of an attribute says how its value is read and shown:

Type	Value
`STRING`	A string, `nullptr` for a string the structure does not carry.
`BOOL`	A flag, shown with the names of `values` when it has them.
`INTEGER`, `DECIMAL`	A number of the `unit` the attribute gives, a decimal one with `scale` digits after the point.
`SIZE`	A number of bytes, shown in the unit which suits it.
`ADDRESS`	An address, shown in hexadecimal.
`ENUM`	One of the values of an enumeration, named by `values`.
`SET`	A set of the bits of a flag word, named by `values`.
`VERSION`, `DATE`, `UUID`	Values of those types.
`BINARY`	Bytes taken as they are, which the flags may say to show as a MAC or an IP address.
`HANDLE`	The handle of another structure.
`STRUCT`	A nested structure, described by attributes of its own.
`VARIANT`	A value whose type depends on another member, described by one of several attributes.

The parameters of an attribute are:

`code`, **`name`**
: The name of the value in the exported data, which is unique within its structure, and the one the output shows, which is translated.

`unit`, **`scale`**, **`flags`**
: How the value is shown: its unit of measurement, the digits after the point, and whether it is signed, hexadecimal, a binary-coded decimal or an address.

`unspec`, **`unknown`**
: The values which stand for "unspecified" and "unknown" rather than for themselves, which the output shows by those words, and which the lint rules of the values leave out.

`values`
: The names of the values of an enumeration or of the bits of a set, see Name sets.

`minimum`, `maximum`

The limits the specification sets for the value, which the lint rule `value.range` checks.

`level`

The version of the specification the value appears in, so that the output leaves out the values a structure of an older version does not have.

`targets`, `link`

For a handle: the types of the structures it may refer to, and the member the referenced structure is put into once the table is linked.

`attrs`, `variants`

The attributes of a nested structure, and the variants of a variant value.

An array of values is described by `DMI_ATTRIBUTE_ARRAY`, which names the member holding the number of the elements along with the array, and a vector by `DMI_ATTRIBUTE_VECTOR`, whose number of elements is the one of the array member. Both are shown the same way, and `dmi_attribute_is_array` and `dmi_attribute_get_elements` tell the code which walks the attributes where the elements are. The processor information structure refers to its caches this way:

```
DMI_ATTRIBUTE(dmi_processor_t, l1_cache_handle, HANDLE, {
    .code    = "l1-cache-handle",
    .name    = "L1 Cache handle",
    .targets = dmi_types(DMI_TYPE(cache)),
    .level   = DMI_VERSION(2, 1, 0),
    .link    = dmi_member(dmi_processor_t, l1_cache)
}),
```

Linking the table resolves the handle held in `l1_cache_handle` into the cache information structure it refers to, which goes into `l1_cache`, and reports a handle which refers to no structure, or to a structure of another type. The lint rules `link.dangling` and `link.wrong-type` check the same attribute.

Attributes are what the tool is built on: `opendmi show` shows the values of the attributes, and `opendmi export` writes them under their codes. A member which has no attribute is known to the program which reads the structure, and to nothing else.

13.5. Handlers of the structure

Some structures cannot be described by fields alone, and their specification gives handlers, in `handlers`, which do what the fields and the attributes cannot:

`decode`, `encode`

Decode and encode the structure by hand, in place of the fields, see Decoders and encoders. A few types are decoded this way, e.g. the additional information, whose entries declare lengths of their own, and the management controller host interface, whose data is read again according to its type once it has been read.

`derive`

Fills in the members which are computed from the others rather than read, once the fields have been read, e.g. whether a checksum the structure carries matches. It never writes a member a field decodes

into: those are what the structure is encoded from.

`link`

Resolves what the attributes cannot say by themselves, once the references they declare have been resolved, e.g. the references back from the referenced structures, or the handles whose type another member names.

`cleanup`

Frees what the handlers of the specification have allocated beyond the decoded structure itself, e.g. the arrays a decoding or deriving handler fills in. The arrays the fields declare and the arrays of the linked structures belong to the library, which frees them on its own, so most specifications need no cleanup handler.

Chapter 14. Name sets

Enumerated values of the data are numbers, and a name set, `dmi_name_set_t`, gives each of them two names: a code, which the exported data and the command line options use, and a name, which the output shows. The cache types are named this way, in `libopendmi/src/entity/cache-names.c`:

```
const dmi_name_set_t dmi_cache_type_names =
{
    .code  = "cache-type",
    .names = DMI_NAMES({
        DMI_NAME_UNSPEC(DMI_CACHE_TYPE_UNSPEC),
        DMI_NAME_OTHER(DMI_CACHE_TYPE_OTHER),
        DMI_NAME_UNKNOWN(DMI_CACHE_TYPE_UNKNOWN),
        {
            .id   = DMI_CACHE_TYPE_INSTRUCTION,
            .code = "instruction",
            .name = "Instruction"
        },
        {
            .id   = DMI_CACHE_TYPE_DATA,
            .code = "data",
            .name = "Data"
        },
        {
            .id   = DMI_CACHE_TYPE_UNIFIED,
            .code = "unified",
            .name = "Unified"
        },
        {}
    })
};
```

The values almost every enumeration of the specification has are named by helpers, which give them the same codes and names everywhere: `DMI_NAME_UNSPEC`, `DMI_NAME_UNKNOWN`, `DMI_NAME_OTHER`, `DMI_NAME_NONE`, `DMI_NAME_UNSUPPORTED` and `DMI_NAME_RESERVED`.

14.1. Ranges

Values the specification gives one meaning to a whole range of, e.g. the ones it leaves to the vendors, are named by ranges rather than one by one, which the name set lists after the names:

```
const dmi_name_set_t dmi_example_names =
{
    .code   = "example",
    .names  = DMI_NAMES({
        DMI_NAME_UNKNOWN(0x00),
        {
```

```
            .id   = 0x01,
            .code = "fan",
            .name = "Fan"
        },
        {}
    }),
    .ranges = DMI_NAME_RANGES({
        {
            .start_id = 0x80,
            .end_id   = 0xFF,
            .code     = "vendor-specific",
            .name     = "Vendor-specific"
        },
        {}
    })
};
```

A value is looked up among the names first, and among the ranges then, so a name given to a value of a range takes precedence over the range.

14.2. Looking the names up

Name sets of the structure types are private to the library, which looks the names up with `dmi_code_lookup` and `dmi_name_lookup`, giving the code and the name of a value, or `nullptr` for a value the set does not name, and with `dmi_code_lookup_rev`, giving the value of a code back, which is how the options of the tool take values by their codes:

```
const char *code = dmi_code_lookup(&dmi_cache_type_names,
                                   DMI_CACHE_TYPE_DATA);
const char *name = dmi_name_lookup(&dmi_cache_type_names,
                                   DMI_CACHE_TYPE_DATA);
int         id   = dmi_code_lookup_rev(&dmi_cache_type_names, "unified");
```

A program gets the names through the functions most enumerations have, e.g. `dmi_cache_type_name`, and through the attributes, which refer to the name sets. The functions are defined with `DMI_NAME_FUNCTION`, see `DMI_NAMES`, which names the function, the enumeration type and the name set after one prefix:

```
DMI_NAME_FUNCTION(dmi_cache_type)  // dmi_cache_type_name(), dmi_cache_type_t,
dmi_cache_type_names
```

Name sets are what the attributes of the enumerated values and of the sets of bits refer to by `values`, see Describing a structure, so a value is shown by its name and exported by its code wherever it appears, and the lint rule `value.invalid-enum` reports the values a set does not name.

14.3. Translations

Names are translated into the locale the library is set to, when it is built with the translations, while codes never are: they are what the data is exchanged by. A translation is looked up by the code of the set and the code of the value, in the resources of the locale, `libopendmi/locale/ru_RU.txt` for Russian:

```
cache-type {
    unspecified { "Не указан" }
    other { "Другой" }
    unknown { "Неизвестный" }
    instruction { "Инструкции" }
    data { "Данные" }
    unified { "Унифицированный" }
}
```

A value the resources have no translation of is shown by the name the set gives it, so a name set works with no translations at all, and the codes of a set and of its values are all a translator needs to know. The names of the structure types and of the attributes are translated the same way, by their codes.

Chapter 15. The registry

The registry of a context, `dmi_registry_t`, holds the structures of the table as entities, `dmi_entity_t`, in the order the table holds them, and indexes them by their handles. `dmi_get_registry` gives it once the context is open.

15.1. Entities

An entity is created for every structure the table holds when the table is scanned, and knows what the header and the string set of the structure say: its type, its handle, its length and its strings. The rest comes later, and the state of an entity tells how far it has got:

`DMI_ENTITY_STATE_DECODED`
: The structure has been decoded, and `dmi_entity_info` gives the decoded structure.

`DMI_ENTITY_STATE_LINKED`
: The references of the structure have been resolved.

`DMI_ENTITY_STATE_PARTIAL`
: The structure ends where an older version of the specification ends it, and carries fewer fields than the decoder knows of.

`DMI_ENTITY_STATE_INCOMPLETE`
: The structure ends in the middle of the fields of a version, and the fields before that point have been decoded.

Each flag has a predicate of its own, e.g. `dmi_entity_is_decoded` and `dmi_entity_is_incomplete`, which reads better than testing the bits.

A structure which cannot be decoded, e.g. because it is shorter than the minimum length of its type, stays in the registry undecoded: its header and its strings are still there, and `dmi_entity_info` gives `nullptr` for it.

15.2. Looking structures up

`dmi_registry_lookup` finds a structure by its handle, and checks its type, unless it is given `DMI_TYPE_ANY`; `dmi_registry_lookup_first` finds the first structure of a type. A structure which is not there is an error, raised to the queue of the context, unless the lookup is told it is optional:

```
dmi_registry_t *registry = dmi_get_registry(context);
dmi_entity_t   *system   = dmi_registry_lookup_first(registry,
                                                     DMI_TYPE(system), true);
dmi_entity_t   *entity   = dmi_registry_lookup(registry, 0x0004,
                                               DMI_TYPE_ANY, true);
```

Handles 0xFFFE and 0xFFFF stand for no structure, and looking them up gives `nullptr` with no error.

The structures are walked in the order of the table by an iterator, which yields the ones a filter matches, or all of them for no filter. A filter matches the structures of the types and of the handles added to it:

```
dmi_filter_t       *filter = dmi_filter_create(context);
dmi_registry_iter_t iter;
dmi_entity_t       *entity;

if ((filter != nullptr) &&
    dmi_filter_add_type(filter, DMI_TYPE_ID(MEMORY_DEVICE)) &&
    dmi_registry_iter_initialize(&iter, registry, filter)) {
    while ((entity = dmi_registry_iter_next(&iter)) != nullptr) {
        dmi_memory_device_t *device =
                dmi_entity_info(entity, DMI_TYPE(memory_device));

        if ((device != nullptr) && (device->device_locator != nullptr))
            printf("%s\n", device->device_locator);
    }
}

dmi_filter_destroy(filter);
```

An iterator is a plain variable, which needs no destroying, and several of them may walk the registry at once.

15.3. Linking

Structures refer to each other by handles, e.g. a memory device to the memory array it belongs to. Linking the table, which the context does when it is created with `DMI_CONTEXT_FLAG_LINK`, resolves the handles into pointers to the entities they refer to, so that a program follows the references without looking them up:

```
dmi_memory_device_t *device = dmi_entity_info(entity,
                                              DMI_TYPE(memory_device));
dmi_entity_t        *array  = device->array;
```

Most references are resolved by the attributes of the handles, which name the member the referenced structure goes into, see Describing a structure, and the link handler of the type does the rest. A link handler resolves a handle with `dmi_registry_resolve`, which tells a reference which is not set, and succeeds for it, from a broken one, which it reports:

```
static bool cooling_device_link(dmi_entity_t *entity)
{
    dmi_context_t        *context  = dmi_entity_context(entity);
    dmi_registry_t       *registry = dmi_get_registry(context);
    dmi_cooling_device_t *info     = dmi_entity_info(entity,
                                                     DMI_TYPE(cooling_device));

    return dmi_registry_resolve(registry, info->probe_handle,
                                DMI_TYPE(temperature_probe), &info->probe);
```

```
}
```

Every reference of the table is resolved, whether the ones before have failed or not, so that all the broken references are reported at once, and a structure with a broken reference is left unlinked. In the relaxed mode the rest of the table is linked all the same, and in the strict mode the opening of the context fails.

Some firmware writes the handle 0x0000 for a reference which is not set, while 0x0000 is a handle a structure may have. A reference of 0x0000 to a structure of another type than the expected one is taken as not set unless the context is strict.

15.4. Additional information

Additional information structures, of type 40, carry values which replace bytes of other structures, e.g. a string the firmware has corrected after the structure has been written. With `DMI_CONTEXT_FLAG_OVERLAY`, their entries are attached to the structures they refer to once the table has been scanned, and before anything is decoded:

1. Every additional information structure is decoded.
2. Every entry is attached to the structure it refers to, which is reported when it is missing, or when the entry refers to the header of the structure or past its end.
3. A structure with entries attached is decoded from a copy of its bytes with the values of the entries applied, in the order they are attached. The bytes of the table are left as they are.

The entries applied to a structure are listed in the `overlays` member of its entity, which the tool shows with the `-O` option.

Chapter 16. Extension modules

The specification leaves the structure types from 128 to 255 to the vendors of the platforms, and nothing keeps two vendors from giving the same number to different structures. The structure types of a vendor are therefore brought by a module of their own, which is enabled when the data comes from a platform of that vendor.

16.1. Modules

A module, `dmi_module_t`, is a list of the specifications of the structure types it brings, which are written the same way the standard ones are, see Describing a structure:

```
const dmi_module_t dmi_dell_module =
{
    .code     = "dell",
    .name     = "Dell extensions",
    .entities = (const dmi_entity_spec_t *[]){
        &dmi_dell_bios_flags_spec,
        &dmi_dell_revisions_spec,
        ...
        nullptr
    }
};
```

The modules of the library are built in, and listed in `dmi_builtin_modules`, so that they are linked into a static build as well: `acer`, `ami`, `apple`, `cisco`, `dell`, `hpe`, `intel`, `intel-rsd`, `lenovo` and `sun`. The tool lists them with `opendmi modules`, and `opendmi types -m dell` lists the types a module brings.

16.2. Enabling a module

A module is enabled for the platforms it lists, as the context is opened, when the context has `DMI_CONTEXT_FLAG_AUTO_MODULES`, which the tool always gives it unless told otherwise by its `--no-auto-modules` option:

```
const dmi_module_t dmi_dell_module =
{
    ...
    .platforms = DMI_PLATFORMS({
        { .firmware_vendor = DMI_VENDOR_DELL },
        DMI_PLATFORM_NULL
    })
};
```

A program enables a module explicitly before the data is opened, e.g. when the data does not tell the vendor, and the tool does with its `-m` option:

```
dmi_context_t *context = dmi_create(DMI_CONTEXT_FLAG_LINK);

if ((context == nullptr) ||
    !dmi_add_extension(context, dmi_module_find("dell")) ||
    !dmi_open(context, nullptr)) {
    dmi_destroy(context);
    return 1;
}
```

Enabling a module maps the types it brings to their specifications in the context. A type is mapped to one specification at a time, so a module which brings a type another enabled module already has, or the same module enabled twice, fails with `DMI_ERROR_MODULE_CONFLICT`. Modules enabled explicitly take precedence: a module of the platform whose types conflict with them is skipped. Structures of the types no enabled module brings are left undecoded, with their headers and strings only.

Modules enabled for the platform stay enabled until the context is closed, while the ones enabled explicitly stay for the next data to be opened.

16.3. Platform

The platform the data comes from, `dmi_platform_t`, is told as the context is opened, before the rest of the table is decoded: the vendor of the firmware from the platform firmware information (type 0), the vendor and the product name of the system from the system information (type 1), the vendor of the baseboard from the baseboard information (type 2), and the vendor of the processors from the processor information (type 4). The vendors are told apart, since each of them adds structures of its own: the firmware of AMI, e.g., is found on the boards of many vendors. The vendor of the firmware tells the family and the generation of the product by the `detect` handler of its `dmi_vendor_spec_t`, since the structures of a vendor often depend on them. HP and HPE, e.g., give the same type numbers to different structures in their workstations and their servers, and to different layouts in different generations of the servers.

A module lists the platforms it is enabled for by conditions on the vendors and the family, whose members left out are satisfied by any platform:

```
.platforms = DMI_PLATFORMS({
    { .firmware_vendor = DMI_VENDOR_HP,  .family = DMI_HPE_FAMILY_SERVER },
    { .firmware_vendor = DMI_VENDOR_HPE, .family = DMI_HPE_FAMILY_SERVER },
    DMI_PLATFORM_NULL
})
```

and a specification the generations it applies to, which the vendor numbers so that a later generation has a greater number:

```
.params = {
    .generations = { .minimum = DMI_HPE_GEN9 },
    ...
}
```

Both bounds are included, and a zero bound leaves the range open on its side. A module may bring several specifications of the same type, whose ranges do not overlap, and the one of the generation of the platform is chosen. A bounded range is never chosen while the generation is unknown, so that no structure is decoded by the layout of another generation. The rest of the specifications leave the range zeroed, and apply to all generations.

The platform is available through `dmi_get_platform`, and belongs to the context. A program sets another one when the data does not tell it, or to decode structures without opening any data: it creates a platform with `dmi_platform_create`, which owns the strings set with `dmi_platform_set_product` and `dmi_platform_set_family`, and gives it to the context with `dmi_set_platform`, which keeps a copy of it:

```
dmi_platform_t *platform = dmi_platform_create(context);

platform->firmware_vendor = DMI_VENDOR_HPE;
platform->generation      = DMI_HPE_GEN10;
dmi_platform_set_family(platform, DMI_HPE_FAMILY_SERVER);

dmi_set_platform(context, platform);
dmi_platform_destroy(platform);
```

16.4. Structures of any platform

Some structures are defined neither by the vendor of the platform nor by the specification, but by a third party whose code the firmware of many vendors includes, e.g. the Intel reference code. Their module is enabled for the platforms of any vendor, by a condition on the vendor of the processors rather than on the vendor of the firmware. Vendors give the same type numbers to structures of their own, so the module has `DMI_MODULE_FLAG_YIELD`, and its specifications give way to the specifications of the other enabled modules of the same types, rather than conflicting with them:

```
const dmi_module_t dmi_intel_module =
{
    .code      = "intel",
    .name      = "Intel extensions",
    .entities  = ...,
    .flags     = DMI_MODULE_FLAG_YIELD,
    .platforms = DMI_PLATFORMS({
        { .processor_vendor = DMI_VENDOR_INTEL },
        DMI_PLATFORM_NULL
    })
};
```

The structures of Intel Rack Scale Design are only found on the platforms built to it, at type numbers of vendor structures, so their module, `intel-rsd`, is enabled explicitly.

16.5. Relocations

Some vendors place structures defined by others at type numbers of their own, since the original numbers are taken by their own structures. Dell, e.g., places the structures of the Intel reference code 16 types down. The module of such a vendor lists the relocations, which apply while it is enabled, to the specifications of the other enabled modules:

```
.relocations = DMI_RELOCATIONS({
    { &dmi_intel_fvi_spec, 205 },
    {}
})
```

A vendor which places a structure at a type number or another, e.g. on platforms of different generations, relocates its specification more than once, and the structure is decoded at every one of the type numbers:

```
.relocations = DMI_RELOCATIONS({
    { &dmi_intel_fvi_spec, 205 },
    { &dmi_intel_fvi_spec, 237 },
    {}
})
```

A vendor whose platforms never carry a structure, and give its type number to a structure the module of the vendor does not describe, relocates it to `DMI_TYPE_ID_INVALID`, so that the type is left undecoded rather than decoded by a wrong layout:

```
.relocations = DMI_RELOCATIONS({
    { &dmi_intel_fvi_spec, DMI_TYPE_ID_INVALID },
    {}
})
```

A relocated structure is decoded by its original specification, since its data is laid out as that specification describes it. `dmi_entity_type_id` gives the type number found in the table, while `dmi_entity_type` gives the structure type of the specification, which `dmi_entity_info` and `dmi_entity_data` check, so that the code which handles the structure does not depend on where the vendor places it.

16.6. Signatures

Some tables hold structures of different layouts at the same type number, since two vendors whose code the firmware includes give it to structures of their own. Lenovo tables, e.g., hold both Intel vPro information and Lenovo ThinkVantage Technologies enablement at type 131. Neither the type number nor the platform tells them apart, so their specifications tell the content of the structures they describe by a signature, which is declared rather than checked by code of their own:

```
.params = {
    .signature = DMI_SIGNATURE({
        .offset = 0x38,
```

```
        .bytes  = "vPro",
        .size   = 4
    })
}
```

A signature may name bytes at an offset, the length of the formatted area, the text of a string, and that the structure carries no strings at all, and a structure matches it if it meets all of the conditions it names. A type number is mapped to one specification without a signature at most, and to a few with signatures, up to `DMI_TYPE_CANDIDATES` in all. As a structure is decoded, the specifications with signatures are tried in the order they are mapped, and the one without a signature is taken for the structures no signature matches, which are left undecoded if there is none. The fields of a signature are kept in the decoded structure, so that it is written back with it.

The specifications told apart by signatures decode into different structure types, e.g. `DMI_TYPE(intel_vpro)` and `DMI_TYPE(lenovo_tvt)`, so asking for the information of one of them never gives the other, see Structure types.

16.7. Groups

Firmware often lists the structures of a kind by a group association (type 14) of a well-known name, e.g. the Intel reference code lists its firmware version information by the `Firmware Version Info` group. A module names such groups along with the specifications of their members:

```
.groups = DMI_GROUPS({
    { "Firmware Version Info", &dmi_intel_fvi_spec },
    {}
})
```

Groups are not used to tell the structures, which their type numbers and signatures do, but to check them: the `group-assoc.member` lint rule reports the members of such a group which are not of the structure type of the specification, whatever the layout they are decoded by. A report tells either a structure the vendor places at a type number no module knows of, which calls for a relocation, or a group the firmware lists a structure in by mistake.

16.8. Modules of a program

A program brings structure types of its own by a module of its own, which it registers once, before the contexts which use it are created. The library links the registered modules through their `next` member, so a module is not constant:

```
static dmi_module_t example_module =
{
    .code     = "example",
    .name     = "Example extensions",
    .entities = (const dmi_entity_spec_t *[]){
        &example_spec,
        nullptr
    }
```

```
};

if (!dmi_module_register(&example_module))
    return 1;
```

A registered module is found by `dmi_module_find` and listed by `dmi_module_next` along with the built-in ones, and is enabled the same way. The library does not copy it, so it stays valid for as long as the program runs. Registration is not thread-safe, and belongs to the start of the program.

Chapter 17. Checking the data

The data is checked against the rules of the specification by `dmi_lint`, which `opendmi lint` is built on. The rules themselves are described in `opendmi-lint`; this chapter describes how they are checked, and how a structure type brings rules of its own.

17.1. Rules

A rule, `dmi_lint_rule_t`, is a code, e.g. `cache.size`, a check, a description, and the severities of the issues it finds in the two profiles: reading the data of a platform, and producing data which is expected to conform. The scope of a rule says when it is checked:

`DMI_LINT_SCOPE_ENTRY`
: Once, against the entry point.

`DMI_LINT_SCOPE_ENTITY`
: Against every structure of the table.

`DMI_LINT_SCOPE_TABLE`
: Once, after every structure has been checked, with the totals of the table, e.g. how many structures of each type it has.

The rules which apply to any structure belong to the library, and the ones which know a type belong to the specification of that type. The rules of the cache information structure are declared with the rest of its specification:

```
.lint_rules = DMI_LINT_RULES({
    DMI_LINT_RULE("cache.size", dmi_cache_lint_size, {
        .name              = "Installed size of the cache fits its "
                             "maximum size",
        .reader_severity   = DMI_LINT_SEVERITY_WARNING,
        .producer_severity = DMI_LINT_SEVERITY_ERROR
    }),
    {}
})
```

A module brings the rules of its types the same way, so the rules checked are the ones of the modules which are enabled.

17.2. Checks

A check is given the structure being checked, and reports what it finds with `dmi_lint_issue`, naming the attribute and the offset of the issue:

```
void dmi_cache_lint_size(dmi_lint_t *lint, const dmi_entity_t *entity)
{
    const dmi_cache_t *info = dmi_entity_info(entity, DMI_TYPE(cache));
```

```
    if ((info == nullptr) || (info->maximum_size == 0) ||
        (info->maximum_size == DMI_SIZE_MAX))
        return;

    if ((info->installed_size == DMI_SIZE_MAX) ||
        (info->installed_size <= info->maximum_size))
        return;

    dmi_lint_issue(lint, entity, "installed-size",
                   dmi_lint_entity_offset(lint, entity),
                   "installed size of %" PRIu64 " bytes is above the "
                   "maximum of %" PRIu64 " bytes",
                   info->installed_size, info->maximum_size);
}
```

A check reads the decoded structure rather than its bytes whenever it can, so that it sees the values the way the rest of the library does. The ones which catch what the decoder puts right, e.g. the words of a processor identifier stored the other way round, read the bytes with a reader, see Buffers, readers and writers. The severity of an issue is not the business of the check: it comes from the rule and the profile.

Many checks are not written at all: the rules of the values, e.g. `value.range` and `value.invalid-enum`, walk the attributes of every structure and check them against the limits and the name sets the attributes give, so a type whose attributes say what its values may be is checked by them.

17.3. Checking the data of a program

A program checks the data of an open context with `dmi_lint`, which reports every issue to a handler as it is found:

```
static void print_issue(void *data, const dmi_lint_issue_t *issue)
{
    (void)data;

    printf("%s: %s [%s]\n",
           dmi_name_lookup(&dmi_lint_severity_names, issue->severity),
           issue->message, issue->rule->code);
}

...

const dmi_lint_options_t options = {
    .profile = DMI_LINT_PROFILE_READER
};

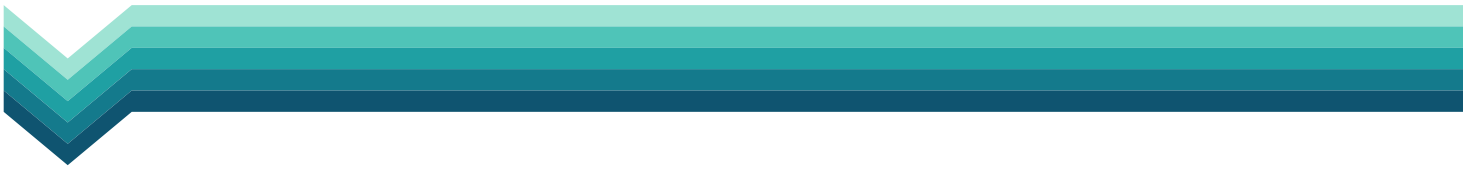
if (!dmi_lint(context, &options, print_issue, nullptr))
    return 1;
```

The options select the profile, the version of the specification to check against in place of the one of the

entry point, whether the rules disabled by default are checked too, and a filter which decides on every rule, which is how `--enable` and `--disable` of the tool are done.

PART VI

File formats



Chapter 18. Exported documents

The `opendmi export` command writes the SMBIOS data as a document of YAML, JSON or XML, for a program to read rather than for a person, see `opendmi-export`. The three formats hold the same document, which this chapter describes, and differ in the way it is written down, which the chapters about each of them describe. The document is what `opendmi import` is meant to read back, once it is implemented, see `opendmi-import`.

The text format `export` writes too is the output of `opendmi show`, which is meant for a person, and has no layout a program could rely on. It is not described here.

The formats available depend on how the tool was built, since each of them needs a library of its own: `libyaml` for YAML, `yajl` for JSON and `libxml2` for XML. YAML is the default format when it is available.

18.1. Layout of the document

A document holds the data of the entry point and the structures of the table, in the order the table holds them:

```
document
 ├─ entry            entry point data
 └─ table            structures of the table
     └─ entity       a structure
         ├─ header   handle, type, length, level, state, description
         ├─ either   attributes, properties, overlays   (decoded)
         └─ or       data, strings                      (raw)
```

Which structures the table holds is decided by the filter options of the command, the same way `opendmi show` decides it. By default the structures of the standard and the OEM-specific types are exported, and the inactive ones and the ones of the types no enabled module describes are not.

18.2. Entry point

The entry point is described by the version of SMBIOS it declares, `smbios-version`, e.g. `3.4`. The XML documents give the address and the size of the table as well.

18.3. Structures

Every structure is described by a header, which is the same for every structure, and by its content, which is either the values decoded from it or its raw bytes.

18.3.1. Header

`handle`
: Handle of the structure.

`type`
: Type of the structure, from 0 to 255.

`length`

Length of the structure in bytes, which counts the formatted area, the header included, the strings and the terminating null bytes, so that the next structure begins that many bytes after this one.

`level`

Version of the specification the layout of the structure follows, e.g. `3.3`, which is told from the length of its formatted area, or null if the structure has no versions or is not decoded. Not written in XML.

`state`

Flags of the state of the structure, which are any of:

`decoded`

Structure has been decoded.

`linked`

Handles the structure refers to have been resolved.

`valid`

Structure has been checked.

`incomplete`

Structure ends within a group of fields, so only the fields present in full are decoded, and its length matches no version of the specification.

`partial`

Structure follows an older version of the specification, and lacks some of the fields the decoder knows.

`description`

Name of the structure type, e.g. `Memory device`, in the language of the locale the tool runs in. Not written in XML.

18.3.2. Decoded structures

A structure the tool has decoded is described by its values:

`attributes`

Values of the fields of the structure, and of what is derived from them, named by the codes of the attributes of the structure type, e.g. `socket-designation`, in the order the type lists them. Attributes the version of the structure lacks are left out. The codes of the attributes do not change with the locale, unlike the labels `opendmi show` prints.

`properties`

String properties which structures of type 46 add to the structure, if any. Every property is described by its identifier, `id`, by its code, `code`, which is null or left out if the identifier is not known, and by its value, `value`, which is null or empty if the property holds no string.

`overlays`

Additional information entries of type 40 structures applied to the structure, which are written only if the global `--overlay` option is given and the structure has any. Every entry is described by:

`source`
: Handle of the type 40 structure holding the entry.

`index`
: Index of the entry within that structure, starting at 0.

`offset`
: Offset of the field of this structure the entry refers to.

`value`
: Value of the entry, as hexadecimal digits in the order of the bytes, or the way a person reads it with `--pretty`.

`string`
: String of the entry, or null if the entry has none.

18.3.3. Raw structures

A structure is written as its raw bytes if it is not decoded: if the `--dump` option is given, or if no enabled module describes its type, in which case it is exported only with the `--unknown` or `--all` option:

`data`
: Formatted area of the structure, the header included, encoded in Base64.

`strings`
: Strings of the structure, the first one being string 1, as they are, with no trailing spaces removed. Left out if the structure has no strings.

Raw structures carry all their bytes, so the bytes of the whole table are kept by the document written with `--all --dump`. The end-of-table structure (type 127) is written with the header only.

18.4. Schemas

The layout of the documents is described by schemas, which are kept in `opendmi/share/opendmi` and are published on the site of the project, at the addresses the schemas give as their identifiers:

Format	Schema	Address
YAML	JSON Schema, applied to the data a YAML reader loads out of the document	`https://opendmi.org/schemas/opendmi.yml`
JSON	JSON Schema	`https://opendmi.org/schemas/opendmi.json`
XML	XML Schema, which is also the namespace of the documents	`https://opendmi.org/schemas/opendmi.xsd`

The schemas describe the layout the documents share, and not the attributes of every structure type, which the schemas leave open.

Chapter 19. Values

Values of the attributes are written by their codes, the way a program matches them, unless the `--pretty` option is given, which writes them the way `opendmi show` prints them for a person. The codes do not change with the locale, and are what this chapter describes. Which of the values a format writes as numbers or booleans of its own, rather than as strings, is described by the chapter about the format.

19.1. Kinds of values

The value of an attribute is written according to its kind:

Kind	Written as	Example
String	String of the structure, as it is.	`DIMM 0`
Boolean	`true` or `false`, or the codes the attribute gives to the two states.	`true`
Integer	Decimal number, or hexadecimal one with the `0x` prefix and uppercase digits for the attributes which are identifiers or masks rather than quantities.	`100`, `0x106A4`
Fixed-point number	Decimal number with as many fractional digits as it needs, and at least one.	`1.2`
Size	Number of bytes.	`1310720`
Address	Hexadecimal number with the `0x` prefix and 8 digits, or 16 if the table uses 64-bit addresses.	`0x000F0000`
Handle	Hexadecimal number with the `0x` prefix and 4 uppercase digits.	`0x0010`
Enumeration	Code of the value, or the value as a hexadecimal number if the specification does not define it.	`write-back`, `0x1f`
Set of flags	Every flag the attribute knows, by its code, with a boolean telling if it is set. XML documents give the value of the whole set as a hexadecimal number too.	`burst: false`
Version	Numbers of the version separated by dots.	`2.0`, `3.1.1`
Date	Date in the ISO 8601 format.	`2024-11-05`
UUID	UUID in the canonical format, with uppercase digits.	`31343834-3438-4247-3839-34343834594E`
Binary data	Hexadecimal digits of the bytes in their order, a MAC address with colons between the bytes, and an IP address in its usual notation.	`0a0b`, `00:1a:2b:3c:4d:5e`
Structure	Mapping of the codes of the nested attributes to their values.	
Array	Sequence of the values of the items.	

Two values are written the same way whatever the kind of the attribute is:

Not specified

The value the data says is not given, e.g. string 0, or the value the specification reserves for that, is written as null, or as an empty element in XML.

Unknown

The value the specification reserves for a value which is not known, where it reserves one, is written as the string `unknown`.

19.2. Units

Numbers are written in the units the attribute is measured in, which are not written in the value: an XML element carries them in its `units` attribute, e.g. `mhz`, and the other formats do not write them. Sizes are always written in bytes.

19.3. Values for a person

With `--pretty`, values are written the way a person reads them: enumerations and booleans by their names, e.g. `Write-back` and `yes`, sizes in the largest unit which keeps them whole, e.g. `1280 KiB`, binary data with uppercase digits separated by spaces, and numbers along with the names of their units, e.g. `4300 MHz`, which XML writes in the `units` attribute instead. Names of the attributes, handles and sets of flags are written the same way as without the option.

Chapter 20. YAML

The YAML document is a single mapping, whose `entry` key holds the entry point and whose `table` key holds the sequence of the structures:

```
entry:
  smbios-version: '3.4'
table:
- handle: 0x000a
  type: 7
  length: 39
  level: '3.1'
  state: [decoded]
  description: Cache information
  attributes:
    socket-designator: "L1 - Cache"
    level: 1
    socketed: false
    location: "internal"
    mode: "write-back"
    installed-size: 1310720
    supported-sram:
      burst: false
      pipeline-burst: true
    speed: 1
    associativity: "8-way"
```

A structure which is not decoded holds its raw bytes instead of the attributes:

```
- handle: 0x0005
  type: 11
  length: 21
  level: '2.0'
  state: [decoded]
  description: OEM strings
  data: !!binary |-
    CwUFAAE=
  strings:
  - "Default string"
```

20.1. Scalars

Scalars are written so that a YAML reader resolves each of them to the type it has in the document, and nothing else:

- Handles, types, lengths, offsets and the values of the numeric attributes, i.e. integers, fixed-point

numbers, sizes, addresses and handles, are plain scalars, which the readers resolve as numbers, the hexadecimal ones included, e.g. `0x000a`.

- Booleans are plain scalars, `true` or `false`, and so are the codes of the states, the ones a boolean attribute gives to its states, and the string `unknown`.
- Versions are single-quoted, so that `2.0` is not read as a number.
- Strings of the structures, codes of the enumerations, dates and UUIDs are double-quoted, so that e.g. `yes` or `2024-11-05` is not read as a boolean or a timestamp.
- Values which are not specified are `null`.
- With `--pretty`, every value of an attribute is a double-quoted string.

A string which is not valid UTF-8, which the firmware may hold, is written as its bytes, in a `!!binary` scalar, since YAML has no strings of other encodings. The raw bytes of a structure, `data`, are always written this way. Readers load `!!binary` scalars as byte strings rather than as text.

20.2. Structures

The header of a structure gives every key described in Exported documents, and `level` is `null` for the structures which have no versions.

`properties` and `overlays` are sequences of mappings of the keys the properties and the entries have, which are all present, with `null` for the ones which have no value. `strings` is a sequence of the strings.

20.3. Schema

The document is described by a JSON Schema, `https://opendmi.org/schemas/opendmi.yml`, which is written in YAML and is applied to the data a reader loads out of the document, e.g. with `check-jsonschema`:

```
$ opendmi export -o smbios.yaml
$ check-jsonschema --schemafile opendmi.yml smbios.yaml
```

The schema expects the plain hexadecimal scalars to be loaded as numbers, which is what readers following YAML 1.1 or the core schema of YAML 1.2 do, and the `!!binary` scalars to be loaded as byte strings, which JSON Schema knows no type of.

Chapter 21. JSON

The JSON document is a single object, whose `entry` member holds the entry point and whose `table` member holds the array of the structures:

```
{
    "entry": {
        "smbios-version": "3.4"
    },
    "table": [
        {
            "handle": 10,
            "type": 7,
            "length": 39,
            "level": "3.1",
            "state": [
                "decoded"
            ],
            "description": "Cache information",
            "attributes": {
                "socket-designator": "L1 - Cache",
                "level": 1,
                "socketed": false,
                "mode": "write-back",
                "installed-size": 1310720,
                "supported-sram": {
                    "burst": false,
                    "pipeline-burst": true
                },
                "speed": 1,
                "associativity": "8-way"
            }
        }
    ]
}
```

A structure which is not decoded holds its raw bytes instead of the attributes:

```
{
    "handle": 5,
    "type": 11,
    "length": 21,
    "level": "2.0",
    "state": [
        "decoded"
    ],
    "description": "OEM strings",
```

```
    "data": "CwUFAAE=",
    "strings": [
        "Default string"
    ]
}
```

21.1. Values

Values are written the same way as in YAML, so that a JSON document and a YAML one, loaded by their readers, hold the same data:

- Handles, types, lengths, the identifiers of the properties, the indexes and the offsets of the additional information entries and the values of the numeric attributes, i.e. integers, fixed-point numbers, sizes, addresses and handles, are numbers. JSON has no hexadecimal numbers, so the ones the other formats write in hexadecimal are written as decimal ones, e.g. `10` for the handle `0x000a`.
- Booleans are `true` or `false`, and so are the flags of the sets. The codes a boolean attribute may give to its states instead are strings.
- Other values, e.g. strings, codes of the enumerations, versions, dates, UUIDs and the string `unknown`, are strings.
- Values which are not specified are `null`.
- With `--pretty`, every value of an attribute is a string.

Numbers are written with all their digits, and sizes and addresses may take 64 bits, which is more than the readers keeping the numbers as floating-point ones, e.g. the ones of JavaScript, hold exactly.

A string which is not valid UTF-8, which the firmware may hold, has the bytes which are not valid replaced by the replacement character, U+FFFD, since JSON has no strings of other encodings. The raw strings of a structure are written this way too, and its raw bytes, `data`, as a Base64 string.

21.2. Structures

The header of a structure gives every member described in Exported documents, and `level` is `null` for the structures which have no versions.

`properties` and `overlays` are arrays of objects of the members the properties and the entries have, which are all present, with `null` for the ones which have no value. `strings` is an array of the strings.

21.3. Schema

The document is described by a JSON Schema, `https://opendmi.org/schemas/opendmi.json`:

```
$ opendmi export -f json -o smbios.json
$ check-jsonschema --schemafile opendmi.json smbios.json
```

Chapter 22. XML

The XML document is made of the elements of the `https://opendmi.org/schemas/opendmi.xsd` namespace, written with the `dmi` prefix. The `dmi:dump` element holds the `dmi:entry` element of the entry point, and a `dmi:entity` element for every structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<dmi:dump created-at="2026-09-30T02:40:02Z"
          xsi:schemaLocation="https://opendmi.org/schemas/opendmi.xsd
                              https://opendmi.org/schemas/opendmi.xsd"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:dmi="https://opendmi.org/schemas/opendmi.xsd">
  <dmi:entry smbios-version="3.4"
             table-area-address="0x20"
             table-area-size="1920"/>
  <dmi:entity handle="0x000a" type="7" length="39" state="decoded">
    <dmi:cache>
      <socket-designator>L1 - Cache</socket-designator>
      <level>1</level>
      <socketed>false</socketed>
      <mode>write-back</mode>
      <installed-size>1310720</installed-size>
      <supported-sram value="0x10">
        <flag name="burst">false</flag>
        <flag name="pipeline-burst">true</flag>
      </supported-sram>
      <speed units="nanosecond">1</speed>
      <associativity>8-way</associativity>
    </dmi:cache>
  </dmi:entity>
</dmi:dump>
```

A structure which is not decoded holds its raw bytes instead of the attributes:

```
<dmi:entity handle="0x0005" type="11" length="21" state="decoded">
  <dmi:data><![CDATA[CwUFAAE=]]></dmi:data>
  <dmi:strings>
    <dmi:string index="1">Default string</dmi:string>
  </dmi:strings>
</dmi:entity>
```

22.1. Document and entry point

`dmi:dump` has the `created-at` attribute, which gives the time the document was written at, in UTC, e.g. `2026-09-30T02:40:02Z`, and the `xsi:schemaLocation` attribute, which tells the validators and the editors where the schema of the namespace is published.

`dmi:entry` has the attributes:

`smbios-version`
: Version of SMBIOS the entry point declares.

`table-area-address`
: Address of the table, as a hexadecimal number with lowercase digits.

`table-area-size`
: Size of the table in bytes.

22.2. Structures

`dmi:entity` gives the header of the structure in its attributes: `handle`, as a hexadecimal number of 4 lowercase digits, `type`, `length` and `state`, which is the list of the flags of the state separated by spaces, and is empty if none is set. The level and the name of the type are not written.

The content of the element is, in this order:

- for a decoded structure, the element of its values, which is named by the code of the structure type, e.g. `dmi:cache`, then the `dmi:properties` element, if the structure has any properties, and the `dmi:overlays` element, if it has any additional information entries applied;
- for a structure which is not decoded, the `dmi:data` element, then the `dmi:strings` element, if the structure has any strings;
- nothing for the end-of-table structure.

22.3. Values

Elements of the values are named by the codes of the attributes, and are not qualified by the namespace. A value is the text of its element, and:

- the value which is not specified is an empty element;
- a nested structure is an element of the elements of its attributes;
- an array is an element of `item` elements, one for every item;
- a set of flags is an element of `flag` elements, one for every flag, whose `name` attribute gives the code of the flag, since the codes are not always valid names of elements, e.g. `5v`, and whose text is `true` or `false`, while the `value` attribute of the set gives the value of the whole set as a hexadecimal number;
- a number measured in units has the `units` attribute, which gives the code of the unit, e.g. `mhz`, or its name with `--pretty`, e.g. `MHz`.

22.4. Properties and overlays

`dmi:properties` holds a `dmi:property` element for every property, whose `id` attribute gives the identifier of the property as a hexadecimal number, and whose `code` attribute, which is left out if the identifier is not known, gives its code. The text of the element is the value of the property, and is empty if the property holds no string.

`dmi:overlays` holds a `dmi:overlay` element for every additional information entry, whose `source`, `index` and `offset` attributes give the handle of the structure holding the entry, the index of the entry and

the offset of the field it refers to. The `dmi:value` element holds the value of the entry, and the `dmi:string` element, which is left out if the entry has none, its string.

22.5. Raw data

`dmi:data` holds the formatted area of the structure, the header included, encoded in Base64. `dmi:strings` holds a `dmi:string` element for every string of the structure, whose `index` attribute gives its number, starting at 1.

Characters XML does not allow, e.g. the control ones, and bytes which are not valid UTF-8, are replaced by the replacement character, U+FFFD.

22.6. Schema

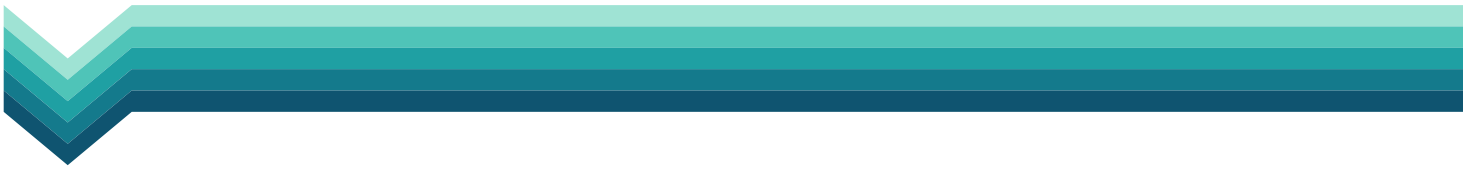
The document is described by an XML Schema, which is published at the address of the namespace, `https://opendmi.org/schemas/opendmi.xsd`:

```
$ opendmi export -f xml -o smbios.xml
$ xmllint --noout --schema opendmi.xsd smbios.xml
```

The elements of the values of the structures are named by the structure types and their attributes, which the schema does not list, and XML Schema 1.0 does not allow the elements of a namespace which are not declared to be told from the ones which are, so the schema checks the content of `dmi:entity` laxly: the elements it declares, e.g. `dmi:data`, are checked wherever they are found, and the order of the content is not checked.

PART VII

Specifications



Chapter 23. Reading the layouts

The layout of a structure is given by a table with a row per field:

Offset

Offset of the field, counted from the beginning of the structure, header included, the way the specification counts it.

Version

Version of the specification the field appears in, e.g. `2.1+`. The data of the platforms following an older version ends before the field.

Name

Name of the field.

Length

Size of the field: `BYTE`, `WORD`, `DWORD` and `QWORD` stand for 8-, 16-, 32- and 64-bit values, stored in the little-endian byte order. `n BYTEs` gives the size through the number `n` a field before it holds, e.g. `2n BYTEs` for `n` values of a `WORD` each, and `Varies` stands for a field whose size depends on the data.

Value

Kind of the value the field holds: `NUMBER` for a number, `ENUM` for one of the values an enumeration defines, `BITFIELD` for a set of bits, `STRING` for the number of a string of the structure, `HANDLE` for the handle of another structure, `ADDRESS` for an address in memory or in the I/O space, and `SPEED` for a speed, in the units the description gives. A number stands for the fixed value of the field, e.g. the type of the structure, and `Varies` for a value the description explains.

Description

Meaning of the field.

Values of the enumerations are given by tables of their hexadecimal and decimal values and their meanings, and the bits of the sets of bits by tables of their positions and meanings.

Chapter 24. Common structures

Chapter 25. Acer-specific structures

Acer laptops carry the firmware of other vendors, e.g. Insyde, and the structures of Acer are decoded for the platforms whose system vendor is Acer.

25.1. Acer Hotkey Functions (Type 170)

Source: the Acer WMI driver of Linux, `acer-wmi.c`, dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells the functions of the hotkeys of a laptop, which the Acer WMI driver of Linux reads. Some laptops of other brands, e.g. Fujitsu-Siemens, Medion, Lenovo and eMachines, are said to carry it too.

The Acer WMI driver reads the structure up to the number of the key of the communication function, at `0x0E`. In the data corpus, that byte begins a list of 4-byte hotkeys, which runs to the end of the structure: the number of a key, a byte of `1` or `2`, the same for all the hotkeys of a structure, and the function of the key, as bits of the bitmap of its button group: the key of the communication function has all the bits of the communication bitmap, e.g. those of WiFi and Bluetooth. The high bits of the key number tell the group: `0x0_` communication, `0x4_` media, `0x6_` display and `0x8_` others, and the functions of the keys of the media and display groups add up to their bitmaps.

Structures of `0x0F` bytes end with the number of the key of the communication function, and hold no hotkeys.

Table 1. Acer Hotkey Functions structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`170`	Acer Hotkey Functions indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Communication Button Functions	`WORD`	`BITFIELD`	Bit 0: WiFi, bit 6: 3G, bit 7: WiMAX, bit 11: Bluetooth, bit 14: radio button
`0x06`	2.0+	Application Button Functions	`WORD`	`BITFIELD`	
`0x08`	2.0+	Media Button Functions	`WORD`	`BITFIELD`	
`0x0A`	2.0+	Display Button Functions	`WORD`	`BITFIELD`	
`0x0C`	2.0+	Other Button Functions	`WORD`	`BITFIELD`	
`0x0E`	2.0+	Hotkeys	4 BYTEs each	Varies	Hotkeys, see below; the key of the first one is the communication function key number

Table 2. Hotkey

Offset	Name	Length	Description
`0x00`	Key	`BYTE`	Number of the key, whose high bits tell the button group
`0x01`	Kind	`BYTE`	Meaning not established, `1` or `2` in the known data
`0x02`	Function	`WORD`	Function of the key, as bits of the bitmap of its button group

25.2. Acer Device List (Type 171)

Source: reverse engineered from the dumps of the data corpus.

Lists the devices of a laptop by their PCI or USB vendor and device IDs. Devices run to the end of the structure, which carries no number of them of its own, and the absent ones have zero IDs.

The IDs are the ones of the PCI and USB ID databases, which tell the devices of the data corpus as follows.

Table 3. Devices of the data corpus by kind

Kind	Devices
`0x01`	Graphics of Intel (`8086:5917`, `8086:191B`) and NVIDIA (`10DE:174D`, `10DE:1C8D`, `10DE:13D8`), and a Realtek card reader (`10EC:5287`) in `acer/aspire-a515-52.bin`. `acer/predator-g9-592.bin` gives the kind to both its graphics
`0x02`	Graphics of NVIDIA (`10DE:174D`, `10DE:1D10`), and a Realtek Ethernet adapter (`10EC:8168`) in `acer/aspire-a515-52g.bin`
`0x03`	Graphics of NVIDIA, the same ones as kind `2` on the same laptops, Realtek card readers (`10EC:5287`, `0BDA:0316`) and a Phison NVMe controller (`1987:5012`)
`0x04`	USB webcams of Quanta, Realtek and Sunplus
`0x05`	HD Audio controllers of Intel
`0x07`	Wireless network adapters of Intel and Qualcomm Atheros
`0x08`	USB Bluetooth adapters of Foxconn, Intel and Qualcomm Atheros
`0x19`	Host bridge of Intel Coffee Lake (`8086:3E34`) in `acer/aspire-a515-52.bin` and `acer/aspire-a515-52g.bin`, the two variants of the same laptop
`0x0D`, `0x11`, `0x13`, `0x15`, `0x16`	None, their devices are always absent

Kinds `4`, `5`, `7` and `8` are given to the devices of a single class on all the laptops of the corpus, while kinds `1` to `3` are given to devices of different classes, and are thus no classes. Kind `25` is found on a single laptop, and the other kinds with no device at all, so their meaning is not established.

Table 4. Acer Device List structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`171`	Acer Device List indicator

Offset	Version	Name	Length	Value	Description
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Devices	5 BYTEs each	Varies	Devices, see below

Table 5. Device

Offset	Name	Length	Description
`0x00`	Kind	`BYTE`	Kind of the device, see below
`0x01`	Vendor ID	`WORD`	PCI or USB vendor ID, zero if the device is absent
`0x03`	Device ID	`WORD`	PCI or USB device ID, zero if the device is absent

The kinds whose meaning the data corpus establishes are named by it, and the other kinds found in the corpus are named by their values. Other values are taken as they are, rather than as invalid ones.

Table 6. Device kind

Value	Meaning
`0x01`, `0x02`, `0x03`	Unknown, e.g. graphics, network adapters and drives
`0x04`	Webcam
`0x05`	Audio
`0x07`	Wireless network adapter
`0x08`	Bluetooth adapter
`0x19`	Unknown, the host bridge on a single laptop
`0x0D`, `0x11`, `0x13`, `0x15`, `0x16`	Unknown

Chapter 26. AMI-specific structures

AMI firmware carries structures of its own on the boards of several vendors. The structures are decoded for the platforms whose firmware vendor is AMI.

26.1. AMI FireWire GUID (Type 139)

Source: reverse engineered from the dumps of the data corpus.

Keeps the data holding the GUID of the IEEE 1394 (FireWire) controller of the board, under the name `V1394GUID`. The data corpus holds a single structure of the kind, on ASUS M5A97 PRO, so the layout of the data is not established: it begins like the bus information block of the configuration ROM of IEEE 1394, with the lengths of the block and of its CRC, `04 04`, followed by the CRC and the options of the bus, and the OUI of VIA, `00 40 63`, is found further on.

The structure is told by its signature at offset `0x04` and by its length. The data identifies the board, and is replaced when the table is anonymized.

Table 7. AMI FireWire GUID structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`139`	AMI FireWire GUID indicator
`0x01`	2.0+	Length	`BYTE`	`0x36`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Signature	8 BYTEs	`FE DC BA 98 76 54 32 10`	
`0x0C`	2.0+	Data	40 BYTEs	Varies	Data holding the GUID of the controller
`0x34`	2.0+	Unknown	`BYTE`	`0`	Meaning not established
`0x35`	2.0+	Name	`BYTE`	`STRING`	Name of the data, `V1394GUID`

Chapter 27. Apple-specific structures

Macs carry structures of Apple, which OpenCore also adds to the tables of the other systems it boots macOS on. The structures are decoded for the platforms whose firmware vendor is Apple.

27.1. Apple Firmware Volume Information (Type 128)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Tells the features of the firmware, which macOS and its boot loader read, and the layout of its flash memory. The features are 64 bits wide: the structure holds their low 32 bits, and newer firmware their high 32 bits as the extended features. `apple/imac11-3.bin` carries the structure of 88 bytes, without the extended features.

Table 8. Apple Firmware Volume Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`128`	Apple Firmware Volume Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Number of Regions	`BYTE`	Varies	Number of the regions of the flash memory
`0x05`	2.0+	Reserved	3 `BYTE`	`0`	
`0x08`	2.0+	Firmware Features	`DWORD`	Bit field	Features of the firmware, bits 0 to 31, see Firmware Features
`0x0C`	2.0+	Firmware Features Mask	`DWORD`	Bit field	Features the firmware tells
`0x10`	2.0+	Region Type Map	8 `BYTE`	Enum	Types of the regions, see Region Types
`0x18`	2.0+	Flash Map	64 `BYTE`	Varies	Start and end addresses of the regions, a `DWORD` each
`0x58`	2.0+	Extended Firmware Features	`DWORD`	Bit field	Features of the firmware, bits 32 to 63, on newer firmware
`0x5C`	2.0+	Extended Firmware Features Mask	`DWORD`	Bit field	Extended features the firmware tells, on newer firmware

27.1.1. Apple Firmware Volume Information — Firmware Features

Table 9. Apple Firmware Volume Information: Firmware Features

Bit	Meaning
0	Supports booting legacy operating systems (CSM)
1	Supports booting from optical drives

Bit	Meaning
2	Supports target disk mode
4	Supports network boot
5	Supports the recovery embedded into the firmware (SlingShot)
8	Supports wireless networks in the firmware
10	Bit 0 of the platform security policy
11	Bit 1 of the platform security policy
12	Firmware is locked
14	Supports USB 2.0 in the firmware
17	USB ports are not substituted with others by macOS
19	Supports APFS
20	Supports APFS, unused by the known Macs
22	Supports Apple tamper resistant boot X
24	Supports the platform security policy
25	Supports the extended features
28	Memory map is not adjusted by the boot loader on hibernation wake
29	Supports UEFI Windows boot
31	Memory map is not adjusted by the boot loader for the boot script
35	Supports the large BaseSystem, which macOS 12 requires

The other bits are of no known meaning.

27.1.2. Apple Firmware Volume Information — Region Types

Table 10. Apple Firmware Volume Information: Region Types

Value	Meaning
`0`	Entry is not used
`1`	Recovery firmware
`2`	Main firmware
`3`	Non-volatile variables (NVRAM)
`4`	Configuration
`5`	Diagnostics vault

27.2. Apple Memory SPD Data (Type 130)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Holds the serial presence detect (SPD) data of a memory module, or a part of it, e.g. the first 176 bytes of the SPD data of the DDR3 modules of `apple/imac11-3.bin`. The firmware adds a structure for each memory module. The Intel reference code gives type 130 to the Intel Active Management Technology information too.

Table 11. Apple Memory SPD Data structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`130`	Apple Memory SPD Data indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	Varies	Handle of the memory device (type 17) of the module
`0x06`	2.0+	Offset	`WORD`	Varies	Offset of the data within the SPD data of the module
`0x08`	2.0+	Size	`WORD`	Varies	Length of the data, in bytes
`0x0A`	2.0+	Data	Varies	Varies	SPD data

27.3. Apple Processor Type Information (Type 131)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Tells the type of the processor, by which macOS names it, e.g. `0x0701` for the first generation of Intel Core i7 in `apple/imac11-3.bin`. The Intel reference code gives type 131 to the Intel vPro information too.

Table 12. Apple Processor Type Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`131`	Apple Processor Type Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Processor Type	`WORD`	Varies	Class of the processor in the high byte, see Processor Classes, and its generation within the class in the low byte
`0x06`	2.0+	Reserved	2 `BYTE`	`0`	On newer firmware

27.3.1. Apple Processor Type Information — Processor Classes

Table 13. Apple Processor Type Information: Processor Classes

Value	Meaning
`0x00`	Unknown
`0x02`	Intel Core
`0x03`	Intel Core 2
`0x04`	Intel Xeon (Penryn)
`0x05`	Intel Xeon (Nehalem)
`0x06`	Intel Core i5
`0x07`	Intel Core i7
`0x09`	Intel Core i3
`0x0A`	Intel Xeon E5
`0x0B`	Intel Core M
`0x0C`	Intel Core m3
`0x0D`	Intel Core m5
`0x0E`	Intel Core m7
`0x0F`	Intel Xeon W
`0x10`	Intel Core i9

27.4. Apple Processor Bus Speed Information (Type 132)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Tells the speed of the bus of the processor, e.g. `4800` for the QuickPath Interconnect of 4.8 GT/s in `apple/imac11-3.bin`. OpenCore sets it to four times the external clock of the processor.

Table 14. Apple Processor Bus Speed Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`132`	Apple Processor Bus Speed Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Processor Bus Speed	`WORD`	Varies	Speed of the bus, in megatransfers per second

27.5. Apple Platform Feature Information (Type 133)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Tells the features of the platform, which macOS reads. Older Macs carry no such structure.

Table 15. Apple Platform Feature Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`133`	Apple Platform Feature Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Platform Feature	`QWORD`	Bit field	Features of the platform, see Platform Features

27.5.1. Apple Platform Feature Information — Platform Features

Table 16. Apple Platform Feature Information: Platform Features

Bit	Meaning
1	System memory is soldered, and cannot be replaced
2	Headless GPU, of no display connectors, is present
3	Supports host power management
4	Supports the chime at boot

The other bits are of no known meaning.

27.6. Apple SMC Version Information (Type 134)

Source: the `AppleSmBios.h` header of OpenCore, `Include/Apple/IndustryStandard`.

Tells the version of the firmware of the System Management Controller (SMC). Newer Macs carry no such structure.

Table 17. Apple SMC Version Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`134`	Apple SMC Version Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	SMC Version	16 `BYTE`	Varies	Version of the SMC firmware, e.g. `2.15f7` as text followed by bytes of zero

Chapter 28. Cisco-specific structures

Cisco UCS servers carry structures of Cisco, which describe the adapters installed in them. The structures are decoded for the platforms whose firmware vendor is Cisco: other Cisco appliances carry the firmware of other vendors, e.g. Dell.

28.1. Cisco PCI Slot Buses (Type 201)

Source: reverse engineered from the dumps of the data corpus.

Lists the PCI buses behind a slot of a Cisco UCS server, presumably. The numbers match the adapters the PCI adapter information (type 202) gives for the same slot: an adapter with a PCI switch, e.g. Cisco UCS VIC 1455, has several buses, and the other adapters one.

Table 18. Cisco PCI Slot Buses structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`201`	Cisco PCI Slot Buses indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure, `0x06` + Bus Count
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Slot	`BYTE`	`STRING`	Slot, as the PCI adapter information names it, e.g. `SlotID:1`, `SlotID:MLOM`, `SlotID:MRAID` or `SlotID:L`
`0x05`	2.0+	Bus Count	`BYTE`	`NUMBER`	Number of buses
`0x06`	2.0+	Buses	Bus Count BYTEs	`NUMBER`	PCI bus numbers, e.g. `5E 5F 61 62 63 64` for Cisco UCS VIC 1455

28.2. Cisco PCI Adapter Information (Type 202)

Source: reverse engineered from the dumps of the data corpus.

Describes an adapter of a Cisco UCS server: its PCI identifiers and class, its model, the slot it is installed in, and the version of its firmware. There is a structure for each adapter, including the ones on board, e.g. the modular LAN on motherboard (`SlotID:MLOM`), the modular RAID controller (`SlotID:MRAID`) and the LAN on motherboard (`SlotID:L`).

The identifiers are held in the big-endian byte order, unlike the other fields of SMBIOS, and match the PCI ID database and the models the structures name, e.g. `1137:0042` of subsystem `1137:0217` for Cisco UCS VIC 1455, `8086:1521` for Intel I350, `1000:0014` of subsystem `1137:020E` for the Cisco 12G modular RAID controller, and `8086:1563` for Intel X550. The class code is laid out as the configuration space of the function holds it.

Table 19. Cisco PCI Adapter Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`202`	Cisco PCI Adapter Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x15`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Version	`BYTE`	`NUMBER`	Version of the structure, presumably, `3` in all known data
`0x05`	2.0+	Vendor ID	`WORD`	`NUMBER`	PCI vendor ID, big-endian
`0x07`	2.0+	Device ID	`WORD`	`NUMBER`	PCI device ID, big-endian
`0x09`	2.0+	Programming Interface	`BYTE`	`NUMBER`	Programming interface of the class
`0x0A`	2.0+	Subclass	`BYTE`	`NUMBER`	PCI subclass, e.g. `0x04` for the RAID controllers
`0x0B`	2.0+	Class	`BYTE`	`NUMBER`	PCI class, e.g. `0x01` for the storage and `0x02` for the network controllers
`0x0C`	2.0+	Revision ID	`BYTE`	`NUMBER`	Revision ID of the function, e.g. `0xA2` for Cisco UCS VIC 1455
`0x0D`	2.0+	Model	`BYTE`	`STRING`	Model of the adapter, e.g. `Cisco UCS VIC 1455`
`0x0E`	2.0+	Slot	`BYTE`	`STRING`	Slot the adapter is installed in, e.g. `SlotID:1`
`0x0F`	2.0+	Subsystem Vendor ID	`WORD`	`NUMBER`	PCI subsystem vendor ID, big-endian
`0x11`	2.0+	Subsystem ID	`WORD`	`NUMBER`	PCI subsystem ID, big-endian
`0x13`	2.0+	Firmware Version	`BYTE`	`STRING`	Version of the firmware of the adapter, `N/A` if it has none
`0x14`	2.0+	Reserved	`BYTE`	`0`	

Chapter 29. Dell-specific structures

29.1. Dell BIOS Flags (Type 177)

Source: the Dell SMBIOS WMI driver of Linux, `dell-smbios-wmi.c`, dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells the features of the firmware the Dell drivers rely on, of which only the support of ACPI WMI is known. dmidecode reads the flags as a `QWORD`, and the Dell SMBIOS WMI driver of Linux as a `WORD`; the bytes past the first word are not known to carry anything. The driver reads the structures of 12 bytes and of handle `0xB100` only, which every structure of the data corpus is.

Table 20. Dell BIOS Flags structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`177`	Dell BIOS Flags indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Flags	`WORD`	`BITFIELD`	Bit 1: ACPI WMI is supported
`0x06`	2.0+	Reserved	6 BYTEs	Varies	Rest of the flags, `0` in all known data

29.2. Dell Hotkeys (Type 178)

Source: the Dell WMI driver of Linux, `dell-wmi-base.c`, and the dumps of the data corpus.

Maps the scan codes of the hotkeys of a laptop to their functions, which the Dell WMI driver of Linux reads. Mappings run to the end of the structure, which carries no number of them of its own. Key code `0xFFFF` is found for the keys of no known function, which the driver takes for an unknown key, while key code `0x00FF` stands for a function of its own, the third programmable key. The structures of the data corpus refer to one string of spaces, which no field refers to.

Table 21. Dell Hotkeys structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`178`	Dell Hotkeys indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Mappings	4 BYTEs each	Varies	Mappings, see below

Table 22. Mapping

Offset	Name	Length	Description
`0x00`	Scan Code	`WORD`	Scan code of the key, as the firmware reports it in the WMI events of the hotkeys
`0x02`	Key Code	`WORD`	Function of the key, as the Dell WMI driver numbers the functions

29.3. Dell Revisions and IDs (Type 208)

Source: Dell `libsmbios`, `doc/smbios23.xml`.

The source gives the structure a length of `0x0C`, while the firmware of Dell systems writes structures of other lengths too: shorter ones end after the system ID extension, at `0x0A`, and have no dates, and longer ones carry bytes past the first power-on date which no source describes.

Table 23. Dell Revisions and IDs structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.2+	Type	`BYTE`	`208`	Dell Revision and IDs indicator
`0x01`	2.2+	Length	`BYTE`	Varies	Length of the structure, `0x0C` according to the source
`0x02`	2.2+	Handle	`WORD`	Varies	
`0x04`	2.2+	Dell Implementation, Major Revision	`BYTE`	`NUMBER`	
`0x05`	2.2+	Dell Implementation, Minor Revision	`BYTE`	`NUMBER`	
`0x06`	2.2+	Dell System ID	`BYTE`	`NUMBER`	A value of `0xFE` indicates that the actual system ID is stored in the system ID extension field.
`0x07`	2.2+	Dell Hardware Revision	`BYTE`	`NUMBER`	
`0x08`	2.2+	Dell System ID Extension	`WORD`	`NUMBER`	The field is valid only when system ID field is set to `0xFE`.
`0x0A`	2.2+	Manufacture Date	`BYTE`	`STRING`	
`0x0B`	2.2+	First Power-on Date	`BYTE`	`STRING`	

29.4. Dell Parallel Port (Type 209)

Source: Dell `libsmbios`, `doc/smbios23.xml`.

Table 24. Dell Parallel Port structure

Offset	Version	Name	Length	Value	Description
0x00	2.2+	Type	BYTE	209	Dell Parallel Port indicator
0x01	2.2+	Length	BYTE	0x0C	Length of the structure
0x02	2.2+	Handle	WORD	Varies	
0x04	2.2+	Base I/O Address	WORD	ADDRESS	
0x06	2.2+	IRQ	BYTE	NUMBER	
0x07	2.2+	Connector Type	BYTE	ENUM	See Connector types for details
0x08	2.2+	Connector Pin-out	BYTE	ENUM	See Connector pin-outs for details
0x09	2.2+	Capabilities	WORD	BITFIELD	See Capabilities for details
0x0B	2.2+	Security Settings	BYTE	ENUM	See Security settings for details

29.4.1. Dell Parallel Port — Connector types

Table 25. Dell Parallel Port: Connector type field

Hex Value	Dec Value	Meaning
0x01	1	Other
0x02	2	Unknown
0x03	3	DB-25 Female IEEE 1284-A Receptacle
0x04	4	DB-25 Male IEEE 1284-A Plug
0x05	5	Centronics IEEE 1284-B Receptacle
0x06	6	Mini-Centronics IEEE 1284-C Receptacle
0x07	7	Proprietary
0xA0	160	Centronics-14
0xA1	161	DB-36 Female
0xA2	162	Mini-Centronics-20

29.4.2. Dell Parallel Port — Connector Pin-outs

Table 26. Dell Parallel Port: Connector pin-out field

Hex Value	Dec Value	Meaning
0x01	1	Other
0x02	2	Unknown
0x03	3	XT/AT
0x04	4	PS/2

Hex Value	Dec Value	Meaning
`0x05`	`5`	IEEE 1284
`0x06`	`6`	Proprietary
`0xA0`	`160`	PC-98
`0xA1`	`161`	PC-98-Hireso
`0xA2`	`162`	PC-H98
`0xA3`	`163`	PC-98Note
`0xA4`	`164`	PC-98Full

29.4.3. Dell Parallel Port — Capabilities

Table 27. Dell Parallel Port: Capabilities field

Word Bit Position	Meaning
Bit 0	XT/AT
Bit 1	PS/2
Bit 2	ECP
Bit 3	EPP
Bit 4	PC-98
Bit 5	PC-98-Hireso
Bit 6	PC-H98
Bits 7:14	Reserved
Bit 15	Device supports DMA

29.4.4. Dell Parallel Port — Security settings

Table 28. Dell Parallel Port: Security settings field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	None
`0x04`	`4`	External Interface Locked Out
`0x05`	`5`	External Interface Enabled
`0x06`	`6`	Boot-bypass

29.5. Dell Serial Port (Type 210)

Source: Dell `libsmbios`, `doc/smbios23.xml`.

Table 29. Dell Serial Port structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.2+	Type	`BYTE`	`210`	Dell Serial Port indicator
`0x01`	2.2+	Length	`BYTE`	`0x0C`	Length of the structure
`0x02`	2.2+	Handle	`WORD`	Varies	
`0x04`	2.2+	Base I/O Address	`WORD`	`ADDRESS`	
`0x06`	2.2+	IRQ	`BYTE`	`NUMBER`	
`0x07`	2.2+	Connector Type	`BYTE`	`ENUM`	See Connector types for details
`0x08`	2.2+	Capabilities	`BYTE`	`ENUM`	See Capabilities for details
`0x09`	2.2+	Maximum BIOS Speed	`WORD`	`SPEED`	Highest speed the BIOS supports on the port, in hundreds of bits per second, e.g. 1152 for 115200 bit/s. The source gives no unit, which is inferred from the data of Dell systems.
`0x0B`	2.2+	Security Settings	`BYTE`	`ENUM`	See Security settings for details

29.5.1. Dell Serial Port — Connector types

Table 30. Dell Serial Port: Connector type field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	DB-9 Male
`0x04`	`4`	DB-9 Female
`0x05`	`5`	DB-25 Male
`0x06`	`6`	DB-25 Female
`0x07`	`7`	RJ-11
`0x08`	`8`	RJ-45
`0x09`	`9`	Proprietary
`0x0A`	`10`	Circular DIN-8 Male
`0x0B`	`11`	Circular DIN-8 Female
`0xA0`	`160`	Mini-Centronics Type-14

Hex Value	Dec Value	Meaning
`0xA1`	`161`	Mini-Centronics Type-26

29.5.2. Dell Serial Port — Capabilities

Table 31. Dell Serial Port: Capabilities field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	XT/AT compatible
`0x04`	`4`	16450 compatible
`0x05`	`5`	16550 compatible
`0x06`	`6`	16550A compatible
`0xA0`	`160`	8251 compatible
`0xA1`	`161`	8251 FIFO compatible

29.5.3. Dell Serial Port — Security settings

Table 32. Dell Serial Port: Security settings field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	None
`0x04`	`4`	External Interface Locked Out
`0x05`	`5`	External Interface Enabled
`0x06`	`6`	Boot-bypass

29.6. Dell Infrared Port (Type 211)

Source: Dell `libsmbios`, `doc/smbios23.xml`.

The structure is told by its length of 13 bytes. Structures of other lengths are found at the same type, e.g. the one of 17 bytes of Studio 1555, which holds text in its formatted area, and are not taken for infrared ports.

Table 33. Dell Infrared Port structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.2+	Type	`BYTE`	`211`	Dell Infrared Port indicator

Offset	Version	Name	Length	Value	Description
`0x01`	2.2+	Length	`BYTE`	`0x0D`	Length of the structure
`0x02`	2.2+	Handle	`WORD`	Varies	
`0x04`	2.2+	Location	`BYTE`	`STRING`	
`0x05`	2.2+	Enable State	`BYTE`	`ENUM`	See Enable states for details
`0x06`	2.2+	Speed Limit Enable	`BYTE`	`ENUM`	See Enable states for details
`0x07`	2.2+	Speed Limit	`WORD`	`SPEED`	Speed the port is limited to, in hundreds of bits per second. The source gives no unit, and no data is known to confirm it: it is taken to be the one of the maximum BIOS speed of the serial port.
`0x09`	2.2+	Physical Port Name	`BYTE`	`STRING`	
`0x0A`	2.2+	Virtual COM Port Name	`BYTE`	`STRING`	
`0x0B`	2.2+	Virtual LPT Port Name	`BYTE`	`STRING`	
`0x0C`	2.2+	Protocol	`BYTE`	`ENUM`	See Protocols for details

29.6.1. Dell Infrared Port — Enable states

Table 34. Dell Infrared Port: Enable state field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	Enabled
`0x04`	`4`	Disabled

29.6.2. Dell Infrared Port — Protocols

Table 35. Dell Infrared Port: Protocol field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Other
`0x02`	`2`	Unknown
`0x03`	`3`	SIR
`0x04`	`4`	FIR
`0x05`	`5`	MIR

29.7. Dell Indexed IO (Type 212)

Source: Dell `libsmbios`, `src/include/smbios_c/obj/token.h`.

Describes tokens, which are stored in CMOS and accessed through the indexed I/O ports, and the checksum of the CMOS area.

Table 36. Dell Indexed IO structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`212`	Dell Indexed IO indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Index Port	`WORD`	`ADDRESS`	I/O port used to select the CMOS location
`0x06`	2.0+	Data Port	`WORD`	`ADDRESS`	I/O port used to access the selected CMOS location
`0x08`	2.0+	Checksum Type	`BYTE`	`ENUM`	See Checksum types for details
`0x09`	2.0+	Checked Range Start Index	`BYTE`	`NUMBER`	First CMOS index of the checked range
`0x0A`	2.0+	Checked Range End Index	`BYTE`	`NUMBER`	Last CMOS index of the checked range
`0x0B`	2.0+	Checksum Value Index	`BYTE`	`NUMBER`	CMOS index of the checksum value. If the checked range and the checksum index are all zeroes, there is no checksum.
`0x0C`	2.0+	Tokens	Varies	Varies	Tokens, see Tokens. The table is terminated by a token with ID `0xFFFF`, tokens with ID `0x0000` are unused.

29.7.1. Dell Indexed IO — Tokens

Table 37. Dell Indexed IO: Token

Offset	Name	Length	Value	Description
`0x00`	Token ID	`WORD`	`NUMBER`	
`0x02`	Location	`BYTE`	`NUMBER`	CMOS index of the token value
`0x03`	AND Mask	`BYTE`	`NUMBER`	Mask applied to the value at the location. Zero for string tokens.
`0x04`	OR Value / String Length	`BYTE`	`NUMBER`	For boolean tokens, value set at the location when the token is activated. For string tokens, length of the string.

29.7.2. Dell — Checksum types

Table 38. Dell: Checksum type field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Running sum in a word
`0x01`	`1`	Running sum in a byte
`0x02`	`2`	CRC in a word
`0x03`	`3`	Running sum in a word, then negated (`~sum + 1`)

29.8. Dell Protected Area Type 1 (Type 213)

Source: Dell `libsmbios`, `src/include/smbios_c/obj/token.h` for the layout, and `src/libsmbios_c/token/token_impl.h` for the type, which is `0xD5`.

Describes a protected value, such as a password, which is stored in CMOS and accessed through the indexed I/O ports, and its checksum.

Table 39. Dell Protected Area Type 1 structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`213`	Dell Protected Area Type 1 indicator
`0x01`	2.0+	Length	`BYTE`	`0x11`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Token ID	`WORD`	`NUMBER`	Token of the protected value
`0x06`	2.0+	Value Length	`BYTE`	`NUMBER`	
`0x07`	2.0+	Value Format	`BYTE`	`ENUM`	See Value formats for details
`0x08`	2.0+	Validation Key	`WORD`	`NUMBER`	
`0x0A`	2.0+	Index Port	`WORD`	`ADDRESS`	I/O port used to select the CMOS location
`0x0C`	2.0+	Data Port	`WORD`	`ADDRESS`	I/O port used to access the selected CMOS location
`0x0E`	2.0+	Checksum Type	`BYTE`	`ENUM`	See Checksum types for details
`0x0F`	2.0+	Value Start Index	`BYTE`	`NUMBER`	CMOS index of the protected value
`0x10`	2.0+	Checksum Index	`BYTE`	`NUMBER`	CMOS index of the checksum of the value

Checksum types are the same as in Checksum types. Value formats are described in Value formats.

29.9. Dell Protected Area Type 2 (Type 214)

Source: Dell `libsmbios`, `src/include/smbios_c/obj/token.h` for the layout, and `src/libsmbios_c/token/token_impl.h` for the type, which is `0xD6`.

Describes a protected value, such as a password, which is stored in CMOS and accessed through the indexed I/O ports, and its checksum.

In addition to the checksum of the value, the checksum of a CMOS range is described.

Table 40. Dell Protected Area Type 2 structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`214`	Dell Protected Area Type 2 indicator
`0x01`	2.0+	Length	`BYTE`	`0x15`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Token ID	`WORD`	`NUMBER`	Token of the protected value
`0x06`	2.0+	Value Length	`BYTE`	`NUMBER`	
`0x07`	2.0+	Value Format	`BYTE`	`ENUM`	See Value formats for details
`0x08`	2.0+	Validation Key	`WORD`	`NUMBER`	
`0x0A`	2.0+	Index Port	`WORD`	`ADDRESS`	I/O port used to select the CMOS location
`0x0C`	2.0+	Data Port	`WORD`	`ADDRESS`	I/O port used to access the selected CMOS location
`0x0E`	2.0+	Checksum Type	`BYTE`	`ENUM`	See Checksum types for details
`0x0F`	2.0+	Value Start Index	`BYTE`	`NUMBER`	CMOS index of the protected value
`0x10`	2.0+	Checksum Index	`BYTE`	`NUMBER`	CMOS index of the checksum of the value
`0x11`	2.0+	Range Checksum Type	`BYTE`	`ENUM`	See Checksum types for details
`0x12`	2.0+	Range Check Start Index	`BYTE`	`NUMBER`	First CMOS index of the checked range
`0x13`	2.0+	Range Check End Index	`BYTE`	`NUMBER`	Last CMOS index of the checked range
`0x14`	2.0+	Range Checksum Index	`BYTE`	`NUMBER`	CMOS index of the checksum of the range

29.9.1. Dell — Protected value formats

Table 41. Dell: Value format field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Alphanumeric, scan codes
`0x01`	`1`	Alphanumeric, ASCII
`0x02`	`2`	Alphanumeric, scan codes (NS)
`0x03`	`3`	Alphanumeric, ASCII (NS)

29.10. Dell Video BIOS Information (Type 216)

Source: reverse engineered from the dumps of the data corpus.

Tells the vendor and the version of the video BIOS, e.g. `"Intel Corp."` and `"2089"` in `dell/g15-5510.bin`, or `ATI` and `RADEON 7000 V6.11` in `dell/poweredge-1800.bin`.

Table 42. Dell Video BIOS Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`216`	Dell Video BIOS Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Vendor	`BYTE`	`STRING`	Vendor of the video BIOS, which may be quoted
`0x05`	2.0+	Version	`BYTE`	`STRING`	Version of the video BIOS, which may be quoted
`0x06`	2.0+	Unknown 1	`BYTE`	Varies	`1`, or `0` on the PowerEdge servers of the 13th and 14th generations, in the known data
`0x07`	2.0+	Unknown 2	`WORD`	Varies	`0`, `0x0088`, `0x0110` or `0x03F0` in the known data

29.11. Dell Calling Interface (Type 218)

Source: Dell `libsmbios`, `src/libsmbios_c/token/token_impl.h`.

Describes the interface for issuing commands to the firmware through an I/O port, and tokens, which are stored in non-volatile storage and accessed with these commands.

Table 43. Dell Calling Interface structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`218`	Dell Calling Interface indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Command I/O Address	`WORD`	`ADDRESS`	I/O port, which is written to issue a command
`0x06`	2.0+	Command I/O Code	`BYTE`	`NUMBER`	Value, which is written to the command I/O port
`0x07`	2.0+	Supported Commands	`DWORD`	`BITFIELD`	

Offset	Version	Name	Length	Value	Description
0x0B	2.0+	Tokens	Varies	Varies	Tokens, see Tokens. The table is terminated by a token with ID 0xFFFF, tokens with ID 0x0000 are unused.

29.11.1. Dell Calling Interface — Tokens

Table 44. Dell Calling Interface: Token

Offset	Name	Length	Value	Description
0x00	Token ID	WORD	NUMBER	
0x02	Location	WORD	NUMBER	Location of the token in non-volatile storage, zero for string tokens
0x04	Value / String Length	WORD	NUMBER	Token value, or length of the string for string tokens

29.12. Dell Device Bay (Type 219)

Source: reverse engineered from the dumps of the data corpus.

Describes a bay of a laptop or of its docking station, which takes one of several devices, e.g. `System Device Bay`, which takes `Floppy, Battery, CD-ROM, CD-RW, Hard Disk, DVD` and holds `Battery` in `dell/latitude-d500.bin`, or `Dock DBay`, which holds `EMPTY` in `dell/latitude-d610.bin`.

Newer platforms give the structure 11 bytes, which refer to two more strings whose meaning is not established: Latitude E6230 has a bay and leaves them out, while Precision M3800 and G15 5515, which have none, carry a structure of blank strings or of `00h`. XPS 13 9350 gives type 219 to the Intel Management Engine interface information, which is told by its signature, and the structure is decoded for the other structures of the type. The numbers of the strings do not tell it, since the structures whose strings are the same are written back with a single one.

Table 45. Dell Device Bay structure

Offset	Version	Name	Length	Value	Description
0x00	2.0+	Type	BYTE	219	Dell Device Bay indicator
0x01	2.0+	Length	BYTE	Varies	Length of the structure
0x02	2.0+	Handle	WORD	Varies	
0x04	2.0+	Unknown 1	BYTE	Varies	3 on the systems which have a bay, 0 or 2 on the ones which have none
0x05	2.0+	Name	BYTE	STRING	Name of the bay
0x06	2.0+	Supported Devices	BYTE	STRING	Devices the bay takes, separated by commas
0x07	2.0+	Installed Device	BYTE	STRING	Device the bay holds, or EMPTY for none, padded with spaces

Offset	Version	Name	Length	Value	Description
`0x08`	2.0+	Unknown 2	`BYTE`	Varies	`0xFF` in most data, `0` on Precision M3800
`0x09`	2.0+	Unknown String 1	`BYTE`	`STRING`	Meaning not established, present in the structures of 11 bytes
`0x0A`	2.0+	Unknown String 2	`BYTE`	`STRING`	Meaning not established, present in the structures of 11 bytes

29.13. Dell Token References, Type 1 (Type 220)

Source: reverse engineered from the dumps of the data corpus.

Lists tokens, most of which the calling interface (type 218) or the indexed I/O access (type 212) of the same table defines, which the structures of a table group by the device they belong to, e.g. `0xF100` and `0xF102`, `0xF110` and `0xF112` in `dell/g15-5510.bin`, or `0xF420`, `0xF410`, `0xF430` and `0xF440` in `dell/precision-490.bin`. The tokens are SMI ones, which are named in no public source, and the meaning of the groups is not established.

Table 46. Dell Token References, Type 1 structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`220`	Dell Token References, Type 1 indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure, `0x14` or `0x16` in the known data
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Tokens	2n BYTEs	Varies	Tokens, a `WORD` each, zero for none, which run to the end of the structure: 8 or 9 of them in the known data

29.14. Dell Token References, Type 2 (Type 221)

Source: reverse engineered from the dumps of the data corpus.

Lists tokens, most of which the calling interface (type 218) or the indexed I/O access (type 212) of the same table defines, e.g. `0xF600` and `0xF601` in `dell/g15-5510.bin`, following two values whose meaning is not established. The structures are 19 bytes long and carry no strings, which tells them from the Intel firmware version information some systems, e.g. XPS 13 9350, keep at type 221.

Table 47. Dell Token References, Type 2 structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`221`	Dell Token References, Type 2 indicator
`0x01`	2.0+	Length	`BYTE`	`0x13`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Unknown 1	`BYTE`	Varies	`0`, `2` or `3` in the known data

Offset	Version	Name	Length	Value	Description
`0x05`	2.0+	Unknown 2	`WORD`	Varies	`0` or `1` in the known data
`0x07`	2.0+	Tokens	6 WORDs	Varies	Tokens, zero for none

29.15. Dell Memory Module Identifiers (Type 223)

Source: reverse engineered from the dumps of the data corpus.

Tells the manufacturer and the serial number of the memory modules, as their serial presence detect (SPD) data holds them, which the memory devices (type 17) of the older systems carry no fields for, e.g. in `dell/poweredge-400sc.bin` and `dell/optiplex-gx110.bin`. Each memory device is referred to by one record, and the records of the sockets which hold no module usually carry bytes of `0xFF`: PowerEdge 1300 repeats the identifiers of a module in the record of an empty socket.

The structures whose identifiers are of another length than 12 bytes are not known, and are left undecoded.

Table 48. Dell Memory Module Identifiers structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`223`	Dell Memory Module Identifiers indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Identifiers Length	`BYTE`	`12`	Length of the identifiers of a module, which follow the handle of its memory device
`0x05`	2.0+	Modules	Varies	Varies	Modules, see Modules, which run to the end of the structure

29.15.1. Dell Memory Module Identifiers — Modules

Table 49. Dell Memory Module Identifiers: Module

Offset	Name	Length	Value	Description
`0x00`	Handle	`WORD`	Varies	Handle of the memory device (type 17) of the socket
`0x02`	Manufacturer	8 `BYTE`	Varies	Identifier of the manufacturer, normally its JEDEC identifier, whose leading bytes of `0x7F` are continuation codes, e.g. `CE` for Samsung, `2C` for Micron, or `7F 98` for Kingston
`0x0A`	Serial Number	4 `BYTE`	Varies	Serial number of the module

29.16. Dell Device Names (Type 225)

Source: reverse engineered from the dumps of the data corpus.

Gives the devices of a kind the names the management controller (iDRAC) of a PowerEdge server knows them by, its fully qualified device descriptors (FQDD), e.g. `CPU.Socket.1` and `CPU.Socket.2` to the processors (type 4) of handles `0x0400` and `0x0401`, or `DIMM.Socket.A1` to the memory device (type 17) of handle `0x1100` in `dell/poweredge-r640.bin`. The structures are found on PowerEdge servers of the 13th and 14th generations, one for the processors and one for the memory devices.

Table 50. Dell Device Names structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`225`	Dell Device Names indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Unknown	`BYTE`	Varies	`1` in all known data
`0x05`	2.0+	Devices	Varies	Varies	Devices, see Devices, which run to the end of the structure

29.16.1. Dell Device Names — Devices

Table 51. Dell Device Names: Device

Offset	Name	Length	Value	Description
`0x00`	FQDD	`BYTE`	`STRING`	Fully qualified device descriptor of the device
`0x01`	Handle	`WORD`	Varies	Handle of the structure describing the device, a processor (type 4) or a memory device (type 17) in all known data
`0x03`	Unknown	`BYTE`	Varies	`0` in all known data

29.17. Dell System ID Record (Type 255)

Source: reverse engineered from the dumps of the data corpus.

Holds the system ID of the platform in hexadecimal digits, e.g. `0A64` in `dell/g15-5510.bin`, which the revisions and IDs (type 208) and the OEM strings (type 11, key `1`) hold too, along with an identifier, whose meaning is not established, beginning with `_SID` on all systems of the data corpus but OptiPlex 9030 AIO, which holds `dellDelldellDELL`.

Most structures are 8 bytes long, and hold `0` in their last two bytes, while the ones of G15 5510 and XPS 13 9300 are 6 bytes long. Most of the longer ones carry two more strings no field refers to: the identifier again, and `DELL`.

Table 52. Dell System ID Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`255`	Dell System ID Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Identifier	`BYTE`	`STRING`	Identifier, beginning with `_SID` in most known data
`0x05`	2.0+	System ID	`BYTE`	`STRING`	System ID, in hexadecimal digits
`0x06`	2.0+	Unknown	`WORD`	Varies	`0` in all known data

Chapter 30. HP/HPE-specific structures

HP and HPE servers add structures of their own, some of which have changed their layouts or have been deprecated over the generations of the servers: up to G7, the generations are named `G1` to `G7`, and `Gen8` onwards. The generation is told from the product name of the system information (type 1). Servers whose product name tells no generation are taken for Gen10 Plus when their firmware is of HPE, and for G6 when it is of HP, the way dmidecode takes them. Type 219 and type 221 are taken on these servers by structures of their own, and the Intel reference code structures of these types are not decoded there.

30.1. HP/HPE Other ROM Information (Type 193)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Describes the redundant system ROM and the OEM ROM image of the server. dmidecode decodes the structure from Gen9 onwards, while the older servers of the data corpus carry the same layout.

Table 53. HP/HPE Other ROM Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`193`	HP/HPE Other ROM Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	ROM Flags	`BYTE`	`BITFIELD`	Bit 0 tells that a redundant ROM is installed
`0x05`	2.0+	Redundant ROM Version	`BYTE`	`STRING`	Version of the redundant ROM, which is its release date. Reserved from Gen12 onwards
`0x06`	2.0+	Boot Block Version	`BYTE`	`STRING`	Version whose meaning is not documented, reserved from Gen9 onwards. Its dates are earlier than the ones of the ROM, which suggests the boot block
`0x07`	2.0+	OEM ROM File Name	`BYTE`	`STRING`	File name of the OEM ROM image. Firmware which has no image leaves the string out or fills it with blanks
`0x08`	2.0+	OEM ROM Build Date	`BYTE`	`STRING`	Build date of the OEM ROM image, meaningful when the file name is
`0x09`	2.0+	Unknown	`BYTE`	`STRING`	String whose meaning is not established, e.g. `2.9` on a G6 server. dmidecode does not decode it

The version of the redundant ROM is shown when the redundant ROM is installed, up to Gen11. The file name and the build date of the OEM ROM image are shown when the file name is neither empty nor begins with two blanks, which is the check dmidecode makes.

30.2. HP/HPE Super I/O Enable/Disable Indicator (Type 194)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells which ports of the Super I/O controller are enabled.

Table 54. HP/HPE Super I/O Enable/Disable Indicator structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`194`	HP/HPE Super I/O Enable/Disable Indicator indicator
`0x01`	2.0+	Length	`BYTE`	`0x05`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Device Status	`BYTE`	`BITFIELD`	Bit 0: serial port A, bit 1: serial port B, bit 2: parallel port, bit 3: floppy disk port, bit 4: virtual serial port

30.3. HP/HPE Server System ID (Type 195)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Gives the unique ID of the system, which replaces the EISA ID of the older systems where the EISA ID port is absent, and the platform ID iLO tells the platform and its flash deliverables by. The oldest servers carry the system ID only, in structures of 5 bytes.

Table 55. HP/HPE Server System ID structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`195`	HP/HPE Server System ID indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	System ID	`BYTE`	`STRING`	Server system ID, e.g. `$0E110761`
`0x05`	2.0+	Platform ID	2 BYTEs	Varies	Low and high bytes of the platform ID, from the XREG register of the CPLD
`0x07`	2.0+	GUID	16 BYTEs	`UUID`	Reserved from Gen11 onwards

30.4. HP/HPE Processor Specific Information (Type 197)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Supplements the processor information (type 4) of each processor socket or slot with the numbers the server tells its processors by. The structure is ignored for the empty sockets. It has grown with the generations of the servers, and its fields past the physical socket are absent from the older ones.

Table 56. HP/HPE Processor Specific Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`197`	HP/HPE Processor Specific Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Processor Handle	`WORD`	`HANDLE`	Handle of the processor information structure (type 4)
`0x06`	2.0+	APIC ID	`BYTE`	`NUMBER`	Local APIC ID of the processor
`0x07`	2.0+	Status	`BYTE`	`BITFIELD`	Bit 0: bootstrap processor, bit 1: x2APIC mode, bit 2: advanced thermal margining
`0x08`	2.0+	Physical Slot	`BYTE`	`NUMBER`	As the silk screen names it, `0xFF` if the processor is not in a slot
`0x09`	2.0+	Physical Socket	`BYTE`	`NUMBER`	As the silk screen names it, `0xFF` if the processor is not in a socket
`0x0A`	2.0+	Maximum Power	`WORD`	`NUMBER`	Rated maximum power in watts, zero if unknown
`0x0C`	2.0+	x2APIC ID	`DWORD`	`NUMBER`	x2APIC ID of the processor, valid in the x2APIC mode only, which bit 1 of the status tells
`0x10`	2.0+	Processor UUID	`QWORD`	`NUMBER`	Unique identifier of the processor
`0x18`	2.0+	Interconnect Speed	`WORD`	`NUMBER`	Speed of the interconnect in MT/s
`0x1A`	2.0+	QDF/S-Spec	6 BYTEs	Varies	QDF or S-Spec number of Intel processors, in ASCII padded with spaces
`0x20`	2.0+	Reserved	`DWORD`	Varies	Reserved from Gen11 onwards

30.5. HP/HPE CPU Microcode Patch Support Information (Type 199)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Lists the CPU microcode patches the firmware carries. Patches run to the end of the structure, which carries no number of them of its own. dmidecode decodes the structure from Gen9 onwards, while the older servers of the data corpus carry the same layout.

Table 57. HP/HPE CPU Microcode Patch Support Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`199`	HP/HPE CPU Microcode Patch Support Information indicator

Offset	Version	Name	Length	Value	Description
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Patches	12 BYTEs each	Varies	Patches, see below

Table 58. Patch

Offset	Name	Length	Description
`0x00`	Patch ID	`DWORD`	Microcode revision
`0x04`	Date	`DWORD`	Release date in binary-coded decimal, `0xMMDDYYYY`, as the microcode header gives it
`0x08`	CPUID	`DWORD`	Processor signature (CPUID leaf 1 EAX) the patch applies to. AMD platforms leave the base family out, see below

AMD platforms leave out the byte of the signature which holds the base family, bits 8 to 15, and move the bytes above it down by one. The base family is 15 on all AMD processors from the Athlon 64 onwards, whose extended family tells them apart, so the signature is put back by taking 15 for the base family, the way dmidecode does:

- bits 0 to 7 are taken from byte 0;
- bits 8 to 15 are set to `0x0F`;
- bits 16 to 23 are taken from byte 1;
- bits 24 to 27 are taken from the low nibble of byte 2.

The signature put back is valid for the families of 15 onwards only.

30.6. HP/HPE DIMM Location Record (Type 202)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells where the DIMM socket of a memory device (type 17) is: the memory board, the processor and the channel it belongs to. There is a record for each socket, including the ones of the memory boards which are not installed. The structure has grown with the generations of the servers, and the older ones end after the DIMM number. dmidecode decodes the structure from Gen9 onwards, while the older servers of the data corpus carry the same layout.

Table 59. HP/HPE DIMM Location Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`202`	HP/HPE DIMM Location Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	

Offset	Version	Name	Length	Value	Description
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Handle of the memory device structure (type 17)
`0x06`	2.0+	Board Number	`BYTE`	`NUMBER`	1-based number of the memory board, `0xFF` for the system board
`0x07`	2.0+	DIMM Number	`BYTE`	`NUMBER`	1-based number of the DIMM
`0x08`	2.0+	Processor Number	`BYTE`	`NUMBER`	1-based number of the processor, `0xFF` if not shown
`0x09`	2.0+	Logical DIMM Number	`BYTE`	`NUMBER`	1-based logical number of the DIMM, as ACPI numbers it
`0x0A`	2.0+	UEFI Device Path	`BYTE`	`STRING`	
`0x0B`	2.0+	UEFI Device Name	`BYTE`	`STRING`	UEFI device structured name
`0x0C`	2.0+	Device Name	`BYTE`	`STRING`	
`0x0D`	2.0+	Memory Controller Number	`BYTE`	`NUMBER`	1-based
`0x0E`	2.0+	Memory Channel Number	`BYTE`	`NUMBER`	1-based, as the silk screen names it
`0x0F`	2.0+	IE DIMM Number	`BYTE`	`NUMBER`	0-based number of the DIMM, as the Innovation Engine reports it, `0xFF` if not supported. Reserved from Gen12 onwards
`0x10`	2.0+	IE PLDM ID	`BYTE`	`NUMBER`	Sensor ID of the Innovation Engine, `0xFF` if not supported. Reserved from Gen12 onwards
`0x11`	2.0+	Vendor ID	`WORD`	`NUMBER`	Manufacturer ID code of the module, as the SPD gives it
`0x13`	2.0+	Device ID	`WORD`	`NUMBER`	Product ID code of the module, NVDIMMs only
`0x15`	2.0+	Controller Vendor ID	`WORD`	`NUMBER`	Manufacturer ID code of the controller, NVDIMMs only, as the SPD gives it: the number of continuation codes in the low byte, whose bit 7 is a parity bit, and the JEP106 code in the high byte
`0x17`	2.0+	Controller Device ID	`WORD`	`NUMBER`	Product ID code of the controller, NVDIMMs only
`0x19`	2.0+	Interleave	`BYTE`	`NUMBER`	1-based interleave set within the processor
`0x1A`	2.0+	Part Number	`BYTE`	`STRING`	HPE part number, from the OEM area of the SPD

Offset	Version	Name	Length	Value	Description
0x1B	2.0+	DIMM Index	BYTE	NUMBER	0-based index of the DIMM within its channel, present in the structures of 28 bytes or more

30.7. HP/HPE Device Correlation Record (Type 203)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Correlates a device with the system slot (type 9) or the onboard device (type 41) it is found at, and with the UEFI boot entries naming it. The structure is decoded from Gen9 onwards. It has grown over the generations, and the shortest structures end after the device name, at `0x1F`. The PCI location, from `0x24` on, is present in the structures of 40 bytes or more.

Table 60. HP/HPE Device Correlation Record structure

Offset	Version	Name	Length	Value	Description
0x00	2.0+	Type	BYTE	203	HP/HPE Device Correlation Record indicator
0x01	2.0+	Length	BYTE	Varies	Length of the structure
0x02	2.0+	Handle	WORD	Varies	
0x04	2.0+	Device Handle	WORD	HANDLE	Handle of the system slot (type 9) or onboard device (type 41) structure, 0xFFFE if not applicable
0x06	2.0+	SMBus Handle	WORD	HANDLE	Handle of the SMBus segment record (type 228), 0xFFFE if not applicable
0x08	2.0+	PCI Vendor ID	WORD	NUMBER	0xFFFF if the device is not present
0x0A	2.0+	PCI Device ID	WORD	NUMBER	0xFFFF if the device is not present
0x0C	2.0+	PCI Subsystem Vendor ID	WORD	NUMBER	0xFFFF if the device is not present
0x0E	2.0+	PCI Subsystem ID	WORD	NUMBER	0xFFFF if the device is not present
0x10	2.0+	PCI Class Code	BYTE	NUMBER	0xFF if the device is not present
0x11	2.0+	PCI Subclass Code	BYTE	NUMBER	0xFF if the device is not present
0x12	2.0+	Parent Handle	WORD	HANDLE	Handle of the parent device, 0xFFFE if not applicable. The structure it refers to is not documented, and is presumably the device correlation record of the parent device
0x14	2.0+	Flags	WORD	BITFIELD	Bit 0: peer bifurcated device, bit 1: upstream device
0x16	2.0+	Device Type	BYTE	ENUM	UEFI only, see below

Offset	Version	Name	Length	Value	Description
`0x17`	2.0+	Device Location	`BYTE`	`ENUM`	See below
`0x18`	2.0+	Device Instance	`BYTE`	`NUMBER`	
`0x19`	2.0+	Device Sub-Instance	`BYTE`	`NUMBER`	Port of a NIC, or bay of an NVMe drive
`0x1A`	2.0+	Bay	`BYTE`	`NUMBER`	`0` if unknown, `0xFF` if not shown
`0x1B`	2.0+	Enclosure	`BYTE`	`NUMBER`	`0` if unknown, `0xFF` if not shown
`0x1C`	2.0+	UEFI Device Path	`BYTE`	`STRING`	
`0x1D`	2.0+	UEFI Device Structured Name	`BYTE`	`STRING`	
`0x1E`	2.0+	UEFI Device Name	`BYTE`	`STRING`	
`0x1F`	2.0+	UEFI Location	`BYTE`	`STRING`	
`0x20`	2.0+	Physical Slot Handle	`WORD`	`HANDLE`	Handle of the system slot structure (type 9) of the real physical slot, valid if bit 0 of the flags is set
`0x22`	2.0+	Part Number	`BYTE`	`STRING`	Part number of the PCI device
`0x23`	2.0+	Serial Number	`BYTE`	`STRING`	Serial number of the PCI device
`0x24`	2.0+	Segment Group	`WORD`	`NUMBER`	PCI segment group, zero for a single group topology
`0x26`	2.0+	Bus	`BYTE`	`NUMBER`	PCI bus
`0x27`	2.0+	Device and Function	`BYTE`	`NUMBER`	PCI device and function, `(device << 3) | function`

Table 61. Device type

Value	Meaning
`0x00`	Unknown
`0x03`	Flexible LOM
`0x04`	Embedded LOM
`0x05`	NIC in a slot
`0x06`	Storage controller
`0x07`	Smart Array storage controller
`0x08`	USB hard disk
`0x09`	Other PCI device
`0x0A`	RAM disk
`0x0B`	Firmware volume

Value	Meaning
`0x0C`	UEFI shell
`0x0D`	Generic UEFI USB boot entry
`0x0E`	Dynamic Smart Array controller
`0x0F`	File
`0x10`	NVMe hard drive
`0x11`	NVDIMM
`0x12`	Embedded GPU

Table 62. Device location

Value	Meaning
`0x00`	Unknown
`0x01`	Embedded
`0x02`	iLO virtual media
`0x03`	Front USB port
`0x04`	Rear USB port
`0x05`	Internal USB
`0x06`	Internal SD card
`0x07`	Internal virtual USB (embedded NAND)
`0x08`	Embedded SATA port
`0x09`	Embedded Smart Array
`0x0A`	PCI slot
`0x0B`	RAM memory
`0x0C`	USB
`0x0D`	Dynamic Smart Array controller
`0x0E`	URL
`0x0F`	NVMe drive bay
`0x10`	NVDIMM processor
`0x11`	NVDIMM board
`0x12`	NVMe riser
`0x13`	NVDIMM namespace
`0x14`	VROC SATA
`0x15`	VROC NVMe

30.8. HP/HPE System/Rack Locator (Type 204)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells where a blade server is: the rack, the enclosure and the bay.

Table 63. HP/HPE System/Rack Locator structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`204`	HP/HPE System/Rack Locator indicator
`0x01`	2.0+	Length	`BYTE`	`0x0B`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Rack Name	`BYTE`	`STRING`	
`0x05`	2.0+	Enclosure Name	`BYTE`	`STRING`	
`0x06`	2.0+	Enclosure Model	`BYTE`	`STRING`	
`0x07`	2.0+	Server Bay	`BYTE`	`STRING`	
`0x08`	2.0+	Enclosure Bays	`BYTE`	`NUMBER`	Number of the bays of the enclosure
`0x09`	2.0+	Bays Filled	`BYTE`	`NUMBER`	Number of the filled bays of the enclosure
`0x0A`	2.0+	Enclosure Serial Number	`BYTE`	`STRING`	

30.9. HP/HPE BIOS NIC PCI and MAC Information (Type 209, 221)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Lists the network ports the firmware boots from: over PXE with type 209, and over iSCSI with type 221. Ports run to the end of the structure, which carries no number of them of its own. A port is disabled if its bus and device are zero, and not installed if they have all bits set.

Type 221 is deprecated from Gen8 onwards. What the newer servers give it to, if anything, is not established, and OpenDMI decodes no structure of that type on them.

Table 64. HP/HPE BIOS NIC PCI and MAC Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`209, 221`	HP/HPE BIOS NIC PCI and MAC Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Ports	8 BYTEs each	Varies	Ports, see below

Table 65. Port

Offset	Name	Length	Description
`0x00`	Device and Function	`BYTE`	PCI device and function, `(device << 3) | function`
`0x01`	Bus	`BYTE`	PCI bus
`0x02`	MAC Address	6 BYTEs	

30.10. HP/HPE Processor TControl Information (Type 211)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Gives the Tcontrol value of a processor, which Intel programs into each processor after the thermal properties of its transistors. The servers begin to spin up the fans at this temperature. The processor information structure (type 4) tells whether the processor is installed.

Table 66. HP/HPE Processor TControl Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`211`	HP/HPE Processor TControl Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x07`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Processor Handle	`WORD`	`HANDLE`	Handle of the processor information structure (type 4)
`0x06`	2.0+	Tcontrol	`BYTE`	`NUMBER`	Tcontrol value, zero if not available

30.11. HP/HPE 64-bit CRU Information (Type 212)

Source: dmidecode, `dmioem.c`, the `hpwdt` watchdog timer driver of Linux, and the dumps of the data corpus.

Tells where the 64-bit entry point of the Compaq ROM Utility (CRU) services of the firmware is, which the watchdog timer driver calls. The structure is known up to Gen8. Servers of AMD carry a record of another signature too, `$SHF`, whose meaning is not known: dmidecode and the watchdog timer driver read the rest of the fields of the records of the `$CRU` signature only.

Table 67. HP/HPE 64-bit CRU Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`212`	HP/HPE 64-bit CRU Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x18`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	

Offset	Version	Name	Length	Value	Description
`0x04`	2.0+	Signature	4 BYTEs	`$CRU`	
`0x08`	2.0+	Physical Address	`QWORD`	Varies	Physical address of the service area
`0x10`	2.0+	Length	`DWORD`	`NUMBER`	Length of the service area
`0x14`	2.0+	Entry Point Offset	`DWORD`	`NUMBER`	Offset of the entry point within the service area

30.12. HP/HPE Version Indicator Record (Type 216)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Tells the version of a firmware component of the server, e.g. of a CPLD or of a power management controller, which type 193 does not scale to. There is a record for each component. The structure is decoded from Gen8 onwards.

The version data is laid out as its format tells. Format 5 is laid out differently on Gen9 servers, where it is the one of format 4, and on Gen10 and Gen10 Plus servers, where it takes the low nibbles of bytes 1, 3, 5 and 6. Words are held low byte first.

Table 68. HP/HPE Version Indicator Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`216`	HP/HPE Version Indicator Record indicator
`0x01`	2.0+	Length	`BYTE`	`0x17`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Firmware Type	`WORD`	`ENUM`	See below
`0x06`	2.0+	Firmware Name	`BYTE`	`STRING`	
`0x07`	2.0+	Firmware Version	`BYTE`	`STRING`	
`0x08`	2.0+	Data Format	`BYTE`	`ENUM`	Format of the version data, see below
`0x09`	2.0+	Version Data	12 BYTEs	Varies	
`0x15`	2.0+	Unique ID	`WORD`	`NUMBER`	Unique ID of the firmware flash, zero if none

Table 69. Data format

Value	Version
`0`	No version data
`1`	`0x` byte 1 bits 0-6, followed by ` B.0x` byte 0 bits 0-6 if bit 7 of byte 0 is set
`2`	Byte 0 bits 4-7 `.` byte 0 bits 0-3

Value	Version
`3`	Reserved
`4`	Byte 0 bits 4-7 `.` byte 0 bits 0-3 `.` byte 1 bits 0-6
`5`	Depends on the generation, see above
`6`	Byte 1 `.` byte 0
`7`	`v`byte 0`.`byte 1 `(`byte 2`/`byte 3`/`word 4`)`
`8`	Word 4 `.` word 0
`9`	Byte 0 `.` byte 1 `.` word 2
`10`	Byte 0 `.` byte 1 `.` byte 2 `Build` byte 3
`11`	Word 2 `.` word 0, then double word 4
`12`	Word 0 `.` word 2 `.` word 4 `.` word 6
`13`	Byte 0
`14`	Byte 0 `.` byte 1 `.` byte 2 `.` word 3
`15`	Word 0 `.` word 2 `.` word 4 `.` word 6 `(`byte 8`/`byte 9`/`word 10`)`
`16`	Characters of bytes 0 to 3 `.` byte 4 and byte 5
`17`	Double word 0 in hexadecimal
`18`	Byte 0 `.` byte 1 of two digits
`19`	`0x`byte 0 `.0x`byte 1 `.0x`byte 2
`20`	Byte 0 `.` byte 1 `.` byte 2 `.` byte 3

Table 70. Firmware type

Value	Meaning
`0x01`	System ROM
`0x02`	Redundant system ROM
`0x03`	System ROM bootblock
`0x04`	Power management controller firmware
`0x05`	Power management controller firmware bootloader
`0x06`	SL chassis firmware
`0x07`	SL chassis firmware bootloader
`0x08`	Hardware PAL/CPLD
`0x09`	SPS firmware (ME firmware)
`0x0A`	SL chassis PAL/CPLD
`0x0B`	Compatibility support module (CSM)

Value	Meaning
`0x0C`	APML
`0x0D`	Smart storage battery (Megacell) firmware
`0x0E`	Trusted module (TPM or TCM) firmware
`0x0F`	NVMe backplane firmware
`0x10`	Intelligent Provisioning
`0x11`	SPI descriptor version
`0x12`	Innovation Engine firmware (IE firmware)
`0x13`	UMB backplane firmware
`0x14`	Embedded diagnostics
`0x20`	EL chassis abstraction revision
`0x21`	EL chassis firmware revision
`0x22`	EL chassis PAL/CPLD
`0x23`	EL cartridge abstraction revision
`0x30`	Embedded video controller
`0x31`	PCIe riser programmable logic device
`0x32`	PCIe card CPLD
`0x33`	Intel NVMe VROC
`0x34`	Intel SATA VROC
`0x35`	Intel SPS firmware
`0x36`	Secondary system programmable logic device
`0x37`	CPU mezzanine board CPLD
`0x38`	Intel Arctic Sound-M accelerator firmware
`0x39`	Ampere system control processor (SCP)
`0x3A`	Intel CFR information
`0x3B`	OCP card
`0x3C`	DC-SCM CPLD
`0x3D`	Power distribution board CPLD
`0x3E`	PCIe switch board CPLD
`0x3F`	Sideband board CPLD
`0x40`	PCIe riser MCU firmware
`0x41`	PCIe switch board firmware

Value	Meaning
`0x42`	Power supply firmware
`0x43`	BMC firmware

30.13. HP/HPE ProLiant Information (Type 219)

Source: dmidecode, `dmioem.c`, the `hpwdt` watchdog timer driver of Linux, and the dumps of the data corpus.

Gives the feature flags of the server, which the watchdog timer driver reads. The servers give type 219 to this structure, which Intel reference code gives to the Management Engine interface information. The oldest servers carry the power features only, in structures of 8 bytes.

Table 71. HP/HPE ProLiant Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`219`	HP/HPE ProLiant Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Power Features	`DWORD`	`BITFIELD`	Whose meaning is not documented
`0x08`	2.0+	Omega Features	`DWORD`	`BITFIELD`	Whose meaning is not documented
`0x0C`	2.0+	Reserved	`DWORD`	Varies	
`0x10`	2.0+	Miscellaneous Features	`DWORD`	`BITFIELD`	Bit 0: integrated CRU (iCRU) is supported, bits 10 and 12: UEFI firmware, see below

The sources disagree on the bits of the UEFI firmware: dmidecode takes either of bits 10 and 12, mask `0x1400`, while the `hpwdt` driver takes either of bits 3 and 10, mask `0x408`. Bit 10 is common to both, and the meaning of bits 3 and 12 is not established. OpenDMI follows dmidecode.

30.14. HP/HPE Trusted Module Status (Type 224)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells whether a trusted module (TPM or TCM) is present and enabled, and what kind of module it is. The older servers hold the status only.

Table 72. HP/HPE Trusted Module Status structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`224`	HP/HPE Trusted Module Status indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure

Offset	Version	Name	Length	Value	Description
0x02	2.0+	Handle	WORD	Varies	
0x04	2.0+	Status	BYTE	BITFIELD	Bits 0-1: presence, bit 2: option ROMs are measured, bit 3: hidden
0x05	2.0+	Extended Status	BYTE	BITFIELD	Bits 0-1: disable reason, meaningful for a disabled module only; bits 2-5: error condition, meaningful for a module disabled for an error only
0x06	2.0+	Module Type	BYTE	BITFIELD	Bits 0-3: type, bit 4: standard algorithms supported, bit 5: Chinese algorithms supported
0x07	2.0+	Module Attributes	BYTE	BITFIELD	Bits 0-1: mounting, bits 2-3: FIPS certification
0x08	2.0+	Version Handle	WORD	HANDLE	Handle of the version indicator (type 216) of the firmware of the module
0x0A	2.0+	Chip Identifier	WORD	ENUM	The low byte tells the chip, the high byte is reserved

Table 73. Presence

Value	Meaning
0	Not present
1	Present and enabled
2	Present and disabled

Table 74. Disable reason

Value	Meaning
0	Not specified
1	Disabled by the user
2	Error condition

Table 75. Type

Value	Meaning
0	Not specified
1	TPM 1.2
2	TPM 2.0
3	Intel PTT firmware TPM

Table 76. Mounting

Value	Meaning
`0`	Not specified
`1`	Pluggable and optional
`2`	Pluggable but standard
`3`	Soldered down on the system board

Table 77. FIPS certification

Value	Meaning
`0`	Not specified
`1`	Not FIPS certified
`2`	FIPS certified

Table 78. Error condition

Value	Meaning
`0`	Not specified
`1`	Self-test failure

Table 79. Chip identifier

Value	Meaning
`0`	None
`1`	STMicro TPM of Gen10
`2`	Intel firmware TPM (PTT)
`3`	Nationz TPM
`4`	STMicro TPM of Gen10 Plus
`5`	STMicro TPM of Gen11
`6`	STMicro TPM of Gen12

30.15. HP/HPE Physical Attribute Information (Type 226)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Keeps the physical serial number and UUID of the server, whose standard fields hold virtual ones, e.g. on Synergy, where a workload moves from one server to another along with its identity.

Up to G7, the servers of the data corpus hold 16 ASCII characters instead of the UUID: the product number and the serial number, e.g. `484184GB894484YN`. dmidecode decodes the field as a UUID on every server. The data corpus holds the characters on a G6 server only, so the boundary between G7 and Gen8 is an inference, taken where the naming of the generations changes.

Table 80. HP/HPE Physical Attribute Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`226`	HP/HPE Physical Attribute Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x15`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	UUID	16 BYTEs	`UUID`	Physical UUID from Gen8 onwards, zero if not specified. Product number and serial number up to G7
`0x14`	2.0+	Serial Number	`BYTE`	`STRING`	Physical serial number

30.16. HP/HPE Reserved Memory Location (Type 229)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells where the memory regions the firmware reserves at boot are, which it reports to the operating system as reserved, e.g. the one the storage controller and iLO exchange the temperatures of the drives in. Regions run to the end of the structure, which carries no number of them of its own, and a table may hold several structures.

Table 81. HP/HPE Reserved Memory Location structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`229`	HP/HPE Reserved Memory Location indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Regions	16 BYTEs each	Varies	Regions, see below

Table 82. Region

Offset	Name	Length	Description
`0x00`	Signature	4 BYTEs	What the region is for, e.g. `$HDD`
`0x04`	Physical Address	`QWORD`	
`0x0C`	Size	`DWORD`	Bits 0-30: size, bit 31: the size is in kilobytes rather than in bytes

30.17. HP/HPE Power Supply Information (Type 230)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Supplements the system power supply information (type 39) with the actual manufacturer of the power supply and the way its FRU is accessed.

Table 83. HP/HPE Power Supply Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`230`	HP/HPE Power Supply Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x0B`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Power Supply Handle	`WORD`	`HANDLE`	Handle of the system power supply structure (type 39)
`0x06`	2.0+	Manufacturer	`BYTE`	`STRING`	Actual manufacturer
`0x07`	2.0+	Revision	`BYTE`	`STRING`	
`0x08`	2.0+	FRU Access Method	`BYTE`	`ENUM`	`0`: not available, `1`: IPMI I2C, `2`: iLO, `3`: chassis manager
`0x09`	2.0+	I2C Bus	`BYTE`	`NUMBER`	I2C bus, or I2C segment with iLO, `0xFF` if not available
`0x0A`	2.0+	I2C Address	`BYTE`	`NUMBER`	I2C address shifted left by one bit, `0xFF` if not available

30.18. HP/HPE DIMM Attributes Record (Type 232)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Tells what the memory device structure (type 17) of a DIMM socket does not: whether the module is an HPE one, and the voltages it runs at. There is a record for each socket. The structure is decoded from Gen9 onwards, and the shortest structures end after the configured voltage, at `0x0E`.

Table 84. HP/HPE DIMM Attributes Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`232`	HP/HPE DIMM Attributes Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Handle of the memory device structure (type 17)
`0x06`	2.0+	Attributes	`DWORD`	`BITFIELD`	Bits 0-1: HPE SmartMemory, `0` no, `1` yes, others unknown. Bit 2: bit 3 is defined, bit 3: load reduced DIMM. Bit 4: bit 5 is defined, bit 5: HPE standard memory

Offset	Version	Name	Length	Value	Description
`0x0A`	2.0+	Minimum Voltage	`WORD`	`NUMBER`	Minimum operating voltage in millivolts, zero if unknown
`0x0C`	2.0+	Configured Voltage	`WORD`	`NUMBER`	Configured operating voltage in millivolts, zero if unknown
`0x0E`	2.0+	Reserved	20 BYTEs	Varies	
`0x22`	2.0+	Map-Out Reason	`BYTE`	`BITFIELD`	Bit 0: configuration error, bit 1: training error
`0x23`	2.0+	Encryption Status	`BYTE`	`ENUM`	`0`: not encrypted, `1`: encrypted, `2`: unknown, `3`: not supported

30.19. HP/HPE NIC PCI and MAC Information (Type 233)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Describes a network port the firmware boots from over PXE, one per structure. The shortest structures end after the first 6 bytes of the MAC address, at `0x0E`. A port is disabled if its bus and device are zero, and not installed if they have all bits set.

Table 85. HP/HPE NIC PCI and MAC Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`233`	HP/HPE NIC PCI and MAC Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Segment Group	`WORD`	`NUMBER`	PCI segment group, zero for a single group topology
`0x06`	2.0+	Bus	`BYTE`	`NUMBER`	PCI bus
`0x07`	2.0+	Device and Function	`BYTE`	`NUMBER`	PCI device and function
`0x08`	2.0+	MAC Address	32 BYTEs	Varies	MAC address padded with zeroes
`0x28`	2.0+	Port	`BYTE`	`NUMBER`	Number of the port
`0x29`	2.0+	UEFI Device Path	`BYTE`	`STRING`	UEFI device path of the port

30.20. HP/HPE HDD Backplane FRU Information (Type 236)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Describes a drive backplane: where its FRU is, and the SAS expander and the drive bays it has. The

structure is decoded up to Gen10 Plus, from which the bay counts behind the ports and the name are deprecated. The shortest structures end after the NVRAM ID, at `0x09`.

Table 86. HP/HPE HDD Backplane FRU Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`236`	HP/HPE HDD Backplane FRU Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	FRU I2C Address	`BYTE`	`NUMBER`	I2C address of the FRU, shifted left by one bit
`0x05`	2.0+	Box Number	`WORD`	`NUMBER`	
`0x07`	2.0+	NVRAM ID	`WORD`	`NUMBER`	
`0x09`	2.0+	SAS Expander WWID	`QWORD`	`NUMBER`	
`0x11`	2.0+	Total Bays	`BYTE`	`NUMBER`	Number of the SAS drive bays
`0x12`	2.0+	A0 Bays	`BYTE`	`NUMBER`	Number of the SAS drive bays behind port `0xA0`
`0x13`	2.0+	A2 Bays	`BYTE`	`NUMBER`	Number of the SAS drive bays behind port `0xA2`
`0x14`	2.0+	Name	`BYTE`	`STRING`	Name of the backplane

30.21. HP/HPE DIMM Vendor Information (Type 237)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Tells the actual manufacturer of the module of a memory device (type 17), whose standard fields hold the ones of HPE. The structure is decoded from Gen9 onwards.

Table 87. HP/HPE DIMM Vendor Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`237`	HP/HPE DIMM Vendor Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Handle of the memory device structure (type 17)
`0x06`	2.0+	Manufacturer	`BYTE`	`STRING`	
`0x07`	2.0+	Part Number	`BYTE`	`STRING`	As the manufacturer gives it

Offset	Version	Name	Length	Value	Description
`0x08`	2.0+	Serial Number	`BYTE`	`STRING`	As the manufacturer gives it
`0x09`	2.0+	Manufacture Year	`BYTE`	`BCD`	Last two digits of the year, from 2000
`0x0A`	2.0+	Manufacture Week	`BYTE`	`BCD`	

30.22. HP/HPE USB Port Connector Correlation Record (Type 238)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Correlates a USB port connector (type 8) with its USB controller, its location and the UEFI device path of its endpoint. The structure is decoded from Gen9 onwards.

Table 88. HP/HPE USB Port Connector Correlation Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`238`	HP/HPE USB Port Connector Correlation Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Port Connector Handle	`WORD`	`HANDLE`	Handle of the port connector structure (type 8)
`0x06`	2.0+	Bus	`BYTE`	`NUMBER`	PCI bus of the USB controller
`0x07`	2.0+	Device and Function	`BYTE`	`NUMBER`	PCI device and function of the USB controller
`0x08`	2.0+	Location	`BYTE`	`ENUM`	`0`: internal, `1`: front, `2`: rear, `3`: embedded internal SD card, `4`: iLO USB, `5`: USB hub of the NAND controller, `7`: debug port, `9`: OCP USB
`0x09`	2.0+	Management Port	`WORD`	`ENUM`	`0`: not shared, `1`: shared with a physical switch, `2`: shared with automatic control
`0x0B`	2.0+	Port Instance	`BYTE`	`NUMBER`	Instance among the ports of the type
`0x0C`	2.0+	Parent Hub Instance	`BYTE`	`NUMBER`	Instance of the internal hub, `0xFE` if not applicable
`0x0D`	2.0+	Port Speed	`BYTE`	`ENUM`	`1`: USB 1.1 full speed, `2`: USB 2.0 high speed, `3`: USB 3.0 super speed
`0x0E`	2.0+	UEFI Device Path	`BYTE`	`STRING`	UEFI device path of the USB endpoint
`0x0F`	2.0+	Segment Group	`WORD`	`NUMBER`	PCI segment group of the USB controller

30.23. HP/HPE USB Device Correlation Record (Type 239)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Describes a USB device the firmware has found at boot on a USB port (type 238), and the UEFI boot entries naming it. Devices plugged in later are not described. The structure is decoded from Gen9 onwards, UEFI firmware only.

Table 89. HP/HPE USB Device Correlation Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`239`	HP/HPE USB Device Correlation Record indicator
`0x01`	2.0+	Length	`BYTE`	`0x17`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	USB Port Handle	`WORD`	`HANDLE`	Handle of the USB port connector correlation record (type 238)
`0x06`	2.0+	Vendor ID	`WORD`	`NUMBER`	USB vendor ID
`0x08`	2.0+	Flags	`WORD`	`BITFIELD`	Bit 0: an embedded SD card is present
`0x0A`	2.0+	Class	`BYTE`	`NUMBER`	USB class code
`0x0B`	2.0+	Subclass	`BYTE`	`NUMBER`	USB subclass code, see Mass storage subclasses for the mass storage class (`0x08`)
`0x0C`	2.0+	Protocol	`BYTE`	`NUMBER`	USB protocol code, see Mass storage protocols for the mass storage class (`0x08`) and Hub protocols for the hub class (`0x09`)
`0x0D`	2.0+	Product ID	`WORD`	`NUMBER`	USB product ID
`0x0F`	2.0+	Capacity	`DWORD`	`NUMBER`	Capacity in megabytes, zero if not applicable
`0x13`	2.0+	UEFI Device Path	`BYTE`	`STRING`	
`0x14`	2.0+	UEFI Device Structured Name	`BYTE`	`STRING`	
`0x15`	2.0+	Device Name	`BYTE`	`STRING`	
`0x16`	2.0+	Location	`BYTE`	`STRING`	

The codes are the ones the USB Implementers Forum defines for the classes, named the way dmidecode names them. The codes of the other classes are shown as they are.

Table 90. Mass storage subclass

Value	Meaning
`0x00`	SCSI command set not reported
`0x01`	Reduced block commands (RBC)
`0x02`	ATAPI
`0x03`	QIC-157, obsolete
`0x04`	USB floppy interface (UFI)
`0x05`	SFF-8070i, obsolete
`0x06`	SCSI transparent command set
`0x07`	LSD FS
`0x08`	IEEE 1667
`0xFF`	Vendor-specific

Subclasses `0x09` to `0xFE` are reserved.

Table 91. Mass storage protocol

Value	Meaning
`0x00`	Control/bulk/interrupt transport with command completion interrupt
`0x01`	Control/bulk/interrupt transport without command completion interrupt
`0x02`	Obsolete
`0x50`	Bulk-only transport
`0x62`	USB attached SCSI (UAS)
`0xFF`	Vendor-specific

Table 92. Hub protocol

Value	Meaning
`0x00`	Full speed
`0x01`	Hi-speed with a single transaction translator
`0x02`	Hi-speed with multiple transaction translators

30.24. HP/HPE Firmware Inventory Record (Type 240)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Tells the version of the firmware of a device which reports it through its UEFI driver, and the device correlation record (type 203) of the device.

Table 93. HP/HPE Firmware Inventory Record structure

Offset	Version	Name	Length	Value	Description
0x00	2.0+	Type	BYTE	240	HP/HPE Firmware Inventory Record indicator
0x01	2.0+	Length	BYTE	0x27	Length of the structure
0x02	2.0+	Handle	WORD	Varies	
0x04	2.0+	Device Correlation Handle	WORD	HANDLE	Handle of the device correlation record (type 203)
0x06	2.0+	Package Version	DWORD	NUMBER	Version of the package of all the firmware of the device
0x0A	2.0+	Version String	BYTE	STRING	
0x0B	2.0+	Image Size	QWORD	NUMBER	Size of the firmware image in bytes, zero if not available
0x13	2.0+	Attributes Defined	QWORD	BITFIELD	Bit 0: updatable, bit 1: reset required, bit 2: authentication required, bit 3: in use, bit 4: UEFI image
0x1B	2.0+	Attributes Set	QWORD	BITFIELD	Values of the defined attributes, bit for bit
0x23	2.0+	Lowest Supported Version	DWORD	NUMBER	Zero if not available

30.25. HP/HPE Hard Drive Inventory Record (Type 242)

Source: dmidecode, dmioem.c. No dumps of the data corpus hold the structure.

Describes an NVMe or SATA drive attached to the system directly, which the firmware has found at boot. Drives behind a storage controller, e.g. a Smart Array one, and the ones plugged in later are not described. The structure is decoded from Gen10 onwards, and the shortest structures end after the firmware revision, at 0x2A.

Table 94. HP/HPE Hard Drive Inventory Record structure

Offset	Version	Name	Length	Value	Description
0x00	2.0+	Type	BYTE	242	HP/HPE Hard Drive Inventory Record indicator
0x01	2.0+	Length	BYTE	Varies	Length of the structure
0x02	2.0+	Handle	WORD	Varies	
0x04	2.0+	Device Correlation Handle	WORD	HANDLE	Handle of the device correlation record (type 203)
0x06	2.0+	Drive Type	BYTE	ENUM	0: undetermined, 1: NVMe SSD, 2: SATA, 3: SAS, 4: SATA SSD, 5: NVMe managed by VROC/VMD

Offset	Version	Name	Length	Value	Description
`0x07`	2.0+	Unique ID	`QWORD`	`NUMBER`	WWID of SATA drives, IEEE extended unique identifier of NVMe ones
`0x0F`	2.0+	Capacity	`DWORD`	`NUMBER`	Capacity in megabytes
`0x13`	2.0+	Power-On Hours	16 BYTEs	`NUMBER`	
`0x23`	2.0+	Reserved	`BYTE`	Varies	
`0x24`	2.0+	Power	`BYTE`	`NUMBER`	Power in watts, zero if unknown
`0x25`	2.0+	Form Factor	`BYTE`	`ENUM`	See below
`0x26`	2.0+	Health	`BYTE`	`ENUM`	`0`: OK, `1`: warning, `2`: critical, `0xFF`: unknown
`0x27`	2.0+	Serial Number	`BYTE`	`STRING`	
`0x28`	2.0+	Model Number	`BYTE`	`STRING`	
`0x29`	2.0+	Firmware Revision	`BYTE`	`STRING`	
`0x2A`	2.0+	Location	`BYTE`	`STRING`	
`0x2B`	2.0+	Encryption Status	`BYTE`	`ENUM`	`0`: not encrypted, `1`: encrypted, `2`: unknown, `3`: not supported
`0x2C`	2.0+	Capacity	`QWORD`	`NUMBER`	Capacity in bytes
`0x34`	2.0+	Block Size	`DWORD`	`NUMBER`	Size of a logical block in bytes
`0x38`	2.0+	Rotational Speed	`WORD`	`NUMBER`	Nominal speed in RPM, `0` if not reported, `1` for solid state drives
`0x3A`	2.0+	Negotiated Speed	`WORD`	`NUMBER`	Negotiated speed of the bus in Gbit/s, zero if unknown
`0x3C`	2.0+	Capable Speed	`WORD`	`NUMBER`	Fastest speed of the bus the drive is capable of in Gbit/s, zero if unknown

Table 95. Form factor

Value	Meaning
`0x02`	3.5-inch
`0x03`	2.5-inch
`0x04`	1.8-inch
`0x05`	Smaller than 1.8-inch
`0x06`	mSATA
`0x07`	M.2
`0x08`	MicroSSD

Value	Meaning
`0x09`	CFast
`0x20`	EDSFF of unknown form factor
`0x21`	EDSFF 1U short
`0x22`	EDSFF 1U long
`0x23`	EDSFF E3 short
`0x24`	EDSFF E3 long

30.26. HP/HPE DIMM Current Configuration Record (Type 244)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Describes a memory region configured on a module of a memory device (type 17), e.g. of a persistent memory one, and the interleave set it belongs to. There is a record for each region. dmidecode says the number of the regions of a module is given by the DIMM attributes record (type 232), whose layout it leaves reserved where the number would be, so the number is not known from the data. The shortest structures end after the interleave set index, at `0x14`.

Table 96. HP/HPE DIMM Current Configuration Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`244`	HP/HPE DIMM Current Configuration Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Handle of the memory device structure (type 17)
`0x06`	2.0+	Region ID	`BYTE`	`NUMBER`	Unique on the module
`0x07`	2.0+	Region Type	`WORD`	`BITFIELD`	Bit 0: volatile, bit 1: byte accessible persistent, bit 2: block I/O
`0x09`	2.0+	Region Size	`QWORD`	`NUMBER`	Size of the region in mebibytes
`0x11`	2.0+	Passphrase State	`BYTE`	`NUMBER`	Non-zero if a passphrase is enabled, byte accessible persistent regions only
`0x12`	2.0+	Interleave Set Index	`WORD`	`NUMBER`	1-based, unique per set; regions of the same index are interleaved
`0x14`	2.0+	Interleave DIMM Count	`BYTE`	`NUMBER`	Number of the DIMMs of the interleave set

Offset	Version	Name	Length	Value	Description
`0x15`	2.0+	Interleave Set Health	`BYTE`	`ENUM`	`0`: healthy, `1`: DIMM missing, `2`: configuration inactive, `3`: SPA missing, `4`: new goal, `5`: locked. Values `6` to `0xFF` are reserved.

30.27. HP/HPE Extension Board Inventory Record (Type 245)

Source: dmidecode, `dmioem.c`. No dumps of the data corpus hold the structure.

Describes an extension board installed in the server, e.g. a PCIe riser, as ML, DL and Alletra servers do from Gen11 onwards. The board type selects the layout of the rest of the structure, and the structures of other board types are not decoded.

Table 97. HP/HPE Extension Board Inventory Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`245`	HP/HPE Extension Board Inventory Record indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Board Type	`BYTE`	`ENUM`	`0`: PCIe riser, `1`: PCIe riser of an MHS platform

Table 98. PCIe riser (board type 0)

Offset	Name	Length	Description
`0x05`	Riser Position	`BYTE`	`1`: primary, `2`: secondary, `3`: tertiary, `4`: quaternary, `10`: front
`0x06`	Riser ID	`BYTE`	
`0x07`	CPLD Version	`BYTE`	Bits 0-6: version, bit 7: `B.` release; zero if the riser has no CPLD
`0x08`	Riser Name	`STRING`	

Table 99. PCIe riser of an MHS platform (board type 1)

Offset	Name	Length	Description
`0x05`	Riser ID	`BYTE`	
`0x06`	Firmware Major Version	`BYTE`	Zero if the riser has no firmware
`0x07`	Firmware Minor Version	`BYTE`	
`0x08`	Miscellaneous Attributes	`BYTE`	Bit 0: the firmware may be downgraded

Offset	Name	Length	Description
`0x09`	Riser Name	`STRING`	
`0x0A`	Slot Count	`BYTE`	
`0x0B`	Slot IDs	Slot Count BYTEs	One per slot

Chapter 31. Intel reference code structures

The Intel reference code, which the firmware of Intel platforms includes whatever its vendor is, adds structures of its own. Their types are taken by structures of some vendors, which place them at other types or not at all.

31.1. Intel ASF Information (Type 129)

Source: reverse engineered from the dumps of the data corpus.

Names the Alert Standard Format (ASF) support of the platform. The structure is found on Intel platforms of Acer, Dell, Framework, Lenovo, Samsung and Toshiba, with the strings of the Intel reference code, e.g. `Intel_ASF` and `Intel_ASF_001`, of Dell, e.g. `Dell_ASF_001`, or of Insyde, e.g. `Insyde_ASF_001`. The meaning of the fields other than the strings is not established.

The structure is told by the bytes at offset `0x04`, `01 01`, since other vendors give type 129 to structures of their own, e.g. Gigabyte, whose structures begin with zeroes there. The number of the second string is not checked: the structures whose strings are the same are written back with a single one, to which both fields refer.

Table 100. Intel ASF Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`129`	Intel ASF Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x08` or `0x10`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Version	`BYTE`	`NUMBER`	Version of the structure, presumably, `1` in all known data
`0x05`	2.0+	Name	`BYTE`	`STRING`	Name of the ASF support, e.g. `Intel_ASF`
`0x06`	2.0+	Identifier	`BYTE`	`STRING`	Identifier of the ASF support, e.g. `Intel_ASF_001`
`0x07`	2.0+	Parameter	`BYTE`	Varies	Meaning not established, `1` in most data and `0` on ThinkPad T61p
`0x08`	2.0+	Extra Data	8 BYTEs	Varies	Meaning not established, present in the structures of 16 bytes, e.g. `00 00 00 01 00 00 08 01` on Acer Aspire 3680

31.2. Intel Active Management Technology Information (Type 130)

Source: Intel AMT Implementation and Reference Guide, "AMT Information Table Parameter Definitions"; MeshAgent, `modules/smbios.js`; and the dumps of the data corpus.

Tells whether the platform supports Intel Active Management Technology (AMT), and which of its features are enabled. The structure is told by the `$AMT` signature at offset `0x04`.

The guide gives the flags of the features as 0 when disabled, 1 when enabled, and values from 2 on as not applicable, and puts a structure version of two bytes after the signature, which no firmware of the data corpus holds. The flags are not consistent with each other on every platform, e.g. some give the network as enabled while AMT is not supported, so they are best taken as the firmware reports them rather than as the state of the platform.

Table 101. Intel Active Management Technology Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`130`	Intel Active Management Technology Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x14` or `0x18`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Signature	4 BYTEs	`$AMT`	
`0x08`	2.0+	AMT Supported	`BYTE`	`BOOLEAN`	
`0x09`	2.0+	AMT Enabled	`BYTE`	`BOOLEAN`	
`0x0A`	2.0+	IDE-R Enabled	`BYTE`	`BOOLEAN`	Storage redirection: IDE redirection (IDE-R), or USB redirection (USB-R) from AMT 11.0 onwards
`0x0B`	2.0+	SOL Enabled	`BYTE`	`BOOLEAN`	Serial over LAN
`0x0C`	2.0+	Network Enabled	`BYTE`	`BOOLEAN`	Network interface of AMT
`0x0D`	2.0+	Extended Data	`BYTE`	Varies	Left to the platform vendor, `0xA5` in all known data
`0x0E`	2.0+	OEM Capabilities 1	`BYTE`	`BITFIELD`	Features of AMT the platform supports, see below
`0x0F`	2.0+	OEM Capabilities 2	`BYTE`	`ENUM`	Bits 0 to 3: terminal emulation of Serial over LAN, see below. Bits 4 to 7 are reserved
`0x10`	2.0+	OEM Capabilities 3	`BYTE`	`BITFIELD`	Bits 0 to 5 are reserved, bit 6: Remote Secure Erase is supported, bit 7: the BIOS supports Secure Boot
`0x11`	2.0+	OEM Capabilities 4	`BYTE`	`BITFIELD`	Targets of the remote boot the platform supports, see below
`0x12`	2.0+	KVM Enabled	`BYTE`	`BOOLEAN`	Keyboard, video and mouse redirection
`0x13`	2.0+	Reserved	`BYTE`	`0`	
`0x14`	2.0+	Reserved	4 BYTEs	Varies	Present in the structures of length `0x18`, whose meaning is not established

Table 102. OEM capabilities 1

Bits	Meaning
0	Storage redirection is supported
1	Serial over LAN is supported
2	The BIOS can be reflashed remotely
3	The setup of the BIOS can be entered remotely
4	The BIOS can be paused remotely
5	Booting from the floppy disk can be prevented
6	Booting from the CD can be prevented
7	Reserved, while most firmware of the data corpus sets it

Table 103. Terminal emulation

Value	Meaning
1	VT52
2	VT100+
3	VT-UTF8
4	PC-ANSI
5-15	Reserved

Table 104. OEM capabilities 4

Bits	Meaning
0	Thunderbolt dock is supported
1	Booting over HTTPS is supported
2	Booting to the local pre-boot authentication (PBA) is supported
3	Booting to the Windows recovery environment (WinRE) is supported
4	Booting without Secure Boot is supported
5	Coexistence with Wi-Fi is supported
6-7	Reserved

31.3. Intel vPro Information (Type 131)

Source: Intel AMT Implementation and Reference Guide, "vPro Verification Table Parameter Definitions"; MeshAgent, `modules/smbios.js`; and the dumps of the data corpus, whose versions match the ones of Intel firmware version information (type 221).

Intel calls the structure the vPro verification table. It describes the parts of the platform Intel vPro technology relies on: the capabilities of the processor, of the Management Engine firmware, of the TPM

and of the BIOS, the versions of the firmware and of its BIOS extension (MEBx), and the PCI functions of the chipset and of the network controller the Management Engine uses. The structure is found on Intel platforms whether they are of vPro or not.

The guide gives the offsets from a point 8 bytes before the one of this chapter, i.e. its `0x0C` is `0x04` here.

The structure is told by the `vPro` signature at offset `0x38`, since some vendors give type 131 to structures of their own in the same table, e.g. Lenovo ThinkVantage Technologies enablement. PCI functions are given as their device and function numbers, `(device << 3) | function`, followed by their bus.

Table 105. Intel vPro Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`131`	Intel vPro Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x40`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	CPU Capabilities	`DWORD`	`BITFIELD`	Capabilities of the processor, see below
`0x08`	2.0+	MEBx Major Version	`WORD`	`NUMBER`	Version of the Management Engine BIOS extension, all zeroes if not reported. The layout of the guide holds the capabilities of the memory controller hub from here to `0x0F`, see below
`0x0A`	2.0+	MEBx Minor Version	`WORD`	`NUMBER`	
`0x0C`	2.0+	MEBx Hotfix	`WORD`	`NUMBER`	
`0x0E`	2.0+	MEBx Build Number	`WORD`	`NUMBER`	
`0x10`	2.0+	LPC Device and Function	`BYTE`	`NUMBER`	PCI device and function of the LPC bridge, `0xF8` for 0:1F.0
`0x11`	2.0+	LPC Bus	`BYTE`	`NUMBER`	PCI bus of the LPC bridge
`0x12`	2.0+	LPC Device ID	`WORD`	`NUMBER`	PCI device ID of the LPC bridge, which tells the chipset
`0x14`	2.0+	Reserved	`DWORD`	`0`	Reserved by the guide, `0xFFFFFFFF` on older platforms
`0x18`	2.0+	ME Capabilities	`DWORD`	`BITFIELD`	Capabilities of the Management Engine firmware, see below
`0x1C`	2.0+	ME Minor Version	`WORD`	`NUMBER`	Version of the Management Engine firmware, as the firmware reports it
`0x1E`	2.0+	ME Major Version	`WORD`	`NUMBER`	
`0x20`	2.0+	ME Build Number	`WORD`	`NUMBER`	
`0x22`	2.0+	ME Hotfix	`WORD`	`NUMBER`	

Offset	Version	Name	Length	Value	Description
0x24	2.0+	TPM Capabilities	DWORD	BITFIELD	Capabilities of the Trusted Platform Module, see below
0x28	2.0+	Wired NIC Device and Function	BYTE	NUMBER	PCI device and function of the wired network controller, 0xFE for 0:1F.6, 0xC8 for 0:19.0
0x29	2.0+	Wired NIC Bus	BYTE	NUMBER	PCI bus of the wired network controller
0x2A	2.0+	Wired NIC Device ID	WORD	NUMBER	PCI device ID of the wired network controller, 0xFFFF or zero if absent, e.g. on the platforms of Meteor Lake, which have no integrated one
0x2C	2.0+	Reserved	WORD	0	
0x2E	2.0+	Wireless NIC Device and Function	BYTE	NUMBER	PCI device and function of the wireless network controller, see below
0x2F	2.0+	Wireless NIC Bus	BYTE	NUMBER	PCI bus of the wireless network controller
0x30	2.0+	Wireless NIC Device ID	WORD	NUMBER	PCI device ID of the wireless network controller, 0xFF00 or 0x00FF if not given
0x32	2.0+	Reserved	WORD	0	
0x34	2.0+	BIOS Capabilities	DWORD	BITFIELD	Capabilities of the BIOS, see below
0x38	2.0+	Signature	4 BYTEs	vPro	
0x3C	2.0+	Reserved	DWORD	0	

The wireless network controller is given as the guide describes it by some firmware, e.g. the one of NEC PC-LZ750LS, which gives Intel Centrino Advanced-N 6235 (0x088F) at 2:00.0. The firmware of HP laptops gives the host bridge at 0:0.0 in its place, e.g. device ID 0x0104 of Intel Sandy Bridge, 0x0A04 of Haswell-ULT, 0x1604 of Broadwell-U and 0x2A40 of GM45. Most firmware gives no controller, and holds 0xFF00 or 0x00FF at 0x30, which tells so, as zero and 0xFFFF do.

Table 106. CPU capabilities

Bits	Meaning
0	Virtual machine extensions (VMX) are enabled
1	Safer mode extensions (SMX) are enabled
2	The processor supports Trusted Execution Technology (TXT)
3	TXT is enabled
4	The processor supports Virtualization Technology (VT-x)
5	VT-x is enabled
6-31	Reserved

Table 107. ME capabilities

Bits	Meaning
0	The Management Engine is enabled
1	The firmware supports Quiet System Technology (QST)
2	The firmware supports Alert Standard Format (ASF)
3	The firmware supports Active Management Technology (AMT)
4	Reserved
5	The firmware supports Small Business Technology (SBT)
6	The firmware supports the level III manageability
7-31	Reserved by the guide. The newer firmware sets some of them, e.g. bits 13 to 15, whose meaning is not established

Table 108. TPM capabilities

Bits	Meaning
0	A TPM is on board
1	The TPM is enabled
2-15	Reserved
16-23	Major version of the TCG specification the TPM is designed to
24-31	Minor version of the TCG specification the TPM is designed to

Table 109. BIOS capabilities

Bits	Meaning
0	VT-x can be set in the setup of the BIOS
1	VT-d can be set in the setup of the BIOS
2	TXT can be set in the setup of the BIOS
3	The TPM can be set in the setup of the BIOS
4	The Management Engine can be set in the setup of the BIOS
5	The BIOS supports the extensions of the Virtual Appliance, the ACPI operation region
6	The platform data area of the SPI flash is reserved
7-9	Highest version of the Virtual Appliance supported: 0 for 2.6, 1 for 3.0, the others reserved. Some firmware gives 3
10-31	Reserved

31.3.1. Older Layout

The guide gives the capabilities of the memory controller hub (MCH) from `0x08` to `0x0F`, where the newer firmware puts the version of the BIOS extension, and firmware of both kinds is found in the data corpus, e.g. the one of HP EliteBook 6930p holds the hub. The hub is found at 0:0.0, so its device and function and its bus are zero, while its device ID is not, which tells the layout: a version of the BIOS extension is either given in full, or not at all.

Table 110. Memory controller hub capabilities

Offset	Name	Length	Description
`0x08`	MCH Device and Function	`BYTE`	PCI device and function of the hub
`0x09`	MCH Bus	`BYTE`	PCI bus of the hub
`0x0A`	MCH Device ID	`WORD`	PCI device ID of the hub, e.g. `0x2A40` of Intel GM45
`0x0C`	MCH Capabilities	`DWORD`	Bit 0: the chipset supports VT-d, bit 1: VT-d is enabled, bit 2: the chipset supports TXT, bit 3: TXT is enabled

31.4. Intel Platform Information (Type 148)

Source: Intel Bay Trail reference code, `Vlv2TbltDevicePkg`, as EDK II published it up to edk2-stable201903: the OEM type `0x94` of `SmBiosMiscDxe`.

Gives the versions of the firmware components and some settings of an Intel Bay Trail platform, all of them as strings: the numbers the firmware writes in hexadecimal, and the settings as their values, e.g. `0` or `1`. The data corpus holds it on Lenovo IdeaPad 100S, whose firmware is of Insyde.

The structure is told by its length of 26 bytes, since Intel server boards give type 148 to structures of other layouts, e.g. S2600WTT to one of 48 bytes, which gives the versions of the firmware components of the board.

Table 111. Intel Platform Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`148`	Intel Platform Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x1A`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	GOP Version	`BYTE`	`STRING`	Version of the graphics output protocol (GOP) driver, e.g. `7.2.1013`
`0x05`	2.0+	Microcode Version	`BYTE`	`STRING`	Revision of the microcode in hexadecimal, e.g. `832`
`0x06`	2.0+	MRC Version	`BYTE`	`STRING`	Version of the memory reference code (MRC), e.g. `1.02`

Offset	Version	Name	Length	Value	Description
`0x07`	2.0+	SEC Version	`BYTE`	`STRING`	Version of the firmware of the security engine, the Trusted Execution Engine (TXE), e.g. `1.2.0.1149`
`0x08`	2.0+	ULPMC Version	`BYTE`	`STRING`	Version of the firmware of the ultra low power microcontroller (ULPMC), or `Non ULPMC!!` if there is none
`0x09`	2.0+	PMC Version	`BYTE`	`STRING`	Version of the firmware of the power management controller (PMC), e.g. `0x4_45`
`0x0A`	2.0+	P-unit Version	`BYTE`	`STRING`	Version of the firmware of the P-unit, e.g. `0x27`
`0x0B`	2.0+	SoC Version	`BYTE`	`STRING`	Stepping of the system on a chip, e.g. `0F (C0 Stepping)`
`0x0C`	2.0+	Board Version	`BYTE`	`STRING`	Board, e.g. `BAY LAKE CR (6)`
`0x0D`	2.0+	Fab Version	`BYTE`	`STRING`	Fab of the board, e.g. `0`
`0x0E`	2.0+	CPU Flavor	`BYTE`	`STRING`	Flavor of the processor, e.g. `VLV-QC Notebook (3)`
`0x0F`	2.0+	BIOS Version	`BYTE`	`STRING`	Version of the firmware, e.g. `E2CN13WW`
`0x10`	2.0+	PMIC Version	`BYTE`	`STRING`	Vendor and revision of the power management IC (PMIC), e.g. `41.01`
`0x11`	2.0+	Touch Version	`BYTE`	`STRING`	Version of the firmware of the touch screen, `NA` if there is none
`0x12`	2.0+	Secure Boot	`BYTE`	`STRING`	Secure Boot setting, `0` or `1`
`0x13`	2.0+	Boot Mode	`BYTE`	`STRING`	Boot mode of the firmware, as a number
`0x14`	2.0+	SpeedStep Mode	`BYTE`	`STRING`	Enhanced Intel SpeedStep setting
`0x15`	2.0+	Turbo Mode	`BYTE`	`STRING`	Turbo Boost setting
`0x16`	2.0+	Maximum C-state	`BYTE`	`STRING`	Deepest C-state of the processor, e.g. `3`
`0x17`	2.0+	Graphics Turbo	`BYTE`	`STRING`	Turbo setting of the graphics
`0x18`	2.0+	Idle Reserve	`BYTE`	`STRING`	S0ix setting
`0x19`	2.0+	RC6	`BYTE`	`STRING`	RC6 power saving setting of the graphics

31.5. Intel Management Engine Interface Information (Type 219)

Source: reverse engineered from the dumps of the data corpus. The layouts of the firmware status registers are the ones coreboot describes in `src/soc/intel/common/block/include/intelblocks/cse.h` and the `me.h` headers of the systems

on a chip.

Keeps the firmware status registers of the Host Embedded Controller Interfaces (HECI) of the Intel Management Engine, as the firmware of the host has read them at boot. Most tables hold a group association (type 14) named `$MEI`, which is not a reliable way to find the structure: in about half of the tables of the data corpus the group gives handle 0, and in some it refers to Intel firmware version information (type 221) or to other structures.

The structures are found on Intel platforms whatever the vendor of the firmware is. The reference code gives them type 219 by default, which some vendors give to structures of their own: Dell places the structures at type 203 or 235, or keeps them at type 219 on some systems, e.g. XPS 13 9350, while HP servers carry none. The structure is told by its version, `1`, and by the name of the first interface, `MEI1`, which every known structure holds, so that the structures vendors give the same types are not taken for it.

Table 112. Intel Management Engine Interface Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`219`	Intel Management Engine Interface Information indicator, `203` on Dell platforms
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure, `0x06` + 25 × Interface Count
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Version	`BYTE`	`NUMBER`	`1` in all known data
`0x05`	2.0+	Interface Count	`BYTE`	`NUMBER`	Number of interfaces, 3 or 4
`0x06`	2.0+	Interfaces	Varies	Varies	Interfaces, see Interfaces

31.5.1. Intel Management Engine Interface Information — Interfaces

The interfaces are the PCI functions 0:16.0, 0:16.1, 0:16.4 and 0:16.5, named `MEI1` to `MEI4`. The registers are the ones of the PCI configuration space of the function, at offsets 0x40, 0x48, 0x60, 0x64, 0x68 and 0x6C. An absent function reads with all bits set. Interfaces whose registers are all zeroes are not reported, e.g. `MEI4` on most platforms of the data corpus.

Table 113. Intel Management Engine Interface Information: Interface

Offset	Name	Length	Value	Description
`0x00`	Name	`BYTE`	`STRING`	Name of the interface, e.g. `MEI1`
`0x01`	HFSTS1	`DWORD`	`BITFIELD`	Firmware status register 1, see HFSTS1
`0x05`	HFSTS2	`DWORD`	`BITFIELD`	Firmware status register 2: boot progress, which depends on the generation
`0x09`	HFSTS3	`DWORD`	`BITFIELD`	Firmware status register 3, see HFSTS3
`0x0D`	HFSTS4	`DWORD`	`BITFIELD`	Firmware status register 4, which depends on the generation

Offset	Name	Length	Value	Description
`0x11`	HFSTS5	`DWORD`	`BITFIELD`	Firmware status register 5, which depends on the generation
`0x15`	HFSTS6	`DWORD`	`BITFIELD`	Firmware status register 6, which depends on the generation. coreboot describes only bits 30 and 31 of it, for Skylake, as the commit state of the field programmable fuses (FPF)

31.5.2. Intel Management Engine Interface Information—HFSTS1

The layout of the first register of the first interface is the same in all generations of the Management Engine.

Table 114. Intel Management Engine Interface Information: HFSTS1

Bits	Name	Description
0-3	Working State	`0` reset, `1` initializing, `2` recovery, `3` test, `4` M3 without UMA (disabled on the 6 series chipsets), `5` normal, `6` waiting for the platform to be disabled, `7` transition of the operation state, `8` invalid CPU plugged in
4	Manufacturing Mode	Firmware is in the manufacturing mode, in which the platform is not locked down yet
5	FPT Bad	Flash partition table is corrupted
6-8	Operation State	Power state of the firmware: `0` preboot, `1` M0 with UMA, `2` M0 power gated, `4` M3 without UMA, `5` M0 without UMA, `6` bring-up, `7` M0 without UMA with an error
9	Initialization Complete	Firmware has completed its initialization
10	Bring-up Load Failure	
11	Update in Progress	Firmware is being updated
12-15	Error Code	`0` no error, `1` uncategorized failure, `3` image failure, `4` debug failure
16-19	Operation Mode	`0` normal, `2` debug, `3` temporarily disabled, `4` security override by jumper, `5` security override by message
20-23	Reserved	
24	Boot Options Present	
25-29	Reserved	
30	D3 Support Valid	
31	D0i3 Support Valid	

The bits from 20 on are the ones coreboot gives for Skylake and later.

31.5.3. Intel Management Engine Interface Information — HFSTS3

Table 115. Intel Management Engine Interface Information: HFSTS3

Bits	Name	Description
0-3	Reserved	
4-6	Firmware SKU	`2` consumer, `3` corporate, `5` lite, `0` for none on some firmware of Management Engine 11.0. Matches the version string of the `ME Firmware Version` component of Intel firmware version information otherwise
7-31	Reserved	Depend on the generation

31.6. Intel Firmware Version Information (Type 221)

Source: reverse engineered from the dumps of the data corpus, and consistent with the firmware version information of the Intel reference code.

Lists the versions of the firmware components of a platform, as the Intel reference code knows them: the reference code modules, the microcode, the Management Engine firmware, the option ROMs and others. Each module of the reference code, e.g. the CPU, the Management Engine, the PCH or the System Agent, adds a structure of its own, whose first item is the version of the module itself, named `Reference Code - <module>`. A group association (type 14) named `Firmware Version Info` often lists the structures of the platform.

The structures are found on Intel platforms whatever the vendor of the firmware is. The reference code gives them type 221 by default, which some vendors give to structures of their own: Dell places the structures at type 205 or 237, or keeps them at type 221 on some systems, e.g. XPS 13 9350, where its own structures of that type are told apart by their length of 19 bytes and by carrying no strings, while HP servers carry none.

Table 116. Intel Firmware Version Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`221`	Intel Firmware Version Information indicator, `205` on Dell platforms
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure, `0x05` + 7 × Item Count
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Item Count	`BYTE`	`NUMBER`	Number of items
`0x05`	2.0+	Items	Varies	Varies	Versions of the firmware components, see Items

31.6.1. Intel Firmware Version Information — Items

Each part of the version is unspecified on its own, with all bits set. Components which carry a single number set that part only, e.g. the revision identifiers of the PCH and the System Agent set the build number, and the HSIO versions set the major version. Components which carry a state rather than a

version, e.g. `PCH-CRID Status`, give it as the version string.

The parts are written in decimal or in hexadecimal depending on the component: the versions of the Management Engine firmware in decimal, e.g. 16.1.32.2473, and the versions of the reference code modules in hexadecimal, e.g. 7.0.5C.50.

The revision of the microcode, `uCode Version`, is the revision followed by the low byte of the build number, `(revision << 8) | build`: e.g. revision 4 and build 55 stand for 0x437 of Alder Lake-P, and revision 0x41 and build 14 for 0x410E of Raptor Lake-P. The older platforms give it in the build number alone, e.g. build 234 for 0xEA.

Some items whose other parts are all unspecified give the build number as `0x00FF` rather than `0xFFFF`, e.g. `UNDI Version` and `MEBx version` on some platforms, which is presumably a byte of all bits set.

Table 117. Intel Firmware Version Information: Item

Offset	Name	Length	Value	Description
`0x00`	Component Name	`BYTE`	`STRING`	Name of the component, e.g. `Reference Code - CPU` or `ME Firmware Version`
`0x01`	Version String	`BYTE`	`STRING`	Version written as a string, or state of the component, e.g. `Consumer SKU` or `Disabled`
`0x02`	Major Version	`BYTE`	`NUMBER`	`0xFF` if unspecified
`0x03`	Minor Version	`BYTE`	`NUMBER`	`0xFF` if unspecified
`0x04`	Revision	`BYTE`	`NUMBER`	`0xFF` if unspecified
`0x05`	Build Number	`WORD`	`NUMBER`	`0xFFFF` if unspecified

31.7. Intel Silicon View Technology Milestones (Type 222)

Source: reverse engineered from the dumps of the data corpus.

Lists the milestones of the boot the firmware reports to the debug and validation tools of Intel Silicon View Technology (SVT). A group association (type 14) named `Intel® Silicon View Technology` often refers to the structure. The reference code gives the structure type 222, which Dell places at type 206, while HP servers carry none.

Table 118. Intel Silicon View Technology Milestones structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`222`	Intel Silicon View Technology Milestones indicator, `206` on Dell platforms
`0x01`	2.0+	Length	`BYTE`	`0x08` + 2 × Milestone Count	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Version	`BYTE`	`NUMBER`	`1` in all known data

Offset	Version	Name	Length	Value	Description
`0x05`	2.0+	Parameter	`WORD`	Varies	Meaning not established, `0x0099` in all known data
`0x07`	2.0+	Milestone Count	`BYTE`	`NUMBER`	
`0x08`	2.0+	Milestones	Varies	Varies	Milestones, see Milestones

31.7.1. Intel Silicon View Technology Milestones — Milestones

Table 119. Intel Silicon View Technology Milestones: Milestone

Offset	Name	Length	Value	Description
`0x00`	Code	`BYTE`	`NUMBER`	`0x10`, `0x20` and `0x30` in all known data
`0x01`	Name	`BYTE`	`STRING`	`Memory Init Complete`, `End of DXE Phase` and `BIOS Boot Complete`

31.7.2. Intel Silicon View Technology Milestones — Aligned Layout

Some firmware does not pack the structure, e.g. the one of Dell XPS 13 9365, which carries `01 00 99 00 03 10 01 20 02 30 03 00` where other firmware carries `01 99 00 03 10 01 20 02 30 03`: the parameter is aligned to two bytes, and the milestones are followed by a byte which makes the length of the structure even. The length of such structures is `0x0A` + 2 × Milestone Count, e.g. `0x10` for three milestones, where the packed layout has `0x0E`.

Both layouts are told by the version and the parameter, `1` and `0x0099`, which every known structure holds, so that the structures vendors give type 222 are not taken for the milestones on the systems whose firmware names no vendor, e.g. the ones of Dell on PowerEdge servers.

Table 120. Intel Silicon View Technology Milestones structure, aligned layout

Offset	Name	Length	Value	Description
`0x04`	Version	`BYTE`	Varies	`1` in all known data
`0x05`	Padding	`BYTE`	`0`	Aligns the parameter to two bytes
`0x06`	Parameter	`WORD`	Varies	Meaning not established, `0x0099` in all known data
`0x08`	Milestone Count	`BYTE`	Varies	Number of milestones
`0x09`	Milestones	Varies	Varies	Milestones, see Milestones
`0x09` + 2 × Milestone Count	Padding	`BYTE`	`0`	Makes the length of the structure even

Chapter 32. Intel RSD structures

The structures of Intel Rack Scale Design (RSD) describe the devices of a system to the management software of the rack, which reads them from the management controller the firmware passes the table on to. The specification reserves types 190 to 209 for them, of which it defines types 190 and 192 to 200. Memory is described by the standard memory structures (types 17 to 20), which type 197 extends.

32.1. Intel RSD Network Interface Card Information (Type 190)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.1.

Declares a network interface card present or configured in the system, with its PCI identifiers, its speeds and the address of its network port.

Table 121. Intel RSD Network Interface Card Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`190`	Network Interface Card Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	PCI Class Code	`BYTE`	`NUMBER`	PCI base class of the device, which tells its general function
`0x05`	2.0+	Slot Number	`WORD`	`NUMBER`	Physical number of the slot connected to the PCIe port of the card
`0x07`	2.0+	Vendor ID	`WORD`	`NUMBER`	PCI vendor identifier of the card
`0x09`	2.0+	Device ID	`WORD`	`NUMBER`	PCI device identifier of the card
`0x0B`	2.0+	SubVendor ID	`WORD`	`NUMBER`	PCI subsystem vendor identifier, which tells the model of the card along with the subsystem identifier
`0x0D`	2.0+	SubDevice ID	`WORD`	`NUMBER`	PCI subsystem identifier, which tells the model of the card along with the subsystem vendor identifier
`0x0F`	2.0+	Maximum Speed	`DWORD`	`SPEED`	Highest speed of the card, in Mbit/s, e.g. 10000 for 10 Gbit/s
`0x13`	2.0+	Current Speed	`DWORD`	`SPEED`	Speed the card currently works at, in Mbit/s

Offset	Version	Name	Length	Value	Description
`0x17`	2.0+	Port Index	`WORD`	`NUMBER`	Index of the port in the system. It is the value the firmware gives to the device instance selector of the boot flags (parameter 5, data byte 5) of the IPMI Get/Set System Boot Options commands.
`0x19`	2.0+	MAC Address	32 BYTEs	Varies	Network MAC address of the port
`0x39`	2.0+	Firmware Version	`BYTE`	`STRING`	Version of the firmware installed on the card

32.2. Intel RSD PCIe Information (Type 192)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.2.

Declares a PCIe device present and enabled in the system, with its PCI identifiers and the state of its link.

Table 122. Intel RSD PCIe Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`192`	PCIe Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	PCI Class Code	`BYTE`	`NUMBER`	PCI base class of the device, which tells its general function
`0x05`	2.0+	Slot Number	`WORD`	`NUMBER`	Physical number of the slot connected to the PCIe port of the device. It is the slot ID of the system slot structure (type 9) of the slot the device is in.
`0x07`	2.0+	Vendor ID	`WORD`	`NUMBER`	PCI vendor identifier of the device
`0x09`	2.0+	Device ID	`WORD`	`NUMBER`	PCI device identifier of the device
`0x0B`	2.0+	SubVendor ID	`WORD`	`NUMBER`	PCI subsystem vendor identifier, which tells the model of the device along with the subsystem identifier
`0x0D`	2.0+	SubDevice ID	`WORD`	`NUMBER`	PCI subsystem identifier, which tells the model of the device along with the subsystem vendor identifier
`0x0F`	2.0+	Link Speed	`DWORD`	`NUMBER`	Current speed of the link. The specification gives no unit or encoding of the value.
`0x13`	2.0+	Link Width	`DWORD`	`NUMBER`	Current number of the lanes of the link, e.g. 1, 2, 4, 8 or 16

32.3. Intel RSD Processor CPUID Information (Type 193)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.3.

Declares CPUID leaves of the processors in the system. Leaves 00h to 10h and 14h to 80000008h are stored in structures of different subtypes. The firmware constructs the structures as late as possible on every boot, so that they reflect the current configuration of the system.

Table 123. Intel RSD Processor CPUID Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`193`	Processor CPUID Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Socket Designation	`BYTE`	`STRING`	Physically labelled socket or board position of the processor, e.g. "Socket 0"
`0x05`	2.0+	Subtype	`BYTE`	`ENUM`	Set of leaves stored in the structure, see Subtypes
`0x06`	2.0+	Leaves	Varies	Varies	CPUID leaves, see Leaves and Leaf

32.3.1. Intel RSD Processor CPUID — Subtypes

Table 124. Intel RSD Processor CPUID: Subtype field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Leaves 00h to 10h
`0x02`	`2`	Leaves 14h to 80000008h

32.3.2. Intel RSD Processor CPUID — Leaves

Leaves are stored in the following order, 16 bytes each. For some leaves the input value of ECX (subleaf) is specified.

Table 125. Intel RSD Processor CPUID: Leaves

Offset	Subtype 1	Subtype 2
`0x06`	EAX = 00h	EAX = 14h, ECX = 00h
`0x16`	EAX = 01h	EAX = 15h
`0x26`	EAX = 02h	EAX = 16h
`0x36`	EAX = 03h	EAX = 17h, ECX = 00h
`0x46`	EAX = 04h	EAX = 17h, ECX = 01h
`0x56`	EAX = 05h	EAX = 17h, ECX = 02h

Offset	Subtype 1	Subtype 2
0x66	EAX = 06h	EAX = 17h, ECX = 03h
0x76	EAX = 07h	EAX = 80000000h
0x86	EAX = 09h	EAX = 80000001h
0x96	EAX = 0Ah	EAX = 80000002h
0xA6	EAX = 0Bh	EAX = 80000003h
0xB6	EAX = 0Dh, ECX = 00h	EAX = 80000004h
0xC6	EAX = 0Fh, ECX = 00h	EAX = 80000006h
0xD6	EAX = 10h, ECX = 00h	EAX = 80000007h
0xE6	—	EAX = 80000008h

32.3.3. Intel RSD Processor CPUID — Leaf

Table 126. Intel RSD Processor CPUID: Leaf

Offset	Name	Length	Value	Description
0x00	EAX	DWORD	Varies	EAX register value returned by the CPUID instruction
0x04	EBX	DWORD	Varies	EBX register value returned by the CPUID instruction
0x08	ECX	DWORD	Varies	ECX register value returned by the CPUID instruction
0x0C	EDX	DWORD	Varies	EDX register value returned by the CPUID instruction

32.4. Intel RSD Storage Device Information (Type 194)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.5.

Declares a storage device of the system: where it is connected, how it is accessed, what it is, and, for a PCIe device, its PCI identifiers.

Table 127. Intel RSD Storage Device Information structure

Offset	Version	Name	Length	Value	Description
0x00	2.0+	Type	BYTE	194	Storage Device Information indicator
0x01	2.0+	Length	BYTE	Varies	Length of the structure
0x02	2.0+	Handle	WORD	Varies	

Offset	Version	Name	Length	Value	Description
`0x04`	2.0+	Port Designation	`BYTE`	`STRING`	Physically labelled port or board position of the device, e.g. "SATA Port 0" or "PCIe Port 1"
`0x05`	2.0+	Device Index	`BYTE`	`NUMBER`	Index of the device in the IPMI Get/Set System Boot Options commands, as the firmware gives it. The management software uses it to choose the device to boot from.
`0x06`	2.0+	Connector Type	`BYTE`	`ENUM`	Connector of the device, see Connector types
`0x07`	2.0+	Device Protocol	`BYTE`	`ENUM`	Protocol the device is accessed with, see Device protocols
`0x08`	2.0+	Device Type	`BYTE`	`ENUM`	Type of the device, see Device types
`0x09`	2.0+	Device Capacity	`DWORD`	`NUMBER`	Capacity of the device, in GB
`0x0D`	2.0+	Device RPM	`WORD`	`SPEED`	Rotational speed of a hard disk drive, in revolutions per minute, or 0 for any other device
`0x0F`	2.0+	Device Model	`BYTE`	`STRING`	Model of the device
`0x10`	2.0+	Device Serial	`BYTE`	`STRING`	Serial number of the device
`0x11`	2.0+	PCI Class Code	`BYTE`	`NUMBER`	PCI base class of a PCIe device, or 0 for any other device
`0x12`	2.0+	Vendor ID	`WORD`	`NUMBER`	PCI vendor identifier of a PCIe device, or 0 for any other device
`0x14`	2.0+	Device ID	`WORD`	`NUMBER`	PCI device identifier of a PCIe device, or 0 for any other device
`0x16`	2.0+	Sub Vendor ID	`WORD`	`NUMBER`	PCI subsystem vendor identifier of a PCIe device, or 0 for any other device
`0x18`	2.0+	Sub Device ID	`WORD`	`NUMBER`	PCI subsystem identifier of a PCIe device, or 0 for any other device
`0x1A`	2.0+	Firmware Version	`BYTE`	`STRING`	Version of the firmware of the device

32.4.1. Intel RSD Storage Device — Connector types

Table 128. Intel RSD Storage Device: Connector type field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Unknown
`0x01`	`1`	SATA
`0x02`	`2`	SAS

Hex Value	Dec Value	Meaning
`0x03`	`3`	PCIe
`0x04`	`4`	M.2
`0x05`	`5`	USB
`0x06`	`6`	U.2

32.4.2. Intel RSD Storage Device — Device protocols

Table 129. Intel RSD Storage Device: Device protocol field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Unknown
`0x01`	`1`	IDE
`0x02`	`2`	AHCI
`0x03`	`3`	NVMe
`0x04`	`4`	USB

32.4.3. Intel RSD Storage Device — Device types

Table 130. Intel RSD Storage Device: Device type field

Hex Value	Dec Value	Meaning
`0x01`	`1`	Hard disk drive (HDD)
`0x02`	`2`	Solid-state drive (SSD)
`0x03`	`3`	Optical, DVD
`0x04`	`4`	Optical, Blu-ray
`0x05`	`5`	USB

The specification defines no value `0x00` for the device type, unlike for the connector type and the device protocol.

32.5. Intel RSD TPM Information (Type 195)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.6.

Declares a Trusted Platform Module (TPM) configuration the system can be set to. The structure is present only if the system has a TPM, even a disabled one, and the system has a structure for every TPM version it can work with.

Table 131. Intel RSD TPM Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`195`	TPM Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	TPM Configuration Index	`BYTE`	`NUMBER`	One-based index of the configuration, which the management software gives to the Set TPM Configuration IPMI command to choose it
`0x05`	2.0+	TPM Version	`BYTE`	`STRING`	Version of the TPM, e.g. "TPM 2.0"
`0x06`	2.0+	TPM Status	`BYTE`	`ENUM`	Current state of the TPM, see TPM status

32.5.1. Intel RSD TPM — Status

Table 132. Intel RSD TPM: TPM status field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Disabled
`0x01`	`1`	Enabled

32.6. Intel RSD TXT Information (Type 196)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.7.

Declares the state of Intel Trusted Execution Technology (TXT). The structure is present only if the system supports TXT, even when it is disabled.

Table 133. Intel RSD TXT Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`196`	TXT Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	TXT Status	`BYTE`	`ENUM`	Current state of TXT, see TXT status

32.6.1. Intel RSD TXT — Status

Table 134. Intel RSD TXT: TXT status field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Disabled
`0x01`	`1`	Enabled

32.7. Intel RSD Memory Device Extended Information (Type 197)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.8.

Extends the memory device information (type 17) with the kind of the memory module, its firmware, its power and the SMBus address it is managed at. Every structure belongs to a memory device structure of the table.

Table 135. Intel RSD Memory Device Extended Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`197`	Memory Device Extended Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Memory device structure (type 17) the structure extends
`0x06`	2.0+	Memory Type	`BYTE`	`ENUM`	Kind of the memory module, see Memory types
`0x07`	2.0+	Memory Media	`BYTE`	`ENUM`	Media the memory is built on, see Memory media
`0x08`	2.0+	Memory Firmware Revision	`BYTE`	`STRING`	Revision of the firmware of the memory module
`0x09`	2.0+	Memory Firmware API Version	`BYTE`	`STRING`	Version of the interface the firmware of the memory module provides
`0x0A`	2.0+	Max TDP	`DWORD`	`NUMBER`	Highest power the memory module consumes, its thermal design power, in milliwatts
`0x0E`	2.0+	Memory System Management Model (SMBus) Address	`BYTE`	`NUMBER`	SMBus address of the memory module, which gives access to its serial presence detect (SPD) data or to the firmware interface of a 3D XPoint module. The management software passes it on to Intel Intelligent Power Node Manager to query the memory subsystem.

32.7.1. Intel RSD Memory Device Extended — Memory types

Table 136. Intel RSD Memory Device Extended: Memory type field

Hex Value	Dec Value	Meaning
`0x00`	`0`	DIMM

Hex Value	Dec Value	Meaning
`0x01`	`1`	NVDIMM-N, persistent memory accessed by bytes
`0x02`	`2`	NVDIMM-F, persistent memory accessed by blocks
`0x03`	`3`	NVDIMM-P

32.7.2. Intel RSD Memory Device Extended — Memory media

Table 137. Intel RSD Memory Device Extended: Memory media field

Hex Value	Dec Value	Meaning
`0x00`	`0`	DRAM
`0x01`	`1`	NAND
`0x03`	`3`	Proprietary

The specification defines no value `0x02` for the memory media.

32.8. Intel RSD FPGA Information (Type 198)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.9.

Declares a field-programmable gate array (FPGA) of the system: what it is, its hard processor subsystem (HPS), its high-speed serial interface (HSSI), where it is reached on PCIe, and the memory on its package. The structure is present only if the system has FPGAs.

Table 138. Intel RSD FPGA Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`198`	FPGA Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	FPGA Index	`BYTE`	`NUMBER`	One-based index of the FPGA in the system
`0x05`	2.0+	FPGA Type	`BYTE`	`ENUM`	Kind of the FPGA, see FPGA types
`0x06`	2.0+	FPGA Status	`BYTE`	`ENUM`	Current state of the FPGA, see FPGA status
`0x07`	2.0+	Socket Identifier	`BYTE`	`NUMBER`	Socket the FPGA is integrated into. Meaningful for an integrated FPGA only.
`0x08`	2.0+	FPGA Vendor	`BYTE`	`STRING`	Vendor of the FPGA, as a name or, for a discrete FPGA, as the vendor identifier of its PCI configuration space
`0x09`	2.0+	FPGA Family	`BYTE`	`STRING`	Family of the FPGA

Offset	Version	Name	Length	Value	Description
`0x0A`	2.0+	FPGA Model	`BYTE`	`STRING`	Model of the FPGA
`0x0B`	2.0+	FPGA Bit Stream Version	`BYTE`	`STRING`	Version of the bit stream loaded into the FPGA
`0x0C`	2.0+	FPGA HPS Core Count	`BYTE`	`NUMBER`	Number of the cores of the hard processor subsystem, if the FPGA has one
`0x0D`	2.0+	—	`BYTE`	Varies	Not defined by the specification, which goes on from the core count to the next field at `0x0E`
`0x0E`	2.0+	FPGA HPS ISA	`BYTE`	`ENUM`	Instruction set of the cores of the hard processor subsystem, see HPS instruction sets
`0x0F`	2.0+	FPGA HSSI Configuration	`BYTE`	`ENUM`	What the high-speed serial interface of the FPGA is configured for, see HSSI configurations
`0x10`	2.0+	FPGA HSSI Port Count	`BYTE`	`NUMBER`	Number of the ports of the high-speed serial interface, e.g. 4 for "4x10G" or 2 for "2x16 PCIe"
`0x11`	2.0+	FPGA HSSI Port Speed	`BYTE`	`NUMBER`	Speed of a port of the high-speed serial interface, in Gbit/s, or its number of the lanes for PCIe, e.g. 10 for "2x10G" or 16 for "2x16 PCIe"
`0x12`	2.0+	FPGA HSSI Side Band Configuration	`BYTE`	`STRING`	Side band the high-speed serial interface is configured through, e.g. "SPI", "I2C-0" or "SMBus"
`0x13`	2.0+	FPGA Reconfiguration Slots	`BYTE`	`NUMBER`	Number of the partial reconfiguration (PR) slots of the FPGA
`0x14`	2.0+	PCIe Slot Number	`WORD`	`NUMBER`	Physical number of the slot connected to the PCIe port the FPGA is reached at
`0x16`	2.0+	PCIe Bus Identifier	`BYTE`	`NUMBER`	PCI bus number the FPGA is reached at
`0x17`	2.0+	PCIe Device Identifier	`BYTE`	`NUMBER`	PCI device number the FPGA is reached at
`0x18`	2.0+	PCIe Function Identifier	`BYTE`	`NUMBER`	PCI function number the FPGA is reached at
`0x19`	2.0+	Thermal Design Power	`DWORD`	`NUMBER`	Thermal design power of the FPGA, in milliwatts
`0x1D`	2.0+	On Package Memory Technology	`BYTE`	`ENUM`	Technology of the memory on the package of the FPGA, see Memory technologies

Offset	Version	Name	Length	Value	Description
`0x1E`	2.0+	On Package Memory Capacity	`DWORD`	`NUMBER`	Capacity of the memory on the package, in MB of 2^{20} bytes each, the way the memory sizes of SMBIOS are given
`0x22`	2.0+	On Package Memory Speed	`WORD`	`SPEED`	Speed of the memory on the package, in MHz

32.8.1. Intel RSD FPGA — Types

Table 139. Intel RSD FPGA: FPGA type field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Integrated
`0x01`	`1`	Discrete
`0x02`	`2`	Discrete, with a system on chip (SoC) or a hard processor subsystem (HPS)

32.8.2. Intel RSD FPGA — Status

Table 140. Intel RSD FPGA: FPGA status field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Disabled
`0x01`	`1`	Enabled

32.8.3. Intel RSD FPGA — HPS instruction sets

Table 141. Intel RSD FPGA: FPGA HPS ISA field

Hex Value	Dec Value	Meaning
`0x00`	`0`	x86
`0x01`	`1`	x86-64
`0x02`	`2`	IA-64
`0x03`	`3`	ARM A32
`0x04`	`4`	ARM A64
`0x05`	`5`	MIPS32
`0x06`	`6`	MIPS64
`0x07`	`7`	OEM-specific

32.8.4. Intel RSD FPGA — HSSI configurations

Table 142. Intel RSD FPGA: FPGA HSSI configuration field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Networking
`0x01`	`1`	PCIe
`0xFF`	`255`	Information not available. The other fields of the high-speed serial interface are to be ignored.

32.8.5. Intel RSD FPGA — Memory technologies

Table 143. Intel RSD FPGA: On package memory technology field

Hex Value	Dec Value	Meaning
`0x00`	`0`	None, the FPGA has no memory attached
`0x01`	`1`	Embedded DRAM (eDRAM)
`0x02`	`2`	High Bandwidth Memory (HBM)
`0x03`	`3`	High Bandwidth Memory of the second generation (HBM2)

32.9. Intel RSD Cabled PCIe Port Information (Type 199)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.10.

Declares a cabled PCIe port on the fixed side of a system: a port which carries PCIe lanes over a cable, usually through an adapter on the board or in a PCIe slot. A structure describes a cable management interface (CMI) controller of the port, and is present only if the system has cabled PCIe ports. The ports follow the PCI Express External Cabling Specification, revision 3.0, and are reached by the management controller over SMBus to identify and manage the cables.

The firmware and the management controller give the same slot ID to a physical slot. How they agree on it, and on the slots and link widths of the ports, is left out of the specification.

Table 144. Intel RSD Cabled PCIe Port Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`199`	Cabled PCIe Port Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Slot ID	`WORD`	`NUMBER`	Slot the port is connected to. It is the slot ID of the system slot structure (type 9) of the slot, and the one the management controller uses for it, which is given to the Get Cable EEPROM Data IPMI command to choose the port.
`0x06`	2.0+	Cable Port Link Width	`BYTE`	`NUMBER`	Number of the lanes of the port, e.g. 4 for a x4 port or 8 for a x8 port

Offset	Version	Name	Length	Value	Description
`0x07`	2.0+	Cable Index Count	`BYTE`	`NUMBER`	Number `n` of the cable indices of the port, with the lanes they cover
`0x08`	2.0+	Cable Port Indices	2n BYTEs	Varies	Cable indices of the port, 2 bytes each, see Cable port index. The specification lays out four of them at most, up to offset `0x0F`.

32.9.1. Intel RSD Cabled PCIe Port — Cable port index

Table 145. Intel RSD Cabled PCIe Port: Cable port index

Offset	Name	Length	Value	Description
`0x00`	Cable Port Index	`BYTE`	`NUMBER`	Zero-based index of the cable within the slot, which the management controller uses for it too. It is given to the Get Cable EEPROM Data IPMI command along with the slot ID to choose the cable.
`0x01`	Cable Port Index Start Lane	`BYTE`	`NUMBER`	First lane of the four lanes the cable carries: 0, 4, 8 or 12

32.10. Intel RSD Physical Device Mapping Information (Type 200)

Source: Intel® RSD Firmware Extension Specification Software v2.3.2 (document 337201-003US), section 3.11.

Declares the relationship between physical devices and their SMBIOS representation, allowing to link structures to physical locations of the devices in the system.

Each structure describes devices of one type only. There may be several structures describing devices of the same type, if they do not fit into a single structure.

Table 146. Intel RSD Physical Device Mapping Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`200`	Physical Device Mapping Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Physical Device Type	`BYTE`	`ENUM`	Type of the devices described by the structure, see Physical device types
`0x05`	2.0+	Reserved	`BYTE`	`0x00`	
`0x06`	2.0+	Physical Device Mappings	Varies	Varies	Locations of the devices, 4 bytes each, see Physical device mappings

32.10.1. Intel RSD Physical Device Mapping—Device types

Table 147. Intel RSD Physical Device Mapping: Physical device type field

Hex Value	Dec Value	Meaning
`0x00`	`0`	Undefined or invalid
`0x01`	`1`	Processor (type 4)
`0x02`	`2`	PCIe system slot (type 9)
`0x03`	`3`	Memory device (type 17)

Other values are undefined or invalid.

32.10.2. Intel RSD Physical Device Mapping—Device mappings

All numbers are zero-based.

Table 148. Intel RSD Physical Device Mapping: Processor

Offset	Name	Length	Value	Description
`0x00`	SMBIOS Handle	`WORD`	`HANDLE`	Processor information structure (type 4)
`0x02`	Socket Number	`BYTE`	`NUMBER`	Physical number of the socket
`0x03`	Reserved	`BYTE`	`0x00`	

Table 149. Intel RSD Physical Device Mapping: PCIe system slot

Offset	Name	Length	Value	Description
`0x00`	SMBIOS Handle	`WORD`	`HANDLE`	System slot structure (type 9)
`0x02`	Riser Number	`BYTE`	`NUMBER`	Physical number of the riser
`0x03`	Slot Number	`BYTE`	`NUMBER`	Physical number of the slot within the riser

Table 150. Intel RSD Physical Device Mapping: Memory device

Offset	Name	Length	Value	Description
`0x00`	SMBIOS Handle	`WORD`	`HANDLE`	Memory device structure (type 17)
`0x02`	CPU Socket Number	`BYTE`	`NUMBER`	Physical number of the processor, which the memory device belongs to
`0x03`	DIMM Slot Number	`BYTE`	`NUMBER`	Physical number of the slot within the processor

Chapter 33. Lenovo-specific structures

33.1. Lenovo ThinkVantage Technologies Enablement (Type 131)

Source: dmidecode, `dmioem.c`, and the dumps of the data corpus.

Tells which ThinkVantage Technologies the platform enables, of which only the diagnostics are known. The structure is told by its length and by its first string, `TVT-Enablement`, since Intel vPro information takes type 131 in the same table.

Table 151. Lenovo ThinkVantage Technologies Enablement structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`131`	Lenovo ThinkVantage Technologies Enablement indicator
`0x01`	2.0+	Length	`BYTE`	`0x16`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Version	`BYTE`	`NUMBER`	`1` in all known data
`0x05`	2.0+	Features	16 BYTEs	`BITFIELD`	Bits of the technologies, of which bit 127 tells that the diagnostics (PC-Doctor) are available
`0x15`	2.0+	Signature	`BYTE`	`STRING`	`TVT-Enablement`

33.2. Lenovo Date Record (Type 134)

Source: reverse engineered from the dumps of the data corpus.

Holds a date of a ThinkPad, which falls within the years the model has been made in, and so is probably its manufacture date, e.g. November 27, 2021 in `lenovo/thinkpad-t14-g2a-20xk.bin`. Structures of type 134 holding TPM information instead are told by their first string, `TPM INFO`.

Table 152. Lenovo Date Record structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`134`	Lenovo Date Record indicator
`0x01`	2.0+	Length	`BYTE`	`0x0D`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Day	`BYTE`	`BCD`	
`0x05`	2.0+	Month	`BYTE`	`BCD`	
`0x06`	2.0+	Year	`WORD`	`BCD`	Low byte first, e.g. `21 20` for 2021
`0x08`	2.0+	Reserved	5 BYTEs	`0`	

33.3. Lenovo TPM Information (Type 134)

Source: reverse engineered from the dumps of the data corpus.

Tells the vendor of the TPM of a Lenovo laptop by its TCG vendor ID, e.g. `ATML` for Atmel in `lenovo/thinkpad-t61p-6460-6xg.bin`, or `STM` for STMicroelectronics in `lenovo/thinkpad-x220-4290le6.bin`. The structures are told from the date records of the same type by their first string, `TPM INFO`.

Table 153. Lenovo TPM Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`134`	Lenovo TPM Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x10`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Unknown 1	`BYTE`	Varies	`0` in all known data
`0x05`	2.0+	TPM Vendor ID	4 BYTEs	Varies	TCG vendor ID of the TPM, in ASCII
`0x09`	2.0+	Unknown 2	2 BYTEs	Varies	`01 01` in all known data
`0x0B`	2.0+	Unknown 3	`WORD`	Varies	`0` in all known data
`0x0D`	2.0+	Unknown 4	`BYTE`	Varies	`0`, `2` or `3`
`0x0E`	2.0+	Signature	`BYTE`	`STRING`	`TPM INFO`
`0x0F`	2.0+	Description	`BYTE`	`STRING`	`System Reserved`

33.4. Lenovo Mobile PC OEM Data (Type 135)

Source: Compal hel81 Service Manual Software Specification, dmidecode, `dmioem.c`, and the dumps of the data corpus.

Holds one OEM structure of a Lenovo laptop, told by its number and revision. The service manuals name the type "Bulk for Lenovo Mobile PC Unique OEM Data". OEM structures 0, 1, 2 and 4 are found in the data corpus too, and only structures 2 and 3 are decoded.

Table 154. Lenovo Mobile PC OEM Data structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`135`	Lenovo Mobile PC OEM Data indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Signature	2 BYTEs	`TP`	
`0x06`	2.0+	OEM Structure Offset	`BYTE`	`0x07`	Offset of the OEM structure number

Offset	Version	Name	Length	Value	Description
`0x07`	2.0+	OEM Structure Number	`BYTE`	`NUMBER`	
`0x08`	2.0+	OEM Structure Revision	`BYTE`	`NUMBER`	
`0x09`	2.0+	Data	Varies	Varies	Data of the OEM structure, see below

Table 155. Device presence detection (OEM structure 3, revision 1)

Offset	Name	Length	Description
`0x09`	Device Presence	`BYTE`	Bit 0: a fingerprint reader is present

Table 156. Bay I/O (OEM structure 2)

Offset	Name	Length	Description
`0x08`	Signature	8 BYTEs	`BAY I/O `, in place of the revision
`0x10`	Version	`BYTE`	`1` to `4` in the data corpus
`0x11`	Data	Varies	Layout not known

33.5. Lenovo OEM Data (Type 140)

Source: dmidecode, `dmioem.c`, the README files of the firmware updates of Lenovo, and the dumps of the data corpus.

Holds one OEM structure of a Lenovo system, told by its number and revision. OEM structures 0 to 9 are found in the data corpus, and only structure 7, the embedded controller program, is decoded. Structure 9 refers to two strings too, which hold the same version, e.g. `1.32`, whose meaning is not established; ThinkPad E14 Gen 5 carries the first of them only, so that the second reference is left dangling.

Table 157. Lenovo OEM Data structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`140`	Lenovo OEM Data indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Signature	6 BYTEs	`LENOVO`	
`0x0A`	2.0+	OEM Structure Offset	`BYTE`	`0x0B`	Offset of the OEM structure number
`0x0B`	2.0+	OEM Structure Number	`BYTE`	`NUMBER`	
`0x0C`	2.0+	OEM Structure Revision	`BYTE`	`NUMBER`	

Offset	Version	Name	Length	Value	Description
`0x0D`	2.0+	Data	Varies	Varies	Data of the OEM structure, see below

Table 158. ThinkPad embedded controller program (OEM structure 7, revision 1)

Offset	Name	Length	Description
`0x0D`	Version ID	`STRING`	Version of the program, which the README files of the firmware updates of Lenovo name, e.g. `N20HT28W`
`0x0E`	Release Date	`STRING`	Release date of the program, e.g. `10/08/2020`

33.6. Lenovo Machine Type Model (Type 200)

Source: reverse engineered from the dumps of the data corpus.

Tells the brand of a consumer or small business laptop of Lenovo, e.g. an IdeaPad, a Yoga, a Legion, a V series one or a ThinkBook, `IdeaPad` for all of them, and its full machine type model, e.g. `20VE00U9RU`, of which the system information (type 1) holds only the machine type, e.g. `20VE`. The structures are 16 bytes long. ThinkPads carry structures of 7 bytes at the same type, which refer to three strings and whose meaning is not known, and which are left undecoded.

Table 159. Lenovo Machine Type Model structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`200`	Lenovo Machine Type Model indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Brand	`BYTE`	`STRING`	
`0x05`	2.0+	Machine Type Model	`BYTE`	`STRING`	
`0x06`	2.0+	Data	10 BYTEs	Varies	Meaning not established, e.g. `01 27 01 14 01 50 B2 80 01 05`

Chapter 34. Sun-specific structures

34.1. Sun Processor Extended Information (Type 132)

Source: illumos, `usr/src/uts/common/sys/smbios_impl.h`.

Extends the processor information (type 4) with initial APIC IDs of the processor strands.

Table 160. Sun Processor Extended Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`132`	Sun Processor Extended Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Processor Handle	`WORD`	`HANDLE`	Processor information structure (type 4) being extended
`0x06`	2.0+	FRU	`BYTE`	`NUMBER`	FRU indicator
`0x07`	2.0+	Number of APIC IDs	`BYTE`	`NUMBER`	Number of initial APIC IDs (n)
`0x08`	2.0+	Initial APIC IDs	2n BYTEs	Varies	Initial APIC IDs of the processor strands, a `WORD` each

34.2. Sun Port Extended Information (Type 136)

Source: illumos, `usr/src/uts/common/sys/smbios_impl.h`.

Extends the port connector information (type 8) with the chassis containing the port and the device connected to it.

Table 161. Sun Port Extended Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`136`	Sun Port Extended Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x0C`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Chassis Handle	`WORD`	`HANDLE`	Chassis containing the port
`0x06`	2.0+	Port Connector Handle	`WORD`	`HANDLE`	Port connector information structure (type 8) being extended
`0x08`	2.0+	Device Type	`BYTE`	`NUMBER`	Type of the connected device. Values are not described in the source.
`0x09`	2.0+	Device Handle	`WORD`	`HANDLE`	Connected device
`0x0B`	2.0+	PHY	`BYTE`	`NUMBER`	PHY number

34.3. Sun PCI-Express Root Complex Information (Type 138)

Source: illumos, `usr/src/uts/common/sys/smbios_impl.h`.

Describes the PCI address of a PCI-Express root complex or root port and the baseboard containing it.

Table 162. Sun PCI-Express Root Complex Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`138`	Sun PCI-Express Root Complex Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x08`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Baseboard Handle	`WORD`	`HANDLE`	Baseboard containing the root complex or root port
`0x06`	2.0+	Bus/Device/Function	`WORD`	`NUMBER`	PCI address of the root complex or root port

Bus, device and function numbers are packed as in PCI routing IDs: bus number in bits 15:8, device number in bits 7:3, function number in bits 2:0. The structure definition leaves the packing out, and it is known from how illumos uses the field: the `x86pi` topology module matches it against the bits 23:8 of the `reg` property of PCI devices, see `x86pi_bdf` in `usr/src/lib/fm/topo/modules/i86pc/x86pi/x86pi_subr.c`.

34.4. Sun Memory Array Extended Information (Type 144)

Source: illumos, `usr/src/uts/common/sys/smbios_impl.h`.

Extends the physical memory array information (type 16) with the parent component and the PCI address of the memory controller.

Table 163. Sun Memory Array Extended Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`144`	Sun Memory Array Extended Information indicator
`0x01`	2.0+	Length	`BYTE`	`0x0A`	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Array Handle	`WORD`	`HANDLE`	Physical memory array structure (type 16) being extended
`0x06`	2.0+	Component Parent Handle	`WORD`	`HANDLE`	Parent component of the memory array
`0x08`	2.0+	Bus/Device/Function	`WORD`	`NUMBER`	PCI address of the memory controller

Bus, device and function numbers are packed as in PCI routing IDs: bus number in bits 15:8, device number in bits 7:3, function number in bits 2:0. The structure definition leaves the packing out, and it is known from how illumos uses the field: the `x86pi` topology module matches it against the bits 23:8 of the `reg` property of PCI devices, see `x86pi_bdf` in `usr/src/lib/fm/topo/modules/i86pc/x86pi/x86pi_subr.c`.

34.5. Sun Memory Device Extended Information (Type 145)

Source: illumos, `usr/src/uts/common/sys/smbios_impl.h`.

Extends the memory device information (type 17) with the DRAM channel and chip selects of the memory device.

Table 164. Sun Memory Device Extended Information structure

Offset	Version	Name	Length	Value	Description
`0x00`	2.0+	Type	`BYTE`	`145`	Sun Memory Device Extended Information indicator
`0x01`	2.0+	Length	`BYTE`	Varies	Length of the structure
`0x02`	2.0+	Handle	`WORD`	Varies	
`0x04`	2.0+	Memory Device Handle	`WORD`	`HANDLE`	Memory device structure (type 17) being extended
`0x06`	2.0+	DRAM Channel	`BYTE`	`NUMBER`	
`0x07`	2.0+	Number of Chip Selects	`BYTE`	`NUMBER`	Number of chip selects (n)
`0x08`	2.0+	Chip Selects	n BYTEs	Varies	Chip selects, a `BYTE` each

Chapter 35. Other structures

Structures of the data corpus whose meaning is not established, which are left undecoded.

35.1. Undocumented structures

Structures found in the dumps of the data corpus which no public source describes, and whose meaning could not be established from the data. They are left undecoded, and what is known of them is gathered here, so that the dumps which will tell more can be recognized.

35.1.1. Structures of several vendors

Type 128: sample structure of the firmware kit

Found on Insyde firmware: Lenovo IdeaPad and ThinkBook, Toshiba Satellite, HP 250 notebooks, Framework laptops, Gigabyte G5 and Neousys Nuvo.

The formatted area holds the bytes `55 AA 55 AA`, and the structure carries the strings `Oem Test 1` and `Oem Test 2`, or `Oem Type 128 Test 1` and `Oem Type 128 Test 2`, which no field refers to. The structure is a sample of the firmware kit which the vendors of the systems leave in place.

Other firmware carries structures of type 128 of its own. Microsoft Surface Laptop 3 carries one referring to a single string, `SHA` followed by 40 hexadecimal digits, as many as the ones of a SHA-1 hash. Fujitsu ESPRIMO Q958 carries the structures of Fujitsu described with type 143.

Type 129

The Intel ASF information is decoded, see the chapter of the Intel reference code structures. Other firmware gives type 129 to structures of its own: Lenovo IdeaPad Z565 and Toshiba Satellite C655D carry one of 5 bytes, whose strings are the ones of a type 128 sample, cut short, Gigabyte carries structures of 38 bytes on a server board, beginning with zeroes and holding `0xFF` from `0x0C` on, and Precision M3800 one of 9 bytes of zeroes.

Type 133

Lenovo firmware of all the known models, from ThinkPad T61p to ThinkPad T14 Gen 3, carries a structure of 5 bytes, which refers to its only string at `0x04`: `KHOIHGIUCCHHII`, the same in all known data. Xiaomi TM1701, of Insyde firmware, carries a structure of the same form, whose string `TIMIPCR001V001` is of the same length.

AMI firmware of Intel platforms of the 12th generation and later (ASUS Prime H610M-R and ROG STRIX Z690-A, Gigabyte B760M DS3H and H610M H V2, Beelink SEi) carries a structure of 44 bytes, identical on all of them, which begins with the GUID `7895B4F9-924D-410A-9639-123862B28A63` and holds the PCI vendor ID of Intel, `0x8086`, twice. ThinkPad T14 Gen 3 carries one of the same GUID with other bytes and strings.

AMI firmware of servers (ASUS RS520-E9, iXsystems TrueNAS H30 and M50) carries a structure of 12 bytes holding two `DWORD` values, e.g. `0x67B6D000` and `0x4000`, which look like the address of a region of memory aligned to 4 KiB below 4 GiB, and its size.

Type 136

Found on the firmware of every vendor, mostly on Intel platforms. The formatted area holds a single `WORD`, whose value depends on the code base of the firmware rather than on the system: `0x0000` on AMI firmware, `0xFFFF` on Insyde firmware, and `0x5A5A` on the firmware of Phoenix code base, e.g. ThinkPad, Dell Precision 490 and HP Z600. Newer AMI tables carry a second structure of type 136, of a `DWORD` of zero, which is the only one of Dell G15 5510.

Type 143

Fujitsu LifeBook S 4546 carries two structures, the first of which holds the signature *`FJ_OEM`* at `0x05`. Fujitsu ESPRIMO Q958 carries the same two structures as type 128, followed by structures of the same type beginning with `03`, a byte of 9, `0x0B` or `0x0D`, and `01 01`, some of which refer to strings such as `[U:SCB1` and `[V:CSM1`.

MSI boards carry one referring to eleven strings, which name the firmware components and their versions, e.g. `GOP1-[N/A]`, `RAID-1…16`, `PXEU1-v.2`, `AMI-AptioV` and `MsiFlash-17` on PRO B650-S WIFI, and `MEFW-9.0.20.1447`, `VBIOS-2179` and `PXEL1-2.53` on MS-7816. The strings of the components the board lacks are `N/A`.

Type 148

Intel S2600WTT server board carries a structure of 48 bytes whose bytes from `0x05` on are the numbers of its strings, which give the versions of the firmware components of the board: the BIOS ID, the BMC firmware, the Management Engine, the SDR package, the FRU and the SDR file, the microcode patches and more. The structure of 26 bytes of the Intel Bay Trail reference code, e.g. of Lenovo IdeaPad 100S, is decoded as the Intel platform information.

Type 248

Acer Aspire A515 and Nitro AN515 carry `00 02 6C 00 00 00 00`, and Predator G9-592 carries `00 02 6C 00` followed by the string `BDF`, which no field refers to. Dell Inspiron 5370, Latitude 7390 and 7490 and XPS 13 9300 carry `00 00 00 00` followed by the number of their only string, `BDF_DE_P87G`, `BDF_DE_1901` or `12345678901`.

Lenovo IdeaPad, Legion, Yoga, ThinkBook and V130 carry 14 bytes beginning with `AC AC AA 17`, followed by the `WORD` values `0` or `0xFFFF` at `0x08` and `0`, `8` or `0x0B` at `0x0A`, and zeros.

35.1.2. Acer

Type 172

Acer Aspire A515, Nitro AN515 and Predator G9-592 carry `02` and a byte which differs from one model to another, from `0x26` to `0x44`, followed by 3-byte records of an ID, from `1` to `7`, and a `WORD` value: `0x00FF`, `1`, `0x00FF`, `1`, `0x000F`, `0x00FF` and `0x00FF`, the same on all of them. Predator G9-592 carries the records `1` to `6` only, and Aspire A515-52 and A515-52G the records `8` to `11` too, of the values `0`, `1`, `2` and `1`. The IDs `1` to `7` are the kinds of the devices of the device list (type 171). The second byte is not the length of that structure, although it is on Nitro AN515-31.

35.1.3. Dell

Type 179

Dell G15 5510 carries a structure referring to a single string, `1[` followed by 64 hexadecimal digits and `]`, in the `key[value]` form of the OEM strings (type 11) of Dell. The digits are as many as the ones of a SHA-256 hash.

Type 217

Dell notebooks and workstations, from Latitude D500 to G15 5510 and XPS 13 9300, carry a structure referring to the strings `US-101` and `Proprietary` at `0x04` and `0x05`, followed by `01 03`, the same on all of them, which may describe the keyboard. Precision M3800 carries one of blank strings, followed by `00 00`, and G15 5515 one referring to `Keyboard` twice, followed by `02 02`.

Type 222

Found on Dell firmware of every kind of system, from OptiPlex GX110 and PowerEdge 1300 to PowerEdge R740xd and XPS 13 9300, of 13 bytes, or of 15 or 16 bytes on newer systems.

Offset	Length	Value
`0x04`	`BYTE`	`1` on most systems, `0xC1` on OptiPlex GX110, PowerEdge 400SC and SC420 and Precision 390 and 490, `0` on PowerEdge C6320, R630 to R740xd, Precision T7600 and G15 5510
`0x05`	`BYTE`	A single bit set: `1`, `2`, `4`, `8`, `0x40` or `0x80`
`0x06`	`WORD`	`0xFFFF` when the date which follows is not set, `0` or `0x0D` otherwise
`0x08`	5 `BYTE`	Date and time in BCD: year of two digits, month, day, hour and minute, e.g. `22 04 21 10 24` for 2022-04-21 10:24, or zeros
`0x0D`	2 or 3 `BYTE`	Only in structures of 15 or 16 bytes, e.g. `01 00 00` on notebooks, or `03 00 01` on PowerEdge R640 to R740xd

The date is set on most notebooks and desktops, and on a few servers. It is mostly weeks or months later than the release date of the firmware, but it is earlier on PowerEdge 4400 and R740xd, so it is not the date the firmware was updated.

35.1.4. HP

ProLiant servers

The ProLiant G1 to G6 servers carry structures of types 196, 198, 205, 210, 213, 214, 215, 217, 218, 220, 222, 223, 227 and 228, which neither dmidecode nor the drivers of HP describe. Type 196 holds a single byte of zero, type 198 e.g. `17 00 00 01 3C 00 01`, and type 205 the ASCII text `FATx` at `0x06`.

HP Z600

The workstation carries structures of types 153, 197 and 207, of which type 197 is not the processor information of the ProLiant servers.

35.1.5. Lenovo

Type 131: CMOS layout

ThinkPads of Phoenix firmware, T61p, T410, W510 and X201, carry a structure of type 131 of 17 bytes referring to the strings `BOOTINF 20h`, `BOOTDEV 21h` and `KEYPTRS 23h`, which name registers of the CMOS memory, along with the Intel vPro information and the ThinkVantage Technologies enablement of the same type. The numbers of the strings are followed by `FF FF 1F 00 00 00 00 00`, a byte of `2` on T61p and `3` on the others, and a byte of zero.

Type 132

ThinkPads and Lenovo B590 carry three bytes, sometimes followed by a byte of zero: `1` or `2`, a byte from `0xC0` to `0xDB`, e.g. `0xD8` on most of them, and `0x36`, the same on all of them.

Types 137 to 139

ThinkPad T61p carries structures of types 137 to 139.

Type 141

ThinkPads of 2019 and later (T14 Gen 1 to Gen 3, T490, E14 Gen 5, L15 Gen 1) carry several structures of 18 to 56 bytes, each of which begins with the text `THNK`, followed by a `WORD` of zero and the kind of the structure, a `DWORD` at `0x0A`: `0`, `0x10`, `0x30`, `0x40`, `0x50` or `0x60`. The data of the structure begins at `0x0E`.

The structures of the kinds `0` to `0x40` hold a `DWORD` value, e.g. a small number, `0xFFFFFFFF`, or the text `LiP` on T14 Gen 2a. The one of the kind `0x50` holds two `QWORD` values. The one of the kind `0x60` is a log of five events: a `WORD` count of the events, e.g. `22`, followed by 8-byte records of the date and the time of the latest ones in BCD: year of four digits as a `WORD`, month, day, hour, minute, second and a byte of zero, e.g. `24 20 05 26 18 08 39 00` for 2024-05-26 18:08:39, the latest first.

35.1.6. Phoenix and Award

Foxconn M61PMV and Packard Bell iMedia S3210, of Phoenix firmware, carry structures of types 188, 190, 192 and 194, and Gigabyte GA-MA74GMT-S2 and GA-MA78GM-S2H, of Award firmware, structures of types 188, 190, 192, 194 and, on GA-MA74GMT-S2, 196 and 198, of 212 to 244 bytes, which hold tables of small numbers. Award Software was acquired by Phoenix Technologies, and the structures are of the code base the firmware of both descends from.

All of these boards are AMD platforms of DDR2 memory, e.g. Phenom 9550 on AMD 780G and Athlon II X2 215 on NVIDIA MCP78, and the structures share their layout whatever the chipset is. They look like the state of the memory controllers the reference code of AMD keeps as it trains the memory, the `MCTStatStruc` and `DCTStatStruc` of the code coreboot carries: types 192 and 194 hold per-byte-lane delays, e.g. `0F 10 10 10` and `17 17 16 16`, for the two DRAM controllers. The fields of coreboot have changed over time, and do not match the structures closely enough to decode them.

35.1.7. Cisco

Types 203 and 204

Cisco UCS C220 M5 carries a structure of type 203 of 36 bytes and one of type 204 of 28 bytes along with the decoded ones, both beginning with `03`, as the PCI adapter information does, and referring to no strings: `03 6A 0A 04 04` followed by small numbers, e.g. `3` at `0x0D` and `0x11`, and `03 00 01 02 00 00 08` followed by zeros.