# AppZone C API Reference Guide

1vv0301130 Rev. 4– 2017-12-12

TELIT
TECHNICAL
DOCUMENTATION

[01.2017]

SPECIFICATIONS ARE SUBJECT TO CHANGE WITHOUT NOTICE

## NOTICES LIST

While reasonable efforts have been made to assure the accuracy of this document, Telit assumes no liability resulting from any inaccuracies or omissions in this document, or from use of the information obtained herein. The information in this document has been carefully checked and is believed to be reliable. However, no responsibility is assumed for inaccuracies or omissions. Telit reserves the right to make changes to any products described herein and reserves the right to revise this document and to make changes from time to time in content hereof with no obligation to notify any person of revisions or changes. Telit does not assume any liability arising out of the application or use of any product, software, or circuit described herein; neither does it convey license under its patent rights or the rights of others.

It is possible that this publication may contain references to, or information about Telit products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that Telit intends to announce such Telit products, programming, or services in your country.

## COPYRIGHTS

This instruction manual and the Telit products described in this instruction manual may be, include or describe copyrighted Telit material, such as computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and its licensors certain exclusive rights for copyrighted material, including the exclusive right to copy, reproduce in any form, distribute and make derivative works of the copyrighted material. Accordingly, any copyrighted material of Telit and its licensors contained herein or in the Telit products described in this instruction manual may not be copied, reproduced, distributed, merged or modified in any manner without the express written permission of Telit. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit, as arises by operation of law in the sale of a product.

## COMPUTER SOFTWARE COPYRIGHTS

The Telit and 3rd Party supplied Software (SW) products described in this instruction manual may include copyrighted Telit and other 3rd Party supplied computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and other 3rd Party supplied SW certain exclusive rights for copyrighted computer programs, including the exclusive right to copy or reproduce in any form the copyrighted computer program. Accordingly, any copyrighted Telit or other 3rd Party supplied SW computer programs contained in the Telit products described in this instruction manual may not be copied (reverse engineered) or reproduced in any manner without the express written permission of Telit or the 3rd Party SW supplier. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit or other 3rd Party supplied SW, except for the normal non-exclusive, royalty free license to use that arises by operation of law in the sale of a product.

# USAGE AND DISCLOSURE RESTRICTIONS

## I. License Agreements

The software described in this document is the property of Telit and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

## II. Copyrighted Materials

Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of Telit

## III. High Risk Materials

Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities"). Telit and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

## IV. Trademarks

TELIT and the Stylized T Logo are registered in Trademark Office. All other product or service names are the property of their respective owners.

## V. Third Party Rights

The software may include Third Party Right software. In this case you agree to comply with all terms and conditions imposed on you in respect of such separate software. In addition to Third Party Terms, the disclaimer of warranty and limitation of liability provisions in this License shall apply to the Third Party Right software.

TELIT HEREBY DISCLAIMS ANY AND ALL WARRANTIES EXPRESS OR IMPLIED FROM ANY THIRD PARTIES REGARDING ANY SEPARATE FILES, ANY THIRD PARTY MATERIALS INCLUDED IN THE SOFTWARE, ANY THIRD PARTY MATERIALS FROM WHICH THE SOFTWARE IS DERIVED (COLLECTIVELY “OTHER CODE”), AND THE USE OF ANY OR ALL THE OTHER CODE IN CONNECTION WITH THE SOFTWARE, INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO THIRD PARTY LICENSORS OF OTHER CODE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND WHETHER MADE UNDER CONTRACT, TORT OR OTHER LEGAL THEORY, ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE OTHER CODE OR THE EXERCISE OF ANY RIGHTS GRANTED UNDER EITHER OR BOTH THIS LICENSE AND THE LEGAL TERMS APPLICABLE TO ANY SEPARATE FILES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

# APPLICABILITY TABLE

## PRODUCTS

| | Platform Version ID[1] | Technology |
|---|---|---|
| GE910 SERIES | 13 | 2G |
| HE910 SERIES | 12 | 3G |
| UE910 SERIES | | |
| UL865 SERIES | | |
| UE866 SERIES | | |
| LE910 Cat1 SERIES | 20 | 4G |
| LE910 V2 SERIES | | |
| LE866 SERIES | 23 | |
| ME866A1 SERIES | | |
| LE910D1 SERIES | | |

[1] Platform Version ID is a reference used in the document. It defines the different SW versions, e.g. 13 for SW version 13.xx.xxx, 12 for software version 12.xx.xxx, etc.

# CONTENTS

**NOTICES LIST** ..... **2**

**COPYRIGHTS** ..... **2**

**COMPUTER SOFTWARE COPYRIGHTS** ..... **2**

**USAGE AND DISCLOSURE RESTRICTIONS** ..... **3**

I. License Agreements ..... 3

II. Copyrighted Materials ..... 3

III. High Risk Materials ..... 3

IV. Trademarks ..... 3

V. Third Party Rights ..... 3

**APPLICABILITY TABLE** ..... **4**

**CONTENTS** ..... **5**

**TABLES LIST** ..... **16**

**1. INTRODUCTION** ..... **17**

**2. PRELIMINARY INFORMATION** ..... **20**

**3. M2M_CLOCK_API** ..... **21**

Date/Time Section ..... 22

3.1. m2m_rtc_set_date ..... 22

3.2. m2m_rtc_get_date ..... 22

3.3. m2m_rtc_set_time ..... 22

3.4. m2m_rtc_get_time ..... 22

3.5. m2m_set_timeofday ..... 23

3.6. m2m_get_timeofday ..... 23

Alarm Section ..... 24

3.7. m2m_rtc_set_alarm ..... 24

3.8. m2m_rtc_clear_alarm ..... 24

**4. M2M_FS_API** ..... **25**

File System Section ..... 26

4.1. m2m_fs_open ..... 26

4.2. m2m_fs_close ..... 27

4.3. m2m_fs_create ..... 28

4.4. m2m_fs_write ..... 28

4.5. m2m_fs_read ..... 28
4.6. m2m_fs_delete ..... 29
4.7. m2m_fs_clear ..... 29
4.8. m2m_fs_copy ..... 30
4.9. m2m_fs_find_first ..... 30
4.10. m2m_fs_find_next ..... 30
4.11. m2m_fs_rename ..... 31
4.12. m2m_fs_get_size ..... 31
4.13. m2m_fs_get_size_with_handle ..... 32
4.14. m2m_fs_tell ..... 32
4.15. m2m_fs_seek ..... 32
4.16. m2m_fs_truncate ..... 33
4.17. m2m_fs_getc ..... 33
4.18. m2m_fs_gets ..... 33
4.19. m2m_fs_set_exec_permission ..... 34
4.20. m2m_fs_set_run_permission ..... 34
4.21. m2m_fs_mk_dir ..... 34
4.22. m2m_fs_rename_dir ..... 35
4.23. m2m_fs_rm_dir ..... 35
4.24. m2m_fs_get_free_space ..... 35
4.25. m2m_fs_get_nof_files ..... 36
4.26. m2m_fs_last_error ..... 36

**5. M2M_HW_API ..... 37**

GPIO Section ..... 38
5.1. m2m_hw_gpio_read ..... 40
5.2. m2m_hw_gpio_write ..... 40
5.3. m2m_hw_gpio_conf ..... 40
5.4. m2m_hw_gpio_int_enable ..... 41
5.5. m2m_hw_gpio_int_disable ..... 41
5.6. m2m_hw_gpio_int_enable_on_front ..... 41
5.7. m2m_hw_gpio_int_trigger_bms_sem ..... 42
HW Timer Section ..... 43
5.8. m2m_hw_timer_start ..... 43
5.9. m2m_hw_timer_stop ..... 43
5.10. m2m_hw_timer_state ..... 43
5.11. m2m_hw_set_ms_count ..... 44
5.12. m2m_hw_get_ms_count ..... 44

Main, Auxiliary Ports Section ..... 45
5.13. m2m_hw_uart_open ..... 45
5.14. m2m_hw_uart_aux_open ..... 45
5.15. m2m_hw_uart_close ..... 45
5.16. m2m_uart_close_hwch ..... 46
5.17. m2m_hw_uart_read ..... 46
5.18. m2m_hw_uart_write ..... 46
5.19. m2m_hw_uart_ioctl ..... 47
5.20. m2m_hw_uart_get_state ..... 51
5.21. m2m_hw_uart_aux_get_state ..... 51
USB Section ..... 52
5.22. m2m_hw_usb_open ..... 52
5.23. m2m_hw_usb_set_open_blocking ..... 52
5.24. m2m_hw_usb_close ..... 53
5.25. m2m_usb_close_hwch ..... 53
5.26. m2m_usb_close_hw_allch ..... 53
5.27. m2m_hw_usb_read ..... 54
5.28. m2m_hw_usb_write ..... 54
5.29. m2m_hw_usb_ioctl ..... 55
5.30. m2m_hw_usb_get_state ..... 56
5.31. m2m_hw_usb_getch_from_handle ..... 56
5.32. m2m_hw_usb_get_instance ..... 57
5.33. m2m_hw_usb_cable_check ..... 57
5.34. m2m_hw_usb_get_ownership ..... 58
5.35. m2m_hw_usb_wait_attach_forever ..... 58
5.36. m2m_hw_usb_wait_attach_timeout ..... 58
Power Down Section ..... 59
5.37. m2m_hw_sleep_mode_cfg ..... 59
5.38. m2m_hw_sleep_mode ..... 59
5.39. m2m_hw_power_down ..... 59
OTA Section ..... 60
5.40. m2m_OTA_write_mem_data ..... 60
5.41. m2m_OTA_read_mem_data ..... 60
5.42. m2m_OTA_erase_mem_data ..... 61
**6. M2M_SPI_API ..... 62**
SPI Section ..... 63
6.1. m2m_spi_init ..... 63

6.2. m2m_spi_write ..... 64
6.3. m2m_spi_close ..... 64
**7. M2M_I2C_API ..... 65**
I2C Section ..... 66
7.1. m2m_hw_i2c_conf ..... 66
7.2. m2m_hw_i2c_read ..... 66
7.3. m2m_hw_i2c_write ..... 67
7.4. m2m_hw_i2c_cmb_format ..... 67
**8. M2M_NETWORK_API ..... 68**
Network Registration Section ..... 69
8.1. m2m_network_enable_registration_location_unsolicited ..... 69
8.2. m2m_network_disable_registration_location_unsolicited ..... 69
8.3. m2m_network_get_reg_status ..... 69
8.4. m2m_network_get_cell_information ..... 70
8.5. m2m_network_get_currently_selected_operator ..... 70
8.6. m2m_network_list_available_networks ..... 70
8.7. m2m_network_get_signal_strength ..... 71
8.8. m2m_network_enable_gprs_registration_location_unsolicited ..... 71
8.9. m2m_network_disable_gprs_registration_location_unsolicited ..... 72
8.10. m2m_network_get_gprs_reg_status ..... 72
**9. M2M_OS_API ..... 73**
Module Information Section ..... 74
9.1. m2m_info_get_model ..... 74
9.2. m2m_info_get_manufacturer ..... 74
9.3. m2m_info_get_factory_SN ..... 74
9.4. m2m_info_get_serial_num ..... 75
9.5. m2m_info_get_sw_version ..... 75
9.6. m2m_info_get_fw_version ..... 75
9.7. m2m_info_get_MSISDN ..... 75
9.8. m2m_info_get_IMSI ..... 76
9.9. m2m_os_set_version ..... 76
9.10. m2m_os_get_version ..... 76
Task Section ..... 77
9.11. m2m_os_create_task ..... 77
9.12. m2m_os_get_current_task_id ..... 78
9.13. m2m_os_cooperate_task ..... 78

9.14. m2m_os_destroy_task ..... 78
9.15. m2m_os_send_message_to_task ..... 79
9.16. m2m_os_task_get_enqueued_msg ..... 79
9.17. m2m_os_set_argc ..... 79
9.18. m2m_os_get_argc ..... 80
9.19. m2m_os_set_argv ..... 80
9.20. m2m_os_get_argv ..... 80
9.21. m2m_os_iat_set_at_command_instance ..... 80
9.22. m2m_os_iat_send_at_command ..... 81
9.23. m2m_os_iat_send_atdata_command ..... 81
Memory Pool Section ..... 82
9.24. m2m_os_mem_pool ..... 82
9.25. m2m_os_mem_alloc ..... 82
9.26. m2m_os_mem_realloc ..... 82
9.27. m2m_os_mem_free ..... 83
9.28. m2m_os_get_mem_info ..... 83
System Tick and Sleep Section ..... 84
9.29. m2m_os_retrieve_clock ..... 84
9.30. m2m_os_sleep_ms ..... 84
9.31. m2m_os_sys_reset ..... 84
Trace Section ..... 85
9.32. m2m_os_trace_out ..... 85
**10. M2M_OS_LOCK_API ..... 86**
Semaphore Section ..... 87
10.1. m2m_os_lock_init ..... 87
10.2. m2m_os_lock_binary_init ..... 87
10.3. m2m_os_lock_lock ..... 87
10.4. m2m_os_lock_unlock ..... 88
10.5. m2m_os_lock_unlock_limit ..... 88
10.6. m2m_os_lock_binary_unlock ..... 89
10.7. m2m_os_lock_prioritize ..... 90
10.8. m2m_os_lock_wait ..... 90
10.9. m2m_os_lock_info ..... 91
10.10. m2m_os_lock_destroy ..... 91
msSemaphore Section ..... 92
10.11. m2m_os_mslock_init ..... 92
10.12. m2m_os_mslock_binary_init ..... 92

10.13. m2m_os_mslock_lock ..... 93
10.14. m2m_os_mslock_unlock ..... 93
10.15. m2m_os_mslock_unlock_limit ..... 93
10.16. m2m_os_mslock_binary_unlock ..... 94
10.17. m2m_os_mslock_prioritize ..... 95
10.18. m2m_os_mslock_setlocklimit ..... 96
10.19. m2m_os_mslock_wait ..... 96
10.20. m2m_os_pause_ms ..... 97
10.21. m2m_os_mslock_info ..... 97
10.22. m2m_os_mslock_destroy ..... 97
Mutex Section ..... 98
10.23. m2m_os_mtx_init ..... 98
10.24. m2m_os_mtx_lock ..... 98
10.25. m2m_os_mtx_unlock ..... 100
10.26. m2m_os_mtx_prioritize ..... 100
10.27. m2m_os_mtx_lock_wait ..... 101
10.28. m2m_os_mtx_info ..... 101
10.29. m2m_os_mtx_destroy ..... 102
**11. M2M_SMS_API ..... 103**
SMS Section ..... 104
11.1. m2m_sms_enable_new_message_indication ..... 104
11.2. m2m_sms_disable_new_message_indication ..... 104
11.3. m2m_sms_get_all_messages ..... 104
11.4. m2m_sms_get_text_message ..... 105
11.5. m2m_sms_delete_message ..... 105
11.6. m2m_sms_send_SMS ..... 105
11.7. m2m_sms_set_PDU_mode_format ..... 106
11.8. m2m_sms_set_text_mode_format ..... 106
11.9. m2m_sms_set_preferred_message_storage ..... 106
11.10. m2m_sms_get_preferred_message_storage ..... 107
**12. M2M_SOCKET_API ..... 108**
Socket Creation/Closure/Options Section ..... 109
12.1. m2m_socket_bsd_socket ..... 109
12.2. m2m_socket_bsd_socket_cid ..... 109
12.3. m2m_socket_bsd_close ..... 110
12.4. m2m_socket_bsd_socket_state ..... 110
12.5. m2m_socket_bsd_set_sock_opt ..... 110

12.6. m2m_socket_bsd_get_sock_opt ..... 111
12.7. m2m_socket_bsd_accept ..... 112
12.8. m2m_socket_bsd_shutdown ..... 112
Address Conversion Section ..... 113
12.9. m2m_socket_bsd_addr_str ..... 113
12.10. m2m_socket_bsd_addr_str_ip6 ..... 113
12.11. m2m_socket_bsd_inet_addr ..... 114
12.12. m2m_socket_bsd_inet_addr_ip6 ..... 114
Connect Section ..... 115
12.13. m2m_socket_bsd_connect ..... 115
12.14. m2m_socket_bsd_bind ..... 115
12.15. m2m_socket_bsd_select ..... 116
12.16. m2m_socket_bsd_fd_set_func ..... 117
12.17. m2m_socket_bsd_fd_clr_func ..... 117
12.18. m2m_socket_bsd_fd_isset_func ..... 117
12.19. m2m_socket_bsd_fd_zero_func ..... 118
Domain Name Conversion Section ..... 119
12.20. m2m_socket_bsd_get_host_by_name ..... 119
12.21. m2m_socket_bsd_get_host_by_name_cid ..... 119
12.22. m2m_socket_bsd_get_host_by_name_ip6 ..... 119
12.23. m2m_socket_bsd_get_host_by_name_ip6_cid ..... 120
12.24. m2m_socket_bsd_get_peer_name ..... 120
12.25. m2m_socket_bsd_get_sock_name ..... 121
DNS Section ..... 122
12.26. m2m_socket_bsd_get_dns_ip ..... 122
12.27. m2m_socket_bsd_get_dns_ip_cid ..... 122
12.28. m2m_socket_bsd_get_dns_ip6 ..... 123
12.29. m2m_socket_bsd_get_dns_ip6_cid ..... 123
Network and Host byte order Section ..... 125
12.30. m2m_socket_bsd_htonl ..... 125
12.31. m2m_socket_bsd_htons ..... 125
12.32. m2m_socket_bsd_ntohl ..... 125
12.33. m2m_socket_bsd_ntohs ..... 125
Listen/Receive/Send Section ..... 126
12.34. m2m_socket_bsd_ioctl ..... 126
12.35. m2m_socket_bsd_listen ..... 127
12.36. m2m_socket_bsd_recv ..... 127

12.37. m2m_socket_bsd_recv_data_size ..... 128
12.38. m2m_socket_bsd_recv_from ..... 128
12.39. m2m_socket_bsd_send ..... 129
12.40. m2m_socket_bsd_send_buf_size ..... 129
12.41. m2m_socket_bsd_send_to ..... 130
12.42. m2m_socket_errno ..... 130
Firewall Section ..... 131
12.43. m2m_firewall ..... 131
12.44. m2m_firewall_list ..... 132
12.45. m2m_firewall_ip6 ..... 132
12.46. m2m_firewall_ip6_list ..... 133
PDP Section ..... 134
12.47. m2m_pdp_apn_set ..... 134
12.48. m2m_pdp_apn_get ..... 134
12.49. m2m_pdp_activate ..... 134
12.50. m2m_pdp_activate_cid ..... 135
12.51. m2m_pdp_activate_ip6 ..... 135
12.52. m2m_pdp_activate_ip6_cid ..... 135
12.53. m2m_pdp_deactive ..... 136
12.54. m2m_pdp_deactive_cid ..... 136
12.55. m2m_pdp_get_status ..... 136
12.56. m2m_pdp_get_status_cid ..... 137
12.57. m2m_pdp_get_my_ip ..... 137
12.58. m2m_pdp_get_my_ip_cid ..... 137
12.59. m2m_pdp_get_my_ip6 ..... 137
12.60. m2m_pdp_get_my_ip6_cid ..... 138
12.61. m2m_pdp_get_datavol ..... 138
12.62. m2m_pdp_get_datavol_cid ..... 138

**13. M2M_IPRAW_API ..... 140**

Configuration Section ..... 141
13.1. m2m_ipraw_cfg ..... 141
IPv4 Section ..... 142
13.2. m2m_ip4_send ..... 142
13.3. m2m_ip4_recv ..... 142
IPv6 Section ..... 143
13.4. m2m_ip6_raw ..... 143
13.5. m2m_ip6_send ..... 143

13.6. m2m_ip6_recv ..... 143
UDP Section ..... 144
13.7. m2m_udp_recv_from_ip6raw ..... 144
**14. M2M_SSL_API ..... 145**
SSL Section ..... 146
14.1. m2m_ssl_create_service_from_file ..... 146
14.2. m2m_ssl_delete_service ..... 146
14.3. m2m_ssl_create_context ..... 147
14.4. m2m_ssl_delete_context ..... 147
14.5. m2m_ssl_securesocket ..... 147
14.6. m2m_ssl_connect ..... 148
14.7. m2m_ssl_new_connection (obsolete) ..... 148
14.8. m2m_ssl_delete_connection ..... 149
14.9. m2m_ssl_encode_send ..... 149
14.10. m2m_ssl_decode ..... 150
**15. M2M_TIMER_API.H ..... 151**
Timer Section ..... 152
15.1. m2m_timer_create ..... 152
15.2. m2m_timer_start ..... 152
15.3. m2m_timer_stop ..... 152
15.4. m2m_timer_free ..... 153
**16. M2M_DUMP_API ..... 154**
Dump Section ..... 155
16.1. m2m_dump_init ..... 155
16.2. m2m_dump_save ..... 155
16.3. m2m_dump_clear ..... 155
**17. M2M_SEC_API ..... 156**
Digest Section ..... 157
17.1. m2m_DIGEST_alloc_res ..... 157
17.2. m2m_DIGEST_destroy_res ..... 157
MD5 Section ..... 158
17.3. m2m_MD5_Init ..... 158
17.4. m2m_MD5_Update ..... 158
17.5. m2m_MD5_Final ..... 159
SHA Section ..... 160
17.6. m2m_SHA_Init ..... 160

17.7. m2m_SHA_Update ..... 160
17.8. m2m_SHA_Final ..... 160
17.9. m2m_SHA256_Init ..... 161
17.10. m2m_SHA256_Update ..... 161
17.11. m2m_SHA256_Final ..... 162
**18. M2M_HW_WATCHDOG ..... 163**
Watchdog Section ..... 164
18.1. m2m_hw_watchdog_conf ..... 164
18.2. m2m_hw_watchdog_enable ..... 165
18.3. m2m_hw_watchdog_disable ..... 165
18.4. m2m_hw_watchdog_refresh ..... 166
**19. APPENDIXES ..... 167**
19.1. Examples ..... 167
19.1.1. Open and Close ..... 167
19.1.2. Listing all Files ..... 168
19.1.3. GPIO ..... 169
19.1.4. Semaphore Used for Critical Section ..... 169
19.1.5. Semaphore Use for Inter Process Communication ..... 169
19.1.6. Timer ..... 170
19.1.7. HW Timer ..... 170
19.1.8. PrintToUart ..... 170
19.1.9. Set Alarm ..... 171
19.1.10. SMS Storage ..... 173
19.1.11. Socket ioctl ..... 174
19.1.12. TCP-Client ..... 175
19.1.13. TCP-Server ..... 177
19.1.14. UDP-Client ..... 179
19.1.15. UDP-Server ..... 181
19.1.16. SSL/TCP Client ..... 182
19.1.17. Get DNS IPv6 addresses ..... 184
19.1.18. M2M_SOCKET_BSD_TCP_CONNTIME Option ..... 185
19.1.19. M2M_SOCKET_BSD_TCP_KEEPALIVE Option ..... 185
19.1.20. MD5 Hash Function ..... 186
19.1.21. Watchdog ..... 187
19.2. Declarations of C Identifiers ..... 188
19.2.1. m2m_cb_app_func C Identifiers ..... 188
19.2.2. m2m_clock_api C Identifiers ..... 188

19.2.3. m2m_fs_api C Identifiers ..... 189
19.2.4. m2m_hw_api C Identifiers ..... 190
19.2.5. m2m_spi_api C Identifiers ..... 191
19.2.6. m2m_i2C_api C Identifiers..... 191
19.2.7. m2m_network_api C Identifiers..... 192
19.2.8. m2m_os_api C Identifiers ..... 194
19.2.9. m2m_os_lock_api C Identifiers ..... 195
19.2.10. m2m_sms_api C Identifiers..... 195
19.2.11. m2m_socket_api C Identifiers..... 196
19.2.12. m2m_ssl_api C Identifiers..... 200
19.2.13. m2m_timer_api C Identifiers ..... 201
19.2.14. m2m_dump_api C Identifiers ..... 201
19.2.15. m2m_sec_api C Identifiers ..... 202
19.2.16. m2m_ipraw_api C Identifiers..... 202
19.2.17. m2m_hw_watchdog_api C Identifiers ..... 202
19.2.18. m2m_type C Identifiers valid for multiple APIs sets..... 202

**20. GLOSSARY AND ACRONYMS ..... 203**

**21. DOCUMENT HISTORY ..... 204**

# Tables List

Tab. 1: 3G GPIO Conflict between Interrupt and HW Timer Functions ..... 38
Tab. 2: 3G GPIO and Interrupts ..... 39
Tab. 3: 4G GPIO Conflict between Interrupt and HW Timer Functions ..... 39
Tab. 4: 4G GPIO and Interrupts ..... 39

# 1. INTRODUCTION

## 1.1. Scope

This document describes the set of the APIs provided by Telit's AppZone software. The user M2M applications can access the features offered by the module through these APIs.

## 1.2. Audience

The present guide is intended for those users who want to develop M2M applications running on the CPU of the Telit's module as an embedded user application.

## 1.3. Contact Information, Support

For general contact, technical support services, technical questions and report documentation errors contact Telit Technical Support at:

- TS-EMEA@telit.com
- TS-AMERICAS@telit.com
- TS-APAC@telit.com

Alternatively, use:

http://www.telit.com/support

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:

http://www.telit.com

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.

Telit appreciates feedback from the users of our information.

## 1.4. Text Conventions


Danger – This information MUST be followed or catastrophic equipment failure or bodily injury may occur.

Caution or Warning – Alerts the user to important points about integrating the module, if these points are not followed, the module and end user equipment may fail or malfunction.

Tip or Information – Provides advice and suggestions that may be useful when integrating the module.

All dates are in ISO 8601 format, i.e. YYYY-MM-DD.

## 1.5. Related Documents

[1] AppZone C User Guide, 1vv0301335

[2] Refer to the document according to the module used:
GE910 Hardware User Guide, 1vv0300962
HE910 Hardware User Guide, 1vv0300925
UE910 Hardware User Guide, 1vv0301012
UL865 Hardware User Guide, 1vv0301050
UE866 hardware user Guide, 1vv0301157
LE910 V2 Hardware User Guide, 1vv0301200
LE866 Hardware Design Guide, 1vv0301355

[3] Refer to the document according to the module used:
AT Commands Reference Guide, 80000ST10025a
Telit 3G Modules AT Commands Reference Guide, 80378ST10091A
LE910 V2 Series AT Command Reference Guide, 80446ST10707A
LE866 Series AT Commands Reference Guide, 80471ST10691A

[4] Refer to the document according to the module used:
Telit 3G Modules Ports Arrangements User Guide, 1vv0300971
GE910 Series Ports Arrangements User Guide, 1vv0301049
LE910 V2, LE910 Cat1 Ports Arrangements User Guide, 1vv0301252
LE866, ME866A1 Ports Arrangements User Guide, 1vv0301469

[6] Telit's Modules Software User Guide, 1vv0300784

# 2. PRELIMINARY INFORMATION

The present User Guide covers the Modules Series listed in the Applicability Table. There are slightly differences about the number of APIs supported by the different Series. To have information on the actual set of APIs concerning the module used, run the Telit Application Development Environment (ADE) tool, and refer to the header files, see document [1].

Use the ADE software version aligned with the software version of the module. The header files included in chapter 19.2 describe only the declarations of C identifiers used by the APIs covered by this document.

Each header file included in the Telit ADE tool - see document [1] - is associated with a chapter of this guide having the same name of the header file. The chapter describes the APIs set defined in the header file.

# 3. M2M_CLOCK_API

Chapter 19.2.2 contains the declarations of the C identifiers used by the APIs set regarding the management of the following features:

- Date/Time
- Alarm
- Epoch Time
- Time Zone/Daylight Saving Time

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Date/Time Section

## 3.1. m2m_rtc_set_date

M2M_T_RTC_RESULT **m2m_rtc_set_date(**M2M_T_RTC_DATE **date)**

*Description*: sets the date.

*Parameters*:
date: allocated data structure filled with the setting date: year/month/day.

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

## 3.2. m2m_rtc_get_date

M2M_T_RTC_RESULT **m2m_rtc_get_date(**M2M_T_RTC_DATE ***date)**

*Description*: gets the date expressed in year/month/day.

*Parameters*:
date: pointer to the allocated data structure.
Output data:
data structure filled with year/month/day

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

## 3.3. m2m_rtc_set_time

M2M_T_RTC_RESULT **m2m_rtc_set_time(**M2M_T_RTC_TIME **time)**

*Description*: sets the time.

*Parameters*:
time: allocated data structure filled with the setting time: hour/minute/second.

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

## 3.4. m2m_rtc_get_time

M2M_T_RTC_RESULT **m2m_rtc_get_time(**M2M_T_RTC_TIME ***time)**

*Description*: gets the time.

*Parameters*:
time: pointer to the allocated data structure.
Output data:

data structure filled with hour/minute/second.

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

## 3.5. m2m_set_timeofday

**INT32 m2m_set_timeofday(struct** M2M_T_RTC_TIMEVAL ***tv, void *tz)**

*Description*: sets the current time expressed as seconds and milliseconds since the epoch (1/1/1970), the time zone, and the Daylight-Saving Time adjustment. Be careful, the setting will be successful only if the date is after 1/1/2000.

*Parameters*:
tv: pointer to the allocated data structure filled with seconds and milliseconds since 1/1/1970;
tz: pointer to the allocated M2M_T_RTC_TIMEZONE data structure filled with the time zone expressed in quarter of hour (range: -47...+48), and the Daylight-Saving Time adjustment (range: 0÷2); tz parameter is a pointer to void for backward compatibility. It is up to the programmer to use **M2M_T_RTC_TIMEZONE** structure or **NULL** value. The function accepts both configurations.

*Return value*:
on success: 0
on failure: -1

*Examples*: 19.1.9 Set Alarm

## 3.6. m2m_get_timeofday

**INT32 m2m_get_timeofday(struct** M2M_T_RTC_TIMEVAL ***tv, void *tz)**

*Description*: gets the current time expressed as seconds and milliseconds since the epoch (1/1/1970), the time zone and the Daylight-Saving Time adjustment.

*Parameters*:
tv: pointer to the allocated data structure;
Output data:
data structures filled with the current time since the epoch, the time zone, and the Daylight-Saving Time adjustment.
tz: pointer to the allocated M2M_T_RTC_TIMEZONE data structure; tz parameter is a pointer to void for backward compatibility. It is up to the programmer to use **M2M_T_RTC_TIMEZONE** structure or **NULL** value. The function accepts both configurations.

*Return value*:
on success: 0
on failure: -1

*Examples: 19.1.9* Set Alarm

# Alarm Section

## 3.7. m2m_rtc_set_alarm

M2M_T_RTC_RESULT **m2m_rtc_set_alarm(**M2M_T_RTC_DATE **date,**
M2M_T_RTC_TIME **time)**

*Description*: sets a wake-up alarm. On the expiration of the alarm timer, the control calls the M2M_onWakeUp(…) callback function contained in the M2M_hwEvents.c file, refer to document [1]. The system supports one alarm at a time.

*Parameters*:
date: allocated data structure filled with year/month/day.
time: allocated data structure filled with hour/minute/second.

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

*Examples: 19.1.9* Set Alarm

## 3.8. m2m_rtc_clear_alarm

M2M_T_RTC_RESULT **m2m_rtc_clear_alarm(INT32 index)**

*Description*: clears a wake-up alarm.

*Parameters*:
index: must be set to 0 because the system supports only one alarm at a time.

*Return value*:
refer to **M2M_T_RTC_RESULT** enum.

# 4. M2M_FS_API

Chapter 19.2.3 contains the declarations of the C identifiers used by the set of APIs regarding the management of M2M File System.

Some parameters of **m2m_fs_xx(…)** functions need full path having drive and separators, for example: "A:\user1\goofy". The table below shows the drives supported by the modules.

| Drives | | Notes |
|---|---|---|
| A: (default) | FLASH_DISK | Maximum 8 files could be simultaneously opened on FLASH volume |
| B: | RAM_DISK | Maximum 2 files could be simultaneously opened on RAM volume |

As described in the next pages, use **m2m_fs_create(...)** API to create a file and its directory, if missing. Here are some examples of paths, valid separators are / and \\.

**m2m_fs_create("B:/temp1/dummy/file1")** /* use B: at the beginning of the string if you do not want to use default drive A: */

**m2m_fs_create("nav/APP/app1.bin")** /* equivalent of "A:\\ nav/APP/app1.bin" */

**m2m_fs_create( "B:/temp2/dummy/file2" )**
**m2m_fs_create( "nav/APP/app2.bin" )**

**m2m_fs_create( "B:/temp3\\file3" )** /* equivalent of "B:\\temp3\\file3" */

**m2m_fs_create( "C:/goofy" )** /* C: doesn't exist, invalid drive */

The table below shows the maximum lengths (characters) of the file names, directory names and full paths.

| max length of the file name | max length of the directory name | max length of the full path including characters as A:\\ |
|---|---|---|
| 63 | 63 | 128 |

## ➢ 4G: Platform Version ID 23

The drives supported by the modules with Platform Version ID 23 are:

| Drives | |
|---|---|
| /home/appzone (default) | FLASH_DISK |
| /tmp | RAM_DISK |

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# File System Section

## 4.1. m2m_fs_open

M2M_T_FS_HANDLE **m2m_fs_open(CHAR *filename, INT32 mode)**

*Description*: opens the file using the given path/filename, and mode. It returns the file handle. The function does not create the required directory if missing. See **m2m_fs_create(...)** to create the directories.

*Parameters*:
filename: pointer to the zero-terminated string containing the file name and path if it is already existing.
mode: open mode, see table below:

| OPEN MODE | INITIAL CONDITIONS:<br>The file must exist, if it does not exist → Error: M2M_F_ERR_NOTFOUND |
|---|---|
| **M2M_FS_OPEN_READ** | - For Reading:<br>- After opening, the reading starts from the first character of the file.<br>- File data pointer can be moved to perform reading from the selected file position.<br>The file data pointer must not point to the last file character, if reading is tried → Error: M2M_F_ERR_ UNKNOWN.<br><br>- No Writing:<br>- If writing is tried → Error: M2M_F_ERR_NOTOPEN |
| **M2M_FS_OPEN_MODIFY** | - For Reading:<br>- After opening, the reading starts from the first character of the file.<br>- File data pointer can be moved to perform reading from the selected file position.<br>The file data pointer must not point to the last file character, if reading is tried → Error: M2M_F_ERR_ UNKNOWN<br><br>- For Writing:<br>- After opening, the writing start from the beginning of the file. The writing puts new characters on the old ones.<br>- File data pointer can be moved to perform writing on existing characters starting from the selected position. The writing is not limited; consequently, the file size can be incremented. |

| OPEN MODE | INITIAL CONDITIONS:<br>Creates the file if it does not exist. If the file exists, appends or puts new characters on the old ones |
|---|---|
| **M2M_FS_OPEN_APPEND** | - For Writing:<br>- After opening, the writing starts from the end of the file (self-seek to the **EOF**).<br>- File data pointer can be moved to perform writing on existing characters starting from the selected position. The writing is not limited; consequently, the file size can be incremented.<br><br>- No Reading:<br>- If reading is tried →Error: M2M_F_ERR_NOTOPEN |

| OPEN MODE | INITIAL CONDITIONS:<br>Creates the file if it does not exist. If the file exists, truncates it to '0' bytes. |
|---|---|
| **M2M_FS_OPEN_WRITE**<br>**M2M_FS_OPEN_CREATE** | - For Writing:<br>- After opening, allows consecutive writings in append mode (self-seek to the **EOF**) starting from the beginning of the file.<br>- File data pointer can be moved to perform writing on existing characters starting from the selected position. The writing is not limited; consequently, the file size can be incremented.<br><br>- No Reading:<br>- If reading is tried →Error: M2M_F_ERR_NOTOPEN, regardless of the position of the file data pointer. |
| **M2M_FS_OPEN_TRUNCATE** | - For Writing:<br>- After opening, allows consecutive writings in append mode (self-seek to the **EOF**) starting from the beginning of the file.<br>- File data pointer can be moved to perform writing on existing characters starting from the selected position. The writing is not limited; consequently, the file size can be incremented.<br><br>- For Reading:<br>- File data pointer can be moved to perform reading. The file data pointer must not point to the last file character. If reading is tried → Error M2M_F_ERR_ UNKNOWN |

*Return value*:
on success: pointer to the file handle
on failure: **NULL**

**NOTE**: m2m_fs_last_error function returns the failure reason.

*Examples*: 19.1.1 Open and Close

## 4.2. m2m_fs_close

M2M_API_RESULT **m2m_fs_close(**M2M_T_FS_HANDLE **filehandle)**

*Description*: closes a previously opened file.

*Parameters*:
filehandle: file handle

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

*Examples*: 19.1.1 Open and Close

## 4.3. m2m_fs_create

M2M_API_RESULT **m2m_fs_create(CHAR *filename)**

*Description*: creates the file using the given full path and filename. It performs the following activities:

- Creates the file if it does not exist and the required directory if missing.
- Truncates the file to 0 bytes if the file exists
- Closes the just opened file containing 0 bytes.

*Parameters*:
filename: pointer to the zero-terminated string containing the full path of the file that you are creating.

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: **m2m_fs_last_error** function returns the failure reason.

## 4.4. m2m_fs_write

**UINT32 m2m_fs_write(**M2M_T_FS_HANDLE **filehandle, CHAR *data, UINT32 length)**

*Description*: writes data into a file opened in suitable mode, refer to **m2m_fs_open(...)**.

*Parameters*:
filehandle: file handle
data: pointer to the allocated buffer containing the characters to be written in the file starting from the current file data pointer.
length: number of characters to be written. If two consecutive writings are performed, the last one starts from the file data pointer left by the first writing plus 1.

*Return value*:
on success: the number of characters written is equal to length value.
on failure: if the number of character written is less than the length value, an error could be happened. In this case, call the **m2m_fs_last_error(...)** API.

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.5. m2m_fs_read

**UINT32 m2m_fs_read(**M2M_T_FS_HANDLE **filehandle, CHAR *buf, UINT32 length)**

*Description*: reads data from an already opened in suitable mode, refer to **m2m_fs_open (...)**.

*Parameters*:
filehandle: file handle

buf: pointer to the allocated buffer that will be filled with characters read from the current file data pointer.
Output data:
on success: the buffer contains the characters read from the file.

length: number of characters to be read from the file. If two consecutive readings are performed, the last one starts from the file data pointer left by the first reading plus 1. If the length parameter forces the reading over the size of the file, the reading stops automatically when the **EOF** is reached, (**EOF** = -1).

*Return value*:
on success: the number of characters read is equal to length value.
on failure: if the number of character read is less than the length value, an error could be happened. In this case, call the **m2m_fs_last_error(...)** API.

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.6. m2m_fs_delete

M2M_API_RESULT **m2m_fs_delete(CHAR *filename)**

*Description*: deletes the selected file. You must close the file before calling the function.

*Parameters*:
filename: pointer to the zero-terminated string containing the file name

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: **m2m_fs_last_error** function returns the failure reason.

## 4.7. m2m_fs_clear

M2M_API_RESULT **m2m_fs_clear(void)**

*Description*: saves the M2M application that is currently running, and completely formats the drive A.

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.8. m2m_fs_copy

M2M_API_RESULT **m2m_fs_copy(CHAR *srcfilename, CHAR *dstfilename)**

*Description*: copies a file. You must close the file and use the full path.

*Parameters*:
srcfilename: pointer to the zero-terminated string containing the full path of the file to be copied
dstfilename: pointer to the zero-terminated string containing the full path of the new file

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.9. m2m_fs_find_first

M2M_API_RESULT **m2m_fs_find_first(CHAR *filename_buffer, CHAR *file_spec)**

*Description*: finds the first file or directory having the name matching the provided pattern. To find the next files or directories matching the pattern, you must use the m2m_fs_find_next function.

*Parameters*:
filename_buffer: pointer to the allocated buffer that will be filled with the file name found.
Output data:
The buffer contains the first directory or file name matching the pattern. The format is shown in the table below:

| | | | | |
|---|---|---|---|---|
| < directory name> | '\0' | | | the directory name is closed in angle brackets |
| (system file name) | '\t' | file size in bytes | '\0' | the system file name is closed in parentheses |
| user file name | '\t' | file size in bytes | '\0' | the user file name is not closed in parentheses |

file_spec: pointer to the zero-terminated string containing the pattern you want to find. It can include full path with drive and separators, example: "A:\MOD\m2mapz.bin".

*Return value*:
refer to **M2M_API_RESULT** enum

*Examples*: 19.1.2 Listing all Files

## 4.10. m2m_fs_find_next

M2M_API_RESULT **m2m_fs_find_next(CHAR *filename_buffer)**

*Description*: finds the next files or directories having the file name matching the pattern provided by the m2m_fs_find_first function.

*Parameters:*
filename_buffer: pointer to the allocated buffer that will be filled with the file name found.
Output data:

if existing, the buffer contains the directories and file names. The format is shown in the table below:

| | | | | |
|---|---|---|---|---|
| < directory name> | '\0' | | | the directory name is closed in angle brackets |
| (system file name) | '\t' | file size in bytes | '\0' | the system file name is closed in parentheses |
| user file name | '\t' | file size in bytes | '\0' | the user file name is not closed in parentheses |

*Return value*:
refer to **M2M_API_RESULT** enum

*Examples*: 19.1.2 Listing all Files

## 4.11. m2m_fs_rename

M2M_API_RESULT **m2m_fs_rename(CHAR *oldfilename, CHAR *newfilename)**

*Description*: renames a file. Close the file before attempting to rename it.

*Parameters*:
oldfilename: pointer to the zero-terminated string containing the full path of the file to be renamed.
newfilename: pointer to the zero-terminated string containing only the new file name. Do not use the full path.

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.12. m2m_fs_get_size

**UINT32 m2m_fs_get_size(CHAR *filename)**

*Description*: returns the size of the selected file identified by name. Be careful: it returns the size of the saved counterpart of the file. If the file size is changed, the result of the function is not updated until the file has been saved (closed). See **m2m_fs_get_size_with_handle(...)**.

*Parameters*:
filename: pointer to the zero-terminated string containing the name of the file.

*Return value*:
on success: file size in bytes
on failure: if the return value is zero, the file size is zero or an error could be happened. In this case, call the **m2m_fs_last_error(...)** API.

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.13. m2m_fs_get_size_with_handle

**UINT32 m2m_fs_get_size_with_handle(**M2M_T_FS_HANDLE **filehandle)**

*Description*: returns the size of a file identified by the handle. It moves the data pointer at the beginning of the file, calculates the file size, and places again the data pointer at the beginning of the file. The function returns the current size of the file even if it has not been saved. See **m2m_fs_get_size(...)**.

*Parameters*:
filehandle: file handle

*Return value*:
on success: file size in bytes
on failure: **M2M_FS_ERROR**

**NOTE**: **m2m_fs_last_error** function returns the failure reason.

## 4.14. m2m_fs_tell

**UINT32 m2m_fs_tell(**M2M_T_FS_HANDLE **filehandle)**

*Description*: returns the current data pointer to write or read in/from the file.

*Parameters*:
filehandle: file handle

*Return value*:
on success: the current data pointer.
on failure: if the current data pointer is equal to zero it could be right, the pointer is at the beginning of the file. In any case, to verify if an error is happened call the **m2m_fs_last_error(...)** API.

**NOTE**: **m2m_fs_last_error** function returns the failure reason.

## 4.15. m2m_fs_seek

**UINT32 m2m_fs_seek(**M2M_T_FS_HANDLE **filehandle, UINT32 offset)**

*Description*: moves the current data pointer to the new data pointer (offset value). The data writing/reading starts from the offset+1 position. If the offset value is greater than the current file size, accordingly the file size becomes larger. See also **m2m_fs_ tell(...)** API.

*Parameters*:
filehandle: file handle
offset: new current data pointer

*Return value*:
on success: new current data pointer
on failure: if the return value is zero, the offset value is zero or an error could be happened. In this case, call the **m2m_fs_last_error(...)** API.

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.16. m2m_fs_truncate

M2M_T_FS_HANDLE **m2m_fs_truncate(CHAR *filename, UINT32 new_size)**

*Description*: truncates the file by discarding the needed last bytes to obtain the new size. The truncation will be effective when you close the file after truncation.

*Parameters*:
filename: pointer to the zero-terminated string containing the file name
new_size: new file size

*Return value*:
on success: pointer to the file handle
on failure: **NULL**

**NOTE**: m2m_fs_last_error function returns the failure reason.

```
Example:

M2M_T_FS_HANDLE file_handle = m2m_fs_truncate( filename, 10 );

if ( file_handle )
{
  m2m_fs_close(file_handle );        /* after the closing the file is effectively truncated  */
}
```

## 4.17. m2m_fs_getc

**CHAR m2m_fs_getc(**M2M_T_FS_HANDLE **filehandle)**

*Description*: reads from an open file the character pointed by the current file data pointer plus 1.

*Parameters*:
filehandle: file handle

*Return value*:
on success: the character read
on failure: **EOF** (**E**nd **O**f **F**ile = -1)

## 4.18. m2m_fs_gets

**CHAR *m2m_fs_gets(CHAR *buf, INT32 length,** M2M_T_FS_HANDLE **filehandle)**

*Description*: reads a character string from an open file. It reads starting from the character pointed by the current file data pointer plus 1 until finds a new line (\r\n) or the **EOF** (-1), if the length value is not exceeded.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the characters read from the file.

Output data:
on success: the buffer contains the characters string read from the file. The string is a zero-terminated string.

length: number of characters to be read from the file. The number must be less than the file size

filehandle: file handle

*Return value*:
on success: buf points to the allocated buffer
on failure: **NULL**

## 4.19. m2m_fs_set_exec_permission

M2M_API_RESULT **m2m_fs_set_exec_permission(CHAR *filename)**

*Description*: has the same feature of the <permission> parameter of the AT#M2MWRITE AT command, refer to document [1]. You can assign to the selected file the executable permission. The filename parameter must be without path, and the file must be stored in \MOD directory.

*Parameters*:
filename: pointer to the zero-terminated string containing only the file name.

*Return value*:
refer to **M2M_API_RESULT** enum

## 4.20. m2m_fs_set_run_permission

M2M_API_RESULT **m2m_fs_set_run_permission(**
M2M_T_FS_RUN_PERM_MODE_TYPE **mode, CHAR *filename)**

*Description*: has the same feature of the <mode> parameter of the AT#M2MRUN AT command, refer to document [1]. You can assign to one file with executable permission, the RUN permission to enable its running. The filename parameter must be without path, and the file must be stored in \MOD directory.

*Parameters*:
mode: refer to M2M_T_FS_RUN_PERM_MODE_TYPE enum.
**M2M_F_RUN_PERM_MODE_SET** value is not supported.
filename: pointer to the zero-terminated string containing only the file name.

*Return value*:
refer to **M2M_API_RESULT** enum

## 4.21. m2m_fs_mk_dir

M2M_API_RESULT **m2m_fs_mk_dir(CHAR *path)**

*Description*: creates a directory entry. You must use the full path.

*Parameters*:

path: pointer to the zero-terminated string containing the full path including the name of the directory to be created.

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.22. m2m_fs_rename_dir

M2M_API_RESULT **m2m_fs_rename_dir(CHAR *oldpath, CHAR *newdirname)**

*Description*: renames a directory entry.

*Parameters*:
oldpath: pointer to the zero-terminated string containing the full path of the directory to be renamed.
newdirname: pointer to the zero-terminated string containing only the new directory name. Do not use the full path.

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.23. m2m_fs_rm_dir

M2M_API_RESULT **m2m_fs_rm_dir(CHAR *path)**

*Description*: deletes a directory entry. The full path must be used, and directory must be empty.

*Parameters*:
path: pointer to the zero-terminated string containing the full path including the directory to be deleted

*Return value*:
refer to M2M_API_RESULT enum

**NOTE**: m2m_fs_last_error function returns the failure reason.

## 4.24. m2m_fs_get_free_space

**UINT32 m2m_fs_get_free_space(void)**

*Description*: returns the available space in the global file system expressed in bytes.

## 4.25. m2m_fs_get_nof_files

**INT32 m2m_fs_get_nof_files(void)**

*Description*: returns the number of files in the global file system.

## 4.26. m2m_fs_last_error

M2M_T_FS_ERROR_TYPE **m2m_fs_last_error(void)**

*Description*: returns the error code of the last file operation.

*Return value*:
refer to **M2M_T_FS_ERROR_TYPE** enum

# 5. M2M_HW_API

Chapter 19.2.4 contains the declarations of the C identifiers used by the APIs set regarding the management of the following devices:

- GPIO
- I2C
- HW timers
- UART
- USB

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2

# GPIO Section

In general, the APIs use the same module resources to manage GPIO, Hwtimer, msTimerCounter. Therefore, the developer must avoid employing APIs that use simultaneously the same resource. Refer to document [2] to get hardware information, as for example, GPIO port numbers. Here is the information to avoid the possible configuration conflicts.

## ➢ 2G: Platform Version ID 13

No configuration conflicts.

## ➢ 3G: Platform Version ID 12

Tab. 1 shows on which GPIO can occur a configuration conflict between interrupt and the HW Timer function **m2m_hw_timer_start()**. For example, consider the HE910 series:

- if GPIO_01 is configured to generate an interrupt, HW Timer having timer_id=2 must not be used.
- if GPIO_02 is configured to generate an interrupt, HW Timer having timer_id=3 must not be used. And so on, as shown by the table below.
- GPIO_05 does not support interrupt. And so on, as shown by the table below.

| | 3G | | | |
|---|---|---|---|---|
| **GPIO** | **HE910** | **UE910** | **UL865** | **UE866** |
| GPIO_01 | T2 | T2 | - | - |
| GPIO_02 | T3 | T3 | - | - |
| GPIO_03 | T4 | T4 | - | - |
| GPIO_04 | T5 | T5 | - | - |
| GPIO_05 | - | - | - | - |
| GPIO_06 | T6 | T6 | T6 | T2 |
| GPIO_07 | T1 | T1 | T5 | T3 |
| GPIO_08 | no conflict | no conflict | see (*) | - |
| GPIO_09 | no conflict | no conflict | - | - |
| GPIO_10 | - | - | - | - |

Tab. 1: 3G GPIO Conflict between Interrupt and HW Timer Functions

(*): if GPIO_08 is configured to generate an interrupt, **m2m_hw_set_ms_count()**, and **m2m_hw_get_ms_count()** must be not used, and conversely.

With reference to the GPIOs interrupt APIs described in the following chapters, Tab. 2 shows which are the GPIOs that can generate an interrupt. These GPIOs are marked with "•".

| | 3G | | | |
|---|---|---|---|---|
| **GPIO** | **HE910** | **UE910** | **UL865** | **UE866** |
| GPIO_01 | • | • | - | - |
| GPIO_02 | • | • | - | - |
| GPIO_03 | • | • | - | - |
| GPIO_04 | • | • | - | - |
| GPIO_05 | - | - | - | - |
| GPIO_06 | • | • | • | • |
| GPIO_07 | • | • | • | • |
| GPIO_08 | • | • | • | - |
| GPIO_09 | • | • | - | - |
| GPIO_10 | - | - | - | - |

Tab. 2: 3G GPIO and Interrupts

➢ 4G: Platform Version ID 20, 23

| | 4G | | | |
|---|---|---|---|---|
| **GPIO** | **LE910 Cat1** | **LE910 V2** | **LE866/ME866A1** | **LE910D1** |
| GPIO_01 | T2 | T2 | no conflict | no conflict |
| GPIO_02 | T3 | T3 | no conflict | no conflict |
| GPIO_03 | T4 | T4 | no conflict | no conflict |
| GPIO_04 | T5 | T5 | no conflict | no conflict |
| GPIO_05 | - | - | no conflict | no conflict |
| GPIO_06 | T6 | T6 | no conflict | no conflict |
| GPIO_07 | T1 | T1 | no conflict | no conflict |
| GPIO_08 | - | - | - | no conflict |
| GPIO_09 | no conflict | no conflict | - | no conflict |
| GPIO_10 | - | - | - | no conflict |

Tab. 3: 4G GPIO Conflict between Interrupt and HW Timer Functions

| | 4G | | | |
|---|---|---|---|---|
| **GPIO** | **LE910 V2** | **LE910 V2** | **LE866/ ME866A1** | **LE910D1** |
| GPIO_01 | • | • | • | • |
| GPIO_02 | • | • | • | • |
| GPIO_03 | • | • | • | • |
| GPIO_04 | • | • | • | • |
| GPIO_05 | - | - | • | • |
| GPIO_06 | • | • | • | • |
| GPIO_07 | • | • | • | • |
| GPIO_08 | - | - | - | • |
| GPIO_09 | • | • | - | • |
| GPIO_10 | - | - | - | • |

Tab. 4: 4G GPIO and Interrupts

## 5.1. m2m_hw_gpio_read

**INT32 m2m_hw_gpio_read (INT32 io)**

*Description*: reads the status of the selected GPIO port. Call the **m2m_hw_gpio_conf** function to set the GPIO in input direction.

*Parameters*:
io: GPIO port number

*Return value*:
on success: 1 = high, 0 = low
on failure: -1

*Examples*: 19.1.3 GPIO

## 5.2. m2m_hw_gpio_write

**INT32 m2m_hw_gpio_write (INT32 io, INT32 val)**

*Description*: sets automatically the selected GPIO port in output direction, then the GPIO output is set to "val" value.

*Parameters*:
io: GPIO port number
val: 1 = high, 0 = low

*Return value*:
on success: 1
on failure: 0

*Examples*: 19.1.3 GPIO

## 5.3. m2m_hw_gpio_conf

**INT32 m2m_hw_gpio_conf (INT32 io, INT32 dir)**

*Description*: sets the selected GPIO port direction. It is mandatory to call this API before using the m2m_hw_gpio_read function.

*Parameters*:
io: GPIO port number
dir: 1 = output, 0 = input

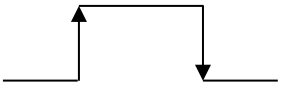
*Return value*:
on success: 1
on failure: 0

## 5.4. m2m_hw_gpio_int_enable

**void m2m_hw_gpio_int_enable(INT32 io)**

*Description*: enables the selected GPIO port to generate interrupts. When interrupt occurs, the control calls the M2M_onInterrupt(…) application callback function contained in the M2M_hwEvents.c file, refer to document [1]

The interrupts are generated on both edges:



*Parameters*:
io: GPIO port number

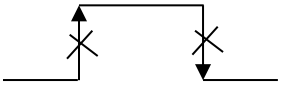
**NOTE**: see tables at the beginning of the present section to have information on the GPIOs that can generate interrupts.

## 5.5. m2m_hw_gpio_int_disable

**void m2m_hw_gpio_int_disable(INT32 io)**

*Description*: disables the capability of the selected GPIO port to generate interrupts.

The interrupts are disabled on both edges:



*Parameters*:
io: GPIO port number

**NOTE**: see tables at the beginning of the present section to have information on the GPIOs that can generate interrupts.

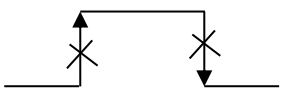
## 5.6. m2m_hw_gpio_int_enable_on_front

**void m2m_hw_gpio_int_enable_on_front(INT32 io,** M2M_INT_FRONT **front)**

*Description*: specifies which edge of the selected GPIO triggers the interrupt. It comprises also the functionalities of the following functions:

- **m2m_hw_gpio_int_enable(...)**
- **m2m_hw_gpio_int_disable(...)**.

*Parameters*:
io: GPIO port number
front: specifies the GPIO edge that trigger the interrupt, refer to **M2M_INT_FRONT** enum



**M2M_NO_EDGE**

**M2M_RISING_EDGE**

**M2M_FALLING_EDGE**

**M2M_BOTH_EDGES**

**NOTE**: see tables at the beginning of the present section to have information on the GPIOs that can generate interrupts.

## 5.7. m2m_hw_gpio_int_trigger_bms_sem

M2M_API_RESULT **m2m_hw_gpio_int_trigger_bms_sem(INT32 io,** M2M_INT_FRONT **front, void *sema)**

*Description*: is used to unlock binary msSemaphore (bms_sem) when interrupt occurs on selected GPIO.

*Parameters*:
io: GPIO port number.
front: specifies the GPIO edge that trigger the interrupt, refer to **M2M_INT_FRONT** enum. To disable the interrupt, the front must be set to **M2M_NO_EDGE**.
sema: binary msSemaphore handle.

*Return value*:
refer to **M2M_API_RESULT** enum. On failure, the msSemaphore is not unlocked.

**NOTE**: see tables at the beginning of the present section to have information on the GPIOs that can generate interrupts.

➢ 4G: Platform Version ID 23

For the products with Platform Version ID 23, after the call of this API, at most 1 second should be waited before the trigger is set up.

# HW Timer Section

## 5.8. m2m_hw_timer_start

M2M_API_RESULT **m2m_hw_timer_start(INT32 timer_id, UINT32 span)**

*Description*: starts the selected HW timer. On timer expiration, the control calls the M2M_onHWTimer(…) application callback function contained in the M2M_hwEvents.c file, refer to document [1].

*Parameters*:
timer_id: 1÷7.
span: 1÷3000, unit 100us.

*Return value*:
refer to **M2M_API_RESULT** enum. On failure, the timer is not started.

**NOTE**: if you use a timer_id already started and still running, the call fails. Only in this case, to avoid the call failure, use **m2m_hw_timer_stop(timer_id)** before calling **m2m_hw_timer_start(…)**.

**NOTE**: see tables at the beginning of the present section to avoid configuration conflict between interrupt and HW Timer function.

*Examples*: 19.1.7 HW Timer

## 5.9. m2m_hw_timer_stop

**void m2m_hw_timer_stop(INT32 timer_id)**

*Description*: stops the selected HW timer.

*Parameters*:
timer_id: 1÷7

*Examples*: 19.1.7 HW Timer

## 5.10. m2m_hw_timer_state

**UINT8 m2m_hw_timer_state(void)**

*Description*: shows the activities of the HW timers.

*Return value*:
the bit value assigned to the HW timer is one if the timer is running, zero if stopped. Refer to the table below:

| Bits of return value | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| HW timer Identifier | / | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Running/stopped | / | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |

## 5.11. m2m_hw_set_ms_count

M2M_API_RESULT **m2m_hw_set_ms_count(INT32 on_off)**

*Description*: starts/stops/resets the counter that increments by 1 every msec.

*Parameters*:
on_off: 1 starts the counter;
0 stops and resets the counter.

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.12. m2m_hw_get_ms_count

M2M_API_RESULT **m2m_hw_get_ms_count(UNT32 *m_secs)**

*Description*: returns the current counter value.

*Parameters*:
m_secs: pointer to the allocated variable that will be filled with the current counter value.
Output data:
on success: m_secs points to the current counter value.

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.13. m2m_hw_uart_open

M2M_T_HW_UART_HANDLE **m2m_hw_uart_open(void)**

*Description*: opens the Main Port (USIF0) and returns the related handle. Any time you call the function, the same handle is returned. For USIF0 port information see document [2].

*Return value*:
on success: Main Port handle
on failure: **M2M_HW_UART_HANDLE_INVALID**

## 5.14. m2m_hw_uart_aux_open

M2M_T_HW_UART_HANDLE **m2m_hw_uart_aux_open(void)**

*Description*: opens the Auxiliary Port (USIF1) and returns the related handle. Any time you call the function, the same handle is returned. The Auxiliary Port is managed by means of the Main Port (USIF0) APIs using the handle returned by the **m2m_hw_uart_ aux_open()** function.

Return value:
on success: Auxiliary Port handle
on failure: **M2M_HW_UART_HANDLE_INVALID**

- 2G: Platform Version ID 13

The Auxiliary Port (USIF1) can be opened only when the module is using one of the following #PORTCFG variants: 1,4 refer to document [4]. The Auxiliary Port does not provide flow control, see document [2].

- 3G: Platform Version ID 12

The Auxiliary Port (USIF1) can be opened only when the module is using one of the following #PORTCFG variants: 0,3,7,8,9,11, refer to document [4]. The Auxiliary port does not provide flow control, see document [2].

- 4G: Platform Version ID 20, 23

The Auxiliary Port (USIF1) can be opened only when the module is using one the following #PORTCFG variant: 3, refer to documents [4]. The Auxiliary Port does not provide flow control, see document [2].

## 5.15. m2m_hw_uart_close

M2M_T_HW_UART_RESULT **m2m_hw_uart_close(**M2M_T_HW_UART_HANDLE **handle)**

*Description*: closes the port identified by the handle. The logical connection defined by the #PORTCFG variant, set before the user application execution, is not restored. AppZone Layer does not release the resource even if the port is closed, see also **m2m_uart_ close_hwch(…)**. For #PORTCFG configuration see documents [4].
*Parameters*:

handle: port handle

*Return value*:
refer to **M2M_T_HW_UART_RESULT** enum

## 5.16. m2m_uart_close_hwch

M2M_API_RESULT **m2m_uart_close_hwch(**M2M_T_HW_UART_HANDLE ***handle)**

*Description*: closes the port identified by the handle. The logical connection defined by the #PORTCFG variant, set before the user application execution, is restored. AppZone Layer releases the closed port, see also **m2m_uart_close(…)**. For #PORTCFG configuration see documents [4].

*Parameters*:
handle: pointer to the port handle

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.17. m2m_hw_uart_read

M2M_T_HW_UART_RESULT **m2m_hw_uart_read(**M2M_T_HW_UART_HANDLE **handle,CHAR *buffer, INT32 len, INT32 *len_read)**

*Description*: receives data from Main or Auxiliary Port according to the used handle. The receiving port mode is set by the function **m2m_hw_uart_ioctl(…)**.

*Parameters*:
handle: port handle
buffer: pointer to the allocated buffer that will be filled with received data. Output data.
len: number of characters to be read.
len_read: pointer to the allocated variable that will be filled with the number of data read (bytes). Output data.

*Return value*:
refer to **M2M_T_HW_UART_RESULT** enum

## 5.18. m2m_hw_uart_write

M2M_T_HW_UART_RESULT **m2m_hw_uart_write(**M2M_T_HW_UART_HANDLE **handle, CHAR *buffer, INT32 len, INT32 *len_sent)**

*Description*: sends data to the Main or Auxiliary Port according to the used handle. The sending port mode is set by the function **m2m_hw_uart_ioctl(…)**.

*Parameters*:
handle: port handle.
buffer: pointer to the allocated buffer containing the data to be sent.
len: number of characters to be sent.

len_sent: pointer to the allocated variable that will be filled with the number of data sent (bytes). Output data

*Return value*:
refer to **M2M_T_HW_UART_RESULT** enum

*Examples*: 19.1.8 PrintToUart

## 5.19. m2m_hw_uart_ioctl

M2M_T_HW_UART_RESULT **m2m_hw_uart_ioctl(**M2M_T_HW_UART_HANDLE **handle, INT32 arg, INT32 value)**

*Description*: sets the port mode.

*Parameters*:
handle: port handle.
arg: selector of the port mode, refer to the tables below.
value: option related to the selected port mode, refer to the tables below.

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_IO_BLOCKING_SET**<br>valid for TX and RX | **M2M_HW_UART_IO_BLOCKING_ON**: sets blocking to ON, default mode.<br><br>TX: The API returns the control when "len" bytes have been moved to the UART transmitter buffer. "len_sent" points to the number of bytes moved, and it is equal to "len".<br><br>Note: the task that uses this mode could be blocked if UART transmitter buffer has no more room until RTS line of the host is not ready (hardware flow control).<br><br>RX: The API returns the control when "len" or more bytes have been received. "len_read" is the number of bytes moved into the buffer, it is equal to "len".<br>Bytes exceeding "len" remain in the UART receiver buffer. You can delete them using the following function:<br>**m2m_hw_uart_ioctl (handle, M2M_UART_CLEAR_RX, (INT32) M2M_HW_UART_IO_NO_ARG)** |
| | **M2M_HW_UART_IO_BLOCKING_OFF**: sets blocking to OFF<br><br>TX: The API tries to move "len" bytes in UART transmitter buffer, in accordance with the available resources. In any case, returns immediately the control. "len_sent" points to the number of bytes moved into UART transmitter buffer<br><br>RX: The API collects "len" bytes if they are available in the UART receiver buffer, and returns immediately the control. "len_read" is the number of available bytes moved from UART receiver buffer into the allocated buffer pointed by "buffer". |
| | **M2M_HW_UART_IO_BLOCKING_RELEASE**: releases both pending RX and TX activities, and set blocking to OFF. The new mode is blocking OFF |

Example **M2M_HW_UART_IO_BLOCKING_ON**:

```
/* Set the next RX and TX activities in blocking ON */
m2m_hw_uart_ioctl (fd, M2M_HW_UART_IO_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_ON);

/* Start RX and TX activities */
m2m_hw_uart_read (fd, buffer_rx, len_rx, &len_read);
m2m_hw_uart_write (fd, buffer_tx, len_tx, &len_sent);
```

Example **M2M_HW_UART_IO_BLOCKING_OFF**:

```
/* Set the next RX and TX activities in blocking OFF */
m2m_hw_uart_ioctl (fd, M2M_HW_UART_IO_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_OFF);

/* Start RX and TX activities */
m2m_hw_uart_read (fd, buffer_rx, len_rx, &len_read);
m2m_hw_uart_write (fd, buffer_tx, len_tx, &len_sent);
```

Example **M2M_HW_UART_IO_BLOCKING_RELEASE**:

**Task N**:

```
.....
/* Set RX and TX activities in blocking ON mode */
m2m_hw_uart_ioctl (fd, M2M_HW_UART_IO_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_ON);
.....
/* Start RX and TX activities */
m2m_hw_uart_read (fd, buffer_rx, len_rx, &len_read);
m2m_hw_uart_write (fd, buffer_tx, len_tx, &len_sent);
.....
```

**Task M**:

```
/* RX and TX pending activities are released */
.....
m2m_hw_uart_ioctl (fd, M2M_HW_UART_IO_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_RELEASE);
.....
.....
```

**Task N**:

```
/* After releasing, start again RX and TX activities. Now, they are in blocking OFF mode  */
m2m_hw_uart_read (fd, buffer_read, len, &len_read);
m2m_hw_uart_write (fd, buffer_write, len, &len_sent);
.....
```

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_RX_BLOCKING_SET** valid only for RX | **M2M_HW_UART_IO_BLOCKING_ON**: sets blocking to ON only for RX activity, default. |
| | **M2M_HW_UART_IO_BLOCKING_OFF**: sets blocking to OFF only for RX activity. |
| | **M2M_HW_UART_IO_BLOCKING_RELEASE** releases RX pending activity if any, and lets unchanged the blocking mode. |

Example **M2M_HW_UART_IO_BLOCKING_ON**:

```
/* Set the next RX activity in blocking OFF */
m2m_hw_uart_ioctl (fd, M2M_HW_UART_RX_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_OFF);

/* Start RX activity */
m2m_hw_uart_read (fd, buffer_rx, len_rx, &len_read);
```

Example **M2M_HW_UART_IO_BLOCKING_RELEASE**:

**Task N**:

```
.....
/* Set RX activity in blocking ON mode */
m2m_hw_uart_ioctl (fd, M2M_HW_UART_RX_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_ON);
.....
/* Start RX activity */
m2m_hw_uart_read (fd, buffer_read, len, &len_read);
.....
```

**Task M**:

```
/* RX pending activity is released */
.....
m2m_hw_uart_ioctl (fd, M2M_HW_UART_RX_BLOCKING_SET, (INT32) M2M_HW_UART_IO_BLOCKING_RELEASE);
.....
.....
```

**Task N**:

```
/* After releasing, start again RX activities. RX blocking mode remains unchanged, therfore –in this example– remains in
blocking ON mode  */
m2m_hw_uart_read (fd, buffer_read, len, &len_read);
.....
```

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_TX_BLOCKING_SET** valid only for TX | **M2M_HW_UART_IO_BLOCKING_ON**: sets blocking to ON only for TX activity, default. |
| | **M2M_HW_UART_IO_BLOCKING_OFF**: sets blocking to OFF only for TX activity |
| | **M2M_HW_UART_IO_BLOCKING_RELEASE**: releases TX pending activity if any, and lets unchanged the blocking mode. |

Refer to the previous **M2M_HW_UART_RX_BLOCKING_SET** examples.

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_IO_AT_MODE_SET** Set AT command mode | **M2M_HW_UART_IO_AT_MODE_OFF**: default option. It routes the data received from the UART to the user AppZone application as they are, see document [4]. |
| | **M2M_HW_UART_IO_AT_MODE_ON**: this option routes the data received from the UART to AT1 parser by means of AZ1 logical port, see document [4]. In addition, it sets RX in blocking OFF, and TX blocking mode is unchanged. |

Example **M2M_HW_UART_IO_AT_MODE_SET**:

```
void M2M_main(…)
{
  ......
 /* It is not mandatory to set the RX and TX blocking mode to OFF */
 m2m_hw_uart_ioctl( uart_fd, M2M_HW_UART_IO_BLOCKING_SET, M2M_HW_UART_IO_BLOCKING_OFF );

 /* Sets RX in blocking OFF, and TX blocking mode is unchanged */
 m2m_hw_uart_ioctl (uart_fd, M2M_HW_UART_IO_AT_MODE_SET, M2M_HW_UART_IO_AT_MODE_ON);
 ......
}

/* The results of the entered AT commands are managed by the M2M_onReceiveResultCmd(…) callback function
    contained in the M2M_atRsp.c file, refer to document [1] */
INT32 M2M_onReceiveResultCmd (…)
{
.
}
```

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_CLEAR_RX**<br>Clear the RX buffer of the UART channel | **M2M_HW_UART_IO_NO_ARG:** the m2m_hw_uart_ioctl function uses an "arg" parameter that does not need options. |

Example **M2M_HW_UART_CLEAR_RX**:

```
m2m_hw_uart_ioctl (handle, M2M_HW_UART_CLEAR_RX, (INT32) M2M_HW_UART_IO_NO_ARG);
```

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_IO_RCV_FUNC**<br>Indicates that the **value** parameter is a callback to manage received data | static INT32 **hw_uart_read_cb** ( M2M_T_HW_UART_HANDLE handle, CHAR *buffer, INT32 len ) |

Example **M2M_HW_UART_IO_RCV_FUNC**:

```
static INT32 hw_uart_read_cb ( M2M_T_HW_UART_HANDLE handle, CHAR *buffer, INT32 len )
{
  CHAR serRxStr[128];

  strncpy( serRxStr, buffer, len );
  serRxStr[len]= 0;
  PRINT(serRxStr);

  return M2M_HW_UART_RESULT_SUCCESS;
}

void M2M_main(…)
{
 .
 m2m_hw_uart_ioctl (handle, M2M_HW_UART_IO_RCV_FUNC,  (INT32) hw_uart_read_cb);
 .
}
```

| **arg**: sets the port mode | **value**: sets the port mode option |
|---|---|
| **M2M_HW_UART_IO_HW_OPTION_GET**<br>Get the UART setting | Get the UART setting: baud rate, data bits, stop bits, flow ctrl, and parity.<br>See M2M_T_HW_UART_IO_HW_OPTIONS structure |
| **M2M_HW_UART_IO_HW_OPTION_SET**<br>Set the UART setting | Set the UART: baud rate, data bits, stop bits, flow ctrl, and parity.<br>See M2M_T_HW_UART_IO_HW_OPTIONS structure |

Example **M2M_HW_UART_IO_HW_OPTION_GET**:

```
m2m_hw_uart_ioctl (handle, M2M_HW_UART_IO_HW_OPTIONS_GET, (INT32) &setting);
```

Example **M2M_HW_UART_IO_HW_OPTION_SET**:

```
m2m_hw_uart_ioctl (handle, M2M_HW_UART_IO_HW_OPTIONS_SET, (INT32) &setting);
```

*Return value*:

refer to **M2M_T_HW_UART_RESULT** enum

## 5.20. m2m_hw_uart_get_state

**void m2m_hw_uart_get_state(**USB_UART_STATE ***ser_state)**

*Description*: gets the current Main Port (USIF0) setting.

*Parameters*:
ser_state: pointer to the allocated buffer that will be filled with the current Main Port setting. Output data.

## 5.21. m2m_hw_uart_aux_get_state

**void m2m_hw_uart_aux_get_state(**USB_UART_STATE ***ser_state)**

*Description*: gets the current Auxiliary Port (USIF1) setting.

*Parameters*:
ser_state: pointer to the allocated buffer that will be filled with the current Auxiliary Port setting. Output data.

# USB Section

## 5.22. m2m_hw_usb_open

M2M_API_RESULT **m2m_hw_usb_open (**M2M_USB_CH **channel,**
M2M_T_HW_USB_HANDLE ***handle)**

*Description*: opens the selected USB channel and returns its handle, its behaviors depends on the USB opening mode that is set by **m2m_hw_usb_set_open_ blocking()** API. The default USB opening mode is in blocking ON mode. In this mode, it is mandatory that the USB cable is connected to the USB port, otherwise the function does not return the control until the cable is plugged in. If USB opening mode is set in blocking OFF mode, and the USB cable is not plugged in, the function returns a failure indication and an error code.

*Parameters*:

channel: USB channel to open as serial USB channel, refer to **M2M_USB_CH** enum.

handle: pointer to the allocated variable that will be filled with the handle of the specified USB channel.
Output data:
on success: handle of the specified USB channel
on failure: refer to error codes for USB handle

*Return value*:
refer to **M2M_API_RESULT** enum

**NOTE**: referring to M2M_USB_CH enum, if you use:

**USB_CH_AUTO**: the function opens the first free USB channel, if any is available. The first free USB channel depends on the ports configuration of the module set through the AT#PORTCFG command, refer to document [4].

**USB_CH_DEFAULT**: the function opens the **USB_CH0**.

## 5.23. m2m_hw_usb_set_open_blocking

**void m2m_hw_usb_set_open_blocking(UINT8 on_off)**

*Description*: sets the opening mode of the USB port. The opening can be set in blocking ON or blocking OFF mode.

In blocking ON, the **m2m_usb_open(…)** API waits forever on the USB port if the USB cable is not plugged in. When the USB cable is plugged in, the API returns the control to the calling task. In blocking OFF, the control is returned regardless if the cable is connected or not to the USB port. The blocking ON is the default mode.

*Parameters*:

on_off: 1 = blocking ON,
0 = blocking OFF.

## 5.24. m2m_hw_usb_close

M2M_API_RESULT **m2m_hw_usb_close (**M2M_T_HW_USB_HANDLE **handle)**

*Description*: closes the USB channel identified by the handle. The logical connection defined by the #PORTCFG variant, set before the user application execution, is not restored. AppZone Layer does not release the resource even if the channel is closed, see also **m2m_usb_ close_hwch(…)**. For #PORTCFG configuration see document [4].

Parameters:
handle: handle of the USB channel to be closed.

Return value:
refer to **M2M_API_RESULT** enum

## 5.25. m2m_usb_close_hwch

M2M_API_RESULT **m2m_usb_close_hwch (**M2M_T_HW_USB_HANDLE ***handle)**

*Description*: closes the USB channel identified by the handle. The logical connection defined by the #PORTCFG variant, set before the user application execution, is restored. AppZone Layer releases the closed channel, see also **m2m_usb_close(…)**. For #PORTCFG configuration see document [4].

*Parameters*:
handle: pointer to the handle of the USB channel
to be closed.

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.26. m2m_usb_close_hw_allch

M2M_API_RESULT **m2m_usb_close_hw_allch(void)**

*Description*: closes all the USB channels, available in the current #PORTCFG configuration, and used by the user application. The logical connections defined by the #PORTCFG variant, set before the user application execution, are restored. AppZone Layer releases all the closed channels, see also **m2m_usb_ close_hwch(…)**. For #PORTCFG configuration see document [4].

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.27. m2m_hw_usb_read

M2M_API_RESULT **m2m_hw_usb_read (**M2M_T_HW_USB_HANDLE **handle, CHAR *buffer, INT32 len, INT32 *len_read)**

*Description*: receives data from an USB channel. The receiving mode is set by the function **m2m_hw_usb_ioctl(…)**.
If USB cable is not plugged in, the function returns a fail. If the cable is disconnected during running operation, the function returns the bytes received up to the cable disconnection event. If no bytes have been received, the function returns zero. This behavior is independent from the read blocking mode set.

*Parameters*:

| | |
|---|---|
| handle: | USB channel handle. |
| buffer: | pointer to the allocated buffer that will be filled with received data.<br>Output data:<br>on success: the buffer contains data to be read. len_read is the number of characters in the buffer.<br>on failure: len_read = 0, buffer content unchanged. |
| len: | number of characters to be read. |
| len_read: | pointer to the allocated variable that will be filled with the number of data read (bytes). |

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.28. m2m_hw_usb_write

M2M_API_RESULT **m2m_hw_usb_write (**M2M_T_HW_USB_HANDLE **handle, CHAR *buffer, INT32 len, INT32 *len_sent)**

*Description*: sends data through USB channel. The sending mode is set by the function **m2m_hw_usb_ioctl(…)**.
If USB cable is not plugged in, the function returns a fail. If the cable is disconnected during running operation, the function returns the number of bytes written up to the cable disconnection event. If no bytes have been written, the function returns zero. This behavior is independent from the write blocking mode set.

*Parameters*:

| | |
|---|---|
| handle: | USB channel handle. |
| buffer: | pointer to the allocated buffer containing the data to be sent. |
| len: | number of characters to be sent. |
| len_sent: | pointer to the allocated variable that will be filled with the number of data sent (bytes).<br>Output data:<br>on success: number of bytes that was sent.<br>on failure: len_sent = 0, buffer content unchanged |

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.29. m2m_hw_usb_ioctl

M2M_API_RESULT **m2m_hw_usb_ioctl (**M2M_T_HW_USB_HANDLE **handle,**
M2M_USB_ACTION_SELECTOR **arg,**
**INT32 value)**

*Description*: sets the USB channel mode. If USB opening mode is set to blocking OFF mode, and the USB cable is not plugged in, the function returns a failure indication.

*Parameters*:
handle: USB channel handle.
arg: selector of the USB mode, refer to **M2M_USB_ACTION_SELECTOR** enum
value: option related to the selected USB mode, refer to the tables below:

| **arg**: sets the USB mode | **value**: sets the USB mode option |
|---|---|
| **M2M_USB_BLOCKING_SET**<br>valid for TX and RX | **M2M_HW_USB_IO_BLOCKING_ON**: sets blocking to ON, default mode.<br><br>TX: The API returns the control when "len" bytes have been moved to the USB transmitter buffer. "len_sent" points to the number of bytes moved, and it is equal to "len".<br><br>RX: The API returns the control when "len" or more bytes have been received. "len_read" is the number of bytes moved into the buffer, it is equal to "len".<br>Bytes exceeding "len" remain in the USB receiver buffer. You can delete them using the following function:<br>**m2m_hw_usb_ioctl (usb_handle, M2M_USB_CLEAR_RX, M2M_HW_USB_IO_NO_ARG)** |
| | **M2M_HW_USB_IO_BLOCKING_OFF**: sets blocking to OFF<br><br>TX: The API tries to move "len" bytes in USB transmitter buffer, in accordance with the available resources. In any case, returns immediately the control. "len_sent" points to the number of bytes actually moved into USB transmitter buffer<br><br>RX: The API collects "len" bytes if they are available in the USB receiver buffer, and returns immediately the control. "len_read" is the number of available bytes moved from USB receiver buffer into the allocated buffer pointed by "buffer". |
| | **M2M_HW_USB_IO_BLOCKING_RELEASE**: releases both pending RX and TX activities, and set blocking to OFF. The new mode is blocking OFF |

Refer to the **M2M_HW_UART_IO_BLOCKING_SET** examples.

| **arg**: sets the USB mode | **value**: sets the USB mode option |
|---|---|
| **M2M_USB_RX_BLOCKING_SET**<br>valid only for RX | **M2M_HW_USB_IO_BLOCKING_ON**: sets blocking to ON only for RX activity, default. |
| | **M2M_HW_USB_IO_BLOCKING_OFF**: sets blocking to OFF only for RX activity. |
| | **M2M_HW_USB_IO_BLOCKING_RELEASE**: releases RX pending activity if any, and lets unchanged the blocking mode. |

Refer to the **M2M_HW_UART_RX_BLOCKING_SET** examples.

| **arg**: sets the USB mode | **value**: sets the USB mode option |
|---|---|
| **M2M_USB_TX_BLOCKING_SET**<br>valid only for TX | **M2M_HW_USB_IO_BLOCKING_ON**: sets blocking to ON only for TX activity, default. |
| | **M2M_HW_USB_IO_BLOCKING_OFF**: sets blocking to OFF only for TX activity |
| | **M2M_HW_USB_IO_BLOCKING_RELEASE**: releases TX pending activity if any, and lets unchanged the blocking mode. |

Refer to the **M2M_HW_UART_RX_BLOCKING_SET** examples.

| **arg**: sets the USB mode | **value**: sets the USB mode option |
|---|---|
| **M2M_USB_AT_MODE_SET**<br>Set AT command mode | **M2M_HW_USB_IO_AT_MODE_OFF**: default option. It routes the data received from the USB to the user AppZone application as they are, see document [4]. |
| | **M2M_HW_USB_IO_AT_MODE_ON**: this options routes the data received from the selected USBx channel to AT1 parser by means of AZ1 logical port, see document [4]. In addition, it sets RX in blocking OFF, and TX blocking mode is unchanged. |

Refer to the **M2M_HW_UART_IO_AT_MODE_SET** example.

| **arg**: sets the USB mode | **value**: sets the USB mode option |
|---|---|
| **M2M_USB_CLEAR_RX**<br>Clear the input buffer of the USB channel | **M2M_HW_USB_IO_NO_ARG:** the m2m_hw_usb_ioctl function uses a "arg" parameter that do not need options |

Refer to the **M2M_HW_UART_CLEAR_RX** example.

| **arg**: sets the USB mode | **value**: sets the option related to the selected USB feature |
|---|---|
| **M2M_USB_RCV_FUNC**<br>Indicates that the **value** parameter is a callback to manage received data | static INT32 **hw_usb_read_cb** ( M2M_T_HW_USB_HANDLE handle,<br>CHAR *buffer, INT32 len ) |

Refer to the **M2M_HW_UART_IO_RCV_FUNC** example.

| **arg**: sets the USB mode | **value**: sets the option related to the selected USB feature |
|---|---|
| **M2M_USB_NO_ACTION**<br>For internal use only | / |

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.30. m2m_hw_usb_get_state

**void m2m_hw_usb_get_state(M2M_USB_CH ch,** USB_UART_STATE ***usb_state)**

*Description*: gets the current setting of the selected USB channel.

*Parameters*:
ch: USB channel
usb_state: pointer to the allocated buffer that will be filled with the current USB channel setting

*Output data:* usb_state points to the buffer filled with the current USB setting

## 5.31. m2m_hw_usb_getch_from_handle

M2M_API_RESULT **m2m_hw_usb_getch_from_handle (**M2M_T_HW_USB_HANDLE **handle,** M2M_USB_CH ***channel)**

*Description*: gets the USB channel name related to the used handle.

*Parameters*:
handle: handle of the USB channel which name is unknown.

channel: pointer to the allocated variable that will be filled with the USB channel name related to the handle.
Output data:
on success: USB channel name, see **M2M_USB_CH** enum
on failure: **USB_CH**_NONE

*Return value*:
refer to **M2M_API_RESULT**enum

## 5.32. m2m_hw_usb_get_instance

USER_USB_INSTANCE_T **m2m_hw_usb_get_instance (**M2M_USB_CH **channel)**

*Description*: gets the USB instance of the selected USB channel.

*Parameters*:
channel: USB channel.

*Return value*:
refer to **USER_USB_INSTANCE_T** enum

**NOTE**: referring to **M2M_USB_CH** enum, if you use:

- **USB_CH_NONE** or **USB_CH_AUTO**, the function returns the first free instance, if any is available.
- **USB_CH_DEFAULT**, the function returns always **USER_USB_INSTANCE_0**.

## 5.33. m2m_hw_usb_cable_check

**UINT8 m2m_hw_usb_cable_check (void)**

*Description*: checks if the USB cable is plugged in or not. The check is based on the reading of D+ and D- signals.

*Return value*:
1 when the USB cable is plugged in, **USB_CABLE_ATTACHED**.
0 when the USB cable is unplugged, **USB_CABLE_DETACHED**.

**NOTE**: the AppZone Layer provides the M2M_onUSbCableEvent(...) callback function, refer to document [1]. When the USB cable is plugged in, the execution of the callback starts automatically after a time interval that may span from few seconds to several seconds. When the cable is unplugged, the execution starts immediately. It is up to the programmer write the needed code inside the callback.

➢ 4G: Platform Version ID 23

For the products with Platform Version ID 23, only at startup can be detected if the cable is disconnected, then can be detected only when the cable has been connected. Further disconnections are not detected.

## 5.34. m2m_hw_usb_get_ownership

M2M_API_RESULT **m2m_hw_usb_get_ownership(void)**

*Description*: grabs all USB channels, and the user application becomes the owner of the grabbed channels. To have information about USB channels availability and the logical connections set by #PORTCFG variant, see document [4].

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.35. m2m_hw_usb_wait_attach_forever

M2M_API_RESULT **m2m_hw_usb_wait_attach_forever(void)**

*Description*: sets the USB port in blocking ON mode, waiting forever the USB cable attach event, if not already happened. The programmer can use this function to wait USB cable attach before proceeding.

This API is also used by the **m2m_usb_open()** function when the USB port is in blocking ON mode, refer to **m2m_hw_usb_set_open_blocking()**.

*Return value*:
refer to **M2M_API_RESULT** enum

➢ 4G: Platform Version ID 23

For the products with Platform Version ID 23, only at startup can be detected if the cable is disconnected, then can be detected only when the cable has been connected. Further disconnections are not detected.

## 5.36. m2m_hw_usb_wait_attach_timeout

M2M_API_RESULT **m2m_hw_usb_wait_attach_timeout(UINT32 timeout)**

*Description*: sets the USB port in blocking ON mode, and waits the USB cable attach event for a time equal to timeout. The programmer, before proceeding, can use this function to wait the USB cable attach for a defined time.

*Parameters*:
timeout: 1 ÷ 100 → 100 msec; 101 ÷ 200 → 200 msec and so on.
If set to 0, the function exits immediately if the USB cable is not plugged in.

*Return value*:
refer to **M2M_API_RESULT** enum

➢ 4G: Platform Version ID 23

For the products with Platform Version ID 23, only at startup can be detected if the cable is disconnected, then can be detected only when the cable has been connected. Further disconnections are not detected.

# Power Down Section

## 5.37. m2m_hw_sleep_mode_cfg

M2M_API_RESULT **m2m_hw_sleep_mode_cfg (UINT8 cfun_mode,**
M2M_T_HW_SLEEP_MODE_CFG_OPTIONS ***options);**

*Description*: sets the CFUN power saving mode. If the configuration API is not used, the default power saving mode is CFUN=0. To have more information on CFUN power saving mode see documents [3], and [6].

*Parameters*:
cfun_mode: the available CFUN modes are:
0: default, it is NON-CYCLIC SLEEP mode. It means that when an event is managed, the module enters CFUN=1 mode (full features).
5, 7: are CYCLIC SLEEP modes. It means that when an event is managed, the module enters again CFUN=5 or 7 mode.
options: pointer not used. Set to **NULL**.

*Return value*:
refer to **M2M_API_RESULT** enum

## 5.38. m2m_hw_sleep_mode

**void m2m_hw_sleep_mode(UINT8 enter)**

*Description*: enables/disables the power saving mode configured by the **m2m_hw_sleep_mode_cfg(…)** function. To have information on CFUN power saving mode refer to documents [3], and [6].

*Parameters*:
enter: 1 enables.
0 enables.

## 5.39. m2m_hw_power_down

**void m2m_hw_power_down(void)**

*Description*: sets the module in power down mode. Typical use is to set an alarm using **m2m_rtc_set_alarm(...)**, and then power down the module. Once the alarm expires, the module will be powered up, and the user application will start execution.

# OTA Section

## 5.40. m2m_OTA_write_mem_data

**INT32 m2m_OTA_write_mem_data(UINT8 *buffer, INT32 len, INT32 offset)**

*Description*: works on OTA section memory, it moves "len" bytes from the buffer pointed by "buffer" to the memory starting from "offset" address. It is mandatory to use the m2m_OTA_erase_mem_data(...) before calling the writing function. It is responsibility of the programmer to avoid consecutive writing on the same memory portion.

*Parameters*:
buffer: pointer to the allocated buffer filled with data to be written.
len: number of bytes to write.
offset: memory offset where data is stored, see table below:

| Module Series | Offset Range |
|---|---|
| GE910 | 0 ÷ 0x140000 - 1 |
| HE910 | 0 ÷ 0x280000 - 1 |
| UE910 | 0 ÷ 0x280000 - 1 |
| UL865 | 0 ÷ 0x280000 - 1 |
| UE866 | 0 ÷ 0x280000 - 1 |
| LE910 Cat1 | 0 ÷ 0x5A0000 - 1 |
| LE910 V2 | |

*Return value*:
on success: 1
on failure: 0

➢ 4G: Platform Version ID 23

Platform Version ID 23 uses a compressed file system. Memory for OTA is around 3MB and can increase depending on the compression ratio of the data.

## 5.41. m2m_OTA_read_mem_data

**INT32 m2m_OTA_read_mem_data(INT32 offset, INT32 len, UINT8 *buffer)**

*Description*: works on OTA section memory, it moves "len" bytes starting from "offset" address into buffer pointed by "buffer" pointer.

*Parameters*:
buffer: pointer to the allocated buffer that will be filled with data read
Output data:
on success: the allocated buffer is filled with data read.
len: number of bytes to read
offset: offset in the memory where data is read, see table below:

| Module Series | Offset Range |
|---|---|
| GE910 | 0 ÷ 0x140000 - 1 |
| HE910 | 0 ÷ 0x280000 - 1 |
| UE910 | 0 ÷ 0x280000 - 1 |
| UL865 | 0 ÷ 0x280000 - 1 |
| UE866 | 0 ÷ 0x280000 - 1 |
| LE910 Cat1 | 0 ÷ 0x5A0000 - 1 |
| LE910 V2 | |

*Return value*:
on success: 1
on failure: 0

➢ 4G: Platform Version ID 23

Platform Version ID 23 uses a compressed file system. Memory for OTA is around 3MB and can increase depending on the compression ratio of the data.

## 5.42. m2m_OTA_erase_mem_data

**INT32 m2m_OTA_erase_mem_data(void)**

*Description*: erases all OTA memory section. This API must be called before using m2m_OTA_write_mem_data(...) function.

*Return value*:
on success: 1
on failure: 0

# 6. M2M_SPI_API

Chapter 19.2.5 contains the declarations of the C identifiers used by the APIs set regarding the management of the following device:

- SPI port

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# SPI Section

➢ 4G: Platform Version ID 23

SPI port is not provided.

## 6.1. m2m_spi_init

M2M_T_SPI_RESULT **m2m_spi_init(INT16 usif_num, INT16 mode, INT16 speed, INT16 *device)**

*Description*: sets the SPI port using the TX and RX lines provided by the selected serial port: SPI_MISO line is mapped onto TX, and SPI_MOSI line onto RX. The SPI port is configured as Master, the module provides the clock signal on the dedicate pin called SPI_CLK. To get hardware information, refer to document [2].

*Parameters*:

usif_num: see the table below.

| usif_num | SPI mapped onto Serial Port |
|---|---|
| 3 | USIF1 (Auxiliary Port) |

mode: the SPI interface provides four modes of clock phase (CPHA) and clock polarity (CPOL).

| mode | CPOL | CPHA | |
|---|---|---|---|
| 0 | 0 | 0 | Data are sampled on the rising edge of the clock |
| 1 | 0 | 1 | Data are sampled on the falling edge of the clock |
| 2 | 1 | 0 | Data are sampled on the falling edge of the clock |
| 3 | 1 | 1 | Data are sampled on the rising edge of the clock |

speed: the SPI interface provides four clock speeds.

| speed | Clock [MHz] |
|---|---|
| 1 | 1,0 |
| 2 | 3,25 |
| 3 | 6,5 |
| 4 | 13 |

device: **NULL**

*Return value*:

refer to **M2M_T_SPI_RESULT** enum

## 6.2. m2m_spi_write

M2M_T_SPI_RESULT **m2m_spi_write(INT16 usif_num, UINT8 *bufferToSend, UINT8 *bufferReceive, INT16 len, INT16 *device)**

*Description*: sends and receives data over the initialized SPI port.

*Parameters*:
usif_num: see the table below:

| usif_num | SPI mapped onto Serial Port |
|---|---|
| 3 | USIF1 (Auxiliary Port) |

bufferToSend: pointer to the allocated buffer filled with data to be sent
bufferReceive: pointer to the allocated buffer that will be filled with read data.
Output data:
on success: the buffer contains the read data.
len: number of bytes to send; see M2M_SPI_BUFFER_LEN
device: **NULL**

*Return value*:
refer to **M2M_T_SPI_RESULT**enum

## 6.3. m2m_spi_close

M2M_T_SPI_RESULT **m2m_spi_close(INT16 usif_num)**

*Description*: closes SPI port.

*Parameters*:
usif_num: see table below:

| usif_num | SPI mapped onto Serial Port |
|---|---|
| 3 | USIF1 (Auxiliary Port) |

*Return value*:
refer to **M2M_T_SPI_RESULT**enum

# 7. M2M_I2C_API

Chapter 19.2.6 contains the declarations of the C identifiers used by the APIs set regarding the management the following device:

- i2C bus

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# I2C Section

## 7.1. m2m_hw_i2c_conf

M2M_T_HW_I2C_RESULT **m2m_hw_i2c_conf(UINT8 i2c_sda, UINT8 i2c_scl)**

*Description*: sets an I2C port bus; to get hardware information, refer to document [2].

*Parameters*:
i2c_sda: is the sdaPin
i2c_scl: is the sclPin

To have information on sdaPin/sclPin see the AT command AT#I2CWR=? and AT#I2CRD=?. Refer to document [3].

*Return value*:
refer to **M2M_T_HW_I2C_RESULT** enum

## 7.2. m2m_hw_i2c_read

M2M_T_HW_I2C_RESULT **m2m_hw_i2c_read(UINT16 address, UINT8 reg_addr, UINT8 *buffer, UINT8 len)**

*Description*: reads data from an I2C device.

*Parameters*:
address: I2C device address. Its LSB is used as read\write command. It is up to the function to set rightly the bit 0. 10 bits address is supported
reg_addr: register address identifies where the first byte is read
buffer: pointer to the buffer that will be filled with the data read from I2C device (in hexadecimal format).
Output data:
on success: the buffer contains data read from the I2C device.
len: number of bytes to read, max number: M2M_HW_I2C_MAX_BUF_LEN.

*Return value*:
refer to **M2M_T_HW_I2C_RESULT** enum

Example of "address" parameter format:

Suppose that the I2C device has the address 7 bits long, for example: 0x0F. The I2C device address must be shifted to left by one bit as shown below. It is up to the function to set the bit 0 according to the command.

| | I2C device address | | | | | | | Writing command |
|---|---|---|---|---|---|---|---|---|
| "address" parameter | 0 | 0 | 0 | 1 | 1 | 1 | 1 | x |
| bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

## 7.3. m2m_hw_i2c_write

M2M_T_HW_I2C_RESULT **m2m_hw_i2c_write(UINT16 address, UINT8 reg_addr, UINT8 *buffer, UINT8 len)**

*Description*: writes data on an I2C device.

*Parameters*:

address: I2C device address. Its LSB is used as read\write command. It is up to the function to set rightly the bit 0. 10 bits address is supported

reg_addr: register address identifies where the first byte is written

buffer: pointer to the buffer containing the data to write on I2C device (in hexadecimal format)

len: number of bytes to write, max number: M2M_HW_I2C_MAX_BUF_LEN.

*Return value*:

refer to **M2M_T_HW_I2C_RESULT** enum

Example of "address" parameter format:

Suppose that the I2C device has the address 7 bits long, for example: 0x0F. The I2C device address must be shifted to left by one bit as shown below. It is up to the function to set the bit 0 according to the command.

| | I2C device address | | | | | | | Writing command |
|---|---|---|---|---|---|---|---|---|
| "address" parameter | 0 | 0 | 0 | 1 | 1 | 1 | 1 | x |
| bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

## 7.4. m2m_hw_i2c_cmb_format

M2M_T_HW_I2C_RESULT **m2m_hw_i2c_cmb_format (UINT16 address, UINT8 *buffer, UINT8 len_wr, UINT8 len_rd)**

*Description*: the function allows performing I2C Combined Format.

*Parameters*:

address: I2C device address. Its LSB is used as read\write command. It is up to the function to set rightly the bit 0. 10 bits address is supported

buffer: pointer to the buffer containing the data to write on I2C device if len_wr >0. After the writing, if len_rd>0, the buffer stores the data read from I2C device; data are in hexadecimal format.

len_wr: number of bytes to write. Max is **M2M_HW_I2C_MAX_BUF_LEN**.

len_rd: number of bytes to read. Max is **M2M_HW_I2C_MAX_BUF_LEN.**

*Return value*:

refer to **M2M_T_HW_I2C_RESULT** enum

Example of "address" parameter format:

Suppose that the I2C device has the address 7 bits long, for example: 0x0F. The I2C device address must be shifted to left by one bit as shown below. It is up to the function to set the bit 0 according to the command.

| | I2C device address | | | | | | | Writing command |
|---|---|---|---|---|---|---|---|---|
| "address" parameter | 0 | 0 | 0 | 1 | 1 | 1 | 1 | x |
| bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

# 8. M2M_NETWORK_API

Chapter 19.2.7 contains the declarations of the C identifiers used by the APIs set regarding the management of some network information.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Network Registration Section

## 8.1. m2m_network_enable_registration_location_unsolicited

**INT32 m2m_network_enable_registration_location_unsolicited(void)**

*Description*: enables the notification of the network registration report to the user M2M application. Every network registration update is automatically routed to M2M_onRegStatusEvent(…) callback function contained in the M2M_net.c file; refer to document [1]. See also the AT+CREG command described in document [3].

*Return value*:
on success: 1
on failure: 0

## 8.2. m2m_network_disable_registration_location_unsolicited

**INT32 m2m_network_disable_registration_location_unsolicited(void)**

*Description*: disables the notification of the network registration report to the user M2M application.

*Return value*:
on success: 1
on failure: 0

## 8.3. m2m_network_get_reg_status

**INT32 m2m_network_get_reg_status(**M2M_T_NETWORK_REG_STATUS_INFO ***reg_status_info)**

*Description*: gets the information on the network registration status.

*Parameters*:
reg_status_info: pointer to the allocated structure that the function fills with the network registration information: status, Lac, Ci, etc. The pointer must not be **NULL**.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success: the allocated structure is filled with the network registration information.

## 8.4. m2m_network_get_cell_information

**INT32 m2m_network_get_cell_information(**
M2M_T_NETWORK_CELL_INFORMATION ***cell_info)**

*Description*: gets the current cell information.

*Parameters*:
cell_info: pointer to the allocated structure that will be filled with cell information.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success: the allocated structure filled with cell information.

**NOTE**: the BSIC value returned by the function is expressed in decimal format.

## 8.5. m2m_network_get_currently_selected_operator

**INT32 m2m_network_get_currently_selected_operator(**
M2M_T_NETWORK_CURRENT_NETWORK ***selected_op)**

*Description*: gets the selected network operator. It returns the same information of the AT command "AT+COPS?" described in [3].

*Parameters*:
selected_op: pointer to the allocated structure that will be filled with the selected network operator.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success: the allocated structure filled with the selected operator.

## 8.6. m2m_network_list_available_networks

**INT32 m2m_network_list_available_networks(**
M2M_T_NETWORK_AVAILABLE_NETWORK ****m2m_available_net_list,**
**UINT16 *size)**

*Description*: returns the list of available networks. The list contains the same information returned by the AT command "AT+COPS=?" described in document [3]. The available networks searching is formed by several steps that use heavily the Level 1 resources. The network scan involves the reading of the power of the available frequencies and other activities that stop temporarily the network stack functionality. In this scenario, for example, the sending of an IP packet is delayed. Other events coming from the network are deferred, example, change cell, incoming SMS, etc.

*Parameters*:

m2m_available_net_list: pointer to the pointer pointing to memory allocated and filled by the function. After function execution, it is caller responsibility to free the memory using the function **m2m_os_mem_free(m2m_available_net_list)**

size: pointer to the allocated variable that will be filled with the number of available networks

*Return value*:

on success: 1
on failure: 0

*Output data:*

on success: pointer to the memory filled with the list of available networks.
number of available networks.

## 8.7. m2m_network_get_signal_strength

**INT32 m2m_network_get_signal_strength(INT32 *rssi, INT32 *ber)**

*Description*: returns the network signal strength, and bit error rate. The same information is returned by the AT command "AT+CSQ" described in document [3].

*Parameters*:

rssi: pointer to the allocated variable that will be filled with signal strength indication
ber: pointer to the allocated variable that will be filled with bit error rate

*Return value*:

on success: 1
on failure: 0

*Output data:*

on success: signal strength indication
bit error rate

## 8.8. m2m_network_enable_gprs_registration_location_unsolicited

**INT32 m2m_network_enable_gprs_registration_location_unsolicited(void)**

*Description*: enables the notification of the GPRS network registration status to the user M2M application. Every GPRS network registration update is automatically routed to M2M_onGprsReg StatusEvent(…) callback function contained in the M2M_net.c file; refer to document [1]. See also the AT+CGREG command described in document [3].

*Return value*:

on success: 1
on failure: 0

## 8.9. m2m_network_disable_gprs_registration_location_unsolicited

**INT32 m2m_network_disable_gprs_registration_location_unsolicited(void)**

*Description*: disables the notification of the GPRS network registration status to the user M2M application.

*Return value*:
on success: 1
on failure: 0

## 8.10. m2m_network_get_gprs_reg_status

**INT32 m2m_network_get_gprs_reg_status(**
M2M_T_NETWORK_GREG_STATUS_INFO ***reg_status_info)**

*Description*: gets the information on the GPRS network registration.

*Parameters*:
reg_status_info: pointer to the allocated structure that the function fills with the GPRS network registration information: registration status, LAC, cell_id, etc. The pointer must not be **NULL**.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success: the allocated structure is filled with the registration status information.

# 9. M2M_OS_API

Chapters 19.2.1, and 19.2.8 contain the declarations of the C identifiers used by the APIs set regarding the management of the Operating System features.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Module Information Section

## 9.1. m2m_info_get_model

**void m2m_info_get_model(CHAR *buf)**

*Description*: returns the module model.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the module model. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module model zero-terminated string.

## 9.2. m2m_info_get_manufacturer

**void m2m_info_get_manufacturer(CHAR *buf)**

*Description*: returns the module manufacturer.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the module manufacturer. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module manufacturer zero-terminated string.

## 9.3. m2m_info_get_factory_SN

**void m2m_info_get_factory_SN(CHAR *buf)**

*Description*: returns the module factory SN.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the factory SN. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with factory SN zero-terminated string.

## 9.4. m2m_info_get_serial_num

**void m2m_info_get_serial_num(CHAR *buf)**

*Description*: returns the module serial number.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the module serial number. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module serial number zero-terminated string.

## 9.5. m2m_info_get_sw_version

**void m2m_info_get_sw_version(CHAR *buf)**

*Description*: returns the Telit AppZone software version installed on the module.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the Telit AppZone software version. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the Telit AppZone software version zero-terminated string.

## 9.6. m2m_info_get_fw_version

**void m2m_info_get_fw_version(CHAR *buf)**

*Description*: returns the module software version. It is the same information returned by the AT command "AT+CGMR" described in document [3].

*Parameters*:
buf: pointer to the allocated buffer that will be filled with the module software version. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module software version zero-terminated string.

## 9.7. m2m_info_get_MSISDN

**void m2m_info_get_MSISDN(CHAR *buf)**

*Description*: returns the Mobile Station ISDN Number.

*Parameters*:

buf: pointer to the allocated buffer that will be filled with the MSISDN. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module MSISDN zero-terminated string.

## 9.8. m2m_info_get_IMSI

**void m2m_info_get_IMSI(CHAR *buf)**

*Description*: returns the module IMSI.

*Parameters*:
buf: pointer to the allocated buffer that will be filled with IMSI. Use a buffer length good enough to contain the zero-terminated string, for example 20 bytes length.

*Output data:*
buffer filled with the module IMSI zero-terminated string.

## 9.9. m2m_os_set_version

**INT32 m2m_os_set_version(CHAR *sw_version)**

*Description*: sets the software version of the customer M2M application.

*Parameters*:
sw_version: pointer to the zero-terminated string containing the customer M2M application software version.
It cannot be larger than M2M_OS_MAX_SW_VERSION_STR_LENGTH.

*Return value*:
on success: 1
on failure: -1

## 9.10. m2m_os_get_version

**CHAR *m2m_os_get_version(void)**

*Description*: returns the customer M2M application software version as set by the m2m_os_set_version(…) function.

*Return value*:
pointer to the buffer filled with the customer M2M software version string.

# Task Section

## 9.11. m2m_os_create_task

**INT32 m2m_os_create_task(**M2M_OS_TASK_STACK_SIZE **stackSize,**
**UINT8 priority,**
M2M_OS_TASK_MBOX_SIZE **mboxSize,**
M2M_CB_MSG_PROC **msg_cb)**

*Description*: creates a user task and returns its process id.

*Parameters*:
stackSize: 2, 4, 8, 16 [Kbytes], refer to **M2M_OS_TASK_STACK_SIZE** enum
priority: 1 ÷ 32; 1 = highest priority
mboxSize: 10, 50, 100 [msg], refer to **M2M_OS_TASK_MBOX_SIZE** enum
msg_cb: name of the function called by the task, example: M2M_msgProc11

*Return value*:
on success: process id, range: 1 ÷ 32
on failure: 0, it means that 32 tasks have already been created
-1, invalid parameters

**NOTE**: you must reduce the size of the stack to use the maximum number of tasks (32).

The first row of the table below shows the default AppZone layer configuration concerning the number of the tasks, and the connected M2M_msgProcX() functions; refer to document [1].

| | **Process id of the Task used by m2m_os_send_message_to_task()** | **M2M_msgProcX(…)** | |
|---|---|---|---|
| The default AppZone layer provides one task and one M2M_msgProc1(…) callback function contained in the M2M_proc1.c file. | 1 | M2M_msgProc1 (…) | The default skeleton configuration includes the M2M_proc1.c file containing two callbacks:<br>- M2M_msgProc1()<br>- M2M_msgProcCompl() |
| It is up to the user to create, if needed, new tasks. The AppZone layer supports up to 32 tasks in total. | 2 | M2M_msgProc2(…) | It is responsibility of the user to write the M2M_msgProcX(…) callback functions.<br><br>In this configuration, each task calls a different M2M_msgProcX(…) callback.<br><br>One M2M_procX.c file may contain one or more callbacks. |
| | 3 | M2M_msgProc3(…) | |
| | 4 | M2M_msgProc4(…) | |
| | 5 | M2M_msgProc5(…) | |
| | 6 | M2M_msgProc6(…) | |
| | 7 | M2M_msgProc7(…) | |
| | 8 | M2M_msgProc8(…) | |
| | 9 | M2M_msgProc9(…) | |
| | 10 | M2M_msgProc10(…) | |
| | 11 | M2M_msgProc11(…) | |
| | 12 | M2M_msgProc12(…) | |
| | 13 | M2M_msgProc13(…) | |
| | ... | M2M_msgProc...(…) | |
| | ... | M2M_msgProc...(…) | |
| | 32 | M2M_msgProc32(…) | |

The table below shows an example of max configuration in which each task calls a single M2M_msgProc(…) callback.

| | Process id of the Task used by m2m_os_send_message_to_task(…) | Single M2M_msgProc | |
|---|---|---|---|
| The AppZone layer supports up to 32 tasks in total. | 1(default) | M2M_msgProc(…) | In this configuration, each task is connected to a single M2M_msgProc(…) callback that can uses the m2m_os_get_current_task_id() API to know which is the calling task. |
| | 2 | | |
| | 3 | | |
| | 4 | | |
| | 5 | | |
| | 6 | | |
| | 7 | | |
| | 8 | | |
| | 9 | | |
| | 10 | | |
| | 11 | | |
| | 12 | | |
| | 13 | | |
| | ... | | |
| | ... | | |
| | 32 | | |

## 9.12. m2m_os_get_current_task_id

**UINT8 m2m_os_get_current_task_id(void)**

*Description*: returns the process id of the running task; see also the m2m_os_create_task(…) function and the related tables showing different M2M application configurations.

*Return value*:
task id (or process id), range 1÷32

## 9.13. m2m_os_cooperate_task

**void M2M_os_cooperate_task(void)**

*Description*: releases the process control to other ready-to-run tasks at the same or higher priority.

## 9.14. m2m_os_destroy_task

**INT32 m2m_os_destroy_task(INT8 procId)**

*Description*: deletes the user task identified by the process id procId.

*Parameters*:
procId: identifies the task to be deleted.

*Return value*:
on success: 1
on failure: 0
-1, invalid parameters

## 9.15. m2m_os_send_message_to_task

**INT32 m2m_os_send_message_to_task(INT8 procId, INT32 type, INT32 param1, INT32 param2)**

*Description*: sends a message to the selected user tasks.

*Parameters*:
procId: task number at which is addressed the message. Task numbers (o process id): 1÷32
type: identifies the message type. It is up to the user the message type definition.
param1: auxiliary parameter defined by the user
param2: auxiliary parameter defined by the user

*Return value*:
on success: 1
on failure: -1

## 9.16. m2m_os_task_get_enqueued_msg

**INT32 m2m_os_task_get_enqueued_msg(INT8 procId)**

*Description*: returns the number of pending messages on the queue of the selected task.

*Parameters*:
procId: task numbers (or process id). Range: 1÷ M2M_OS_MAX_PROCESS

*Return value*:
on success: number of pending messages
on failure: -1

## 9.17. m2m_os_set_argc

**INT8 m2m_os_set_argc(INT8 argc)**

*Description*: sets the number of argument strings to be passed to the

**M2M_main(INT32 argc, CHAR argv [**M2M_ARGC_MAX**][**M2M_ARGV_MAXTOKEN **+ 1])**

Refer to document [1].

*Parameters*:
argc: 1 ÷ 4, see **M2M_ARGC_MAX**

*Return value*:
on success: 1
on failure: -1

## 9.18. m2m_os_get_argc

**INT8 m2m_os_get_argc(void)**

*Description*: returns the number of argument strings stored in the internal parameters table.

*Parameters*:
no parameters.

*Return value*:
on success: number of argument strings
on failure: /

## 9.19. m2m_os_set_argv

**INT8 m2m_os_set_argv(UINT8 index, CHAR*arg)**

*Description*: sets a new argument string in the internal parameters table. Refer also to m2m_os_set_argc(...) function.

*Parameters*:
index: 0 ÷ 3, identifies the index of the new argument string
arg: pointer to the new argument string. Its length must be equal or less than 15 bytes, see M2M_ARGV_MAXTOKEN .

*Return value*:
on success: 1
on failure: -1

## 9.20. m2m_os_get_argv

**CHAR*m2m_os_get_argv(UINT8 index)**

*Description*: returns the argument string from the internal parameters table.

*Parameters*:
index: 0 ÷ 3, is the index of the argument string to be read.

*Return value*:
on success: pointer to the argument string read from the internal parameters table
on failure: **NULL**

## 9.21. m2m_os_iat_set_at_command_instance

**INT32 m2m_os_iat_set_at_command_instance(UINT16 logPort, UINT16 atInstance);**

*Description*: sets the logical connection between one M2M logical port (AZ1, AZ2) and one AT Command Parser Instance of the module (AT0, AT1, AT2); the table below shows an example. Refer to document [4] to have more information.

| | M2M logical ports | |
|---|---|---|
| **AT Command Parser Instances** | **AZ1** | **AZ2** |
| AT0 | | |
| AT1 | ✓ | |
| AT2 | | ✓ |

*Parameters*:
logPort: M2M logical port to link to AT Command Parser Instance. Range: 1÷2
atInstance: AT Command Parser Instance. Range: 0÷2

*Return value*:
on success: 1
on failure: -1

## 9.22. m2m_os_iat_send_at_command

**INT32 m2m_os_iat_send_at_command(CHAR *atCmd, UINT16 logPort)**

*Description*: sends AT command to the modem.

*Parameters*:
atCmd: pointer to the zero-terminated string containing the AT command;
logPort: M2M logical port connected to AT Command Parser Instance through m2m_os_iat_set_at_command_instance(…) function.

*Return value*:
on success: 1
on failure: -1

## 9.23. m2m_os_iat_send_atdata_command

**INT32 m2m_os_iat_send_atdata_command(CHAR *atCmd, INT32 atCmdLength, UINT16 logPort)**

*Description*: sends AT data to the modem.

*Parameters*:
atCmd: pointer to the not zero-terminated string containing the AT data;
atCmdLength: length of not zero-terminated string containing the AT data (bytes);
logPort: M2M logical port connected to AT Command Parser Instance through m2m_os_iat_set_at_command_instance(…) function.

*Return value*:
on success: 1
on failure: -1

# Memory Pool Section

## 9.24. m2m_os_mem_pool

**INT32 m2m_os_mem_pool(UINT32 pool_size)**

*Description*: reserves a dynamic memory pool space (HEAP).

*Parameters*:
pool_size: requested HEAP size expressed in bytes. If this function is not used, the system provides a default pool memory size. See table below.

| | Modules Series | | | | | |
|---|---|---|---|---|---|---|
| **Heap Size** | **GE910** | **HE910** | **UE910** | **UL865** | **UE866** | **LE910 V2/Cat1** |
| Max | 512 kB | 2 MB | 2 MB | 2 MB | 2 MB | 16 MB |
| default | 8 kB | 8 kB | 8 kB | 8 kB | 8 kB | 8 kB |

**NOTE**: every time the function is called, the previous memory pool is removed and the new one is created.

*Return value*:
on success: 1
on failure: -1

➢ 4G: Platform Version ID 23

This API doesn't have any effect for Platform Version ID 23

## 9.25. m2m_os_mem_alloc

**void *m2m_os_mem_alloc(UINT32 size)**

*Description*: allocates dynamic memory within the HEAP.

*Parameters*:
size: requested buffer size expressed in bytes. It must be in accordance with the dynamic memory pool space, see m2m_os_mem_pool(…)

*Return value*:
on success: pointer to the allocated memory block;
on failure: **NULL** if the requested memory size is not available.

## 9.26. m2m_os_mem_realloc

**void *m2m_os_mem_realloc(void *ptr, UINT32 size)**

*Description*: reallocates dynamic memory within the HEAP.

*Parameters*:

ptr: pointer to memory
size: requested buffer size expressed in bytes. It must be in accordance with the dynamic memory pool space, see m2m_os_mem_pool(…)

*Return value*:
on success: pointer to the reallocated memory block;
on failure: **NULL** if the requested memory size is not available.

## 9.27. m2m_os_mem_free

**void m2m_os_mem_free(void *mem)**

*Description*: frees an already allocated memory within the HEAP.

*Parameters*:
mem: pointer to the memory to free

## 9.28. m2m_os_get_mem_info

**UINT32 m2m_os_get_mem_info(UINT32 *pool_frags)**

*Description*: returns dynamic memory pool space (HEAP) information:
- the total number of memory fragments
- the total number of available bytes

*Parameters*:
pool_frags: pointer to the allocated variable that will be filled with the total number of memory fragments. If it is **NULL,** no total number of fragments is returned.

*Return value*:
total number of available bytes in the dynamic memory pool space, may be **NULL**.

*Output data:*
"pool_frags" points to the allocated variable filled with the total number of memory fragments.

# System Tick and Sleep Section

## 9.29. m2m_os_retrieve_clock

**INT32 m2m_os_retrieve_clock(void);**

*Description*: returns the system tick.

*Return value*:
system tick, 1 tick = see the table below

| Module Series | System tick, 1 tick = |
|---|---|
| GE910 | 10 ms |
| HE910 | 100 ms |
| UE910 | 100 ms |
| UL865 | 100 ms |
| UE866 | 100 ms |
| LE910 V2/Cat1 | 100 ms |
| LE866/ME866A1 | 10ms |

## 9.30. m2m_os_sleep_ms

**void m2m_os_sleep_ms(UINT32 ms)**

*Description*: forces the current task in sleep mode.

*Parameters*:
ms: is expressed in msec. The resolution is 100 msec: 1 ÷ 100 →100 msec, 101 ÷ 200 → 200 msec, and so on.

## 9.31. m2m_os_sys_reset

**void m2m_os_sys_reset(INT32 id)**

*Description*: resets the entire system (module).

Parameters:
id: use 0, it is a dummy parameter used only for backward compatibility.

# Trace Section

## 9.32. m2m_os_trace_out

**void m2m_os_trace_out(CHAR *msg)**

*Description*: sends messages on the TT Access Point linked to the trace port. The physical trace port is set by the #PORTCFG variant, refer to document [4]. On the physical trace port are running user and/or trace messages as shown in the table below. Use the +TRACE command to choose the format and type of the messages running on the trace port. The factory setting supports user messages only. See document [3]

*Parameters*:
msg: pointer to the zero-terminated string containing the message to be sent.

➢ 2G, 3G: Platform Version ID 13, 12

| Trace disabled | Trace enable | m2m_os_trace_out(…) working | Messages running on the trace port |
|---|---|---|---|
| ✓ | | ✓ | Only user messages are sent to the terminal, they are issued in ASCII format; a hyper-terminal can display them. It is the factory-setting configuration. |
| | ✓ | ✓ | The trace and user messages are coded. A suitable tool is needed to display both messages on the terminal. |

➢ 4G: Platform Version ID 20

| Trace disabled | Trace enable | m2m_os_trace_out(…) working | Messages on terminal connected to the selected port |
|---|---|---|---|
| ✓ | | / | No user messages (*). |
| | ✓ | ✓ | The trace and user messages are coded. A suitable tool is needed to display both messages on the terminal. |

(*) Use the **m2m_hw_uart_aux_open(…)** function to print user messages.

➢ 4G: Platform Version ID 23

Function not supported.

# 10. M2M_OS_LOCK_API

Chapter 19.2.9 contains the declarations of the C identifiers used by the APIs set regarding the management of the semaphores and mutexes.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Semaphore Section

## 10.1. m2m_os_lock_init

M2M_T_OS_LOCK **m2m_os_lock_init(INT32 val)**

*Description*: creates a semaphore, and returns its handle. To have info on the granularity of the timeout used by this semaphore, refer to the **m2m_os_lock_wait()** function.

*Parameters*:
val: initial counter value of the semaphore.

*Return value*:
on success: pointer to the semaphore handle;
on failure: **NULL**.

*Examples*: 19.1.4, 19.1.5

## 10.2. m2m_os_lock_binary_init

M2M_T_OS_LOCK **m2m_os_lock_binary_init(INT32 val)**

*Description*: creates a semaphore, initializes it as a binary semaphore, and returns its handle. The function is similar to **m2m_os_lock_init()** with the exceptions of "val" parameter, which will be limited to one.

*Parameters*:
val: initial counter value of the binary semaphore, range: 0, 1. Greater values will be truncated to 1.

*Return value*:
on success: pointer to the binary semaphore handle;
on failure: **NULL**.

## 10.3. m2m_os_lock_lock

M2M_API_RESULT **m2m_os_lock_lock(**M2M_T_OS_LOCK **lock)**

*Description*: decreases by one the count of the semaphore identified by its handle. If, before decreasing, the retrieved semaphore counter value is zero, the counter value does not change, and the control is not returned to the calling task until another task unlocks or destroys the semaphore.

*Parameters*:
lock: semaphore handle.

*Return value*:
refer to **M2M_API_RESULT**enum.

## 10.4. m2m_os_lock_unlock

M2M_API_RESULT **m2m_os_lock_unlock(**M2M_T_OS_LOCK **lock)**

*Description*: increases by one the counter value of the semaphore identified by its handle.

*Parameters*:
lock: semaphore handle.

*Return value*:
refer to **M2M_API_RESULT** enum.

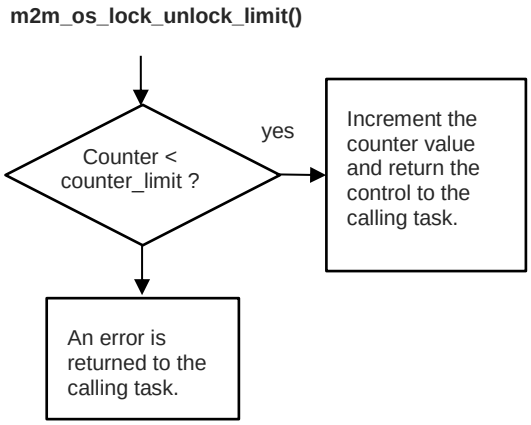
## 10.5. m2m_os_lock_unlock_limit

M2M_API_RESULT **m2m_os_lock_unlock_limit(**M2M_T_OS_LOCK **lock,**
**UINT32 counter_limit)**

*Description*: if current counter value < counter_limit, the function unlocks the semaphore incrementing by one its current counter value, and returns the control to the task. If the current counter value ≥ counter_limit, the function returns an error code. Refer to the flow chart below.
It is up to the programmer to avoid incrementing the semaphore counter value over the counter_limit value through an improper or unlimited use of **m2m_os_lock_unlock()** in another tasks.

*Parameters*:
lock: semaphore handle.
counter_limit: range: 1 ÷ 0xFFFFFFFF

*Return value*:
refer to **M2M_API_RESULT** enum.



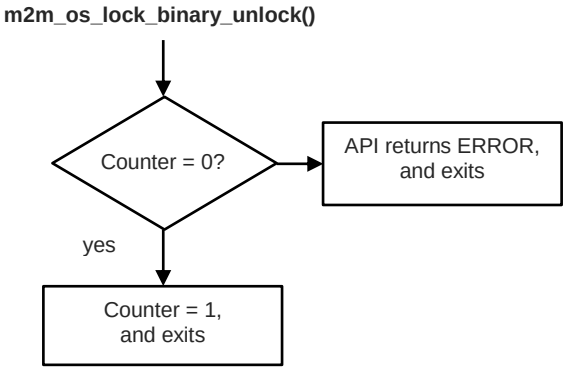
## 10.6. m2m_os_lock_binary_unlock

M2M_API_RESULT **m2m_os_lock_binary_unlock(**M2M_T_OS_LOCK **lock)**

*Description*: unlocks the semaphore as a binary semaphore. The programmer should always use this function to unlock binary semaphores. Refer to the flow chart below. Alternatively, the programmer can use **m2m_os_lock_unlock_limit()** with counter_limit parameter equal to **M2M_OS_LOCK_CS**.

*Parameters*:
lock: semaphore handle.

*Return value*:
refer to **M2M_API_RESULT**enum.




```
Example:
.............
M2M_T_OS_LOCK Bin_Lock;
UINT32 val = 0;
M2M_API_RESULT API_result;
.............

Task N:
...........
/* create a binary semaphore */
Bin_Lock = m2m_os_lock_binary_init(val);

/* Task N is suspended on the binary semaphore */
API_result = m2m_os_lock_lock (Bin_Lock);
...........

Task M:

/* Task M unlocks the binary semaphore on which Task N is suspended */
/* The value of the semaphore is set to 1*/
API_result = m2m_os_lock_binary_unlock(Bin_Lock);
...........

Task Q:

/* Task Q unlocks again the binary semaphore on which Task N was suspended */
/* The value of the semaphore is 1, the function returns an error code */
API_result = m2m_os_lock_binary_unlock(Bin_Lock);
...........
```

## 10.7. m2m_os_lock_prioritize

M2M_API_RESULT **m2m_os_lock_prioritize(**M2M_T_OS_LOCK **lock)**

*Description*: when a task is waiting on a semaphore, it is suspended and inserted into a Suspending List associated with the semaphore itself; refer to the table below on the left side. The function moves the task with the highest priority at the top of the Suspending List, see table on right side.

| Suspending List associated with a semaphore | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_4 | 4 |
| Task_19 | 19 |

| Suspending List associated with a semaphore | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_4 | 4 |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_19 | 19 |

*Parameters*:
lock: semaphore handle.

*Return value*:
refer to **M2M_API_RESULT** enum.

➢ 4G: Platform Version ID 23

This API does not have any effect for Platform Version ID 23. All tasks are performed according to the priority set during the creation of each task. Following the previous example, tasks will be executed with this order: Task_4, Task_7, Task_19 and Task_24

## 10.8. m2m_os_lock_wait

M2M_API_RESULT **m2m_os_lock_wait(**M2M_T_OS_LOCK **lock, UINT32 timeout)**

*Description*: decreases by one the counter of the semaphore identified by its handle. If, before decreasing, the retrieved semaphore counter value is zero, the counter value does not change, and the control is not returned to the calling task until at least one of the two events happens:

- another task calls proper API to unlock or destroy the semaphore
- the timeout is expired.

*Parameters*:
lock: semaphore handle;
timeout: expressed in msec. The resolution is 100 msec: 1 ÷ 100 →100 msec, 101 ÷ 200 →200 msec, and so on.

*Return value*:
refer to **M2M_API_RESULT** enum.

**NOTE**: if the return value is:

- **M2M_API_RESULT_SUCCESS**: the calling task gets the control because another task has unlocked the semaphore;

- **M2M_API_RESULT_FAIL**: the calling task gets the control because the timeout is expired or the semaphore has been destroyed by another task.

## 10.9. m2m_os_lock_info

M2M_API_RESULT **m2m_os_lock_info(**M2M_T_OS_LOCK **lock, INT32 *sem_count, INT32 *count_suspended)**

*Description*: gets info on the semaphore.

*Parameters*:

| | |
|---|---|
| lock: | semaphore handle. |
| sem_count: | pointer to the allocated variable that will be filled with the counter of the semaphore, (output parameter). |
| count_suspended: | pointer to the allocated variable that will be filled with the number of tasks suspended on the semaphore, (output parameter). |

*Return value*:
refer to **M2M_API_RESULT** enum.

**NOTE**: in case of failure the return value is **M2M_API_RESULT_FAIL** and the output parameters:

- sem_count points to **INVALID_COUNT**
- count_suspended points to **INVALID_SUSPENDED**

## 10.10. m2m_os_lock_destroy

M2M_API_RESULT **m2m_os_lock_destroy(**M2M_T_OS_LOCK **lock)**

*Description*: destroys the selected semaphore, and releases all resources allocated for it. It is recommended to assign a **NULL** value to the lock handle after deletion to prevent improper further usage.

*Parameters*:
lock: semaphore handle.

*Return value*:
refer to **M2M_API_RESULT** enum.

# msSemaphore Section

## 10.11. m2m_os_mslock_init

M2M_API_RESULT **m2m_os_mslock_init (**M2M_T_OS_MSLOCK ***msLock,**
**UINT32 val)**

*Description*: creates a msSemaphore, and returns its handle; set ***msLock** to **NULL** (**INVALID_MS_SEM**) before calling the function. To have info on the granularity of the timeout used by this msSemaphore, refer to the **m2m_os_mslock_wait()** function.

*Parameters*:
msLock: pointer to the msSemaphore handle returned by the function, (output parameter). On failure: **NULL**.
val: initial counter value of the msSemaphore.

*Return value*:
refer to **M2M_API_RESULT** enum.

```
Example:

Task N:
.....
M2M_T_OS_MSLOCK new_Lock = INVALID_MS_SEM;
UINT32 new_val = 0;
LOCK_RESULT_T new_lock_result;
.....
/* create an msSemaphore */
m2m_os_mslock_init(&new_Lock, new_val);

/* Task N is suspended until the msSemaphore is unlocked */
m2m_os_mslock_lock (new_Lock, &new_lock_result);
.....
```

## 10.12. m2m_os_mslock_binary_init

M2M_API_RESULT **m2m_os_mslock_binary_init(**M2M_T_OS_MSLOCK ***msLock,**
**UINT32 val)**

*Description*: creates a msSemaphore, initializes it as a binary semaphore, and returns its handle. The function is similar to **m2m_os_mslock_init()** with the exceptions of "val" parameter, which will be limited to one.

*Parameters*:
msLock: pointer to the msSemaphore handle returned by the function, (output parameter). On failure: **NULL**.
val: initial counter value of the binary msSemaphore, range: 0, 1. Greater values will be truncated to 1.

*Return value*:
refer to **M2M_API_RESULT** enum

## 10.13. m2m_os_mslock_lock

M2M_API_RESULT **m2m_os_mslock_lock (**M2M_T_OS_MSLOCK **msLock,**
LOCK_RESULT_T ***lock_res )**

*Description*: decreases by one the counter of the msSemaphore identified by its handle. If, before decreasing, the retrieved msSemaphore counter value is zero, the counter value does not change, and the control is not returned to the calling task until another task unlocks or destroys the msSemaphore. See also the **m2m_os_mslock_wait()** function.

*Parameters*:
msLock: handle of msSemaphore
lock_res: pointer to the allocated variable that will be filled with the msSemaphore status, (output parameter). The used pointer can be **NULL**.

*Return value*:
refer to **M2M_API_RESULT** enum

```
Example:
M2M_T_OS_MSLOCK  msLock;
LOCK_RESULT_T    lock_result;
UINT32           val=0;

Task N:
.....
/* create a binary semaphore */
m2m_os_mslock_binary_init(&msLock, val);

/* Task N is suspended on the binary semaphore */
m2m_os_mslock_lock (msLock, &lock_res);
.....
```

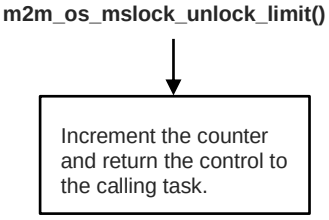
## 10.14. m2m_os_mslock_unlock

M2M_API_RESULT **m2m_os_mslock_unlock (**M2M_T_OS_MSLOCK **msLock )**

*Description*: every time the function is called, it unlocks the msSemaphore incrementing by one its counter value. No counter_limit value is used, see **m2m_os_mslock_unlock _limit()**. It is up to the programmer to use the right msSemaphore handle. If the programmer uses improperly a handle of a binary msSemaphore, the binary msSemaphore could become a counting msSemaphore because no more control of counter_limit will be in place. Refer to the flow chart below.

*Parameters*:
msLock: handle of msSemaphore

*Return value*:
refer to **M2M_API_RESULT** enum




## 10.15. m2m_os_mslock_unlock_limit

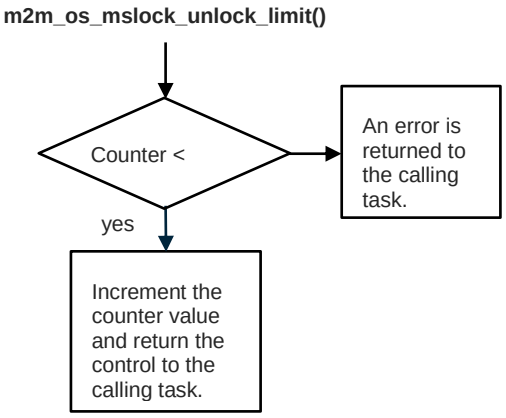
M2M_API_RESULT **m2m_os_mslock_unlock_limit (**M2M_T_OS_MSLOCK **msLock,**
**UINT32 counter_limit)**

*Description*: if the current counter value < counter_limit, the function unlocks the msSemaphore incrementing by one its current counter value, and returns the control to the task. If the current counter value ≥ counter_limit, the function returns an error code. Refer to the flow chart below.
It is up to the programmer to avoid incrementing the msSemaphore counter value over the counter_limit value through an improper or unlimited use of **m2m_os_mslock_unlock()** in another tasks.

**NOTE**: it is up to the programmer to use the right handle. If this function is used with a binary msSemaphore handle and with an improper counter_limit value (> 1), it could became a counting msSemaphore.

*Parameters*:
lock: msSemaphore handle.
counter_limit: range: 1 ÷ 0xFFFFFFFF

*Return value*:
refer to **M2M_API_RESULT** enum



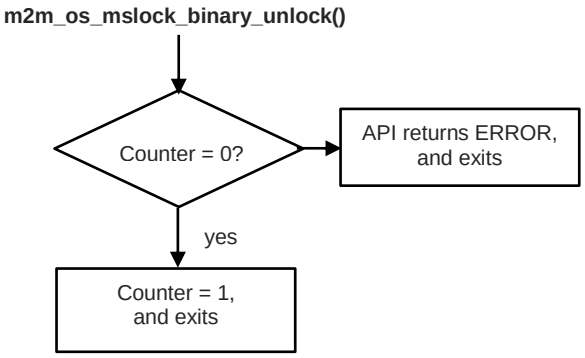

## 10.16. m2m_os_mslock_binary_unlock

M2M_API_RESULT **m2m_os_mslock_binary_unlock(**M2M_T_OS_MSLOCK **msLock)**

*Description*: unlocks the msSemaphore as a binary semaphore. The programmer should always use this function to unlock binary msSemaphore. Refer to the flow chart below.
Alternatively, the programmer can use **m2m_os_mslock_unlock_limit()** with counter_limit parameter equal to **M2M_OS_LOCK_CS**.

*Parameters*:
msLock: handle of msSemaphore

*Return value*:
refer to **M2M_API_RESULT** enum




```
Example:
.............
M2M_T_OS_MSLOCK msLock;
UINT32 val = 0;
LOCK_RESULT_T lock_result;
.............

Task N:
```

```
.....
/* create a binary msSemaphore */
m2m_os_mslock_binary_init(&msLock, val);

/* Task N is suspended on the binary msSemaphore */
m2m_os_mslock_lock (msLock, &lock_res);
.....

Task M:

/* Task M unlocks the binary msSemaphore on which Task N is suspended */
/* The value of the msSemaphore is set to 1*/
m2m_os_mslock_binary_unlock(msLock);
.....

Task Q:

/* Task Q unlocks again the binary msSemaphore on which Task N was suspended */
/* The value of the msSemaphore is 1, the function returns an error code */
m2m_os_mslock_binary_unlock(msLock);
.....
```

## 10.17. m2m_os_mslock_prioritize

M2M_API_RESULT **m2m_os_mslock_prioritize(**M2M_T_OS_MSLOCK **msLock )**

*Description*: when a task is waiting on a msSemaphore, it is suspended and inserted into a Suspending List associated with the msSemaphore itself; refer to the table below on the left side. The function moves the task with the highest priority at the top of the Suspending List, see table on right side

| Suspending List associated with a semaphore | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_4 | 4 |
| Task_19 | 19 |

| Suspending List associated with a semaphore | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_4 | 4 |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_19 | 19 |

*Parameters*:
msLock: msSemaphore handle.

*Return value*:
refer to **M2M_API_RESULT**enum.

➢ 4G: Platform Version ID 23

This API does not have any effect for Platform Version ID 23. All tasks are performed according the priority set during the creation of each task. Following the previous example, tasks will be executed with this order: Task_4, Task_7, Task_19 and Task_24

## 10.18. m2m_os_mslock_setlocklimit

M2M_API_RESULT **m2m_os_mslock_setlocklimit(UINT16 max_lock_limit)**

*Description*: up to 32 tasks can be suspended on a single msSemaphore and inserted into the Suspending List associated with the semaphore itself, see **m2m_os_mslock_ prioritize()** function.

By default setting, the maximum number of tasks that can be simultaneously suspended on all created msSemaphores is also 32. For example, 4 msSemaphores are blocking 4 tasks each one, plus 2 msSemaphores are blocking 8 tasks each one; in general, the msSemaphores can be used with a timeout value greater than zero or with a wait forever option.

This function allows to change the maximum number of suspended tasks. If no semaphores are created, the programmer can increase/decrease the default number of tasks. If one or more semaphores are already created (run time scenario), the programmer can only increase the number of tasks, decreasing is not allowed and the function returns the error code **M2M_API_RESULT _INVALID _ARG**. This function can be used to optimize the memory usage.

*Parameters*:
max_lock_limit: maximum number of locked tasks on overall msSemaphores.

*Return value*:
refer to **M2M_API_RESULT** enum.

**NOTE**: Now, this function is not needed because 32 is the maximum number of tasks supported by the AppZone Layer, see M2M_OS_MAX_PROCESS. It will be useful for future implementations.

## 10.19. m2m_os_mslock_wait

M2M_API_RESULT **m2m_os_mslock_wait (**M2M_T_OS_MSLOCK **msLock,**
**UINT32 timeout,**LOCK_RESULT_T ***result)**

*Description*: the task calling this function is suspended until the msSemaphore is unlocked, destroyed or the timeout is expired. The granularity of the timeout used by the msSemaphore is one msec.

*Parameters*:
msLock: handle of msSemaphore
timeout: time to wait to getting a msSemaphore. Range: 0 ÷ 300 msec
result: pointer to the allocated variable that will be filled with the msSemaphore status, (output parameter). The used pointer can be **NULL**.

*Return value*:
refer to **M2M_API_RESULT** enum:

- **M2M_API_RESULT_SUCCESS**: the calling task gets the control because another task has unlocked the msSemaphore.
- **M2M_API_RESULT_FAIL**:
  - timeout is expired, result → **LOCK_TIMEOUTED**

- msSemaphore is destroyed, result → **LOCK_DESTROYED**
- fail to get msSemaphore when timeout was set to 0, result → **LOCK_NOT_GOT**
- OS error, result → **LOCK_ERROR**

## 10.20. m2m_os_pause_ms

M2M_API_RESULT **m2m_os_pause_ms (UINT32 timeout)**

*Description*: forces the current task in sleep mode, similar to **m2m_os_sleep _ms()**, but in this case with granularity of one ms: max waiting is 300 msec.

*Parameters*:
timeout: time to wait. Range 0-300 ms

*Return value*:
refer to **M2M_API_RESULT** enum

## 10.21. m2m_os_mslock_info

M2M_API_RESULT **m2m_os_mslock_info (**M2M_T_OS_MSLOCK **msLock,**
MS_SEM_STATE ***ms_state)**

*Description*: gets the info on the msSemaphore.

*Parameters*:
msLock: handle of the msSemaphore
ms_state: pointer to the structure that will be filled with the msSemaphore info, (output parameter).

*Return value*:
refer to **M2M_API_RESULT** enum:

- **M2M_API_RESULT_FAIL**, the output parameter indicates the failure reason:
  - count = **INVALID_COUNT**
  - count_suspended = **INVALID_SUSPENDED**

## 10.22. m2m_os_mslock_destroy

M2M_API_RESULT **m2m_os_mslock_destroy (**M2M_T_OS_MSLOCK ***msLock )**

*Description*: destroys the msSemaphore, and releases its resources.

*Parameters*:
msLock: this parameter works as input/output parameter. As input, it points to the handle of the msSemaphore to be destroyed. On success, the function returns **NULL**, on failure the input value is not changed.

*Return value*:
refer to **M2M_API_RESULT** enum

# Mutex Section

## 10.23. m2m_os_mtx_init

M2M_T_OS_MTX **m2m_os_mtx_init(UINT32 inheritance)**

*Description*: creates a mutex, sets its inheritance property, and return its handle. To have info on the granularity of the timeout used by mutex, refer to the **m2m_os_mtx_lock_wait()** function.

*Parameters*:

inheritance: sets the inheritance property of the mutex. **INHERITANCE_ENABLED**: the priority of the task, owner of the mtx, will be temporary elevated to the priority of the task waiting for the mtx, if any, and having the highest priority. With inheritance enabled, priority resumption is done automatically; therefore, it works as if **m2m_os_mtx_prioritize(...)** were automatically executed every time **m2m_os_mtx_unlock(...)** is called. It is used to avoid priority inversion. **INHERITANCE_DISABLED**: disables the inheritance property.

*Return value*:

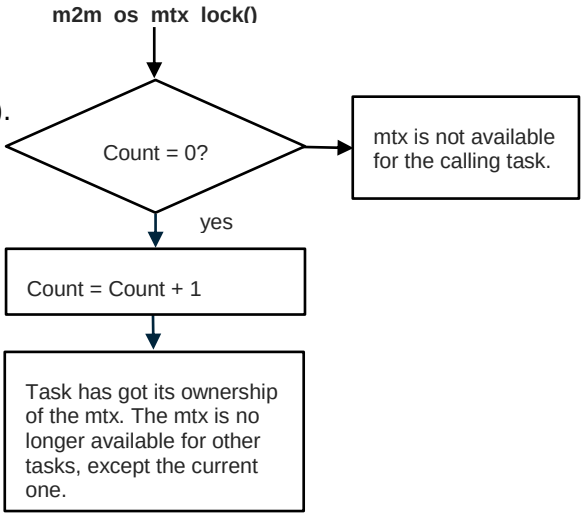
on success: pointer to the mutex handle;
on failure: **NULL**.

## 10.24. m2m_os_mtx_lock

M2M_API_RESULT **m2m_os_mtx_lock(**M2M_T_OS_MTX **mtx,**
LOCK_RESULT_T ***lock_res )**

*Description*: the task, calling this function, tries to get the ownership of the mtx. If the task has gained the ownership it can execute the critical code section as shown by the flow chart below.
The mtx is available for the generic task when the mtx counter value is zero, or the task has gained the ownership of the mtx.
If the mtx is not available for the calling task, the task will be blocked forever, waiting for mtx availability or deletion by another task.

*Parameters*:
mtx: mutex handle
lock_res: result of the lock (output parameter). It can be **NULL** if not needed, refer to **LOCK_RESULT_T**.




*Return value*:
refer to **M2M_API_RESULT** enum:

- **M2M_API_RESULT_SUCCESS**: the calling task has gained the ownership of the mtx.
- **M2M_API_RESULT_FAIL**, the output parameter indicates the failure reason:
  - mtx is destroyed, lock_res → **LOCK_DESTROYED**
  - fail to get mtx, lock_res → **LOCK_NOT_GOT**
  - OS error, lock_res → **LOCK_ERROR**

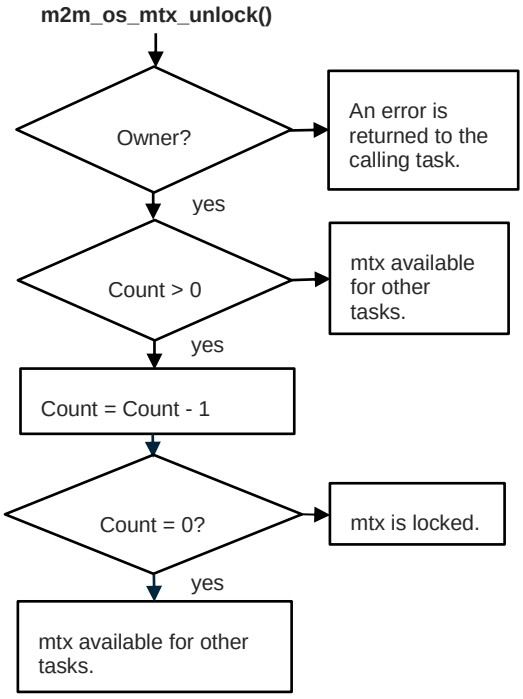
## 10.25. m2m_os_mtx_unlock

M2M_API_RESULT **m2m_os_mtx_unlock(**M2M_T_OS_MTX **mtx)**

*Description*: if the calling task owns the selected mtx, and the counter value is greater than zero the function decrements by one the counter value. If the counter value, after decreasing, is zero the mtx is unlocked for other tasks. Refer to the flow chart below.

*Parameters*:
mtx: mutex handle

*Return value*:
refer to **M2M_API_RESULT** enum




## 10.26. m2m_os_mtx_prioritize

M2M_API_RESULT **m2m_os_mtx_prioritize(**M2M_T_OS_MTX **mtx)**

*Description*: when a task is waiting on a mtx, it is suspended and inserted into a Suspending List associated with the mtx itself; refer to the table below on the left side. The function moves the task with the highest priority at the top of the Suspending List, see table on right side.

| Suspending List associated with a mtx | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_4 | 4 |
| Task_19 | 19 |

| Suspending List associated with a mtx | |
|---|---|
| **Tasks** | **Task's Priority: 1 = highest priority** |
| Task_4 | 4 |
| Task_7 | 7 |
| Task_24 | 24 |
| Task_19 | 19 |

*Parameters*:
mtx: mutex handle.

*Return value*:
refer to **M2M_API_RESULT**enum.

➢ 4G: Platform Version ID 23

This API does not have any effect for Platform Version ID 23. All tasks are performed according the priority set during the creation of each task. Following the previous example, tasks will be executed with this order: Task_4, Task_7, Task_19 and Task_24

## 10.27. m2m_os_mtx_lock_wait

M2M_API_RESULT **m2m_os_mtx_lock_wait(**M2M_T_OS_MTX **mtx, UINT32 timeout,** LOCK_RESULT_T ***lock_res)**

*Description*: the task, calling this function, tries to get the ownership of the mtx. If the task has gained the ownership it can execute the critical code section. If the mtx is not available for it, the task will wait for its availability until the expiration of the timeout or mtx deletion by another task.

*Parameters*:
mtx: mutex handle
timeout: expressed in msec. The resolution is 100 msec: 1 ÷ 100 => 100 ms, 101 ÷ 200 => 200ms, and so on.
lock_res: result of the lock (output parameter). It can be **NULL** if not needed, refer to **LOCK_RESULT_T**.

*Return value*:
refer to **M2M_API_RESULT** enum:

- **M2M_API_RESULT_SUCCESS**: the calling task has gained the ownership of the mtx.
- **M2M_API_RESULT_FAIL**, the output parameter indicates the failure reason:
  - timeout is expired, lock_res → **LOCK_TIMEOUTED**
  - mtx is destroyed, lock_res → **LOCK_DESTROYED**
  - fail to get mtx when timeout was set to 0, lock_res → **LOCK_NOT_GOT**
  - OS error, lock_res → **LOCK_ERROR**

## 10.28. m2m_os_mtx_info

M2M_API_RESULT **m2m_os_mtx_info(**M2M_T_OS_MTX **mtx, INT32 *mtx_count, INT32 *count_suspended)**

*Description*: gets info on the selected mutex.

*Parameters*:
mtx: mutex handle.

mtx_count: counter value of the mtx. 0 means mtx unlocked, (output parameter). It can be **NULL** if not needed.
count_suspended: number of tasks suspended on mtx, (output parameter). It can be **NULL** if not needed.

*Return value*:
refer to **M2M_API_RESULT** enum:

- **M2M_API_RESULT_FAIL**, the output parameter indicates the failure reason:
  - mtx_count → **INVALID_COUNT**
  - count_suspended → **INVALID_SUSPENDED**

```
Example:
.............
/* count_suspended is not requested */
m2m_os_mtx_info(lock, &mtx_count, 0)
.............
```

## 10.29. m2m_os_mtx_destroy

M2M_API_RESULT **m2m_os_mtx_destroy(**M2M_T_OS_MTX **mtx)**

*Description*: destroys the selected mtx, and releases all resources allocated for it. It is recommended to assign a **NULL** value to the mtx handle after deletion to prevent improper further usage.

*Parameters*:
mtx: mutex handle.

*Return value*:
refer to **M2M_API_RESULT** enum

# 11. M2M_SMS_API

Chapter 19.2.10 contains the declarations of the C identifiers used by the APIs set regarding the management of SMS messages.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# SMS Section

## 11.1. m2m_sms_enable_new_message_indication

**INT32 m2m_sms_enable_new_message_indication(void)**

*Description*: enables new message indication, see AT command "AT+CNMI" with <mt>=1, refer to document [3]. After enabling the message indication, every SMS received will trigger a call to the M2M_onMsgIndEvent() application callback function, refer to M2M_net.c file, document [1].

*Return value*:
on success: 1
on failure: 0

## 11.2. m2m_sms_disable_new_message_indication

**INT32 m2m_sms_disable_new_message_indication(void)**

*Description*: disables a new message indication, see AT command "AT+CNMI" with <mt>=0, refer to document [3].

*Return value*:
on success: 1
on failure: 0

## 11.3. m2m_sms_get_all_messages

**INT32 m2m_sms_get_all_messages(**M2M_T_SMS_INFO ****sms_info_list, INT32 *num_of_msg)**

*Description*: retrieves all SMS messages.

*Parameters*:
sms_info_list: pointer to the pointer pointing to the memory allocated and filled by the function. After function execution, it is caller responsibility to free the memory pointed by sms_info_list using the function **m2m_os_mem_free(sms_info_list)**;
num_of_msg: pointer to the allocated variable that will be filled with the number of available SMS messages.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success:
- "sms_info_list" points to the memory filled with the list of available SMS messages
- "num_of_msg" points to the number of available SMS messages

## 11.4. m2m_sms_get_text_message

**INT32 m2m_sms_get_text_message(INT32 index, M2M_T_SMS_INFO *sms_info)**

*Description*: retrieves an SMS message at the location of index.

Parameters:
index: message index to be retrieved;
sms_info: pointer to the allocated structure that will be filled with the SMS message information.

*Return value*:
on success: 1
on failure: 0

*Output data:*
on success: "sms_info" points to the allocated structure filled with the SMS message information.

## 11.5. m2m_sms_delete_message

**INT32 m2m_sms_delete_message(INT32 index)**

*Description*: deletes the selected SMS message.

*Parameters*:
index: identifies the message to be deleted.

*Return value*:
on success: 1
on failure: 0

## 11.6. m2m_sms_send_SMS

**INT32 m2m_sms_send_SMS(CHAR *address, CHAR *message)**

*Description*: sends an SMS message (coded in GSM default 7 bit alphabet, no Class, <pid>=0) to a specific address. SIM must be ready before using this function otherwise a failure indication is returned.

*Parameters*:
address: pointer to the zero-terminated string containing the phone number;
message: pointer to the zero-terminated string containing the SMS message to be sent.

*Return value*:
on success: 1
on failure: 0

## 11.7. m2m_sms_set_PDU_mode_format

**INT32 m2m_sms_set_PDU_mode_format(void)**

*Description*: sets SMS format in PDU mode that affects the following functions:

- m2m_sms_get_all_messages(…)
- m2m_sms_get_text_message(…)

*Return value*:

on success: 1
on failure: 0

## 11.8. m2m_sms_set_text_mode_format

**INT32 m2m_sms_set_text_mode_format(void)**

*Description*: sets SMS format to text mode that affects the following functions:

- m2m_sms_get_all_messages(…)
- m2m_sms_get_text_message(…**)**

*Return value*:

on success: 1
on failure: 0

## 11.9. m2m_sms_set_preferred_message_storage

**INT32 m2m_sms_set_preferred_message_storage(CHAR *memr, CHAR *memw, CHAR*mems)**

*Description*: selects the following memory storages as done by the AT command "AT+CPMS" described in the document [3]:

- <memr>: memory from which messages are read and deleted;
- <memw>: memory to which writing and sending operations are made;
- <mems>: memory to which received SMSs are preferred to be stored.

*Parameters:*

memr: pointer to the zero-terminated string containing "SM" or "ME";
memw: pointer to the zero-terminated string containing "SM" or "ME";
mems: pointer to the zero-terminated string containing "SM" or "ME".

**NOTE***:* "SM", "ME" available memories for SMSs, see document [3]. The three parameters supplied to the function must be equal: all "SM" or all "ME". It is not permitted to use parameters with different values.

*Return value*:

on success: 1
on failure: 0

➢ 2G: Platform Version ID 13

GE910 series does not support "ME" memory, see document [3].

*Examples*: 19.1.10 SMS Storage

## 11.10. m2m_sms_get_preferred_message_storage

**INT32 m2m_sms_get_preferred_message_storage(**M2M_T_SMS_MEM_STORAGE **mem_storages[])**

*Description*: gets the message storage status as done by the AT command "AT+CPMS?" described in the document [3]. <memr>, <memw>, <mems> - shown below - are the memory storage parameters of the "AT+CPMS" command.

*Parameters*:
mem_storages: is an allocated array of **M2M_T_SMS_MEM_STORAGE** elements. The size of the array must be 3, where:

- 1st array element will be filled with message storage status relating to <memr>
- 2nd array element will be filled with message storage status relating to <memw>
- 3rd array element will be filled with message storage status relating to <mems>

*Return value*:

on success: 1
on failure: 0

*Output data*:

on success:

- 1st array element is filled with message storage status relating to <memr>
- 2nd array element is filled with message storage status relating to <memw>
- 3rd array element is filled with message storage status relating to <mems>

*Examples*: 19.1.10 SMS Storage

# 12. M2M_SOCKET_API

Chapter 19.2.11 contains the declarations of the C identifiers used by the APIs set regarding the management of sockets.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Socket Creation/Closure/Options Section

## 12.1. m2m_socket_bsd_socket

M2M_SOCKET_BSD_SOCKET **m2m_socket_bsd_socket(INT32 domain, INT32 type, INT32 protocol)**

*Description*: creates an endpoint for communication and returns the associated socket handle. Up to 10 sockets can be active at a time. The function uses the PDP context dedicated to AppZone.

*Parameters*:
domain: Socket_Address_Families, for example: M2M_SOCKET_BSD_AF_INET
type: Socket_Types, for example: M2M_SOCKET_BSD_SOCK_STREAM
protocol: Socket_Protocols, for example: M2M_SOCKET_BSD_IPPROTO_IP

*Return value*:
on success: socket handle
on failure: refer to Invalid_Socket_handle

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server, 19.1.14 UDP-Client and 19.1.15 UDP-Server

## 12.2. m2m_socket_bsd_socket_cid

M2M_SOCKET_BSD_SOCKET **m2m_socket_bsd_socket_cid(INT32 domain, INT32 type, INT32 protocol, UINT8 cid)**

*Description*: creates an endpoint for communication and returns the associated socket handle. Up to 10 sockets can be active at a time. The function uses the specified PDP context.

*Parameters*:
domain: Socket_Address_Families, for example: M2M_SOCKET_BSD_AF_INET
type: Socket_Types, for example: M2M_SOCKET_BSD_SOCK_STREAM
protocol: Socket_Protocols, for example: M2M_SOCKET_BSD_IPPROTO_IP
cid: PDP context identifier.

*Return value*:
on success: socket handle
on failure: refer to Invalid_Socket_handle

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.3. m2m_socket_bsd_close

**INT32 m2m_socket_bsd_close(**M2M_SOCKET_BSD_SOCKET **s)**

*Description*: closes the specified socket, and releases the resources allocated to the socket. In case of TCP socket, it also terminates the connection.

*Parameters*:
s: socket handle

*Return value*:
on success: 0
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server, 19.1.14 UDP-Client and 19.1.15 UDP-Server

## 12.4. m2m_socket_bsd_socket_state

**INT32 m2m_socket_bsd_socket_state(**M2M_SOCKET_BSD_SOCKET **s)**

*Description*: returns the state of the specified socket.

*Parameters*:
s: socket handle

*Return value*:
refer to Socket_State

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.5. m2m_socket_bsd_set_sock_opt

**INT32 m2m_socket_bsd_set_sock_opt(**M2M_SOCKET_BSD_SOCKET **s, INT32 level, INT32 optname, const void *optval, INT32 optlen)**

*Description*: sets a socket option for the specified socket.

*Parameters*:
s: socket handle
level: Socket_Protocols, for example: M2M_SOCKET_BSD_IPPROTO_IP; or Level_number: M2M_SOCKET_BSD_SOL_SOCKET
optname: Socket_Option_Flags, for example: M2M_SOCKET_BSD_SO_DEBUG
optval: pointer to the allocated buffer containing the Socket Option value to be set
optlen: length of the buffer pointed by "optval".

*Return value*:
on success: 0
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.13 TCP-Server,
19.1.15 UDP-Server,
19.1.18 M2M_SOCKET_BSD_TCP_CONNTIME Option,
19.1.19 M2M_SOCKET_BSD_TCP_KEEPALIVE Option.

## 12.6. m2m_socket_bsd_get_sock_opt

**INT32 m2m_socket_bsd_get_sock_opt(**M2M_SOCKET_BSD_SOCKET **s, INT32 level,INT32 optname, void *optval, INT32 *optlen)**

*Description*: returns the current value of a socket option for the specified socket.

*Parameters*:
s: socket handle
level: Socket_Protocols, for example: M2M_SOCKET_BSD_IPPROTO_IP; or Level_number: M2M_SOCKET_BSD_SOL_SOCKET
optname: Socket_Option_Flags, for example: M2M_SOCKET_BSD_SO_DEBUG
optval: pointer to the allocated buffer that will be filled with the Socket Option value
Output data:
on success:
buffer contains Socket Option value of the specified socket.
optlen: pointer to the allocated variable which is a value-result argument:
Input data:
variable contains the size of the buffer pointed by "optval"
Output data:
on success:
variable contains the actual size of the value returned.

*Return value*:
on success: 0
on failure: < 0

➢ 2G: Platform Version ID 13

| | M2M_SOCKET_BSD_SO_KEEPALIVE, refer to chapter 19.2.11 | |
|---|---|---|
| | Option disabled | Option enabled |
| optval returned→ | 0 | 1 |

| | M2M_SOCKET_BSD_TCP_NODELAY, refer to chapter 19.2.11 | |
|---|---|---|
| | Option disabled | Option enabled |
| optval returned→ | 0 | 1 |

➢ 3G/4G: Platform Version ID 12, 20, 23

| | M2M_SOCKET_BSD_SO_KEEPALIVE, refer to chapter 19.2.11 | |
|---|---|---|
| | Option disabled | Option enabled |
| optval returned→ | 0 | ≠ 0 |

| | M2M_SOCKET_BSD_TCP_NODELAY, refer to chapter 19.2.11 | |
|---|---|---|
| | Option disabled | Option enabled |
| optval returned→ | 0 | ≠ 0 |

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.7. m2m_socket_bsd_accept

M2M_SOCKET_BSD_SOCKET
**m2m_socket_bsd_accept(**M2M_SOCKET_BSD_SOCKET **s,**
M2M_SOCKET_BSD_SOCKADDR ***addr,**
**INT32 *addrlen)**

*Description*: permits an incoming connection attempt on the specified socket. The function is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, creates a new socket associated with the socket address pair of this connection.

*Parameters*:

s: socket handle;

addr: pointer to the allocated address structure used by TCP/IP stack that will be filled with address /port /protocol accepted, see M2M_SOCKET_BSD_SOCKADDR_IN structure. Cast to M2M_SOCKET_BSD_SOCKADDR structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function.
Output data:
on success:
the structure contains address/port/protocol accepted;

addrlen: pointer to the variable that will be filled with the size of the address/port/protocol accepted, see **M2M_SOCKET_BSD_SOCKADDR_IN** structure.
Output data:
on success:
the variable contains the size of address/port/protocol accepted.

*Return value*:

on success: socket handle;
on failure: refer to Invalid_Socket_handle.

**NOTE**: **m2m_socket_errno(…)** function returns the failure reason.

*Example*: 19.1.13 TCP-Server

## 12.8. m2m_socket_bsd_shutdown

**INT32 m2m_socket_bsd_shutdown(**M2M_SOCKET_BSD_SOCKET **s, INT32 how)**

*Description*: is a dummy function, no actions.

*Parameters*:

s: not used;

how: not used.

*Return value*: 0

# Address Conversion Section

## 12.9. m2m_socket_bsd_addr_str

**CHAR *m2m_socket_bsd_addr_str(UINT32 ipAddr)**

*Description*: converts the IPv4 address in UINT32 number format into string format.

*Parameters*:
ipAddr: IPv4 address to be converted into string format.

*Return value*:
on success: pointer to a static buffer holding the zero-terminated string of the IPv4 address. The static buffer is over written with each call to the function.
on failure: **NULL**.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

```
Example:

.............
UINT32 MY_IP_inet_address;
.............
MY_IP_inet_address = m2m_pdp_get_my_ip();
PrintToUart("IPv4 String: %s", m2m_socket_bsd_addr_str(MY_IP_inet_address));
.............
```

## 12.10. m2m_socket_bsd_addr_str_ip6

**CHAR *m2m_socket_bsd_addr_str_ip6(**M2M_SOCKET_BSD_IPV6_ADDR ***ipAddr)**

*Description*: converts the IPv6 address value into string format.

*Parameters*:
ipAddr: IPv6 address to be converted into string format.

*Return value*:
on success: pointer to a static buffer holding the zero-terminated string of the IPv6 address. The static buffer is over written with each call to the function.
on failure: **NULL**.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.11. m2m_socket_bsd_inet_addr

**UINT32 m2m_socket_bsd_inet_addr(const CHAR *ip_addr_str)**

*Description*: converts the IPv4 address string into a UINT32 number.

*Parameters*:
ip_addr_str: pointer to a zero-terminated string containing the IPv4 address string to be converted into UINT32.

*Return value*:
on success: IPv4 address string converted into UNIT32;
on failure: 0.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client, 19.1.14 UDP-Client

## 12.12. m2m_socket_bsd_inet_addr_ip6

**INT32 m2m_socket_bsd_inet_addr_ip6(const CHAR *ip_addr_str,**
**M2M_SOCKET_BSD_IPV6_ADDR *ipAddr)**

*Description*: converts the IPv6 address string into a **M2M_SOCKET_BSD_ IPV6_ADDR** structure.

*Parameters*:
ip_addr_str: pointer to a zero-terminated string containing the IPv6 address string to be converted into a **M2M_SOCKET_BSD_ IPV6_ADDR** structure.
ipAddr: pointer to the allocated **M2M_SOCKET_BSD_ IPV6_ADDR** structure that will be filled with IPv6 address.
Output data:
on success: the **M2M_SOCKET_BSD_ IPV6_ADDR** structure contains IPv6 address.

*Return value*:
on success: 0
on failure: -1

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.13. m2m_socket_bsd_connect

**INT32 m2m_socket_bsd_connect(**M2M_SOCKET_BSD_SOCKET **s, const** M2M_SOCKET_BSD_SOCKADDR ***name, INT32 namelen)**

*Description*: establishes a connection to the specified address. The connect function is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

*Parameters:*

s: socket handle;

name: pointer to the allocated structure filled with the address/port/protocol to connect to, see M2M_SOCKET_BSD_SOCKADDR_IN structure. Cast to **M2M_SOCKET_BSD_SOCKADDR** structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function;

namelen: size of **M2M_SOCKET_BSD_SOCKADDR_IN** structure.

*Return value*:

on success: 0

on failure: < 0

**NOTE**:

I. m2m_socket_errno(…) function returns the failure reason.

II. In case of ERROR, it is strongly recommended to close the socket (with **m2m_socket_bsd_close()** ) and create a new one ( with **m2m_socket_bsd_socket()**).

*Example*: 19.1.12 TCP-Client

## 12.14. m2m_socket_bsd_bind

**INT32 m2m_socket_bsd_bind(**M2M_SOCKET_BSD_SOCKET **s,** M2M_SOCKET_BSD_SOCKADDR ***name, INT32 namelen)**

*Description*: binds the address with the socket. This function is typically used on the server side, associates a socket with a socket address structure that is a specified local port number and IP address.

*Parameters*:

s: socket handle;

name: pointer to the allocated socket address structure, in Internet style, containing the address/port/protocol to bind to, see struct M2M_SOCKET_BSD_SOCKADDR_IN. Cast to **M2M_SOCKET_BSD_SOCKADDR** structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function.

namelen: the allocated variable containing the size of the **M2M_SOCKET_BSD_SOCKADDR** structure.

*Return value*:
on success: 0
on failure: < 0

**NOTE**: **m2m_socket_errno(…)** function returns the failure reason.

*Examples*: 19.1.13 TCP-Server, 19.1.15 UDP-Server

## 12.15. m2m_socket_bsd_select

**INT32 m2m_socket_bsd_select(INT32 nfds,** M2M_SOCKET_BSD_FD_SET ***readfds,**
**M2M_SOCKET_BSD_FD_SET *writefds,**
**M2M_SOCKET_BSD_FD_SET *exceptfds,**
**const** M2M_SOCKET_BSD_TIMEVAL ***timeout)**

*Description*: returns the sockets number matching the following criteria: ready to read, write, and having errors. See also the following functions:

- m2m_socket_bsd_fd_set_func(...),
- m2m_socket_bsd_fd_clr_func(...),
- m2m_socket_bsd_fd_isset_func(...),
- m2m_socket_bsd_fd_zero_func(...).

*Parameters*:

| | |
|---|---|
| nfds: | number of sockets on which information is required; |
| readfds: | pointer to the allocated structure for sockets ready to read. The function verifies for each socket with setting value set to **TRUE** if it is ready to read. If affirmative, the setting value is not changed. If the socket is not ready to read, the value is set to **FALSE**; |
| writefds: | pointer to the allocated structure for sockets ready to write, not supported; |
| exceptfds: | pointer to the allocated structure for sockets having errors, not supported; |
| timeout: | pointer to the allocated structure to set the timeout.<br>If timeout = 0, the function returns immediately the control to the calling task.<br>If the timeout pointer is **NULL**, timeout is: WAIT FOREVER. |

*Return value*:
on success: number of sockets matching the criteria;
on Timeout: 0
on failure: < 0

*Output data:*
on success:
"readfds" is the pointer to the allocated structure for sockets ready to read.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.16. m2m_socket_bsd_fd_set_func

**void m2m_socket_bsd_fd_set_func(INT32 fd,** M2M_SOCKET_BSD_FD_SET ***set)**

*Description*: sets the selected socket to **TRUE** in the selected list.

*Parameters:*
fd: socket handle;
set: pointer to one of the following allocated list:
- sockets ready to read
- sockets ready to write, not supported
- sockets having errors, not supported

**NOTE***:* refer to m2m_socket_bsd_select(…) function.

## 12.17. m2m_socket_bsd_fd_clr_func

**void m2m_socket_bsd_fd_clr_func(INT32 fd,** M2M_SOCKET_BSD_FD_SET ***set)**

*Description*: sets the selected socket to **FALSE** in the selected list.

*Parameters:*
fd: socket handle;
set: pointer to one of the following allocated list:
- sockets ready to read
- sockets ready to write, not supported
- sockets having errors, not supported

**NOTE***:* refer to m2m_socket_bsd_select(…) function.

## 12.18. m2m_socket_bsd_fd_isset_func

**UINT8 m2m_socket_bsd_fd_isset_func(INT32 fd,** M2M_SOCKET_BSD_FD_SET ***set)**

*Description*: returns the setting value of the selected socket. The setting value is stored in the selected list.

*Parameters:*
fd: socket handle;
set: pointer to one of the following allocated list:
- sockets ready to read
- sockets ready to write, not supported
- sockets having errors, not supported

*Return value*:
setting value of the selected socket in the selected list: **TRUE** (1) or **FALSE** (0).

**NOTE***:* refer to m2m_socket_bsd_select(…) function.

## 12.19. m2m_socket_bsd_fd_zero_func

**void m2m_socket_bsd_fd_zero_func(**M2M_SOCKET_BSD_FD_SET ***set)**

*Description*: sets all sockets to **FALSE** in the selected list.

*Parameters:*
fd: socket handle;
set: pointer to one of the following allocated list:

- sockets ready to read
- sockets ready to write, not supported
- sockets having errors, not supported

**NOTE***:* refer to m2m_socket_bsd_select(…) function.

# Domain Name Conversion Section

## 12.20. m2m_socket_bsd_get_host_by_name

**UINT32 m2m_socket_bsd_get_host_by_name(const CHAR *domain_name)**

*Description*: converts the domain name into the host's IPv4 address. The function uses the AppZone dedicated PDP context.

*Parameters*:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into the host's IPv4 address.

*Return value*:
on success: IPv4 address converted into UINT32 number format;
on failure: 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.21. m2m_socket_bsd_get_host_by_name_cid

**UINT32 m2m_socket_bsd_get_host_by_name_cid(const CHAR *domain_name, UNIT8 cid)**

*Description*: converts the domain name into the host's IPv4 address.

*Parameters*:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into the host's IPv4 address.
cid: PDP context identifier.

*Return value*:
on success: IPv4 address converted into UINT32 number format;
on failure: 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.22. m2m_socket_bsd_get_host_by_name_ip6

**INT32 m2m_socket_bsd_get_host_by_name_ip6(const CHAR *domain_name, M2M_SOCKET_BSD_IPV6_ADDR *ipAddr)**

*Description*: converts the domain name into the host's IPv6 address. The function uses the AppZone dedicated PDP context.

*Parameters*:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into the host's IPv6 address;
ipAddr: pointer to the allocated **M2M_SOCKET_BSD_ IPV6_ADDR** structure that will be filled with the host's IPv6 address.

*Return value*:
on success: 0
on failure: -1
*Output data:*
on success: "ipAddr" points to the **M2M_SOCKET_BSD_ IPV6_ADDR** structure filled with the host's IPv6 address.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.23. m2m_socket_bsd_get_host_by_name_ip6_cid

**INT32 m2m_socket_bsd_get_host_by_name_ip6_cid(const CHAR *domain_name, M2M_SOCKET_BSD_IPV6_ADDR *ipAddr, UNIT8 cid)**

*Description*: converts the domain name into the host's IPv6 address.

*Parameters*:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into the host's IPv6 address;
ipAddr: pointer to the allocated **M2M_SOCKET_BSD_ IPV6_ADDR** structure that will be filled with the host's IPv6 address.
cid: PDP context identifier.

*Return value*:
on success: 0
on failure: -1

*Output data:*
on success: "ipAddr" points to the **M2M_SOCKET_BSD_ IPV6_ADDR** structure filled with the host's IPv6 address.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.24. m2m_socket_bsd_get_peer_name

**INT32 m2m_socket_bsd_get_peer_name(M2M_SOCKET_BSD_SOCKET s, M2M_SOCKET_BSD_SOCKADDR *name, INT32 *namelen)**

*Description*: returns the peer name (address, port, and so on).

*Parameters*:
s: socket handle;
name: pointer to the allocated structure that will be filled with address/port/protocol;
namelen: pointer to the allocated variable that will be filled with the size of name parameter.

*Return value*:
on success: 0
on failure: < 0

*Output data*:
on success:

"name" points to the structure filled with address/port/protocol;
"namelen" points to the variable filled with the size of name parameter.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.25. m2m_socket_bsd_get_sock_name

**INT32 m2m_socket_bsd_get_sock_name(**M2M_SOCKET_BSD_SOCKET **s,**
M2M_SOCKET_BSD_SOCKADDR ***name, INT32 *namelen)**

*Description*: returns the local address of the socket (address, port, and so on).

*Parameters:*

| | |
|---|---|
| s: | socket handle; |
| name: | pointer to the allocated structure that will be filled with address/port/protocol of the socket; |
| namelen: | pointer to the allocated variable that will be filled with the size of "name" parameter. |

*Return value*:
on success: 0
on failure: < 0

*Output data*:
on success:
"name" points to the structure filled with address/port/protocol of the socket;
"namelen" points to the variable filled with the size of "name" parameter.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client

# DNS Section

## 12.26. m2m_socket_bsd_get_dns_ip

**INT32 m2m_socket_bsd_get_dns_ip(** M2M_SOCKET_BSD_IN_ADDR ***pDNSAddr, M2M_SOCKET_BSD_IN_ADDR *sDNSAddr)**

*Description*: gets primary and secondary DNS IPv4 address of the AppZone dedicated PDP connection.

*Parameters:*

pDNSAddr: points to the **M2M_SOCKET_BSD_IN_ADDR** structure that will be filled with primary DNS IPv4 address.
sDNSAddr: points to the **M2M_SOCKET_BSD_IN_ADDR** structure that will be filled with secondary DNS IPv4 address.

*Return value*:
on success: 0
on failure: -1

*Output data*:
on success:
- pDNSAddr points to the **M2M_SOCKET_BSD_IN_ADDR** structure filled with primary DNS IPv4 address (all zeros if not present);
- sDNSAddr points to the **M2M_SOCKET_BSD_IN_ADDR** structure filled with secondary DNS IPv4 address (all zeros if not present);

**NOTE**: m2m_socket_errno(…) function not supported.

## 12.27. m2m_socket_bsd_get_dns_ip_cid

**INT32 m2m_socket_bsd_get_dns_ip_cid(** M2M_SOCKET_BSD_IN_ADDR ***pDNSAddr, M2M_SOCKET_BSD_IN_ADDR *sDNSAddr, UINT8 cid)**

*Description*: gets primary and secondary DNS IPv4 address of the selected PDP connection.

*Parameters:*

pDNSAddr: points to the **M2M_SOCKET_BSD_IN_ADDR** structure that will be filled with primary DNS IPv4 address.
sDNSAddr: points to the **M2M_SOCKET_BSD_IN_ADDR** structure that will be filled with secondary DNS IPv4 address.
cid: PDP context identifier.

*Return value*:
on success: 0
on failure: -1

*Output data*:
on success:

- pDNSAddr points to the **M2M_SOCKET_BSD_IN_ADDR** structure filled with primary DNS IPv4 address (all zeros if not present);
- sDNSAddr points to the **M2M_SOCKET_BSD_IN_ADDR** structure filled with secondary DNS IPv4 address (all zeros if not present);

**NOTE**: m2m_socket_errno(…) function not supported.

➢ 2G: Platform Version ID 13

Not supported by GE910 series.

## 12.28. m2m_socket_bsd_get_dns_ip6

**INT32 m2m_socket_bsd_get_dns_ip6(**M2M_SOCKET_BSD_IN6_ADDR ***pDNSAddr, M2M_SOCKET_BSD_IN6_ADDR *sDNSAddr)**

*Description*: gets primary and secondary DNS IPv6 address of the AppZone dedicated PDP connection.

*Parameters:*

pDNSAddr: points to the **M2M_SOCKET_BSD_IN6_ADDR** structure that will be filled with primary DNS IPv6 address
sDNSAddr: points to the **M2M_SOCKET_BSD_IN6_ADDR** structure that will be filled with secondary DNS IPv6 address.

*Return value*:
on success: 0
on failure: -1

*Output data*:
on success:
- pDNSAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with primary DNS IPv6 address (all zeros if not present);
- sDNSAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with secondary DNS IPv6 address (all zeros if not present);

**NOTE**: **m2m_socket_errno(…)** function not supported

## 12.29. m2m_socket_bsd_get_dns_ip6_cid

**INT32 m2m_socket_bsd_get_dns_ip6_cid(**M2M_SOCKET_BSD_IN6_ADDR ***pDNSAddr, M2M_SOCKET_BSD_IN6_ADDR *sDNSAddr, UNIT8 cid)**

*Description*: gets primary and secondary DNS IPv6 address of the selected PDP connection.

*Parameters:*

pDNSAddr: points to the **M2M_SOCKET_BSD_IN6_ADDR** structure that will be filled with primary DNS IPv6 address

sDNSAddr: points to the **M2M_SOCKET_BSD_IN6_ADDR** structure that will be filled with secondary DNS IPv6 address.
cid: PDP context identifier.

*Return value*:
on success: 0
on failure: -1

*Output data*:
on success:
- pDNSAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with primary DNS IPv6 address (all zeros if not present);
- sDNSAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with secondary DNS IPv6 address (all zeros if not present);

**NOTE**: **m2m_socket_errno(…)** function not supported

➢ 2G: Platform Version ID 13

Not supported by GE910 series.

# Network and Host byte order Section

## 12.30. m2m_socket_bsd_htonl

**UINT32 m2m_socket_bsd_htonl(UINT32 x)**

*Description*: converts the unsigned integer x from host byte order to network byte order.

## 12.31. m2m_socket_bsd_htons

**UINT16 m2m_socket_bsd_htons(UINT16 x)**

*Description*: converts the unsigned short integer x from host byte order to network byte order.

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server, 19.1.14 UDP-Client and 19.1.15 UDP-Server

## 12.32. m2m_socket_bsd_ntohl

**UINT32 m2m_socket_bsd_ntohl(UINT32 x)**

*Description*: converts the unsigned integer x from network byte order to host byte order.

## 12.33. m2m_socket_bsd_ntohs

**UINT16 m2m_socket_bsd_ntohs(UINT16 x)**

*Description*: converts the unsigned short integer x from network byte order to host byte order.

*Examples*: 19.1.12 TCP-Client

## 12.34. m2m_socket_bsd_ioctl

**INT32 m2m_socket_bsd_ioctl(**M2M_SOCKET_BSD_SOCKET **s, INT32 cmd, void *argp)**

*Description*: sets the specified socket in blocking or non-blocking mode, or gets the size of data available on the socket. A blocking socket does not return control until it has sent (or received) some or all data specified for the operation. If a blocking socket continues to listen, a user program hangs until some data arrives or internal timeout expires. A non-blocking socket returns whatever is in the receive buffer and immediately continues.

*Parameters*:
s: socket handle
cmd: command, currently are supported the following commands:
- **M2M_SOCKET_BSD_FIONREAD**: gets the number of bytes to read
- **M2M_SOCKET_BSD_FIONBIO**: sets blocking or non-blocking mode

argp: if cmd = **M2M_SOCKET_BSD_FIONREAD**:
"argp" is the pointer to the allocated variable that will be filled with the number of the data to be read.
Output data:
on success: the allocated variable contains the number of the data to be read.

if cmd = **M2M_SOCKET_BSD_FIONBIO**:
"argp" is the pointer to the allocated variable that selects the blocking or non-blocking mode:
- argp=1: sets non-blocking
- argp=0: sets blocking

*Return value*:
on success: 0
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.11 Socket ioctl

## 12.35. m2m_socket_bsd_listen

**INT32 m2m_socket_bsd_listen(**M2M_SOCKET_BSD_SOCKET **s, INT32 backlog)**

*Description*: places the socket in a listening state for an incoming connection. Listen function is used on the server side, and causes a bound TCP socket to enter listening state.

*Parameters*:
s: socket handle
backlog: total number of connections allowed on the specified socket.

*Return value*:
on success: 0
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.13 TCP-Server.

## 12.36. m2m_socket_bsd_recv

**INT32 m2m_socket_bsd_recv(**M2M_SOCKET_BSD_SOCKET **s, void *buf, INT32 len, INT32 flags)**

*Description*: receives data on the specified socket. Depending on the socket configuration, this function can perform in blocking or non-blocking mode. In case of blocking, it will not return the control until data is received (not necessarily all requested data). In case of non-blocking, the function will return the data pending on the specified socket, and will not wait for additional data in case the total number of characters is not reached.

*Parameters*:
s: socket handle
buf: pointer to the allocated buffer that will be filled with the data received.
Output data:
on success:
the buffer contains the data received.
len: total number of characters to be received in the allocated buffer. The size of the allocated buffer should be larger than "len" value.
flags: set to 0. Not supported, ignored.

*Return value*:
on success: number of bytes received.
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server

## 12.37. m2m_socket_bsd_recv_data_size

**INT32 m2m_socket_bsd_recv_data_size(**M2M_SOCKET_BSD_SOCKET **s, UINT32 *len)**

*Description*: retrieves the size of the data pending on the specified socket.

*Parameters*:
s: socket handle;
len: pointer to the allocated variable that will be filled with the size of the data pending.
Output data:
on success:
the variable contains the size of the data pending.

*Return value*:
on success: 0
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

## 12.38. m2m_socket_bsd_recv_from

**INT32 m2m_socket_bsd_recv_from(**M2M_SOCKET_BSD_SOCKET **s, void *buf, INT32 len,INT32 flags,** M2M_SOCKET_BSD_SOCKADDR ***from, INT32 *fromlen)**

*Description*: receives the data on the specified socket and stores the source address. This function is used only for datagram sockets. Depending on the socket configuration, this function can perform in blocking or non-blocking mode. In case of blocking mode, the function will not return the control until data is received (not necessarily all requested data). In case of non-blocking mode, it will return the data pending on the specified socket, but will not wait for additional data in case the size requested is not reached.

*Parameters*:
s: socket handle
buf: pointer to the allocated buffer that will be filled with the data received.
Output data:
on success:
the buffer contains the data received.
len: expected number of characters to be received in the allocated buffer.
flags: set to 0. Not supported, ignored.
from: pointer to the allocated address structure used by TCP/IP stack filled with the address received, see M2M_SOCKET_BSD_SOCKADDR_IN structure. Cast to **M2M_SOCKET_BSD_SOCKADDR** structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function.
fromlen: pointer to the allocated variable that will be filled with the size of the structure of socket address.
Output data:
on success:
the variable contains the size of the socket address

*Return value*:

on success: number of bytes received.
on failure: < 0

**NOTE**: **m2m_socket_errno(…)** function returns the failure reason.

*Examples*: 19.1.15 UDP-Server

## 12.39. m2m_socket_bsd_send

**INT32 m2m_socket_bsd_send(**M2M_SOCKET_BSD_SOCKET **s, const void *buf, INT32 len, INT32 flags)**

*Description*: sends data using the specified socket. Depending on the socket configuration, this function can perform in blocking or non-blocking mode. In case of blocking mode, the function waits for the availability of all the needed stack resources and will not return the control until data is sent (not necessarily all requested data). In case of non-blocking mode, it will try to send the data using only the available stack resources at that time, and will not wait for resources to be free.

*Parameters*:
s: socket handle;
buf: pointer to the allocated buffer filled with the data to be sent.
len: total number of characters written in the allocated buffer. The size of the buffer should be larger enough to contain the data.
flags: set to 0. Not supported, ignored.

*Return value*:
on success: number of bytes sent.
on failure: < 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server

## 12.40. m2m_socket_bsd_send_buf_size

**UINT32 m2m_socket_bsd_send_buf_size(**M2M_SOCKET_BSD_SOCKET **s)**

*Description*: returns the available buffer space (in bytes) for sending on the specified sockets. The function supports TCP socket only. UDP socket will always return 0.

*Parameters*:
s: socket handle

*Return value*:
on success: total number of bytes available for sending on the specified socket.
on failure: 0

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

➢ 4G: Platform Version ID 23

For the products with Platform Version ID 23, the function supports TCP and UDP socket

*Examples*: 19.1.12 TCP-Client, 19.1.13 TCP-Server

## 12.41. m2m_socket_bsd_send_to

**INT32 m2m_socket_bsd_send_to(**M2M_SOCKET_BSD_SOCKET **s, const void *buf, INT32 len, INT32 flags, const** M2M_SOCKET_BSD_SOCKADDR ***to, INT32 tolen)**

*Description*: sends data on the specified socket to the specified address. This function is used only for datagram sockets. Depending on the socket configuration, this function can perform in blocking or non-blocking mode. In case of blocking mode, the function waits for the availability of all the needed stack resources and will not return the control until data is sent (not necessarily all requested data). In case of non-blocking mode, it will try to send the data using only the available stack resources at that time, and will not wait for resources to be free.

*Parameters*:
s: socket handle
buf: pointer to the allocated buffer filled with data to be sent.
len: total number of characters written in the allocated buffer. The size of the buffer should be larger enough to contain the data.
flags: set to 0. Not supported, ignored.
to: pointer to the allocated address structure used by TCP/IP stack filled with the destination address, see M2M_SOCKET_BSD_SOCKADDR_IN structure. Cast to **M2M_SOCKET_BSD_SOCKADDR** structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function.
tolen: size of the structure of socket address in Internet style, see **M2M_SOCKET_BSD_SOCKADDR_IN** structure.

*Return value*:
on success: number of bytes sent.
on failure: < 0.

**NOTE**: m2m_socket_errno(…) function returns the failure reason.

*Examples*: 19.1.14 UDP-Client

## 12.42. m2m_socket_errno

**INT32 m2m_socket_errno(void)**

*Description*: returns the error code of the last "socket" operation.

*Return value*:
on success: 0.
on failure: refer to Socket_Error_Types.

# Firewall Section

## 12.43. m2m_firewall

**UINT32 m2m_firewall(UINT16 enable,** M2M_FIREWALL_ELEMENT **frw_rule)**

*Description*: clears or sets the rules contained in the IPv4 firewall list; the couple of an IPv4 address and a mask build a rule. The firewall monitors and controls the incoming connections.

*Parameters*:

| | | |
|---|---|---|
| enable: | 0 | Clear from the list the rule used in the function. |
| | 1 | Set in the list the rule used in the function. |
| | 2 | Clear the entire IPv4 firewall rule list. Use a dummy rule in the function. |
| | 3 | Enable firewall and save this setting in NVM (*). |
| | 4 | Disable firewall and save this setting in NVM (*). |
| frw_rule: | structure of the IPv4 firewall rule. | |

*Return value*:

| | |
|---|---|
| on success: | 1 |
| on failure: | 0 |

```
Example:
.............
M2M_FIREWALL_ELEMENT fw_rules_ipv4[5];
UINT32 Result;
.............
fw_rules_ipv4[2].ipAddr = m2m_socket_bsd_inet_addr("213.82.124.164");
fw_rules_ipv4[2].ipMask = m2m_socket_bsd_inet_addr("255.255.255.255");

.............
Result = m2m_firewall(1, fw_rules_ipv4[2]);
```

**NOTE**: when a PDP context is created using the **m2m_pdp_activate** API, the firewall is opened and the IPv4 firewall list contains the rule 0.0.0.0/0.0.0.0. Here are the steps to set the firewall: clear all, then set the rules.

➢ 4G: Platform Version ID 20, 23

(*) <enable> parameter values 3, and 4 (default) are supported only by modules having platform versions ID 20 and 23.

## 12.44. m2m_firewall_list

**UINT32 m2m_firewall_list(**M2M_FIREWALL_ELEMENT**** frw_list, UINT16* size)**

*Description*: displays the entire IPv4 firewall list.

*Parameters*:

| | |
|---|---|
| frw_list: | is a pointer to the pointer of the IPv4 firewall list. The function allocates the needed memory. It is responsibility of the programmer to deallocate it. |
| size: | number of rules contained in the IPv4 firewall list. |

*Return value*:

on success: 1
on failure: 0

```
Example:
.............
M2M_FIREWALL_ELEMENT fw_rules_ipv4[5];
M2M_FIREWALL_ELEMENT *firewall_test;
UINT32 Result;
UINT16 size;
.............
fw_rules_ipv4[2].ipAddr = m2m_socket_bsd_inet_addr("213.82.124.164");
fw_rules_ipv4[2].ipMask = m2m_socket_bsd_inet_addr("255.255.255.255");

.............
Result = m2m_firewall(1, fw_rules_ipv4[2]);
.............
Result = m2m_firewall_list(&firewall_test, &size);
.............
```

## 12.45. m2m_firewall_ip6

**UINT32 m2m_firewall_ip6(UINT16 enable,** M2M_FIREWALL_ELEMENT_IP6 **frw_rule)**

*Description*: clears or sets the rules contained in the IPv6 firewall list; the couple of an IPv6 address and a mask build a rule. The firewall monitors and controls the incoming connections.

*Parameters*:

| | | |
|---|---|---|
| enable: | 0 | Clear from the list the rule used in the function. |
| | 1 | Set in the list the rule used in the function. |
| | 2 | Clears the entire IPv6 firewall rule list. Use a dummy rule in the function. |
| | 3 | Enable firewall and save this setting in NVM (*). |
| | 4 | Disable firewall and save this setting in NVM (*). |
| frw_rule: | structure of the IPv6 firewall rule. | |

*Return value*:

on success: 1
on failure: 0

**NOTE**: when a PDP context is created using the **m2m_pdp_activate_ip6** API, the firewall is opened and the IPv6 firewall list contains the rule 0:0:0:0:0:0:0:0/0:0:0:0:0:0:0:0. Here are the steps to set the firewall: clear all, then set the rules.

- 4G: Platform Version ID 20, 23

(*) <enable> parameter values 3, and 4 (default) are supported only by modules having platform versions ID 20 and 23.

## 12.46. m2m_firewall_ip6_list

**UINT32 m2m_firewall_ip6_list(**M2M_FIREWALL_ELEMENT_IP6 **** frw_list, UINT16* size)**

*Description*: displays the entire IPv6 firewall list.

*Parameters*:

| | |
|---|---|
| frw_list: | is a pointer to the pointer of the IPv6 firewall list. The function allocates the needed memory. It is responsibility of the programmer to deallocate it. |
| size: | number of rules contained in the IPv6 firewall list |

*Return value*:

| | |
|---|---|
| on success: | 1 |
| on failure: | 0 |

# PDP Section

## 12.47. m2m_pdp_apn_set

M2M_API_RESULT **m2m_pdp_apn_set(UINT8 cid, CHAR* apn)**

*Description*: sets the apn (Access Point Name) for the selected cid, and store it in NVM memory.

*Parameters*:
cid: PDP context identifier.
apn: pointer to the Access Point Name string identifying the IP Packet Data Network (PDN).

*Return value*:
refer to **M2M_API_RESULT** enum

## 12.48. m2m_pdp_apn_get

M2M_API_RESULT **m2m_pdp_apn_get (UINT8 cid, CHAR* apn, UINT8 size)**

*Description*: gets the apn (APN) associated to the selected cid. The apn is previously associated to the cid by one of the following functions: **m2m_pdp_apn_set()**, **m2m_pdp_activate()**, or **m2m_pdp_activate_cid()**.

*Parameters*:
cid: PDP context identifier.
apn: pointer to the buffer filled with the Access Point Name string.
size: apn string length

*Return value*:
refer to **M2M_API_RESULT** enum

## 12.49. m2m_pdp_activate

**INT32 m2m_pdp_activate(CHAR *apn, CHAR *name, CHAR *pwd)**

*Description*: sets the apn (APN) for the default cid dedicated to the AppZone PDP context, stores the apn in NVM memory, and activates the PDP context. Upon activation, **M2M_SOCKET_EVENT_PDP_ACTIVE** event is received through M2M_onNetEvent(…) application callback function, refer to M2M_NETWORK_EVENT enum, M2M_net.c file, and document [1].

*Parameters*:
apn: pointer to the Access Point Name string identifying the IP packet Data Network (PDN).
name: pointer to the User Name string. Optional, if not required use **NULL**.
pwd: pointer to the User Password string. Optional, if not required use **NULL**.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.50. m2m_pdp_activate_cid

**INT32 m2m_pdp_activate_cid(CHAR *apn, CHAR *name, CHAR *pwd, UINT8 cid)**

*Description*: sets the apn (APN) for the selected cid, stores the apn in NVM memory, and activates the PDP context. Upon activation, **M2M_SOCKET_EVENT_ PDP_ACTIVE** event is received through M2M_onNetEvent(…) application callback function, refer to M2M_NETWORK_EVENT enum, M2M_net.c file, and document [1].

*Parameters*:
apn: pointer to the Access Point Name string identifying the IP packet Data Network (PDN).
name: pointer to the User Name string. Optional, if not required use **NULL**.
pwd: pointer to the User Password string. Optional, if not required use **NULL**.
cid: PDP context identifier.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.51. m2m_pdp_activate_ip6

**INT32 m2m_pdp_activate_ip6(CHAR *apn, CHAR *name, CHAR *pwd)**

*Description*: activates the AppZone dedicated IPV6 PDP context. Upon activation, **M2M_SOCKET_EVENT_PDP_IPV6_ACTIVE** event is received through M2M_onNetEvent(…) application callback function, refer to M2M_NETWORK_EVENT enum, M2M_net.c file, and document [1].

*Parameters*:
apn: pointer to the Access Point Name string identifying the IP packet Data Network (PDN).
name: pointer to the User Name string. Optional, if not required use **NULL**.
pwd: pointer to the User Password string. Optional, if not required use **NULL**.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.52. m2m_pdp_activate_ip6_cid

**INT32 m2m_pdp_activate_ip6_cid(CHAR *apn, CHAR *name, CHAR *pwd, UINT8 cid)**

*Description*: activates the selected IPV6 PDP context. Upon activation, **M2M_SOCKET_EVENT_PDP_IPV6_ACTIVE** event is received through M2M_onNetEvent(…) application callback function, refer to M2M_NETWORK_EVENT enum, M2M_net.c file, and document [1].

*Parameters*:
apn: pointer to the Access Point Name string identifying the IP packet Data Network (PDN).
name: pointer to the User Name string. Optional, if not required use **NULL**.

pwd: pointer to the User Password string. Optional, if not required use **NULL**.
cid: PDP context identifier.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.53. m2m_pdp_deactive

**INT32 m2m_pdp_deactive(void);**

*Description*: deactivates the AppZone dedicated PDP context.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_SUCCESS**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.54. m2m_pdp_deactive_cid

**INT32 m2m_pdp_deactive_cid(UINT8 cid);**

*Description*: deactivates the PDP context identified by cid.

*Parameters*:
cid: PDP context identifier.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_SUCCESS**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.55. m2m_pdp_get_status

**INT32 m2m_pdp_get_status(void)**

*Description*: gets the AppZone dedicated PDP context status.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.56. m2m_pdp_get_status_cid

**INT32 m2m_pdp_get_status_cid(UINT8 cid)**

*Description*: gets the status of the PDP context identified by cid.

*Parameters*:
cid: PDP context identifier.

*Return value* (see PDP_context_status #defines):
on success: **M2M_PDP_STATE_ACTIVE** or **M2M_PDP_STATE_NOT_ACTIVE**
on failure: **M2M_PDP_STATE_FAILURE**

## 12.57. m2m_pdp_get_my_ip

**UINT32 m2m_pdp_get_my_ip(void)**

*Description*: returns the IPv4 address, in UINT32 number format, of the AppZone dedicated PDP connection.

*Return value*:
on success: IPv4 address in UINT32 number format.
on failure: 0, if PDP is not active.

*Examples*: 19.1.13 TCP-Server

## 12.58. m2m_pdp_get_my_ip_cid

**UINT32 m2m_pdp_get_my_ip_cid(UINT8 cid)**

*Description*: returns the IPv4 address, in UINT32 number format, of the PDP connection identified by cid.

*Parameters*:
cid: PDP context identifier.

*Return value*:
on success: IPv4 address in UINT32 number format.
on failure: 0, if PDP is not active.

## 12.59. m2m_pdp_get_my_ip6

**INT32 m2m_pdp_get_my_ip6(**M2M_SOCKET_BSD_IN6_ADDR ***ipAddr)**

*Description*: gets the IPv6 address of the AppZone dedicated PDP connection.

*Return value*:
on success: 0
on failure: -1

*Output data:*

on success: ipAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with the host's IPv6 address.

## 12.60. m2m_pdp_get_my_ip6_cid

**INT32 m2m_pdp_get_my_ip6_cid(**M2M_SOCKET_BSD_IN6_ADDR ***ipAddr, UINT8 cid)**

*Description*: gets the IPv6 address of the PDP connection identified by cid.

*Parameters*:

cid: PDP context identifier.

*Return value*:

on success: 0
on failure: -1

*Output data:*

on success: ipAddr points to the **M2M_SOCKET_BSD_IN6_ADDR** structure filled with the host's IPv6 address.

## 12.61. m2m_pdp_get_datavol

**INT32 m2m_pdp_get_datavol(UINT8 mode,** M2M_PDP_DATAVOLINFO ***dataVol)**

*Description*: reports the amount of data received and transmitted on the PDP context activated through **m2m_pdp_activate(...)** or **m2m_pdp_activate_ip6(...)** APIs.

Parameters:

mode: 0, only total data counters are reset.
1, dataVol will be filled with PDP data counters of current active session.
2, dataVol will be filled with total data counters since last reset.

dataVol: pointer to the allocated **M2M_PDP_DATAVOLINFO** structure that will be filled with PDP received/sent/total data counters.

*Return value*:

on success: 0
on failure: < 0

## 12.62. m2m_pdp_get_datavol_cid

**INT32 m2m_pdp_get_datavol_cid(UINT8 mode,** M2M_PDP_DATAVOLINFO ***dataVol)**

*Description*: reports the amount of data received and transmitted on the selected PDP context activated through **m2m_pdp_activate_cid(...)** or **m2m_pdp_activate_ip6_cid(...)** APIs.

Parameters:

mode: 0, data counters are reset.
1, dataVol will be filled with PDP data counters of current active session.

2, dataVol will be filled with total data counters since last reset.

dataVol: pointer to the allocated **M2M_PDP_DATAVOLINFO** structure that will be filled with PDP received/sent/total data counters.
cid: PDP context identifier.

*Return value*:
on success: 0
on failure: < 0

➢ 2G: Platform Version ID 13

Not supported by GE910 series.

# 13. M2M_IPRAW_API

Chapter 19.2.16 contains the declarations of the C identifiers used by the APIs set regarding the management of IPv4/IPv6 raw mode.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Configuration Section

## 13.1. m2m_ipraw_cfg

**INT32 m2m_ipraw_cfg( UINT8 enable,UINT8 type,UINT16 RecvTout,UINT8 cid);**

*Description*: enables/disables the IPv4 or IPV6 protocol in raw mode, and sets the timeout to receive an IPv4 or IPv6 packet according to the used API: **m2m_ip4_recv(…) or m2m _ip6_recv(…)**.

*Parameters*:

| | |
|---|---|
| enable: | 1 enable, 0 disable. |
| type: | **M2M_IPV4_RAW_TYPE** selects IPv4, see chapter 19.2.16<br>**M2M_IPV6_RAW_TYPE** selects IPv6, see chapter 19.2.16 |
| RecvTout: | receiver timeout expressed in msec. Range 0000÷FFFF, default 1. |
| cid: | PDP context identifier. |

*Return value*:

| | |
|---|---|
| on success: | 1 |
| on failure: | 0 |

## 13.2. m2m_ip4_send

**INT32 m2m_ip4_send( const void *buff, INT32 len );**

*Description*: sends IPv4 packets in raw mode.

*Parameters*:
buff: pointer to the allocated buffer that contains the IPv4 packet to be send.
len: total number of characters written in the allocated buffer.

*Return value*:
on success: 1
on failure: 0

## 13.3. m2m_ip4_recv

**INT32 m2m_ip4_recv( void *buff, INT32 len );**

*Description*: returns the pending IPv4 packets.

*Parameters*:
buff: pointer to the allocated buffer that will be filled with the received data.
len: total number of characters to be received in the allocated buffer. The size of the allocated buffer should be larger than len value.

*Return value*:
on success: 1
on failure: 0

## 13.4. m2m_ip6_raw

**INT32 m2m_ip6_raw(UINT8 enable)**

*Description*: enables/disables IPv6 raw mode.

*Parameters*:
enable: 1 enable, 0 disable.

*Return value*:
on success: 0
on failure: -1

## 13.5. m2m_ip6_send

**INT32 m2m_ip6_send(const void *buff, INT32 len)**

*Description*: sends IPv6 packets in raw mode.

*Parameters*:
buff: pointer to the allocated buffer that contains the IPv6 packet.
len: total number of characters written in the allocated buffer.

*Return value*:
on success: 0
on failure: -1

## 13.6. m2m_ip6_recv

**INT32 m2m_ip6_recv(void *buff, INT32 len)**

*Description*: returns the pending IPv6 packets.

*Parameters*:
buff: pointer to the allocated buffer that will be filled with the received data.
len: total number of characters to be received in the allocated buffer. The size of the allocated
buffer should be larger than len value.

*Return value*:
on success: 0
on failure: -1

## 13.7. m2m_udp_recv_from_ip6raw

**INT32 m2m_udp_recv_from_ip6raw(**M2M_SOCKET_BSD_SOCKET **s, void *buf, INT32 len,INT32 flags,** M2M_SOCKET_BSD_SOCKADDR ***from, INT32 *fromlen, void *ip6pack, UINT16 *ip6packLen)**

*Description*: is the same as **m2m_socket_bsd_recv_from(…)**, only for UDP sockets; but additionally has two more parameters.

*Parameters*:

| | |
|---|---|
| s: | socket handle. |
| buf: | pointer to the allocated buffer that will be filled with the received data. |
| len: | expected number of characters to be received in the allocated buffer. |
| flags: | not supported, ignored. Set it to 0. |
| from: | pointer to the allocated address structure used by TCP/IP stack filled with the received address, see M2M_SOCKET_BSD_SOCKADDR_IN structure. Cast to **M2M_SOCKET_BSD_SOCKADDR** structure the pointer of the **M2M_SOCKET_ BSD_SOCKADDR_IN** structure when calling this function. |
| fromlen: | pointer to the allocated variable that will be filled with the size of the structure of socket address. |
| ip6pack: | pointer to the allocated buffer where to copy the packet without payload. |
| ip6packLen: | in input: to specify how many bytes can be copied into ip6pack buffer.<br>in output: to specify the length of IPv6 packet without payload. |

*Return value*:

| | |
|---|---|
| on success: | 0 |
| on failure: | -1 |

# 14. M2M_SSL_API

Chapter 19.2.12 contains the declarations of the C identifiers used by the APIs set regarding the management of the SSL.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

Only SSL client support is available, follow next steps:

- Create an SSL service session
- For each connection, create a connection context
- Use **m2m_ssl_encode_send(…)** to encrypt and send data
- After receiving data on the socket, use **m2m_ssl_decode (…)** to decrypt and collect data.

# SSL Section

## 14.1. m2m_ssl_create_service_from_file

**M2M_SSL_SERVICE_SESSION m2m_ssl_create_service_from_file(**
**const CHAR *certFile, const CHAR *privFile, const CHAR *privPass,**
**const CHAR *trustedCAFile, INT32 flags, INT32 *result)**

*Description*: creates a set of SSL service session using the certificates stored in a file.

*Parameters*:
certFile: certificate to use with each connection.
privFile: private key file to be used with each connection.
privPass: private key file password, if exist. **NULL** if no password.
trustedCAFile: CA file, containing list of device trusted CA.
flags: for future use.
result: pointer to the allocated variable that will be filled with the result code.

*Output data*:
"result" points to the variable filled with the result code. See **M2M_SSL_result_codes** #defines

*Return value*
on success: pointer to a valid SSL service session; see **M2M_SSL_SERVICE_SESSION**.
on failure: **NULL**.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.2. m2m_ssl_delete_service

**void m2m_ssl_delete_service(M2M_SSL_SERVICE_SESSION service_session)**

*Description*: deletes an existing SSL service session created with m2m_ssl_create_service_from_file(…).

*Parameters*:
service_session: pointer to a valid SSL service session to be deleted.

## 14.3. m2m_ssl_create_context

**M2M_SSL_CONTEXT_ID_TYPE m2m_ssl_create_context(**
M2M_SSL_Protocol_Version_E **ProtVers,**
M2M_SSL_Cipher_Suite_E **CipherSuite,**
M2M_SSL_AUTH_TYPE_E **AuthType,**
**INT32 * result)**

*Description*: returns a handle that identify a specific setting (or context) of the following items: SSL/TLS protocol version, cipher suite, and authentication type.

*Parameters*:
ProtVers: SSL/TLS protocol version, refer to **M2M_SSL_Protocol_Version_E**
CipherSuite: cipher suite, refer to **M2M_SSL_Cipher_Suite_E**
AuthType: authentication type, refer to **M2M_SSL_AUTH_TYPE_E**
result: pointer to the allocated variable that will be filled with the result code.

*Output data:*
"result" points to the variable filled with the result code. See **M2M_SSL_result_codes** #defines.

*Return value:*
on success: pointer to a valid handle, see **M2M_SSL_CONTEXT_ID_TYPE**.
on failure: **NULL**.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.4. m2m_ssl_delete_context

**void m2m_ssl_delete_context(**M2M_SSL_CONTEXT_ID_TYPE **ssl_context)**

*Description*: deletes the setting (or context) identified by the ssl_context parameter created by the **m2m_ssl_create_context(…)** API.

*Parameters:*
ssl_context: identifies the setting (or context) to be deleted.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.5. m2m_ssl_securesocket

**M2M_SSL_CONNECTION_CONTEXT m2m_ssl_securesocket(**
M2M_SSL_SERVICE_SESSION **session_ID,**
M2M_SSL_CONTEXT_ID_TYPE **SSLCtxID,**
**void *socket_fd,**
**INT32 *result)**

*Description*: returns the connection handle used by the **m2m_ssl_connect(…)** API.

*Parameters:*
session_ID: handle of a SSL service session created by **m2m_ssl_create_service_from_file(…)** API.

SSLCtxID: handle that identifies a specific setting (or context) created by **m2m_ssl_create_ context(…)** API.
socket_fd: socket handle created by **m2m_socket_bsd_socket(…)** API.
result: pointer to the allocated variable that will be filled with the result code.

*Output data:*
"result" points to the variable filled with the result code. See M2M_SSL_result_codes #defines.

*Return value:*
on success: pointer to a valid handle, see **M2M_SSL_CONNECTION_CONTEXT**.
on failure: **NULL**.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.6. m2m_ssl_connect

**INT32 m2m_ssl_connect(**M2M_SSL_CONNECTION_CONTEXT **connectionHandle)**

*Description*: creates a new client connection to the server.

*Parameters:*
connectionHandle: connection handle created by **m2m_ssl_securesocket(…)** API.

*Return value*
on success: **M2M_SSL_SUCCESS**, see M2M_SSL_result_codes #defines
on failure: < 0, see Failure_return_codes #defines.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.7. m2m_ssl_new_connection (obsolete)

**INT32 m2m_ssl_new_connection(**M2M_SSL_SERVICE_SESSION **service_session,**
**void *socket_fd,**
**INT32 flags,**
M2M_SSL_CONNECTION_CONTEXT
***connection_ctx)**

*Description*: creates a new client connection to the server. It uses only the following items to create the SSL context:

- **TLS1.0** handshake protocol
- **M2M_TLS_RSA_WITH_RC4_128_MD5** cipher suite
- **M2M_SSL_ SERVER_AUTH_TYPE** authentication type

The user cannot change the SSL context provided by the function.

Instead of this function, it is suggested to use the **m2m_ssl_securesocket(…)** and **m2m_ssl_connect(…)** functions.

*Parameters*:
service_session: pointer to a valid, not **NULL**, SSL service session created with m2m_ssl_create_service_from_file(…).

socket_fd: pointer to a valid, not **NULL**, socket ID created with **m2m_socket_bsd_socket(…)**.
flags: for future use.
connection_ctx: pointer to the allocated variable that will be filled with connection context.

*Output data*:
"connection_ctx" points to the variable filled with connection context.

*Return value*
on success: **M2M_SSL_SUCCESS**, see M2M_SSL_result_codes #defines
on failure: < 0, see Failure_return_codes #defines.

## 14.8. m2m_ssl_delete_connection

**void m2m_ssl_delete_connection(**M2M_SSL_CONNECTION_CONTEXT **connection_ctx)**

*Description*: deletes an existing connection.

*Parameters*:
connection_ctx: pointer to a valid, not **NULL**, connection session created with **m2m_ssl_securesocket()**.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.9. m2m_ssl_encode_send

**INT32 m2m_ssl_encode_send(**M2M_SSL_CONNECTION_CONTEXT **connection_ctx, UINT8 *buf, UINT32 len)**

*Description*: encrypts "len" bytes of plain text data. After encrypting, the buffer is sent through the socket created by **m2m_socket_bsd_socket(…)**.

*Parameters*:
connection_ctx: pointer to a valid, not **NULL**, connection context created with **m2m_ssl_securesocket()**.
buf: pointer to the buffer containing the plain text to be encrypted and sent.
len: length of the buffer to be encrypted and sent.

*Return value*
on success: > 0, indicates the number of bytes still pending to be sent.
on failure: < 0, see Failure_return_codes #defines.

*Examples*: 19.1.16 SSL/TCP Client.

## 14.10. m2m_ssl_decode

**INT32 m2m_ssl_decode(**M2M_SSL_CONNECTION_CONTEXT **connection_ctx, UINT8 *buffer, INT32 length)**

*Description*: decrypts "length" bytes expected from socket. The decrypted data are stored in the buffer.

*Parameters*:

| | |
|---|---|
| connection_ctx: | pointer to a valid, not **NULL**, connection context created with **m2m_ssl_securesocket()**. |
| buffer: | pointer to the buffer that will be filled with the plain text. |
| length: | expected number of bytes of the plain text. |

*Output data*:

"buffer" points to the buffer filled with plain text.

*Return value*

| | |
|---|---|
| on success: | > 0, indicates the effective length of the plain text in bytes. |
| on failure: | < 0, see Failure_return_codes #defines. |

*Examples*: 19.1.16 SSL/TCP Client.

# 15. M2M_TIMER_API.H

Chapter 19.2.13 contains the declarations of the C identifiers used by the APIs set regarding the management of timers.
The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Timer Section

## 15.1. m2m_timer_create

M2M_T_TIMER_HANDLE **m2m_timer_create(**M2M_T_TIMER_TIMEOUT **cb,**
**void *arg)**

*Description*: creates a timer. Upon timer expiration, the timeout callback function will be called to provide the timeout handling.

*Parameters*:
cb: timeout callback function.
arg: pointer to argument, which will be passed to timeout callback function.

*Return value*:
on success: timer handle.
on failure: **NULL**.

*Examples*: 19.1.6 Timer

## 15.2. m2m_timer_start

**void m2m_timer_start(** M2M_T_TIMER_HANDLE **timer, UINT32 msecs)**

*Description*: starts the timer.

*Parameters*:
timer: timer handle.
msecs: timeout value in milliseconds.

*Examples*: 19.1.6 Timer

## 15.3. m2m_timer_stop

**INT32 m2m_timer_stop(** M2M_T_TIMER_HANDLE **timer)**

*Description*: stops the timer.

*Parameters*:
timer: timer handle.

*Return value*:
1: the timer was not running during the attempt to stop it.
0: on success, the timer has been stopped.

## 15.4. m2m_timer_free

**INT32 m2m_timer_free(** M2M_T_TIMER_HANDLE **timer_handle)**

*Description*: stops and destroys the timer identified by "timer_handle".

*Parameters*:
timer_handle: handle of the timer to stop and destroy.

*Return value*:
1: the timer was not running during the attempt to stop and destroy it.
0: on success, the timer has been stopped and destroyed.

# 16. M2M_DUMP_API

Chapter 19.2.14 contains the declarations of the C identifiers used by the APIs set regarding the management the dump functions.

- 2G: Platform Version ID 12

Watchdog feature not supported.

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Dump Section

## 16.1. m2m_dump_init

M2M_T_DUMP_RESULT **m2m_dump_init(void)**

*Description*: initializes the memory structure reserved to save core dump information.

*Return value*:
on success: M2M_MEM_DUMP_RESULT_SUCCESS
Refer to **M2M_T_DUMP_RESULT** enum to see all available return values.

## 16.2. m2m_dump_save

M2M_T_DUMP_RESULT **m2m_dump_save(INT8 *buffer, INT16 size)**

*Description*: stores the provided buffer in the memory location previously initialized with the **m2m_dump_init** function. The maximum buffer size is M2M_MAX_DUMP_BUFFER_LENGTH.

*Parameters*:
buffer: pointer to the data buffer to be saved.
size: size of the data buffer.

*Return value*:
on success: **M2M_MEM_DUMP_RESULT_SUCCESS**
Refer to **M2M_T_DUMP_RESULT** enum to see all available return values.

## 16.3. m2m_dump_clear

M2M_T_DUMP_RESULT **m2m_dump_clear(void)**

*Description*: clears and de-initializes the memory structure initialized by the **m2m_dump_init** function.

*Return value*:
on success: **M2M_MEM_DUMP_RESULT_SUCCESS**
Refer to **M2M_T_DUMP_RESULT** enum to see all available return values.

# 17. M2M_SEC_API

Chapter 19.2.15 contains the declarations of the C identifiers used by the APIs set regarding the management of the following hash functions: MD5, SHA, SHA256.

These functions implement the message-digest algorithms, which take as input a message of arbitrary length and produce as output respectively a 16-byte, 20-byte, 32-byte message-digest of the input.

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Digest Section

## 17.1. m2m_DIGEST_alloc_res

M2M_API_RESULT **m2m_DIGEST_alloc_res(void *dig_handle,**
DIG_RESULT_T ****dig_res);**

*Description*: creates the room where store the message digest according to the used hash context handle. Before calling this API, you must call one of the following functions to get the desired context handle, see **m2m_MD5_Init**, **m2m_SHA_Init**, **m2m_SHA256_Init**.

Referring to **DIG_RESULT_T** structure:
for example, for hash context handle type M2M_T_MD5_HANDLE, dig_res→result has dig_res→size equal to MD5_DIGEST_LENGTH.

| Hash context handle type | Message digest length |
|---|---|
| M2M_T_MD5_HANDLE | MD5_DIGEST_LENGTH |
| M2M_T_SHA_HANDLE | SHA_DIGEST_LENGTH |
| M2M_T_SHA256_HANDLE | SHA256_DIGEST_LENGTH |

*Parameters*:
dig_handle: pointer to the allocated variable that contains the selected hash context handle.
dig_res: pointer to the pointer of the structure allocated and initialized by the API according to the message digest length corresponding to the used hash context handle.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Example*: *19.1.20* MD5 Hash Function, SHA and SHA256 functions follow the same programming sequence.

## 17.2. m2m_DIGEST_destroy_res

M2M_API_RESULT **m2m_DIGEST_destroy_res(**DIG_RESULT_T ****dig_res);**

*Description*: frees resources allocated by the **m2m_DIGEST_alloc_res** API for the message digest storage.

*Parameters*:
dig_res: pointer to the pointer of the structure allocated and initialized by the caller according to the message digest length corresponding to the previously selected context.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.3. m2m_MD5_Init

M2M_API_RESULT **m2m_MD5_Init(**M2M_T_MD5_HANDLE ***h)**

*Description*: initializes the MD5 context structure and returns its handle.
After that, the caller using the **m2m_DIGEST_alloc_res** API must allocate the room where the final message digest will be stored.
The other MD5 functions, **m2m_MD5_Update** and **m2m_MD5_Final**, allow an MD5 digest to be computed over multiple message blocks; between blocks, the state of the MD5 computation is held in an MD5 context structure.

*Parameters*:
h: pointer to the allocated variable that will be filled with the MD5 context handle.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Example*: *19.1.20* MD5 Hash Function

## 17.4. m2m_MD5_Update

M2M_API_RESULT **m2m_MD5_Update(**M2M_T_MD5_HANDLE **h, UINT8 *data, UINT32 len)**

*Description*: processes the bytes in input, adding them to the digest being calculated in the MD5 context. The function can be called repeatedly with chunks of the message to be hashed.

*Parameters*:
h: handle referring to the MD5 context.
data: pointer to the buffer containing the data that must be processed.
len: data length.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.5. m2m_MD5_Final

M2M_API_RESULT **m2m_MD5_Final(**M2M_T_MD5_HANDLE ***h,**
DIG_RESULT_T ***dig_res)**

*Description*: finishes the MD5 operation and places the message digest in the structure initialized by means of the **m2m_DIGEST_alloc_res** API. In addition, the function releases the MD5 context. It must be reinitialized with **m2m_MD5_Init** before being used again.

*Parameters*:
h: pointer to the MD5 context handle.
dig_res: pointer to the structure that define the room for the message digest.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.6. m2m_SHA_Init

M2M_API_RESULT **m2m_SHA_Init(**M2M_T_SHA_HANDLE ***h)**

*Description*: initializes the SHA context structure and returns its handle.
After that, the caller using the **m2m_DIGEST_alloc_res** API must allocate the room where the final message digest will be stored.
The other SHA functions, **m2m_SHA_Update** and **m2m_SHA_Final**, allow an SHA message digest to be computed over multiple message blocks; between blocks, the state of the SHA computation is held in a SHA context structure.

*Parameters*:
h: pointer to the allocated variable that will be filled with the SHA context handle.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Example*: *19.1.20* MD5 Hash Function, SHA functions follow the same programming sequence.

## 17.7. m2m_SHA_Update

M2M_API_RESULT **m2m_SHA_Update(**M2M_T_SHA_HANDLE **h, UINT8 *data, UINT32 len)**

*Description*: processes the bytes in input, adding them to the digest being calculated in the SHA context. The function can be called repeatedly with chunks of the message to be hashed.

*Parameters*:
h: handle referring to the SHA context.
data: pointer to the buffer containing the data that must be processed.
len: data length.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.8. m2m_SHA_Final

M2M_API_RESULT **m2m_SHA_Final(**M2M_T_SHA_HANDLE ***h,**
DIG_RESULT_T ***dig_res)**

*Description*: finishes the SHA operation and places the message digest in the structure initialized by means of the **m2m_DIGEST_alloc_res** API. In addition, the function releases the SHA context. It must be reinitialized with **m2m_SHA_Init** before being used again.

*Parameters*:

h: pointer to the SHA context handle.
dig_res: pointer to the structure that define the room for the message digest.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.9. m2m_SHA256_Init

M2M_API_RESULT **m2m_SHA256_Init(**M2M_T_SHA256_HANDLE ***h)**

*Description*: initializes the SHA256 context structure and returns its handle.
After that, the caller using the **m2m_DIGEST_alloc_res** API must allocate the room where the final message digest will be stored.
The other SHA256 functions, **m2m_SHA256_Update** and **m2m_SHA256_Final**, allow an SHA256 message digest to be computed over multiple message blocks; between blocks, the state of the SHA256 computation is held in a SHA256 context structure.

*Parameters*:
h: pointer to the allocated variable that will be filled with the SHA256 context handle.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Example*: *19.1.20* MD5 Hash Function, SHA256 functions follow the same programming sequence.

## 17.10. m2m_SHA256_Update

M2M_API_RESULT **m2m_SHA256_Update(**M2M_T_SHA256_HANDLE **h, UINT8 *data, UINT32 len)**

*Description*: processes the bytes in input, adding them to the digest being calculated in the SHA256 context. The function can be called repeatedly with chunks of the message to be hashed.

*Parameters*:
h: handle referring to the SHA256 context.
data: pointer to the buffer containing the data that must be processed.
len: data length.

*Return value*:
on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

## 17.11. m2m_SHA256_Final

M2M_API_RESULT **m2m_SHA256_Final(**M2M_T_SHA256_HANDLE ***h,**
DIG_RESULT_T ***dig_res)**

*Description*: finishes the SHA256 operation and places the message digest in the structure initialized by means of the **m2m_DIGEST_alloc_res** API. In addition, the function releases the SHA256 context. It must be reinitialized with **m2m_SHA256_Init** before being used again.

*Parameters*:

| | |
|---|---|
| h: | pointer to the SHA256 context handle. |
| dig_res: | pointer to the structure that define the room for the message digest. |

*Return value*:

| | |
|---|---|
| on success: | **M2M_API_RESULT_SUCCESS**<br>Refer to **M2M_API_RESULT** enum to see all available return values. |

# 18. M2M_HW_WATCHDOG

Chapter 19.2.17 contains the declarations of the C identifiers used by the APIs set regarding the management of the Hardware Watchdog features.

The watchdog hardware register is shared between:

- User Tasks
- AppZone Framework (AppZone tasks not controlled by the user)
- Modem Stack (Firmware)

The table below shows the three possible watchdog configurations set using different APIs sequences.

| Watchdog is monitoring: | APIs sequences |
|---|---|
| User Tasks | **m2m_hw_watchdog_conf(...)** with **timeout = 0**<br>**m2m_hw_watchdog_enable(...)** |
| User Tasks + AppZone Framework | **m2m_hw_watchdog_conf(...)** with **timeout ≥ 5**<br>**m2m_hw_watchdog_enable(...)** |
| Modem Stack (valid on for 3G modules) | **m2m_hw_watchdog_disable ()** |

## ➢ 2G: Platform Version ID 12

Watchdog feature not supported.

## ➢ 4G: Platform Version ID 23

Watchdog feature not supported.

The next chapters describe the use of the APIs. See also the Caution Note in chapter 2.

# Watchdog Section

## 18.1. m2m_hw_watchdog_conf

M2M_API_RESULT **m2m_hw_watchdog_conf (** M2M_WATCHDOG_OPTIONS ***options)**

*Description*: sets the watchdog timeout for the AppZone Framework which is not under user control. Its behavior is affected by the **m2m_hw_watchdog_enable (...)** API**.**

*Parameters*:

options: pointer to the allocated variable. According to the option value, one of the following scenarios occur.

If:

- options = 0, the watchdog timeout for the AppZone Framework is disabled.
- 0 < options < 5, the function returns the **M2M_API_RESULT_INVALID_ARG** error code.
- options ≥ 5 seconds is a valid timeout value.
- the function is not called before the watchdog is enabled, a default value of 20 seconds is set for the watchdog of the AppZone Framework.
- the function is called after the watchdog has been enabled, the **M2M_API_RESULT_FAIL** error code is returned.

| | Right APIs sequence |
|---|---|
| 1 | m2m_hw_watchdog_conf (...) |
| 2 | m2m_hw_watchdog_enable(...) |

| | Right APIs sequence |
|---|---|
| 1 | m2m_hw_watchdog_disable () |
| 2 | m2m_hw_watchdog_conf (...) |
| 3 | m2m_hw_watchdog_enable(...) |

| | Wrong APIs sequence |
|---|---|
| 1 | m2m_hw_watchdog_enable(...) |
| 2 | m2m_hw_watchdog_conf (...) |

*Return value*:

on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Examples*: 19.1.21 Watchdog

## 18.2. m2m_hw_watchdog_enable

M2M_API_RESULT **m2m_hw_watchdog_enable (**M2M_WATCHDOG_TIMEOUT **timeout)**

*Description*: enables the User Tasks watchdog, and sets the related timeout. After calling this function, it is as if there were two distinct watchdogs:

- the User Tasks are monitored by the watchdog using the timeout value set by the timeout parameter of the enable API, and refreshed by the **m2m_hw_watchdog_ refresh (...)** API. If this function is called, and the User Tasks watchdog was previously enabled, the timeout is updated.

- the AppZone Framework is monitored by the watchdog using the timeout value set by **m2m_hw_watchdog_conf(...)** API, and refreshed by internal tasks belonging to the AppZone Framework. If **m2m_hw_ watchdog_conf(...)** API is not called before the enable API, or called with option value > 0 and < 5, by default is used the timeout set to 20 seconds.

*Parameters*:

timeout: specifies the watchdog timeout of the User Tasks.
Refer to **M2M_WATCHDOG_TIMEOUT** enum. If an invalid timeout is passed to the function, the **M2M_API_RESULT_INVALID _ARG** error code is returned.

*Return value*:

on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Examples*: 19.1.21 Watchdog

## 18.3. m2m_hw_watchdog_disable

M2M_API_RESULT **m2m_hw_watchdog_disable (void)**

*Description:* disables the User Tasks and AppZone Framework watchdogs, and restores the status[2] present before the call of **m2m_hw_watchdog_enable(...)** API. If the User Tasks watchdog was not previously enabled, the **M2M_API_RESULT_FAIL** error code is returned.

*Return value*:

on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Examples*: 19.1.21 Watchdog

[2] For 3G modules (Platform ID 12), the Modem Stack watchdog will be restored. Therefore, possible stops of the AppZone Framework or User Tasks will be not more detected, and recovered.

## 18.4. m2m_hw_watchdog_refresh

M2M_API_RESULT **m2m_hw_watchdog_refresh (void)**

*Description:* refreshes the User Tasks watchdog, it must be called periodically to avoid the User Tasks watchdog timer expiration. If this function is called but the watchdog was not previously enabled, the **M2M_API_RESULT_FAIL** error code is returned.

*Return value*:

on success: **M2M_API_RESULT_SUCCESS**
Refer to **M2M_API_RESULT** enum to see all available return values.

*Examples*: 19.1.21 Watchdog

# 19. APPENDIXES

## 19.1. Examples

### 19.1.1. Open and Close

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "m2m_type.h"
#include "m2m_fs_api.h"
#include "m2m_hw_api.h"

CHAR  *file_name = "goofy";
M2M_API_RESULT  API_Result;
M2M_T_FS_HANDLE file_handle = NULL;
.........
.........

file_handle = m2m_fs_open(file_name, M2M_FS_OPEN_WRITE);
if(file_handle == NULL)
 {
   PrintToUart("M2M_FS_OPEN_WRITE, NULL");              /* See chapter 19.1.8 PrintToUart */
   /* use m2m_fs_last_error() to know error details */
   return;
 }
else
 {
   PrintToUart("M2M_FS_OPEN_WRITE, SUCCESS");
 }

.......
.......

API_Result_Close = m2m_fs_close(file_handle);
if(API_Result_Close == M2M_API_RESULT_INVALID_ARG)
 {
   PrintToUart("close %u, INVALID ARGUMENT", Id);
   /* use m2m_fs_last_error() to know error details */
   return;
 }
else if (API_Result_Close == M2M_API_RESULT_FAIL)
 {
   PrintToUart("close %u, FAIL", Id);
   /* use m2m_fs_last_error() to know error details */
   return;
 }
else if (API_Result_Close == M2M_API_RESULT_SUCCESS)
 {
  PrintToUart("close %u, SUCCESS", Id);
 }

...........
...........
```

### 19.1.2. Listing all Files

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "m2m_type.h"
#include "m2m_fs_api.h"
#include "m2m_hw_api.h"
.............................

M2M_API_RESULT  API_Result;
CHAR filename_buffer[256];
UINT32 i;
.............................

API_Result = m2m_fs_find_first(filename_buffer, "*");
if(API_Result == M2M_API_RESULT_INVALID_ARG)
 {
   PrintToUart("m2m_fs_find_first: INVALID ARGUMENT");          /* See chapter 19.1.8 PrintToUart */
   /* use m2m_fs_last_error() to know error details */
   return;
 }
else if (API_Result == M2M_API_RESULT_FAIL)
 {
   PrintToUart("m2m_fs_find_first: FAIL");
   /* use m2m_fs_last_error() to know error details */
   return;
 }
else if (API_Result == M2M_API_RESULT_SUCCESS)
 {
   /* PrintToUart("m2m_fs_find_first: SUCCESS"); */
   /* use m2m_fs_last_error() to know error details */
    PrintToUart("\n");
 }

i=0;
while(filename_buffer[i] != '\0')
  {
     PrintToUart_ON_Line("%c", filename_buffer[i]);
     i++;
  }
    PrintToUart("End first searching\n");

do
 {
   API_Result = m2m_fs_find_next(filename_buffer);
   if(API_Result == M2M_API_RESULT_INVALID_ARG)
    {
     PrintToUart("m2m_fs_find_next: INVALID ARGUMENT");
     /* use m2m_fs_last_error() to know error details */
     return;
    }
   else if (API_Result == M2M_API_RESULT_FAIL)
    {
      PrintToUart("m2m_fs_find_next: FAIL");
      /* use m2m_fs_last_error() to know error details */
      return;
    }
   else if (API_Result == M2M_API_RESULT_SUCCESS)
    {
      /* PrintToUart("m2m_fs_find_next: SUCCESS"); */
      /* use m2m_fs_last_error() to know error details */
      PrintToUart("\n");
    }

   i=0;
   while(filename_buffer[i] != '\0')
    {
      PrintToUart_ON_Line("%c", filename_buffer[i]);
      i++;
    }

    PrintToUart("\n");
 }
```

```
while(API_Result == M2M_API_RESULT_SUCCESS);
```

### 19.1.3. GPIO

```
#include "m2m_type.h"
#include "m2m_hw_api.h"

void GPIO_example(void)
{
   INT32 gpio = 2;
   INT32 value;

   m2m_hw_gpio_conf(gpio, 0);
   value = m2m_hw_gpio_read(gpio);
   m2m_hw_gpio_write(gpio, value);
}
```

### 19.1.4. Semaphore Used for Critical Section

```
#include "m2m_type.h"
#include "m2m_os_lock_api.h"
M2M_T_OS_LOCK semaphore
   .

   /* Initialize a semaphore for a Critical Section: "val" value is equal to M2M_OS_LOCK_CS, it is
      the initial semaphore count = 1*/
      semaphore = m2m_os_lock_init(M2M_OS_LOCK_CS);

   /* The retrieved semaphore count is one, then the semaphore count is decreased. The control is returned to the calling
      task */
      m2m_os_lock_lock(semaphore);

   /* The current task executes its critical code section, any task trying to use m2m_os_lock_lock(semaphore) gets stuck */

   /* Increase the semaphore count by one to unlock the semaphore. */
   /* Pay attention: unlock the semaphore more times than it is locked changes its behavior: it works as a counting
      semaphore, (counter > 1). So, its use is no more suited for Critical Sections which need binary semaphore */
      m2m_os_lock_unlock(semaphore);

   /* Destroy the semaphore */
      m2m_os_lock_destroy(semaphore);
   .
```

### 19.1.5. Semaphore Use for Inter Process Communication

```
#include "m2m_type.h"
#include "m2m_os_lock_api.h"
M2M_T_OS_LOCK semaphore
   .

   /* Initialize a semaphore for Inter Process Communication: "val" value is equal to M2M_OS_LOCK_IPC, it is
      the initial semaphore count = 0 */
      M2M_T_OS_LOCK semaphore = m2m_os_lock_init(M2M_OS_LOCK_IPC);

   /* The retrieved semaphore count is zero, the control is not returned to the calling task */
      m2m_os_lock_lock(semaphore);

   /* The current task waits till the other task unlocks the semaphore by calling m2m_os_lock_unlock(semaphore),
      the function increments the semaphore count by one */

   /* Destroy the semaphore */
      m2m_os_lock_destroy(semaphore);
   .
```

### 19.1.6. Timer

```
#include "m2m_type.h"
#include "m2m_timer_api.h"

void user_timeout_handler(void *arg)
{
    /* do something */
    /* implement 1 second periodic timer functionality */
    m2m_timer_start(user_timer, 1000);
    return;
}

void timer_example(void)
{
    M2M_T_TIMER_HANDLE user_timer = m2m_timer_create(user_timeout_handler, NULL);
    m2m_timer_start(user_timer, 1000);  /* start the timer with 1 sec timeout  */
}
```

### 19.1.7. HW Timer

```
#include "m2m_type.h"
#include "m2m_hw_api.h"

void Timer_example(void)
{
   INT32 timer_id = 1;
   UINT32 span = 100;
   INT32 value;

   value = m2m_hw_timer_start(timer_id, span);

  /* On time out expiration the M2M_onHWTimer(…) callback is called */
}
```

### 19.1.8. PrintToUart

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "m2m_type.h"
#include "m2m_hw_api.h"

void PrintToUart ( const char *fmt, ... )
{
            INT32 sent;
            va_list arg;
            CHAR buf[256];

            M2M_T_HW_UART_HANDLE uart_handle = M2M_HW_UART_HANDLE_INVALID;

            va_start(arg, fmt);
            vsnprintf(buf, 256, fmt, arg);

            /* Get a UART handle first */
            uart_handle = m2m_hw_uart_open();

            if (uart_handle != M2M_HW_UART_HANDLE_INVALID)
             {
               m2m_hw_uart_write(uart_handle, buf, strlen(buf), &sent);
               m2m_hw_uart_write(uart_handle, "\r\n", 2, &sent);
               m2m_hw_uart_close(uart_handle);
             }

            va_end(arg);
}
```

### 19.1.9. Set Alarm

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>

#include "m2m_type.h"
#include "m2m_hw_api.h"
#include "m2m_clock_api.h"

/* See chapter 19.1.8 PrintToUart */

INT32 RTC_example( int days, int hours, int minutes )
{

 /* Sets the RTC alarm clock according to defined time interval expressed in dd:hh:mm */
 /* The function returns 0 on success, else returns -1 */

 /* set epoch in seconds: 1429163640:
  * Year 2015, Month 4, Date 16, Hours 07, Minutes 54, Seconds 0, Time Zone Local */

 const time_t start_sec_prog = 1429163640;


 INT32 sec_prog = minutes*60 + hours*3600 + days*24*3600;
 INT32 sec_cur;

 M2M_T_RTC_DATE          date_mem;
 M2M_T_RTC_TIME          time_mem;
 M2M_T_RTC_RESULT        res_RTC;

 struct M2M_T_RTC_TIMEVAL          tv;
 struct M2M_T_RTC_TIMEZONE         tz;
 struct tm*restart;

 restart = localtime(&start_sec_prog);
 PrintToUart("Initial Local time and date: %s", asctime(restart));

 tv.tv_sec = 1429163640;
 tv.tv_msec = 0;
 tz.tz_tzone = 0;
 tz.tz_dst = 0;

 if (m2m_set_timeofday(&tv, &tz) == -1 )
  {
   PrintToUart("Exit: set_timeofday ERROR\n");
   return -1;
  }

 /* get epoch, it is just an example */

 if (m2m_get_timeofday(&tv, &tz) == -1 )
  {
   PrintToUart("Exit: get_timeofday ERROR\n");
   return -1;
  }

 /* add the time interval. When it is elapsed, the M2M_onWakeup(void) callback is activated. */

 sec_cur = tv.tv_sec;
 sec_prog = sec_cur + sec_prog;
 PrintToUart("epoch when the time interval will be over: %d\n", sec_prog);

 restart = localtime((const time_t *) &sec_prog);

 PrintToUart("Local time and date when the time interval will be over: %s", asctime(restart));

 /* set the alarm when the timer interval is over */

 time_mem.hour =(CHAR)restart->tm_hour;
 time_mem.minute = (CHAR)restart->tm_min;
 time_mem.second = (CHAR)restart->tm_sec;

 date_mem.day = (CHAR)restart->tm_mday;
```

```
  date_mem.month = (CHAR)(restart->tm_mon + 1);
  date_mem.year = (CHAR)(restart->tm_year -100);


  res_RTC = m2m_rtc_set_alarm(date_mem, time_mem);

  if (res_RTC != M2M_RTC_SUCCESS)
    {
      PrintToUart("Exit: set_alarm ERROR\n");
      return -1;
    }

  return 0;
}

void M2M_main ( INT32 argc, CHAR argv[M2M_ARGC_MAX][M2M_ARGV_MAXTOKEN + 1] )
{
  PrintToUart("Start alarm setting");
  PrintToUart("After one minute the M2M_onWakeup(void) callback sends: Alarm is waked up!\n");
  RTC_example( 0, 0, 1);
}


/* Write the code into M2M_onWakeup(void) callback contained in the M2M_HwEvents.c file */

#include "m2m_type.h"
#include "m2m_hw_api.h"

void M2M_onWakeup(void)
{
  M2M_T_HW_UART_HANDLE USIF0_handle;
  INT32 len_sent;

  USIF0_handle = m2m_hw_uart_open();
  m2m_hw_uart_write(USIF0_handle, "Alarm is waked up!", 18, &len_sent);
  m2m_hw_uart_close(USIF0_handle);
}
```

### 19.1.10. SMS Storage

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "m2m_type.h"
#include "m2m_hw_api.h"
#include "m2m_sms_api.h"


void SMS_Storage_Example(void)
{
  INT32 i, Res;
  M2M_T_SMS_MEM_STORAGE memory[3];

  /* Set SMS storage */
  Res = m2m_sms_set_preferred_message_storage("SM", "SM", "SM");
  if ( Res != 1 )
   {
     PrintToUart("Set Error!");                  /* See chapter 19.1.8 PrintToUart */
     return;
   }

 /* Get SMS storage status */
 Res = m2m_sms_get_preferred_message_storage(memory);
 if ( Res != 1 )
  {
    PrintToUart("Get Error!");
    return;
  }

 for ( i = 0; i < 3; i ++ )
  {
    PrintToUart("mem[%d] = %s", i, memory[i].mem );
    PrintToUart("used[%d] = %d", i, memory[i].nUsed );
    PrintToUart("tot[%d] = %d", i, memory[i].nTotal );
  }
}

void M2M_main ( INT32 argc, CHAR argv[M2M_ARGC_MAX][M2M_ARGV_MAXTOKEN + 1] )
{
              PrintToUart("SMS Storage Example\n");
              SMS_Storage_Example();

}
```

### 19.1.11. Socket ioctl

```
/* Set socket in blocking/non-blocking mode. */

#include "m2m_type.h"
#include "m2m_socket_api.h"

INT32 on = 1;

/* ... Create socket and connect ... */

/* Set socket in non blocking mode */
if (0 != m2m_socket_bsd_ioctl (SocketFD, M2M_SOCKET_BSD_FIONBIO, &on))
 {
   /* error setting to non blocking */
 }

 on = 0;

/* Set socket in blocking mode */
if (0 != m2m_socket_bsd_ioctl (SocketFD, M2M_SOCKET_BSD_FIONBIO, &on))
 {
   /* error setting to blocking */
 }
```

### 19.1.12. TCP-Client

Here is a simple TCP client that can be used with the TCP server shown in the chapter 19.1.13. As example, assume to have two modules, one in server mode, the second one in client mode.

Client side:

set the PORT

set the IP address of the server, it is already running and waiting a message from client.

start the PDP context

start the tcp_client(): the message "Hello from AppZone!" is sent to the server

```
void tcp_client()

{
 struct M2M_SOCKET_BSD_SOCKADDR_IN stSockAddr;
 M2M_SOCKET_BSD_SOCKET SocketFD ;
 M2M_SOCKET_BSD_FD_SET set;
 CHAR buf_send[] = "Hello from AppZone!";
 CHAR buf_recv[100];
 INT32 Res;
 UINT32 addr = 0;
 UINT16 PORT = YYYY;                                    /* ← SET PORT */
 CHAR IP_SERVER[] = "XXX.XXX.XXX.XXX";                  /* ← SET THE IP OF THE SERVER */
 INT32 namelen = 0;

 SocketFD = m2m_socket_bsd_socket(M2M_SOCKET_BSD_PF_INET, M2M_SOCKET_BSD_SOCK_STREAM,
                                          M2M_SOCKET_BSD_IPPROTO_TCP);

 if (M2M_SOCKET_BSD_INVALID_SOCKET == SocketFD)
  {
    PrintToUart("Socket handle FAILURE");          /* See chapter 19.1.8 PrintToUart */
    return;
  }

 PrintToUart("Socket handle: SUCCESS");

 memset(&stSockAddr, 0, sizeof(struct M2M_SOCKET_BSD_SOCKADDR_IN));

 stSockAddr.sin_family = M2M_SOCKET_BSD_PF_INET;
 stSockAddr.sin_port = m2m_socket_bsd_htons(PORT);
 stSockAddr.sin_addr.s_addr = m2m_socket_bsd_inet_addr(IP_SERVER);

 if (M2M_SOCKET_BSD_INVALID_SOCKET ==
     m2m_socket_bsd_connect(SocketFD, (M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                                  M2M_SOCKET_BSD_SOCKADDR_IN)))
  {
    PrintToUart("Socket connects: FAILURE");
    m2m_socket_bsd_close(SocketFD);
    return;
  }

  PrintToUart("Socket connects: SUCCESS");

 Res = m2m_socket_bsd_get_sock_name(SocketFD, ( M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, &namelen );

 if (Res == 0)
   {
    /* IP address and Port */
   PrintToUart ("IP: %s",m2m_socket_bsd_addr_str(stSockAddr.sin_addr.s_addr));
   PrintToUart ("PORT: %u", m2m_socket_bsd_ntohs(stSockAddr.sin_port));
   }
 else
   {
    PrintToUart ("get_socket_name: FAILURE");
    return;
   }

 /* send message to server */

 Res = m2m_socket_bsd_send_buf_size(SocketFD);
```

```
 if (Res > sizeof(buf_send))
  {
    PrintToUart ("Available buffer space = %d",Res);
    Res = m2m_socket_bsd_send(SocketFD, buf_send, sizeof(buf_send), 0);
  }

 if(Res < 0)
 {
   PrintToUart ("Sending: FAILURE");
 }
 else
 {
   PrintToUart ("Sending: SUCCESS, character sent = %d", Res);
 }

 /* receive message from server */

 Res = m2m_socket_bsd_recv(SocketFD, buf_recv, sizeof(buf_recv), 0);

 if(Res < 0)
 {

 PrintToUart ("Receiving: FAILURE");
 }
 else
 {
   PrintToUart ("Receiving: SUCCESS, characters received = %d", Res);
   PrintToUart("%s", buf_recv);
 }


 if(m2m_socket_bsd_close(SocketFD) == 0)
 {
   PrintToUart ("Close SUCCESS");
 }
 else
 {
   PrintToUart ("Close FAILURE");
 }

 return;
}
```

### 19.1.13. TCP-Server

Here is a simple TCP server that can be used with the TCP client shown in the chapter 19.1.12. As example, assume to have two modules, one in server mode, the second one in client mode.

Server side:

set the PORT in accordance with the client

start the PDP context

start the TCPServer()

take note of its IP address and use it to set the client

Now, the server is waiting a message from the client. When the client message is received, it sends "Hello World!" to the client.

```
void TCPServer()
{
   struct M2M_SOCKET_BSD_SOCKADDR_IN stSockAddr;
   M2M_SOCKET_BSD_SOCKET SocketFD,ConnectFD;
   INT32 Res, addrlen, MY_IP_inet_address;
   INT32 timeOutVal = 240000;
   UINT16 PORT = YYYY;                          /* ← SET PORT USED BY THE CLIENT */
   CHAR buf_recv[100];
   CHAR buf_send[] = "Hello World!";

   MY_IP_inet_address = m2m_pdp_get_my_ip();
   PrintToUart("IP String: %s", m2m_socket_bsd_addr_str(MY_IP_inet_address));

   for(;;)
   {

   SocketFD = m2m_socket_bsd_socket(M2M_SOCKET_BSD_AF_INET, M2M_SOCKET_BSD_SOCK_STREAM,
                                    M2M_SOCKET_BSD_IPPROTO_TCP);

   if(M2M_SOCKET_BSD_INVALID_SOCKET == SocketFD)
   {
     PrintToUart("Socket handle FAILURE");          /* See chapter 19.1.8 PrintToUart */
     return;
   }

   PrintToUart ("Socket handle: SUCCESS");

   memset(&stSockAddr, 0, sizeof(struct M2M_SOCKET_BSD_SOCKADDR_IN));

   stSockAddr.sin_family = M2M_SOCKET_BSD_AF_INET;
   stSockAddr.sin_port = m2m_socket_bsd_htons(PORT);
   stSockAddr.sin_addr.s_addr = M2M_SOCKET_BSD_INADDR_ANY;

   if(M2M_SOCKET_BSD_INVALID_SOCKET ==
      m2m_socket_bsd_bind(SocketFD,(struct M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                          M2M_SOCKET_BSD_SOCKADDR_IN)))
   {
      PrintToUart("Socket bind FAILURE");
      m2m_socket_bsd_close(SocketFD);
      return;
   }

  PrintToUart ("Socket bind: SUCCESS");

   if(M2M_SOCKET_BSD_INVALID_SOCKET == m2m_socket_bsd_listen(SocketFD, 1))
   {
      PrintToUart("Socket listen FAILURE");
      m2m_socket_bsd_close(SocketFD);
      return;
   }

   PrintToUart ("Socket listen: SUCCESS");
```

```
   m2m_socket_bsd_set_sock_opt(SocketFD, M2M_SOCKET_BSD_SOL_SOCKET,
M2M_SOCKET_BSD_SO_RCVTIMEO,
                                        &timeOutVal, sizeof(timeOutVal));



    PrintToUart ("Accept loop");

   ConnectFD = m2m_socket_bsd_accept(SocketFD, (struct M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr,
                                                  &addrlen);



    if(0 > ConnectFD)
    {
       PrintToUart ("Socket accept: FAILURE");
       m2m_socket_bsd_close(SocketFD);
       return;
    }

    /* receive message from client */

    Res = m2m_socket_bsd_recv(ConnectFD, buf_recv, sizeof(buf_recv), 0);

    if(Res < 0)
    {
      PrintToUart ("Receiving: FAILURE");
    }
    else
    {
      PrintToUart ("Receiving: SUCCESS, characters received = %d", Res);

      PrintToUart ("%s", buf_recv);
     }

      /* send message to client */

     Res = m2m_socket_bsd_send_buf_size(ConnectFD);

     if (Res > sizeof(buf_send))
     {
       PrintToUart ("Available buffer space = %d",Res);
       Res = m2m_socket_bsd_send(ConnectFD, buf_send, sizeof(buf_send), 0);
     }

     if(Res < 0)
     {
             PrintToUart ("Sending: FAILURE");
     }
     else
     {
             PrintToUart ("Sending: SUCCESS, character sent = %d", Res);
     }

     m2m_socket_bsd_close(ConnectFD);


     while(1)
     {

            Res = m2m_socket_bsd_socket_state(ConnectFD);
            if(Res == M2M_SOCKET_STATE_CLOSED )
            {
              break;
            }
            m2m_os_sleep_ms(1000)
      }

      if(m2m_socket_bsd_close(SocketFD) != 0)
      {
            PrintToUart("ERROR closing parent socket. errno: %d", m2m_socket_errno());
      }

      while(1)
      {
```

```
            Res = m2m_socket_bsd_socket_state(SocketFD);
            if(Res == M2M_SOCKET_STATE_CLOSED )
            {
              break;
            }
              m2m_os_sleep_ms(1000);
      }

   }  /* for(;;) */
   return;
}
```

### 19.1.14. UDP-Client

Here is a simple UDP client that can be used with the UDP server shown in the chapter 19.1.15. As example, assume to have two modules, one in server mode, the second one in client mode.

Client side:

set the PORT

set the IP address of the server. The server is already running and waiting a message from client.

start the PDP context

start the UDPClient(): the message "Hello from AppZone!" is sent to the server

```
void UDPClient()
{
   M2M_SOCKET_BSD_SOCKET sock;
   struct M2M_SOCKET_BSD_SOCKADDR_IN sa;
   INT32 bytes_sent, buffer_length;
   UINT16 PORT = YYYY;                                  /* ← SET PORT */
   CHAR IP_SERVER[] = "XXX.XXX.XXX.XXX";                /* ← SET THE IP OF THE SERVER */

   CHAR buf_send[] = "Hello from AppZone!";

   sock = m2m_socket_bsd_socket(M2M_SOCKET_BSD_PF_INET, M2M_SOCKET_BSD_SOCK_DGRAM,
                                          M2M_SOCKET_BSD_IPPROTO_UDP);


   if (M2M_SOCKET_BSD_INVALID_SOCKET == sock)
   {
     PrintToUart("Socket handle FAILURE");
     return;
   }

   PrintToUart("Socket handle: SUCCESS");                /* See chapter 19.1.8 PrintToUart */

   memset(&sa, 0, sizeof(sa));
   sa.sin_family = M2M_SOCKET_BSD_AF_INET;
   sa.sin_addr.s_addr = m2m_socket_bsd_inet_addr(IP_SERVER);
   sa.sin_port = m2m_socket_bsd_htons(PORT);

   bytes_sent = m2m_socket_bsd_send_to(sock, buf_send, sizeof(buf_send), 0,(struct
                                  M2M_SOCKET_BSD_SOCKADDR*)&sa, sizeof (struct
M2M_SOCKET_BSD_SOCKADDR_IN));


   if(bytes_sent >= 0)
   {
     PrintToUart ("Sending: SUCCESS, character sent = %d", bytes_sent);
   }
   else
   {
     PrintToUart ("Sending: FAILURE");
   }

   if(m2m_socket_bsd_close(sock) == 0)
   {
     PrintToUart ("Close: SUCCESS");
```

```
    }
    else
    {
      PrintToUart ("Close: FAILURE");
    }

    return;
}
```

### 19.1.15. UDP-Server

Here is a simple UDP server that can be used with the UDP client shown in the chapter 19.1.14. As example, assume to have two modules, one in server mode, the second one in client mode.

Server side:

set the PORT in accordance with the client

start the PDP context

start the UDPServer()

take note of its IP address and use it to set the client

Now, the server is waiting a message from the client.

```
void UDPServer(void)
{
   struct M2M_SOCKET_BSD_SOCKADDR_IN sa;
   CHAR  buffer[1024];
   INT32 timeOutVal = 240000;
   INT32 fromlen, recsize, MY_IP_inet_address;
   M2M_SOCKET_BSD_SOCKET sock;
   UINT16 PORT = YYYY;                    /* ← SET PORT USED BY THE CLIENT */

   MY_IP_inet_address = m2m_pdp_get_my_ip();
   PrintToUart("IP String: %s", m2m_socket_bsd_addr_str(MY_IP_inet_address));


   sock = m2m_socket_bsd_socket(M2M_SOCKET_BSD_PF_INET,

M2M_SOCKET_BSD_SOCK_DGRAM,M2M_SOCKET_BSD_IPPROTO_UDP);

   if(M2M_SOCKET_BSD_INVALID_SOCKET == sock)
   {
     PrintToUart("Socket handle FAILURE");                    /* See chapter 19.1.8 PrintToUart */
     return;
   }

   PrintToUart ("Socket handle: SUCCESS");

   memset(&sa, 0, sizeof(sa));
   sa.sin_family = M2M_SOCKET_BSD_AF_INET;
   sa.sin_addr.s_addr = M2M_SOCKET_BSD_INADDR_ANY;
   sa.sin_port = m2m_socket_bsd_htons(PORT);

   if (M2M_SOCKET_BSD_INVALID_SOCKET ==
      m2m_socket_bsd_bind(sock,(struct M2M_SOCKET_BSD_SOCKADDR *)&sa, sizeof(struct
                                  M2M_SOCKET_BSD_SOCKADDR)))
   {
      m2m_socket_bsd_close(sock);
      return;
   }

   PrintToUart ("Socket bind: SUCCESS");

   m2m_socket_bsd_set_sock_opt(sock, M2M_SOCKET_BSD_SOL_SOCKET, M2M_SOCKET_BSD_SO_RCVTIMEO,
                                        &timeOutVal, sizeof(timeOutVal));

   for (;;)
   {
      PrintToUart ("recv_from loop");
      recsize = m2m_socket_bsd_recv_from(sock, buffer, 1024, 0, (struct M2M_SOCKET_BSD_SOCKADDR *)&sa,
                &fromlen);

      if (recsize < 0)
      {
       PrintToUart ("Receiving: FAILURE");
       return;
```

```
    }

    PrintToUart ("Character received %d", recsize);
    PrintToUart ("%s", (CHAR *)buffer);

    m2m_os_sleep_ms(1000);
  }
}
```

### 19.1.16. SSL/TCP Client

This chapter shows a simple SSL/TCP client example. Before show it, here is summarized how to create an SSL connection.

Create the sets of the SSL configurations, use:

- **m2m_ssl_create_service_from_file(…)** API, to create a set of SSL service session named, for example, service_1, service_2, service_3, … service_n.

- **m2m_ssl_create_context (…)** API, to create a set of contexts named, for example, context_1, context_2, context _3, … context _m.

Create a socket, use:

- **m2m_socket_bsd_socket(…)** API, to get, for example socket_fd socket handle.

Merge the configurations sets and the socket, use:

- **m2m_ssl_securesocket(…)** API to merge, for example: service_3, context_1, socket_fd

Establish the TCP connection, use

- **m2m_socket_bsd_connect(...)** API.

On the TCP connection, start the SSL handshake. One SSL connection at a time can be activated, use

- **m2m_ssl_connect(…)** API.

Send/Receive, use:

- **m2m_ssl_encode_send(…)/ m2m_ssl_decode(…)** API.

Close SSL connection, use:

- **m2m_ssl_delete_connection(…)** API.

Here is the example.

```
void tcp_client()
{
struct M2M_SOCKET_BSD_SOCKADDR_IN stSockAddr;
M2M_SOCKET_BSD_SOCKET SocketFD ;
M2M_SOCKET_BSD_FD_SET set;
CHAR buf_send[] = "Hello from AppZone!";
CHAR buf_recv[100];
INT32 Res;
UINT32 addr = 0;
UINT16 PORT = YYYY;                                    /* SET PORT */
CHAR IP_SERVER[] = "XXX.XXX.XXX.XXX";                  /* SET THE IP OF THE SERVER */
INT32 namelen = 0;

/*SSL variables*/
const char certFile="certFile_path";
const char privFile="privFile_path";
const char privPass="privPass_path";
const char trustedCAFile="trustedCAFile";
INT32 flags=NULL;
INT32 result=NULL;

M2M_SSL_SERVICE_SESSION session_ID;
M2M_SSL_CONTEXT_ID_TYPE SSLCtxID;
M2M_SSL_CONNECTION_CONTEXT connectionHandle;



/* Create the sets of the SSL configurations */
```

```
/* Create a new SSL service session */
session_ID=m2m_ssl_create_service_from_file(&certFile,&privFile,&privPass,&trustedCAFile,flags,&result );
if(!session_ID) return;

result=NULL;

/* Create a new SSL context by setting Protocol Version, Cipher Suite and Authorization Type */

SSLCtxID=m2m_ssl_create_context(M2M_SSL_Client_Method_TLSV1_2,
            M2M_TLS_RSA_WITH_AES_128_CBC_SHA,M2M_SSL_SERVER_CLIENT_AUTH_TYPE,&result);
if(!ssl_context) return;


/* end of SSL configuration */


/* create a socket */
SocketFD = m2m_socket_bsd_socket(M2M_SOCKET_BSD_PF_INET, M2M_SOCKET_BSD_SOCK_STREAM,
                                                    M2M_SOCKET_BSD_IPPROTO_TCP);



if (M2M_SOCKET_BSD_INVALID_SOCKET == SocketFD)
 {
  PrintToUart("Socket handle FAILURE");          /* See chapter 19.1.8 PrintToUart */
  return;
 }

PrintToUart("Socket handle: SUCCESS");

/* Merge the configurations sets and the socket */
connectionHandle=m2m_ssl_securesocket(session_ID,SSLCtxID,&SocketFD,&result);
if(!connectionHandle) return;


/* Establish the TCP connection */
memset(&stSockAddr, 0, sizeof(struct M2M_SOCKET_BSD_SOCKADDR_IN));

stSockAddr.sin_family = M2M_SOCKET_BSD_PF_INET;
stSockAddr.sin_port = m2m_socket_bsd_htons(PORT);
stSockAddr.sin_addr.s_addr = m2m_socket_bsd_inet_addr(IP_SERVER);

if (M2M_SOCKET_BSD_INVALID_SOCKET ==
    m2m_socket_bsd_connect(SocketFD, (M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                                    M2M_SOCKET_BSD_SOCKADDR_IN)))
 {
  PrintToUart("Socket connects: FAILURE");
  m2m_socket_bsd_close(SocketFD);
   return;
 }

PrintToUart("Socket connects: SUCCESS");

Res = m2m_socket_bsd_get_sock_name(SocketFD, ( M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, &namelen );

if (Res == 0)
 {
   /* IP address and Port */
   PrintToUart ("IP: %s",m2m_socket_bsd_addr_str(stSockAddr.sin_addr.s_addr));
   PrintToUart ("PORT: %u", m2m_socket_bsd_ntohs(stSockAddr.sin_port));
 }
else
 {
  PrintToUart ("get_socket_name: FAILURE");
  return;
 }


/* On the TCP connection, start the SSL handshake, if error return */

if(!m2m_ssl_connect(connectionHandle)) return;


/* Send message to server */
Res = m2m_socket_bsd_send_buf_size(SocketFD);
```

```
if (Res > sizeof(buf_send))
 {
  PrintToUart ("Available buffer space = %d",Res);
  /* SSL send*/
  Res=m2m_ssl_encode_send(connectionHandle,buf_send, sizeof(buf_send));
 }

if(Res < 0)
 {
  PrintToUart ("Sending: FAILURE");
 }
  else
 {
  PrintToUart ("Sending: SUCCESS, character sent = %d", Res);
 }

/* receive message from server */
/* SSL receive*/
Res=m2m_ssl_decode(connectionHandle,buf_send,sizeof(buf_send));

if(Res < 0)
 {
  PrintToUart ("Receiving: FAILURE");
 }
else
 {
  PrintToUart ("Receiving: SUCCESS, characters received = %d", Res);
  PrintToUart("%s", buf_recv);
 }

/* Close SSL connection */
m2m_ssl_delete_connection(connectionHandle);

/* Clean SSL configuration */
m2m_ssl_delete_context(SSLCtxID);
m2m_ssl_delete_service (session_ID);

if(m2m_socket_bsd_close(SocketFD) == 0)
 {
  PrintToUart ("Close SUCCESS");
 }
else
{
 PrintToUart ("Close FAILURE");
}

 return;
}
```

### 19.1.17. Get DNS IPv6 addresses

```
CHAR *pName6_tmpDnsPrim;
CHAR *pName6_tmpDnsSec;
M2M_SOCKET_BSD_SOCKADDR_IN6 pdns_addr6;
M2M_SOCKET_BSD_SOCKADDR_IN6 sdns_addr6;
UINT16 i;


ret = m2m_socket_bsd_get_dns_ip6(( M2M_SOCKET_BSD_IN6_ADDR *)&pdns_addr6.sin6_addr,(
M2M_SOCKET_BSD_IN6_ADDR *)&sdns_addr6.sin6_addr);

if (ret < 0)
   {
     PrintToUart(">> DNS error condition found: return value < 0 ret:%d \n",ret);
   }
else
   {
```

```
  pName6_tmpDnsPrim = m2m_socket_bsd_addr_str_ip6(&pdns_addr6.sin6_addr);
  PrintToUart(" primaryDNS hex : %s\r\n",pName6_tmpDnsPrim);                    /* See chapter 19.1.8 PrintToUart */

  PrintToUart(" primaryDNS dotted: \r\n");

  for (i=0; i<=14; i++)
    {
      PrintToUart("%d.",pdns_addr6.sin6_addr.v6_v.addr8[i]);
    }
  PrintToUart("%d \r\n",pdns_addr6.sin6_addr.v6_v.addr8[15]);

  pName6_tmpDnsSec = m2m_socket_bsd_addr_str_ip6(&sdns_addr6.sin6_addr);
  PrintToUart(" secondaryDNS hex: %s\r\n",pName6_tmpDnsSec);

  PrintToUart(" secondaryDNS dotted: \r\n");

  for (i=0; i<=14; i++)
    {
      PrintToUart("%d.",sdns_addr6.sin6_addr.v6_v.addr8[i]);
    }
  PrintToUart("%d \r\n",sdns_addr6.sin6_addr.v6_v.addr8[15]);

  m2m_os_sleep_ms( 2000 );
 }
```

### 19.1.18. M2M_SOCKET_BSD_TCP_CONNTIME Option

**M2M_SOCKET_BSD_TCP_CONNTIME** option is used to set the connection timeout. If you use the **m2m_socket_bsd_connect(...)** API in blocking mode and it does not return the control, when the timeout is expired the API is forced to return the control.

Note: only HE910/UE910 Modules Series provide this feature.

```
……

INT32 connTime = 60000;

……

m2m_socket_bsd_set_sock_opt(sockID,M2M_SOCKET_BSD_IPPROTO_TCP,M2M_SOCKET_BSD_TCP_CONNTIME,&c
onnTime,(INT32)sizeof(connTime));

memset (&stSockAddr, 0, sizeof (struct M2M_SOCKET_BSD_SOCKADDR_IN));

stSockAddr.sin_family = M2M_SOCKET_BSD_PF_INET;
stSockAddr.sin_port = m2m_socket_bsd_htons(PORT);
stSockAddr.sin_addr.s_addr = m2m_socket_bsd_inet_addr(IP_SERVER);

if (M2M_SOCKET_BSD_INVALID_SOCKET ==
   m2m_socket_bsd_connect(SocketFD, (M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                          M2M_SOCKET_BSD_SOCKADDR_IN)))

……
```

### 19.1.19. M2M_SOCKET_BSD_TCP_KEEPALIVE Option

**M2M_SOCKET_BSD_TCP_KEEPALIVE** option is used to send on the connection the keepalive message to prevent this connection from being broken.

```
M2M_SOCKET_BSD_SOCKET tcp_socket = M2M_SOCKET_BSD_INVALID_SOCKET;
INT32 keep_alive = 0;

tcp_socket = m2m_socket_bsd_socket (M2M_SOCKET_BSD_AF_INET, M2M_SOCKET_BSD_SOCK_STREAM,
                                    M2M_SOCKET_BSD_IPPROTO_TCP);

keep_alive = 1;
```

```
/* Enable the keepalive feature */
res = m2m_socket_bsd_set_sock_opt (tcp_socket, M2M_SOCKET_BSD_SOL_SOCKET,
                                   M2M_SOCKET_BSD_SO_KEEPALIVE, &keep_alive, sizeof(keep_alive));

/*Set the timeout value after which the keepalive will start */
keep_alive = 40000;
res = m2m_socket_bsd_set_sock_opt (tcp_socket, M2M_SOCKET_BSD_IPPROTO_TCP,
                                   M2M_SOCKET_BSD_TCP_KEEPALIVE, &keep_alive, sizeof(keep_alive));

/* Define the remote socket parameters (port, ip address …) */
Memset (&stSockAddr, 0, sizeof (struct M2M_SOCKET_BSD_SOCKADDR_IN));

stSockAddr.sin_family = M2M_SOCKET_BSD_PF_INET;
stSockAddr.sin_port = m2m_socket_bsd_htons(PORT);
stSockAddr.sin_addr.s_addr = m2m_socket_bsd_inet_addr(IP_SERVER);


/* Attempt to connect to the socket */
if (M2M_SOCKET_BSD_INVALID_SOCKET == m2m_socket_bsd_connect (tcp_socket,
(M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct M2M_SOCKET_BSD_SOCKADDR_IN)))
……
```

### 19.1.20. MD5 Hash Function

```
……
#include "m2m_type.h"
#include "m2m_sec_api.h"
……

CHAR body_of_file[] = "Text, Text, Text...";

CHAR *str_apires[] =
            {
             "M2M_API_RESULT_INVALID_ARG",
             "M2M_API_RESULT_FAIL",
             "M2M_API_RESULT_SUCCESS"
            };

INT8 i;

M2M_API_RESULT apires;
M2M_T_MD5_HANDLE MD5 = INVALID_DIG_HANDLE;
M2M_T_SHA_HANDLE SHA = INVALID_DIG_HANDLE;
M2M_T_SHA256_HANDLE SHA256 = INVALID_DIG_HANDLE;

DIG_RESULT_T *MD5_res = NULL;
DIG_RESULT_T *SHA_res = NULL;
DIG_RESULT_T *SHA256_res = NULL;

apires = m2m_MD5_Init(&MD5);
PrintToUart("apires m2m_MD5_Init : %s \r\n", str_apires[apires+1]);

apires = m2m_DIGEST_alloc_res((void *) MD5, &MD5_res);
PrintToUart("apires m2m_DIGEST_alloc_res : %s \r\n", str_apires[apires+1]);

apires = m2m_MD5_Update(MD5, (UINT8*)body_of_file, sizeof(body_of_file));
PrintToUart("apires m2m_MD5_Update : %s \r\n", str_apires[apires+1]);

apires = m2m_MD5_Final(&MD5, MD5_res);
PrintToUart("apires m2m_MD5_Final 1: %s \r\n", str_apires[apires+1]);

PrintToUart("res =  ");
for (i=0; i<MD5_res→size; i++)
 {
   PrintToUart("%d ", *(MD5_res→result+i));
 }

apires = m2m_DIGEST_destroy_res(&MD5_res);
PrintToUart("\r\napires m2m_DIGEST_destroy_res : %s \r\n", str_apires[apires+1]);
```

### 19.1.21. Watchdog

```
#include "m2m_type.h"
#include "m2m_os_api.h"
#include "m2m_hw_watchdog_api.h"

static INT32 refresh_task (INT32 type, INT32 nTimes, INT32 msec)
{
  UINT32 cnt = 0;

  While(cnt < nTimes)
   {
     m2m_hw_watchdog_refresh();
     m2m_os_sleep_ms((UINT32)msec);
     cnt++;
   }
  return 0;
}


void WDG_example ( void )
{
  INT32 taskId;
  M2M_WATCHDOG_OPTIONS wdgOpt;
  int conf_id = 5;                                  /* AZ internal tasks watchdog timeout in sec */

 /* AZ user tasks watchdog timeout */
  M2M_WATCHDOG_TIMEOUT wdg_timeout = M2M_WATCHDOG_TIMEOUT_NORMAL;
  UINT32 nof_kicks = 5;                             /* number of refreshes */
  UINT32 kick_period_ms = 11000*.9;                 /* refresh period in msec */

  taskId = m2m_os_create_task( M2M_OS_TASK_STACK_M, 10, M2M_OS_TASK_MBOX_M,refresh_task );

  m2m_os_sleep_ms(200);
  wdgOpt.timeout = conf_id;
  m2m_hw_watchdog_conf(&wdgOpt);
  m2m_hw_watchdog_enable(wdg_timeout);
  m2m_os_sleep_ms( 100 );
  m2m_os_send_message_to_task(taskId, 0, nof_kicks, kick_period_ms);
  m2m_os_sleep_ms(kick_period_ms*nof_kicks);
  m2m_hw_watchdog_disable();
  m2m_os_sleep_ms( 200 );
  m2m_os_destroy_task(taskId);

  while(1)
   {
     m2m_os_sleep_ms(1000);
   }
}
```

## 19.2. Declarations of C Identifiers

This section collects the declarations of the C identifiers used by the function prototypes described in the present User Guide.

### 19.2.1. m2m_cb_app_func C Identifiers

```
/* M2M_OS_MAX_PROCESS: Max number of tasks */
#define    M2M_OS_MAX_PROCESS                32

/* M2M_ARGC_MAX: Max number of usable argv strings  */
#define M2M_ARGC_MAX                         4

/* M2M_ARGV_MAXTOKEN: Each argv[] param passed contains a token string with this size */
#define M2M_ARGV_MAXTOKEN                    15
```

### 19.2.2. m2m_clock_api C Identifiers

```
/* M2M date structure */
typedef struct {
            CHAR     year;
            CHAR     month;
            CHAR     day;
} M2M_T_RTC_DATE;


/* M2M Time structure */
typedef struct {
            CHAR     hour;
            CHAR     minute;
            CHAR     second;
            CHAR     timeZone;
            CHAR     dst;
} M2M_T_RTC_TIME;


/* M2M Timeval structure (seconds and milliseconds, range 0-999, since epoch) */
struct M2M_T_RTC_TIMEVAL {
            INT32    tv_sec;
            INT32    tv_msec;
};


/* M2M_T_RTC_TIMEZONE: M2M Time zone structure (time zone, expressed in quarter of an hour, range is -47...+48, and
   Daylight Saving Time adjustment, range is 0-2) */
struct M2M_T_RTC_TIMEZONE {
        INT32       tz_tzone;
        INT32       tz_dst;
};

/* M2M RTC result definition. */
typedef enum {
            M2M_RTC_SUCCESS,                    /* Success */
            M2M_RTC_ALARM_LIMIT_EXCEEDED,       /* Too many alarms are set */
            M2M_RTC_INVALID_ARG,                /* Invalid argument */
            M2M_RTC_FAILURE                     /* Failure */
}M2M_T_RTC_RESULT;
```

### 19.2.3. m2m_fs_api C Identifiers

```
/* M2M File Handle type definition */
typedef INT32 *M2M_T_FS_HANDLE;

/* M2M_T_FS_ERROR_TYPE enumeration defining the M2M FS error codes. */
typedef enum {
  M2M_F_NO_ERROR,
  M2M_F_ERR_INVALIDDRIVE,
  M2M_F_ERR_NOTFORMATTED,
  M2M_F_ERR_INVALIDDIR,
  M2M_F_ERR_INVALIDNAME,
  M2M_F_ERR_NOTFOUND,
  M2M_F_ERR_DUPLICATED,
  M2M_F_ERR_NOMOREENTRY,
  M2M_F_ERR_NOTOPEN,
  M2M_F_ERR_EOF,
  M2M_F_ERR_RESERVED,
  M2M_F_ERR_NOTUSEABLE,
  M2M_F_ERR_LOCKED,
  M2M_F_ERR_ACCESSDENIED,
  M2M_F_ERR_NOTEMPTY,
  M2M_F_ERR_INITFUNC,
  M2M_F_ERR_CARDREMOVED,
  M2M_F_ERR_ONDRIVE,
  M2M_F_ERR_INVALIDSECTOR,
  M2M_F_ERR_READ,
  M2M_F_ERR_WRITE,
  M2M_F_ERR_INVALIDMEDIA,
  M2M_F_ERR_BUSY,
  M2M_F_ERR_WRITEPROTECT,
  M2M_F_ERR_INVFATTYPE,
  M2M_F_ERR_MEDIATOOSMALL,
  M2M_F_ERR_MEDIATOOLARGE,
  M2M_F_ERR_NOTSUPPSECTORSIZE,
  M2M_F_ERR_UNKNOWN,
  M2M_F_ERR_DRVALREADYMNT,
  M2M_F_ERR_TOOLONGNAME,
  M2M_F_ERR_NOTFORREAD,
  M2M_F_ERR_DELFUNC,
  M2M_F_ERR_ALLOCATION,
  M2M_F_ERR_INVALIDPOS,
  M2M_F_ERR_NOMORETASK,
  M2M_F_ERR_NOTAVAILABLE,
  M2M_F_ERR_TASKNOTFOUND,
  M2M_F_ERR_UNUSABLE,
  M2M_F_ERR_CRCERROR,
  M2M_F_ERR_CARDCHANGED

} M2M_T_FS_ERROR_TYPE;


/* M2M_T_FS_RUN_PERM_MODE_TYPE enumeration defining the M2M FS run permission set mode. */
typedef enum {
  M2M_F_RUN_PERM_MODE_RESET_ALL,
  M2M_F_RUN_PERM_MODE_SET,
  M2M_F_RUN_PERM_MODE_SET_RESET_OTHERS

} M2M_T_FS_RUN_PERM_MODE_TYPE;
```

### 19.2.4. m2m_hw_api C Identifiers

```
/* Edge configuration type. Use to select the edge to trigger the interrupt on a GPIO. */
typedef enum
{
   M2M_NO_EDGE = 0,          /* INT disable */
   M2M_RISING_EDGE,
   M2M_FALLING_EDGE,
   M2M_BOTH_EDGES,
}M2M_INT_FRONT;


/*Used to get Opening state of communication channel*/
typedef enum
{
 HW_USB_CABLE_DETACHED,
 HW_CLOSED,
 HW_OPENED,
 SW_CLOSED,
}STATE_T;

/*Used to get software state for Usb or Uart channel*/
typedef struct
{
  STATE_T Open;
  UINT8 BlockingRx;
  UINT8 BlockingTx;
  UINT8 IsAt;
  UINT8 IsRcv;
}USB_UART_STATE;

/* UART port handle. */
typedef INT32 M2M_T_HW_UART_HANDLE;


/* M2M UART results definition */
typedef enum {
              M2M_HW_UART_RESULT_SUCCESS = 0,
              M2M_HW_UART_RESULT_FAIL,
              M2M_HW_UART_RESULT_NOT_SUPPORTED,
              M2M_HW_UART_RESULT_INVALID_ARG
}M2M_T_HW_UART_RESULT;


typedef INT32 M2M_T_HW_USB_HANDLE;

/* The MAX number of USB instances is 3. The USB instance identifies one USB channel. Not all USB instances are always
    available, the number depends on module configuration, see AT#PORTCFG command */

typedef enum
{
  USER_USB_INSTANCE_0,
  USER_USB_INSTANCE_1,
  USER_USB_INSTANCE_2,
  USER_USB_INSTANCE_ERR
}USER_USB_INSTANCE_T;

/*All possible USB channels that can be used: NOT ALL are always available as above */
typedef enum
{
  USB_CH_NONE,
  USB_CH0 = 1,
  USB_CH1,
  USB_CH2,
  USB_CH3,
  USB_CH4,
  USB_CH5,
  USB_CH_AUTO,
  USB_CH_DEFAULT,
  USB_CH_NUM,
}M2M_USB_CH;
```

```
/*Command selector for USB channels */
typedef enum
{
  M2M_USB_NO_ACTION = 0,
  M2M_USB_BLOCKING_SET,
  M2M_USB_RCV_FUNC,
  M2M_USB_HW_OPTIONS_GET,          /* Not used for USB channel */
  M2M_USB_HW_OPTIONS_SET,          /* Not used for USB channel */
  M2M_USB_AT_MODE_SET,
  M2M_USB_CLEAR_RX,
  M2M_USB_RX_BLOCKING_SET,
  M2M_USB_TX_BLOCKING_SET,
  M2M_USB_ACTION_SELECTOR_NUM,
}M2M_USB_ACTION_SELECTOR;


/*error codes for USB handle */
#define M2M_HW_USB_UART_HANDLE_GENERIC_ERR                (-10)
#define M2M_HW_USB_UART_HANDLE_PORTCFG_ERR                (-5)
#define M2M_HW_USB_UART_HANDLE_NEW_HWCH_UNAVAILABLE       (-4)
#define M2M_HW_USB_UART_HANDLE_HW_ERR                     (-3)
#define M2M_HW_USB_UART_HANDLE_HWPORT_ALREADY_OPEN        (-2)
#define M2M_HW_USB_UART_HANDLE_INVALID_PORT               (-1)


/* M2M_T_HW_UART_IO_HW_OPTIONS used with M2M_HW_UART_IO_HW_OPTIONS_SET and
   M2M_HW_UART_IO_HW_OPTIONS_GET OPTIONS */
typedef struct {
   UINT32        baudrate;        /* example: 115200 bits/sec */
   UINT8         databits;        /* example: 8  */
   UINT8         stop_bits;       /* example: 1 */
   UINT8         parity;          /* parity: 0 none; 1 odd; 2 even */
   UINT8         flow_ctrl;       /* set flow control: 0 none; 1 hw flow control (default) */
} M2M_T_HW_UART_IO_HW_OPTIONS;


typedef struct{
    UINT32 timeout;          /* maximum sleep timeout */
} M2M_T_HW_SLEEP_MODE_CFG_OPTIONS;
```

### 19.2.5. m2m_spi_api C Identifiers

```
/* M2M_SPI_BUFFER_LEN: Max buffers length (in bytes) */
#define M2M_SPI_BUFFER_LEN 256

/* M2M_T_SPI_RESULT: SPI result definition */
typedef enum {
        M2M_SPI_SUCCESS = 0,                    /* Success */
        M2M_SPI_FAILURE,                        /* Generic failure */
        M2M_SPI_OPEN_ERROR,                     /* Device open error */
        M2M_SPI_OPTS_GET_ERROR,
        M2M_SPI_OPTS_SET_ERROR,
        M2M_SPI_CLOCK_FREQUENCY_ERROR,          /* Frequency speed error */
        M2M_SPI_CLOCK_MODE_ERROR,               /* SPI mode error */
        M2M_SPI_BIT_PER_FRAME_ERROR,
        M2M_SPI_DMA_THRESHOLD_ERROR,
        M2M_SPI_POWER_STATE_ON_ERROR,
        M2M_SPI_POWER_STATE_OFF_ERROR,
        M2M_SPI_DEVICE_SELECTION_ERROR,         /* Chip select error */
        M2M_SPI_USIF_SELECTION_ERROR,
        M2M_SPI_RAW_IO_ERROR,                   /* Reading/Writing error */
        M2M_SPI_USIF_ERROR,                     /* usif_num parameter error */
        M2M_SPI_BUFFER_SIZE_ERROR,              /* len parameter error */
        M2M_SPI_MODE_ERROR,                     /* mode parameter error */
        M2M_SPI_SPEED_ERROR                     /* speed parameter error */
}M2M_T_SPI_RESULT;
```

### 19.2.6. m2m_i2C_api C Identifiers

```
/* M2M_HW_I2C_MAX_BUF_LEN: Max length */
#define M2M_HW_I2C_MAX_BUF_LEN 256

/* M2M_T_HW_I2C_RESULT: I2C result definition */
typedef enum {
        M2M_HW_I2C_RESULT_SUCCESS = 0,
        M2M_HW_I2C_ACK_FAIL,
        M2M_HW_I2C_RESULT_INVALID_ARG,    /* used only by m2m_hw_i2c_write(...) and  m2m_hw_i2c_read(...) */
        M2M_HW_I2C_RESULT_INVALID_PINS    /* used only by m2m_hw_i2c_conf(...) */
}M2M_T_HW_I2C_RESULT;
```

### 19.2.7. m2m_network_api C Identifiers

```
/* M2M max network name (long) */
#define M2M_NETWORK_MAX_LONG_ALPHANUMERIC  16

/* M2M max network name (short) */
#define M2M_NETWORK_MAX_SHORT_ALPHANUMERIC          8

/* M2M max network number */
#define M2M_NETWORK_MAX_NUMERIC                     8

/* M2M max neighbor */
#define M2M_NETWORK_NUM_OF_NEIGHBOR                 7

/* M2M UMTS max neighbor */
#define M2M_NETWORK_NCELL_MAX_TOTAL_UMTS_CELLS      25

/* M2M max cell length */
#define M2M_NETWORK_MAX_CELL_LENGTH                 8

/* M2M max LAC length */
#define M2M_NETWORK_MAX_LAC_LENGTH                  5

/* M2M_T_NETWORK_AVAILABLE_NETWORK: available network information */
typedef struct _M2M_T_NETWORK_AVAILABLE_NETWORK
{
  UINT16 nStat;
  CHAR          longAlphanumeric[M2M_NETWORK_MAX_LONG_ALPHANUMERIC];
  CHAR          shortAlphanumeric[M2M_NETWORK_MAX_SHORT_ALPHANUMERIC];
  CHAR          Numeric[M2M_NETWORK_MAX_NUMERIC];
  UINT16 AcT;
} M2M_T_NETWORK_AVAILABLE_NETWORK;


/* M2M_T_NETWORK_CURRENT_NETWORK: Current network information */
typedef struct _M2M_T_NETWORK_CURRENT_NETWORK
{
  UINT16 nMode;
  UINT16 nFormat;
  CHAR          longAlphanumeric[M2M_NETWORK_MAX_LONG_ALPHANUMERIC];
  UINT16 AcT;
} M2M_T_NETWORK_CURRENT_NETWORK;


/* Network cell neighbor information */
typedef struct _M2M_T_NETWORK_CELL_NEIGHBOR
{
  INT32         nARFCN;
  INT32         nBSIC;
  INT32         nSignalStrength;
} M2M_T_NETWORK_CELL_NEIGHBOR;



/* Type of CELL */
typedef enum _M2M_T_NETWORK_CELL_TYPE
{
   CELL_TYPE_ACTIVE_SET,              /* Cell belongs to the Active set (CELL_DCH)*/
   CELL_TYPE_VIRTUAL_ACTIVE_SET,      /* Cell belongs to the Virtual Active set (CELL_DCH)*/
   CELL_TYPE_MONITORED,               /* Cells in the SIB 11/12 "BA"-list */
   CELL_TYPE_DETECTED,                /* Cell is a detected UMTS cell (CELL_DCH) */
   CELL_TYPE_UMTS_CELL,               /* Cell is a UMTS neighbor cell in GSM mode */
```

```
    CELL_TYPE_UMTS_RANKED,              /* Cell is a UMTS neighbor cell (all states but CELL_DCH) */
    CELL_TYPE_UMTS_NOT_RANKED,          /* Cell is a UMTS neighbor cell (all states but CELL_DCH) */
    CELL_TYPE_SERVING,                  /* Serving Cell*/
    CELL_TYPE_INVALID_CELL_TYPE         /* Indicates empty / invalid entries in cell list  */
} M2M_T_NETWORK_CELL_TYPE;
```

```
/* UMTS Network cell neighbor information */
typedef struct _M2M_T_UMTS_NETWORK_CELL_NEIGHBOR
{
  M2M_T_NETWORK_CELL_TYPE  cellType;          /* type of cell */
  UINT16          psc;            /* Primary scrambling code */
  UINT16          rscp;           /* Received Signal Code Power (dBm - positive value presented positive )(0xFF) */
  UINT8           ecn0;           /* EC2N0 (dB - positive value presented positive) (0xFF) */
  UINT16          uarfcn;         /* DL UARFCN (0xFFFF) */
} M2M_T_UMTS_NETWORK_CELL_NEIGHBOR;


/* M2M_T_NETWORK_CELL_INFORMATION: Network cell information (neighbor list) */
typedef struct _M2M_T_NETWORK_CELL_INFORMATION
{
 M2M_T_NETWORK_CELL_NEIGHBOR        neighbors[M2M_NETWORK_NUM_OF_NEIGHBOR];
                                                    /* serving and neighbor cell info in GSM case
                                                   */
 M2M_T_UMTS_NETWORK_CELL_NEIGHBOR
umtsNeighbors[M2M_NETWORK_NCELL_MAX_TOTAL_UMTS_CELLS];
                                                    /* serving and neighbor cell info in UMTS case
                                                   */
} M2M_T_NETWORK_CELL_INFORMATION;


/* M2M_T_NETWORK_REG_STATUS_INFO: Registration status information */
typedef struct
{
 UINT16          status;
 UINT16          LAC;
 UINT16          cell_id;
 UINT8           LAC_string[M2M_NETWORK_MAX_LAC_LENGTH];
 CHAR            cell_id_string[M2M_NETWORK_MAX_CELL_LENGTH];
 UINT16          AcT;
}M2M_T_NETWORK_REG_STATUS_INFO;

/* M2M_T_NETWORK_GREG_STATUS_INFO: Registration gprs status information */
typedef struct
{
 UINT16  gprs_status;
 UINT16  LAC;
 UINT32  cell_id;
 UINT8   LAC_string[M2M_NETWORK_MAX_LAC_LENGTH];
 CHAR    cell_id_string[M2M_NETWORK_MAX_CELL_LENGTH];
 UINT16  AcT;
}M2M_T_NETWORK_GREG_STATUS_INFO;
```

### 19.2.8. m2m_os_api C Identifiers

```
/* String length (in bytes) of the pool_info ptr to be passed into m2m_os_get_mem_info(). */
#define M2M_OS_MEM_POOL_INFO_STRING_LEN 64

/* M2M_OS_MAX_SW_VERSION_STR_LENGTH: string length (in bytes) of the sw version to be passed into
   m2m_os_set_version(). */
#define M2M_OS_MAX_SW_VERSION_STR_LENGTH     40


/* M2M_CB_MSG_PROC */
typedef INT32 (*M2M_CB_MSG_PROC)(INT32, INT32, INT32);

/*  M2M_OS_TASK_STACK_SIZE: stack size of the task */
typedef enum
{
 M2M_OS_TASK_STACK_S,                /* 2K */
 M2M_OS_TASK_STACK_M,                /* 4K */
 M2M_OS_TASK_STACK_L,                /* 8K */
 M2M_OS_TASK_STACK_XL,               /* 16K */
M2M_OS_TASK_STACK_LIMIT

} M2M_OS_TASK_STACK_SIZE;


#define M2M_OS_TASK_PRIORITY_MAX  1
#define M2M_OS_TASK_PRIORITY_MIN  32
```

```
/*  M2M_OS_TASK_MBOX_SIZE: mbox size of the task */
typedef enum
{
  M2M_OS_TASK_MBOX_S,
  M2M_OS_TASK_MBOX_M,
  M2M_OS_TASK_MBOX_L,
  M2M_OS_TASK_MBOX_LIMIT

} M2M_OS_TASK_MBOX_SIZE;
```

### 19.2.9. m2m_os_lock_api C Identifiers

```
/* M2M_T_OS_LOCK: Lock handle used by m2m_os_lock_init(...)*/
typedef void *M2M_T_OS_LOCK;

/* M2M_T_OS_MSLOCK: Lock handle used by m2m_os_mslock_init(...)*/
typedef void *M2M_T_OS_MSLOCK;

/* M2M_T_OS_MTX: Lock handle used by m2m_os_mtx_init(...) */
typedef void *M2M_T_OS_MTX;

/* LOCK_RESULT_T */
typedef enum
{
 LOCK_ERROR = -1,
 LOCK_GOT = 0,
 LOCK_NOT_GOT,
 LOCK_TIMEOUTED,
 LOCK_DESTROYED,
}LOCK_RESULT_T;

/* MS_SEM_STATE */
typedef struct
{
  INT16              count;
  INT16              count_suspended;
  INT16              count_sem_waiting;
  UINT16             n_sem;
  UINT16             all_waiting;
  UINT16             all_locked;
  UINT16             max_lockable;
}MS_SEM_STATE;
```

### 19.2.10. m2m_sms_api C Identifiers

```
/* Memory storage location length (maximum) according to AT+CPMS command settings */
#define M2M_SMS_NUM_MEM_CHAR          4

/* SMS max status string length */
#define M2M_SMS_NUM_OF_STATUS_CHAR    13

/* SMS max address string length */
#define M2M_SMS_NUM_OF_ADDRESS_CHAR20

/* SMS max date string length */
#define M2M_SMS_DATE_CHAR             10

/* SMS max time string length */
#define M2M_SMS_TIME_CHAR             15

/* SMS max data (text or PDU) length composed of 176 max data length +1 for NULL termination */
#define M2M_SMS_DATA_CHAR             177



/* M2M_T_SMS_MEM_STORAGE */
typedef struct _M2M_T_SMS_MEM_STORAGE
{
   CHAR   mem[M2M_SMS_NUM_MEM_CHAR];          /* selected memory location */
   INT32  nUsed;                              /* space used (in Bytes) */
```

```
  INT32   nTotal;                                   /* total space (Bytes) */
} M2M_T_SMS_MEM_STORAGE;


/* M2M_T_SMS_INFO: SMS information */
typedef struct _M2M_T_SMS_INFO
{
  INT32   index;                                    /* The message index used to retrieve a message */
  CHAR   status[M2M_SMS_NUM_OF_STATUS_CHAR];        /* SMS status, i.e. REC READ, REC UNREAD */
  CHAR   originalAddress[M2M_SMS_NUM_OF_ADDRESS_CHAR];      /* SMS sender */
  CHAR   date[M2M_SMS_DATE_CHAR];                   /* SMS receive date */
  CHAR   time[M2M_SMS_TIME_CHAR];                   /* SMS receive time */
  CHAR   data[M2M_SMS_DATA_CHAR];                   /* SMS receive data (text or PDU) */
} M2M_T_SMS_INFO;
```

### 19.2.11. m2m_socket_api C Identifiers

```
/* Socket handle */
typedef INT32   M2M_SOCKET_BSD_SOCKET;


/* Invalid_Socket_handle */
#define M2M_SOCKET_BSD_INVALID_SOCKET  (M2M_SOCKET_BSD_SOCKET)(~0)

/* M2M Socket_Types */
#define M2M_SOCKET_BSD_SOCK_STREAM   SOCK_STREAM   /* Stream socket type used for TCP */
#define M2M_SOCKET_BSD_SOCK_DGRAM    SOCK_DGRAM    /* Datagram socket type used for UDP */
#define M2M_SOCKET_BSD_SOCK_RAW      SOCK_RAW      /* Raw socket type */

/* M2M Socket_Address_Families */
#define M2M_SOCKET_BSD_AF_UNSPEC     0       /* Unspecified Address Family */
#define M2M_SOCKET_BSD_AF_INET       2       /* Internetwork: UDP, TCP, etc. */
#define M2M_SOCKET_BSD_AF_INET6      10

/* M2M Socket Protocol Families */
#define M2M_SOCKET_BSD_PF_UNSPEC     0       /* Unspecified Protocol Family */
#define M2M_SOCKET_BSD_PF_INET       2       /* Internetwork: UDP, TCP, etc. */

/* M2M Socket_Protocols */
#define M2M_SOCKET_BSD_IPPROTO_IP    0       /* Dummy for IP */
#define M2M_SOCKET_BSD_IPPROTO_TCP   6       /* Transmission Control Protocol */
#define M2M_SOCKET_BSD_IPPROTO_UDP   17      /* User Datagram Protocol */
#define M2M_SOCKET_BSD_IPPROTO_ICMP  1       /* Internet Control Message Protocol */

/* ========================================================================== */

/* Level_number for M2M_socket_bsd_get_sock_opt() and M2M_socket_bsd_set_sock_opt() to apply to socket itself. */

#define M2M_SOCKET_BSD_SOL_SOCKET    0xffff      /* options for socket level */

/* M2M Socket_Option_Flags */
#define M2M_SOCKET_BSD_SO_DEBUG              SO_DEBUG          /* Turn on debugging info recording */
#define M2M_SOCKET_BSD_SO_ACCEPTCONN         SO_ACCEPTCONN /* Socket has had listen(), Not supported */
#define M2M_SOCKET_BSD_SO_REUSEADDR          SO_REUSEADDR   /* Allow local address reuse, always set */
#define M2M_SOCKET_BSD_SO_KEEPALIVE          SO_KEEPALIVE     /* Keep connections alive, not enabled by
                                                                 default */
#define M2M_SOCKET_BSD_SO_DONTROUTE          SO_DONTROUTE  /* Just use interface addresses */
#define M2M_SOCKET_BSD_SO_BROADCAST          SO_BROADCAST  /* Permit sending of broadcast  */

#define M2M_SOCKET_BSD_SO_LINGER             SO_LINGER         /* Linger on close if data present */
#define M2M_SOCKET_BSD_SO_OOBINLINE          SO_OOBINLINE    /* Leave received OOB data in line */
#define M2M_SOCKET_BSD_SO_DONTLINGER         (INT32) (~M2M_SOCKET_BSD_SO_LINGER) /* Don't Linger */

#define M2M_SOCKET_BSD_SO_SNDBUF             SO_SNDBUF       /* Send buffer size */
                                                             /* GE910: not supported */
#define M2M_SOCKET_BSD_SO_RCVBUF             SO_RCVBUF       /* Receive buffer size */
                                                             /* GE910: supported only for
                                                             M2M_socket_bsd_set_sock_opt(...) */

#define M2M_SOCKET_BSD_SO_SNDLOWAT   SO_SNDLOWAT   /* Send low-water mark */
#define M2M_SOCKET_BSD_SO_RCVLOWAT   SO_RCVLOWAT   /* Receive low-water mark */
#define M2M_SOCKET_BSD_SO_SNDTIMEO   SO_SNDTIMEO   /* Send timeout */
```

```
                                                    /* GE910: supported only for
                                                    M2M_socket_bsd_set_sock_opt(...) */
#define M2M_SOCKET_BSD_SO_RCVTIMEO    SO_RCVTIMEO   /* Receive timeout, supported. Only for
                                                    M2M_socket_bsd_set_sock_opt(...)*/
#define M2M_SOCKET_BSD_SO_ERROR       SO_ERROR      /* Get error status and clear  */
#define M2M_SOCKET_BSD_SO_TYPE        SO_TYPE       /* Get socket type, supported */
                                                     /* GE910: not supported */
#define M2M_SOCKET_BSD_TCP_NODELAY    TCP_NODELAY   /* Do not delay, coalesce the sent packets */
                                                     /* GE910: not supported */
#define M2M_SOCKET_BSD_TCP_CONNTIME   0x02          /* Connection timeout, supported. Only for
                                                    M2M_socket_bsd_set_sock_opt(...)*/
                                                     /* GE910: not supported */

#define M2M_SOCKET_BSD_TCP_KEEPALIVE  TCP_KEEPIDLE  /* TCP keepalive timer */

/* ========================================================================== */


/* M2M Socket constants for m2m_socket_bsd_shutdown() */
#define M2M_SOCKET_BSD_SHUT_RD    0x00              /* Read socket */
#define M2M_SOCKET_BSD_SHUT_WR    0x01              /* Write socket */
#define M2M_SOCKET_BSD_SHUT_RDWR  0x02              /* Read Write socket */


/* Structure used for manipulating linger option. */
typedef struct M2M_SOCKET_BSD_LINGER {
   INT32  l_onoff;                                  /* option on/off */
   INT32  l_linger;                                 /* linger time */
} M2M_SOCKET_BSD_LINGER;


/*: Structure used by TCP/IP stack to store most addresses. */
typedef struct M2M_SOCKET_BSD_SOCKADDR {
   UINT8   _internal_sa_len;                        /* INTERNAL USE ONLY */
   UINT8   sa_family;
   CHAR   sa_data[14];
} M2M_SOCKET_BSD_SOCKADDR;


/* ========================================================================== */

/* M2M Internet address.*/

/* Any internet address. */
#define M2M_SOCKET_BSD_INADDR_ANY                (UINT32) 0x00000000

/* Loopback internet address. */
#define M2M_SOCKET_BSD_INADDR_LOOPBACK           (UINT32) 0x7f000001

/* Broadcast internet address. */
#define M2M_SOCKET_BSD_INADDR_BROADCAST          (UINT32) 0xffffffff


/* ========================================================================== */


/* Structure for storing Internet address. */
typedef struct M2M_SOCKET_BSD_IN_ADDR {
   UINT32            s_addr;                       /* 32 bits inet address */
} M2M_SOCKET_BSD_IN_ADDR;


/* M2M_SOCKET_BSD_SOCKADDR_IN: Socket address, internet style. */
typedef struct M2M_SOCKET_BSD_SOCKADDR_IN {
   UINT8             _internal_sin_len;            /* INTERNAL USE ONLY */
   UINT8             sin_family;                   /* M2M Socket Protocol Families, e.g. M2M_SOCKET_BSD_PF_INET. */
   UINT16            sin_port;                     /* 16 bits port number. */
   M2M_SOCKET_BSD_IN_ADDR      sin_addr;           /* 32 bits inet address (IP). */
   CHAR              sin_zero[8];                  /* INTERNAL USE ONLY */
} M2M_SOCKET_BSD_SOCKADDR_IN;
```

```
/* M2M_SOCKET_BSD_IN6_ADDR */
typedef struct M2M_SOCKET_BSD_IN6_ADDR {
  UINT32              s_addr[4];
}M2M_SOCKET_BSD_IN6_ADDR;

/* M2M_SOCKET_BSD_IPV6_ADDR */
typedef struct M2M_SOCKET_BSD_IPV6_ADDR
{
    union
    {
        UINT8          addr8[16];
        UINT16         addr16[8];
        UINT32         addr32[4];
    }v6_v;

    #define addr8_s     v6_v.addr8
    #define addr16_s    v6_v.addr16
    #define addr32_s    v6_v.addr32
} M2M_SOCKET_BSD_IPV6_ADDR;


/* Socket address, internet style. */
typedef struct M2M_SOCKET_BSD_SOCKADDR_IN6 {
    UINT8               _internal_sin6_len;             /* INTERNAL USE ONLY */
    UINT8               sin6_family;                    /* M2M Socket Protocol Families, e.g. M2M_SOCKET_BSD_PF_INET. */
    UINT16              sin6_port;                      /* 16 bits port number. */
    UINT32              sin6_flowinfo;
    M2M_SOCKET_BSD_IPV6_ADDR    sin6_addr;      /* 32 bits inet address (IP). */
    UINT32              sin6_scope_id;
} M2M_SOCKET_BSD_SOCKADDR_IN6;

/* Structure returned by network data base library. */
typedef struct M2M_SOCKET_BSD_HOSTENT {
    CHAR*               h_name;             /* Official name of host */
    CHAR**              h_aliases;          /* Pointer to struct of aliases */
    INT32               h_addrtype;         /* Host address type, equals M2M_SOCKET_BSD_AF_INET */
    INT32               h_length;           /* Length of address */
    CHAR**              h_addr_list;        /* Pointer to array of pointers with inet v4 addresses */
} M2M_SOCKET_BSD_HOSTENT;

/* M2M_SOCKET_BSD_TIMEVAL: Structure used in m2m_socket_bsd_select() call. */
typedef struct M2M_SOCKET_BSD_TIMEVAL {
    INT32  m_tv_sec;                        /* seconds */
    INT32  m_tv_usec;                       /* microseconds */
} M2M_SOCKET_BSD_TIMEVAL;

/* FD set size used by m2m_socket_bsd_select(). */
#define M2M_SOCKET_BSD_FD_SETSIZE      1024


/* M2M_SOCKET_BSD_FD_SET: FD set used by m2m_socket_bsd_select(). */
typedef struct M2M_SOCKET_BSD_FD_SET {
   INT32               fd_count;                                                    /*How many are SET? */
   UINT32 fd_array[(M2M_SOCKET_BSD_FD_SETSIZE + 31)/32];                            /* Bit map of SOCKET Descriptors. */
} M2M_SOCKET_BSD_FD_SET;


typedef struct M2M_PDP_DATAVOLINFO
{
  UINT32        dataRec;
  UINT32        dataSent;
  UINT32        total;
}M2M_PDP_DATAVOLINFO;
```

```
/* M2M_NETWORK_EVENT: M2M Socket Network Event codes */
typedef enum
{
    M2M_SOCKET_EVENT_SOCKET_BREAK,          /* Connection closed by the server */
    M2M_SOCKET_EVENT_SOCKET_FAIL,           /* Connection error */
    M2M_SOCKET_EVENT_PDP_IPV6_ACTIVE,       /* PDP IPV6 activated */
    M2M_SOCKET_EVENT_PDP_ACTIVE,            /* PDP activated */
    M2M_SOCKET_EVENT_PDP_DEACTIVE,          /* PDP deactivated */
    M2M_SOCKET_EVENT_PDP_BREAK              /* PDP broken */

} M2M_NETWORK_EVENT;

/* M2M firewall rule element definitions */
/* M2M_FIREWALL_ELEMENT Ip Address and Mask */
typedef struct M2M_FIREWALL_ELEMENT
{
UINT32 ipAddr;
UINT32 ipMask;
} M2M_FIREWALL_ELEMENT;

/* M2M_FIREWALL_ELEMENT_IP6 Ip Address IPv6 and Mask Ipv6*/
typedef struct M2M_FIREWALL_ELEMENT_IP6
{
M2M_SOCKET_BSD_IPV6_ADDR ipAddr;
M2M_SOCKET_BSD_IPV6_ADDR ipMask ;
} M2M_FIREWALL_ELEMENT_IP6;

/* ========================================================================== */

/* M2M Socket I/O control options for  m2m_socket_bsd_ioctl() */

/* command to get the number of bytes to read */
#define M2M_SOCKET_BSD_FIONREAD                 0

/* command to select the blocking or non-blocking mode */
#define M2M_SOCKET_BSD_FIONBIO                  1

/* command to set a receive callback function. Not supported */
#define M2M_SOCKET_IO_READ_CB_FUNC              2

/* command to set an accept callback function, typically used for server. Not supported */
#define M2M_SOCKET_IO_ACCEPT_CB_FUNC            3


/* ========================================================================== */


/* M2M Socket_Error_Types */
/* Errors can be retrieved via the m2m_socket_errno() */

#define M2M_SOCKET_BSD_SOCKET_ERROR             (-1)
#define M2M_SOCKET_BSD_SOCKNOERROR              0
#define M2M_SOCKET_BSD_EUNDEFINED               1
#define M2M_SOCKET_BSD_EACCES                   2
#define M2M_SOCKET_BSD_EADDRINUSE               3
#define M2M_SOCKET_BSD_EADDRNOTAVAIL            4
#define M2M_SOCKET_BSD_EAFNOSUPPORT             5
#define M2M_SOCKET_BSD_EALREADY                 6
#define M2M_SOCKET_BSD_EBADF                    7
#define M2M_SOCKET_BSD_ECONNABORTED             8
#define M2M_SOCKET_BSD_ECONNREFUSED             9
#define M2M_SOCKET_BSD_ECONNRESET               10
#define M2M_SOCKET_BSD_EDESTADDRREQ             11
#define M2M_SOCKET_BSD_EFAULT                   12
#define M2M_SOCKET_BSD_EHOSTDOWN                13
#define M2M_SOCKET_BSD_EHOSTUNREACH             14
#define M2M_SOCKET_BSD_EINPROGRESS              15
#define M2M_SOCKET_BSD_EINTR                    16
#define M2M_SOCKET_BSD_EINVAL                   17
#define M2M_SOCKET_BSD_EISCONN                  18
#define M2M_SOCKET_BSD_EMFILE                   19
#define M2M_SOCKET_BSD_EMSGSIZE                 20
#define M2M_SOCKET_BSD_ENETDOWN                 21
#define M2M_SOCKET_BSD_ENETRESET                22
#define M2M_SOCKET_BSD_ENETUNREACH              23
#define M2M_SOCKET_BSD_ENOBUFS                  24
```

```
#define M2M_SOCKET_BSD_ENOPROTOOPT          25
#define M2M_SOCKET_BSD_ENOTCONN             26
#define M2M_SOCKET_BSD_ENOTSOCK             27
#define M2M_SOCKET_BSD_EOPNOTSUPP           28
#define M2M_SOCKET_BSD_EPFNOSUPPORT         29
#define M2M_SOCKET_BSD_EPROTONOSUPPORT      30
#define M2M_SOCKET_BSD_EPROTOTYPE           31
#define M2M_SOCKET_BSD_ESHUTDOWN            32
#define M2M_SOCKET_BSD_ESOCKTNOSUPPORT      33
#define M2M_SOCKET_BSD_ETIMEDOUT            34
#define M2M_SOCKET_BSD_EWOULDBLOCK          35

/* ========================================================================== */
/* M2M Socket_State */

/* socket is closed. */
#define M2M_SOCKET_STATE_CLOSED         0

/* socket is opened. */
#define M2M_SOCKET_STATE_OPEN           1

/* socket is connected. */
#define M2M_SOCKET_STATE_CONNECTED      2

/* ========================================================================== */
/* PDP_context_status */

/* PDP context is active. */
#define M2M_PDP_STATE_ACTIVE            0

/* PDP context is not yet active. */
#define M2M_PDP_STATE_NOT_ACTIVE        1

/* PDP context has errors. */
#define M2M_PDP_STATE_FAILURE           2

/* PDP operation is successful. Not used by m2m_pdp_activate(). */
#define M2M_PDP_STATE_SUCCESS           3

/* PDP operation (activate or deactivate) is in progress. Result will be notified through the m2m_cb_on_net_event() callback
*/
#define M2M_PDP_STATE_IN_PROGRESS       4


/* ========================================================================== */
/* network interface adapters */

/* maximum name length of a network interface (like eth0, ps5 etc). */
#define M2M_NET_MAX_IF_NAME             5

/* ========================================================================== */
```

### 19.2.12. m2m_ssl_api C Identifiers

```
/* M2M_SSL_SERVICE_SESSION: the M2M SSL service session is created by m2m_ssl_create_service_from_file(). Each
client app needs it's own SSL service session. */
typedef INT32* M2M_SSL_SERVICE_SESSION;

/* M2M_SSL_CONTEXT_ID_TYPE */
typedef INT32* M2M_SSL_CONTEXT_ID_TYPE;

/* M2M_SSL_CONNECTION_CONTEXT: M2M SSL connection context. Shall be created for each and every connection.
This context shall be used for encode/decode the specific connection. */
typedef INT32* M2M_SSL_CONNECTION_CONTEXT;

/* M2M_SSL_result_codes */
#define M2M_SSL_SUCCESS                 0       /* success */
#define M2M_SSL_REQUEST_SEND            1       /* Not used */
#define M2M_SSL_REQUEST_RECV            2       /* Not used */

/* Failure_return_codes MUST be < 0 */
#define M2M_SSL_FAILURE                 -1      /* Generic failure */
#define M2M_SSL_ARG_FAIL                -6      /* Failure due to bad function param */
```

```
#define M2M_SSL_PLATFORM_FAIL                   -7      /* Not used */
#define M2M_SSL_MEM_FAIL                        -8      /* Not used */
#define M2M_SSL_LIMIT_FAIL                      -9      /* Not used */
#define M2M_SSL_UNSUPPORTED_FAIL                -10     /* Not used */
#define M2M_SSL_PROTOCOL_FAIL                   -12     /* A protocol error occurred */
#define M2M_SSL_TIMEOUT_FAIL                    -13     /* A timeout occurred and MAY be an error */
#define M2M_SSL_INTERRUPT_FAIL                  -14     /* An interrupt occurred and MAY be an error */
#define M2M_SSL_WRITE_ERROR                     -15     /* An error occurred while encoding on socket */
#define M2M_SSL_READ_ERROR                      -16     /* An error occurred while decoding from socket */
#define M2M_SSL_END_OF_FILE                     -17     /* There is no data to read in SSL */
#define M2M_SSL_CLOSE_NOTIFY                    -18     /* SSL connection has been closed by remote host */
#define M2M_SSL_CERT_AUTH_FAIL                  -35     /* Authentication fails */
#define M2M_SSL_FULL                            -50     /* Not used */
#define M2M_SSL_ALERT                           -54     /* We've decoded an alert */
#define M2M_SSL_FILE_NOT_FOUND                  -55     /* File not found */
#define M2M_SSL_BAD_HANDLER                     -100    /* Configuration Handler not valid */
#define M2M_SSL_UNSUPPORTED_CONF                -101    /* Configuration not supported */

#define M2M_SSL_FALSE                           0       /* FALSE */
#define M2M_SSL_TRUE                            1       /* TRUE */


/* Public Key types for M2M_SSL_PUBLIC_KEY */
#define M2M_SSL_RSA                             1       /* Not used */
#define M2M_SSL_ECC                             2       /* Not used */
#define M2M_SSL_DH                              3       /* Not used */

/* M2M_SSL_Protocol_Version_E */
typedef enum _M2M_SSL_Protocol_Version_En {
  M2M_SSL_Client_Method_None       = 0x0000,   /* No Method (ERROR) */
  M2M_SSL_Client_Method_TLSV1      = 0x0002,   /*For TLS1.0 handshake protocol */
  M2M_SSL_Client_Method_TLSV1_1    = 0x0003,   /*For TLS1.1 handshake protocol */
  M2M_SSL_Client_Method_TLSV1_2    = 0x0004,   /*For TLS1.2 handshake protocol */
} M2M_SSL_Protocol_Version_E;

/* M2M_SSL_Cipher_Suite_E */
typedef enum _M2M_SSL_Cipher_Suite_En{
  M2M_SSL_REMOTE_SERVER_CS =0,
  M2M_TLS_RSA_WITH_RC4_128_MD5,
  M2M_TLS_RSA_WITH_RC4_128_SHA,
  M2M_TLS_RSA_WITH_AES_128_CBC_SHA,
  M2M_TLS_RSA_WITH_NULL_SHA,
  M2M_TLS_RSA_WITH_AES_256_CBC_SHA
} M2M_SSL_Cipher_Suite_E;

/* M2M_SSL_AUTH_TYPE_E */
typedef enum _M2M_SSL_AUTH_TYPE_En{
  M2M_SSL_NO_AUTH_TYPE =0,
  M2M_SSL_SERVER_AUTH_TYPE,
  M2M_SSL_SERVER_CLIENT_AUTH_TYPE
} M2M_SSL_AUTH_TYPE_E;
```

### 19.2.13. m2m_timer_api C Identifiers

```
/*Timer handle provided by the m2m_timer_create() function */
typedef void * M2M_T_TIMER_HANDLE;

/*Timer callback function prototype */
typedef void (*M2M_T_TIMER_TIMEOUT)(void *);
```

### 19.2.14. m2m_dump_api C Identifiers

```
#define M2M_MAX_DUMP_BUFFER_LENGTH 256

/*Dump result definition */
typedef enum {
        M2M_MEM_DUMP_RESULT_SUCCESS = 0,
        M2M_MEM_DUMP_ERROR_INVALID_INPUT,
        M2M_MEM_DUMP_ERROR_OUT_OF_MEMORY,
        M2M_MEM_DUMP_ERROR_GENERIC = 15
} M2M_T_DUMP_RESULT;
```

### 19.2.15. m2m_sec_api C Identifiers

```
/* M2M hash output lenght*/
#define MD5_DIGEST_LENGTH        16
#define SHA_DIGEST_LENGTH        20
#define SHA256_DIGEST_LENGTH     32

#define INVALID_DIG_HANDLE       0

/* MD5 handle. */
typedef UINT32 M2M_T_MD5_HANDLE;

/* SHA handle. */
typedef UINT32 M2M_T_SHA_HANDLE;

/* SHA256  handle. */
typedef UINT32 M2M_T_SHA256_HANDLE;

/* DIG_RESULT_T struct to point to result of digest and its size */
typedef struct DIG_RESULT_TAG
{
  UINT8 size;
  UINT8 *result;
}DIG_RESULT_T;
```

### 19.2.16. m2m_ipraw_api C Identifiers

```
#define M2M_IPV4_RAW_TYPE 0      /* IPv4 */
#define M2M_IPV6_RAW_TYPE 1      /* IPv6 */
```

### 19.2.17. m2m_hw_watchdog_api C Identifiers

```
/* User watchdog timeouts */
typedef enum
{
  M2M_WATCHDOG_TIMEOUT_FAST,          /*   1sec */
  M2M_WATCHDOG_TIMEOUT_NORMAL,        /* 11sec */
  M2M_WATCHDOG_TIMEOUT_SLOW,          /* 21sec */
  M2M_WATCHDOG_TIMEOUT_LIMIT          /* timeout values no more valid */
} M2M_WATCHDOG_TIMEOUT;

/* Internal watchdog configuration structure */
typedef struct
{
  UINT32 timeout;                     /* internal timeout expressed in sec */
} M2M_WATCHDOG_OPTIONS;
```

### 19.2.18. m2m_type C Identifiers valid for multiple APIs sets

```
/* API result codes */
typedef enum
{
  M2M_API_RESULT_INVALID_ARG = -1,
  M2M_API_RESULT_FAIL = 0,
  M2M_API_RESULT_SUCCESS = 1

} M2M_API_RESULT;
```

# 20. GLOSSARY AND ACRONYMS

| | Description |
|---|---|
| API | Application Programming Interface |
| APN | Access Point Name |
| GPIO | General Purpose Input/Output |
| I2C | Inter-Integrated Circuit |
| NVM | Non-Volatile Memory |
| PDP | Packet Data Protocol |
| SMS | Short Message Service |
| SPI | Serial Peripheral Interface |
| SSL | Secure Socket Layer |
| UART | Universal Asynchronous Receiver Transmitter |

# 21. DOCUMENT HISTORY

| Revision | Date | Changes |
|---|---|---|
| 0 | 2015-02-16 | First issue |
| 1 | 2015-11-26 | The entire document has been revised, and a new template has been adopted. |
| 2 | 2016-10-31 | New APIs<br>m2m_hw_i2c_cmb_format;<br>m2m_hw_uart_aux_open, m2m_hw_uart_aux_get_state;<br>m2m_dump_init, m2m_dump_save, m2m_dump_clear;<br>m2m_DIGEST_alloc_res, m2m_DIGEST_destroy_res;<br>m2m_MD5_Init, m2m_MD5_Update, m2m_MD5_Final,<br>m2m_SHA_Init, m2m_SHA_Update, m2m_SHA_Final,<br>m2m_SHA256_Init, m2m_SHA256_Update,<br>m2m_SHA256_Final<br>m2m_uart_close_hwch;<br><br>Added:<br>M2M_SOCKET_BSD_TCP_KEEPALIVE option and related example.<br>M2M_SOCKET_BSD_SOCKNOERROR.<br>M2M_SOCKET_STATE_CONNECTED.<br>LE910 V2 SERIES in applicability table.<br>m2m_hw_sleep_mode_cfg()<br>m2m_os_task_get_enqueued_msg()<br><br>Updated:<br>m2m_socket_bsd_addr_str()<br>m2m_socket_bsd_addr_str_ip6()<br>m2m_socket_bsd_get_sock_opt():<br>- M2M_SOCKET_BSD_SO_KEEPALIVE,<br>- M2M_SOCKET_BSD_TCP_NODELAY;<br>m2m_socket_bsd_socket_state()<br>m2m_hw_sleep_mode()<br>m2m_firewall()<br>m2m_firewall_ip6()<br>m2m_network_list_available_networks()<br><br>Changed:<br>m2m_hw_aux_open chapter title in m2m_hw_uart_aux_open.<br>m2m_hw_uart_close function name in m2m_uart_close_hwch.<br><br>Removed:<br>Modules & SW Ver. Tables chapter. |
| 3 | 2017-01-31 | Changed the title of the document and the format of the Applicability Table.<br><br>Changed the note in chapter 5.<br><br>Added notes in chapters 5.4 ÷ 5.7, 12.37, and 12.39.<br><br>Added the following APIs: |

| | | |
|---|---|---|
| | | m2m_ipraw_cfg()<br>m2m_ip4_send()<br>m2m_ip4_recv()<br><br>m2m_pdp_activate_ip6_cid()<br>m2m_pdp_activate_cid()<br>m2m_pdp_deactive_cid()<br>m2m_pdp_get_status_cid()<br>m2m_pdp_get_my_ip6_cid()<br>m2m_pdp_get_my_ip_cid()<br>m2m_pdp_get_datavol_cid()<br>m2m_socket_bsd_socket_cid()<br>m2m_socket_bsd_get_host_by_name_cid()<br>m2m_socket_bsd_get_host_by_name_ip6_cid()<br>m2m_socket_bsd_get_dns_ip6_cid()<br>m2m_socket_bsd_get_dns_ip_cid()<br><br>Added chapter m2m_ipraw_api.h |
| 4 | 2017-12-12 | A new template is used. The title "AppZone C API User Guide" is changed in "AppZone C API Reference Guide".<br><br>Some chapters titles and contents have been revised. Links have been made more visible. Added tables list.<br><br>In the Applicability Table, has been introduced the Platform Version Identifier, and the following modules SERIES: LE910 Cat1, LE910 V2, LE866, ME866A1, LE910D1.<br><br>Added the following APIs:<br>m2m_pdp_apn_set()<br>m2m_pdp_apn_get()<br>m2m_hw_watchdog_conf()<br>m2m_hw_watchdog_enable()<br>m2m_hw_watchdog_disable()<br>m2m_hw_watchdog_refresh()<br><br>m2m_ssl_create_context()<br>m2m_ssl_delete_context()<br>m2m_ssl_securesocket()<br>m2m_ssl_connect()<br><br>Updated the following APIs:<br>m2m_pdp_get_datavol()<br>m2m_ipraw_cfg()<br>m2m_os_mem_pool()<br>m2m_os_retrive_clock()<br>m2m_OTA_write_mem_data()<br>m2m_OTA_read_mem_data<br><br>Updated the following chapters:<br>Related Documents<br>m2m_ssl_api.h (Declaration of C Identifier)<br>m2m_hw_api.h (Declaration of C identifier) |

# SUPPORT INQUIRIES

Link to **www.telit.com** and contact our technical support team for any questions related to technical issues.

## www.telit.com


Telit Communications S.p.A.
Via Stazione di Prosecco, 5/B
I-34010 Sgonico (Trieste), Italy

Telit IoT Platforms LLC
5300 Broken Sound Blvd, Suite 150
Boca Raton, FL 33487, USA

Telit Wireless Solutions Inc.
3131 RDU Center Drive, Suite 135
Morrisville, NC 27560, USA

Telit Wireless Solutions Co., Ltd.
8th Fl., Shinyoung Securities Bld.
6, Gukjegeumyung-ro8-gil, Yeongdeungpo-gu
Seoul, 150-884, Korea

Telit Wireless Solutions Ltd.
10 Habarzel St.
Tel Aviv 69710, Israel

Telit Wireless Solutions
Technologia e Servicos Ltda
Avenida Paulista, 1776, Room 10.C
01310-921 São Paulo, Brazil





[01.2017]

Mod. 0809 2017-01 Rev.8