AppZone APIs User Guide

1vv0301130 Rev. 0 - 2015-02-16

Making machines talk.

APPLICABILITY TABLE

	SW Versions
HE910 Family	
HE910 [1]	**12.00.xx6**
UE910 Family (Embedded)	
UE910-EUR / UE910-EUD	**12.00.xx6**
UE910-NAR / UE910-NAD	**12.00.xx6**
GE910 Family (Embedded) [2]	
GE910-QUAD	**13.00.xx7**
GE910-GNSS	**13.00.xx7**

SERVICES COEXISTENCE TABLE

	Services			
	Embedded GPS	**External GPS**	**Python**	**AppZone**
HE910 Family			**Python and AppZone are mutually exclusive**	
HE910	✓		✓	✓*
UE910 Family (Embedded)				
UE910-EUR / UE910-EUD		✓	✓	✓*
UE910-NAR / UE910-NAD		✓	✓	✓*
GE910 Family (Embedded)				
GE910-QUAD		✓	✓	✓*
GE910-GNSS	✓		✓	✓*

Note: the table summarizes the Services provided by the modules when they are equipped with the suitable software version, and shows the Services coexistence. Embedded/External GPS and Python Services are beyond the scope of this guide.

**(***): AppZone available on demand on specific part numbers.

[1] HE910 is the "type name" of the products marketed as HE910-G & HE910-DG
[2] Currently, the GE910 family does not support some APIs described in this document. To know them refer to the header files containing the function prototypes. In the future software releases also these APIs will be supported.

LEGAL NOTICE

These Specifications are general guidelines pertaining to product selection and application and may not be appropriate for your particular project. Telit (which hereinafter shall include, its agents, licensors and affiliated companies) makes no representation as to the particular products identified in this document and makes no endorsement of any product. Telit disclaims any warranties, expressed or implied, relating to these specifications, including without limitation, warranties or merchantability, fitness for a particular purpose or satisfactory quality. Without limitation, Telit reserves the right to make changes to any products described herein and to remove any product, without notice.

It is possible that this document may contain references to, or information about Telit products, services and programs, that are not available in your region. Such references or information must not be construed to mean that Telit intends to make available such products, services and programs in your area.

USE AND INTELLECTUAL PROPERTY RIGHTS

These Specifications (and the products and services contained herein) are proprietary to Telit and its licensors and constitute the intellectual property of Telit (and its licensors). All title and intellectual property rights in and to the Specifications (and the products and services contained herein) is owned exclusively by Telit and its licensors. Other than as expressly set forth herein, no license or other rights in or to the Specifications and intellectual property rights related thereto are granted to you. Nothing in these Specifications shall, or shall be deemed to, convey license or any other right under Telit's patents, copyright, mask work or other intellectual property rights or the rights of others.

You may not, without the express written permission of Telit: (i) copy, reproduce, create derivative works of, reverse engineer, disassemble, decompile, distribute, merge or modify in any manner these Specifications or the products and components described herein; (ii) separate any component part of the products described herein, or separately use any component part thereof on any equipment, machinery, hardware or system; (iii) remove or destroy any proprietary marking or legends placed upon or contained within the products or their components or these Specifications; (iv) develop methods to enable unauthorized parties to use the products or their components; and (v) attempt to reconstruct or discover any source code, underlying ideas, algorithms, file formats or programming or interoperability interfaces of the products or their components by any means whatsoever. No part of these Specifications or any products or components described herein 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 the prior express written permission of Telit.

HIGH RISK MATERIALS

Components, units, or third-party products contained or used with the products 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, its licensors and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

TRADEMARKS

You may not and may not allow others to use Telit or its third party licensors’ trademarks. To the extent that any portion of the products, components and any accompanying documents contain proprietary and confidential notices or legends, you will not remove such notices or legends.

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.

Contents

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 modem 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] Telit's AppZone User Guide, 1vv0301082

[2] UE910 Hardware User Guide, 1vv0301012

[3] HE910/UE910/UL865 AT Commands Reference Guide, 80378ST10091A

[4] HE910/UE910 Families Ports Arrangements User Guide, 1vv0300971

[5] HE910 Hardware User Guide, 1vv03700925

[6] AT Commands Reference Guide, 80000ST10025a

[7] GE910 Hardware User Guide, 1vv0300962

[8] GE910 Families Ports Arrangements User Guide, 1vv0301049

NOTICE: in the present guide is used the notation [x]/[y] to refer to documents of different families of modules. You have to refer to the document in accordance with the module you are using.

2. m2m_clock_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the following features:

- set/get date/time
- manage alarm
- time since the epoch

2.1. m2m_rtc_set_date

M2M_T_RTC_RESULT m2m_rtc_set_date(M2M_T_RTC_DATE date)

Description: the function 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.

Examples: 17.8 RTC

2.2. m2m_rtc_get_date

M2M_T_RTC_RESULT m2m_rtc_get_date(M2M_T_RTC_DATE *date)

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

Parameters:
date: pointer to the allocated data structure.

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

Output data:
data structure filled with year/month/day

Examples: 17.8 RTC

2.3. m2m_rtc_set_time

M2M_T_RTC_RESULT m2m_rtc_set_time(M2M_T_RTC_TIME time)

Description: the function 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.

Examples: 17.8 RTC

2.4. m2m_rtc_get_time

M2M_T_RTC_RESULT m2m_rtc_get_time(M2M_T_RTC_TIME *time)

Description: the function gets the time.

Parameters:
time: pointer to the allocated data structure.

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

Output data:
data structure filled with hour/minute/second.

Examples: 17.8 RTC

2.5. m2m_rtc_set_alarm

M2M_T_RTC_RESULT m2m_rtc_set_alarm(M2M_T_RTC_DATE date, M2M_T_RTC_TIME time)

Description: the function sets a wake up alarm. On the expiration of the alarm timer, the control calls the M2M_onWakeUp (…) application callback function contained in the M2M_hwEvents.c file, refer to document [1].

NOTICE: the system supports only 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: 17.8 RTC

2.6. m2m_rtc_clear_alarm

M2M_T_RTC_RESULT m2m_rtc_clear_alarm(INT32 index)

Description: the function clears a wake up alarm.

Parameters:
index: 0

NOTICE: 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.

Examples: 17.8 RTC

2.7. m2m_set_timeofday

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

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

NOTICE: the setting will be successful only if the date is subsequent to 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

2.8. m2m_get_timeofday

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

Description: the function 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;
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

Output data:
data structures filled with the current time since the epoch, the time zone, and the Daylight Saving
Time adjustment.

3. m2m_fs_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of M2M File System.

In general, the parameters of the **m2m_fs_xx(…)** functions may include full path using drive and separators, for example: "A:\MOD\m2mapz.bin". The table below shows the drives supported by the modules.

Drives	
A: (default)[3]	B:
FLASH_DISK	RAM_DISK

Valid separators are / and \\.

The **m2m_fs_xx(…)** functions do not take into account the working directory as a start to create the path of the file; they start always from the root. Here are some examples:

```
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 */
```

[3] It is the drive 0 used by the AT#M2MCHDRIVE command, see document [1].

3.1. m2m_fs_create

M2M_API_RESULT m2m_fs_create(CHAR *filename)

Description: the function creates the file using the given filename. In addition, it creates the required directory if missing.

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.

Examples: 17.1 File System

3.2. m2m_fs_open

M2M_T_FS_HANDLE m2m_fs_open(CHAR *filename, INT32 mode)

Description: the function opens the file using the given filename, and returns the file handle. It does not create the required directory if missing.

Parameters:
filename: pointer to the zero-terminated string containing the file name of the file to be opened
mode: open mode:

M2M_FS_OPEN_READ	Open mode for Read only
M2M_FS_OPEN_WRITE	Open mode for Write only. Pointer to the data of the file used to write cannot be changed, always append. The file will be truncated to '0' bytes, if exist.
M2M_FS_OPEN_MODIFY	Open mode for modify only. Pointer to the data of the file used to write can be changed by using the seek API. This mode can not append (file size can not change).
M2M_FS_OPEN_APPEND	Open mode for appending to a file. All write operation will self-seek to the end of file
M2M_FS_OPEN_TRUNCATE	Open mode for truncating a file. Truncates the file to '0' bytes on successful open. Allows writing in append mode only (self seek to the end of file).
M2M_FS_OPEN_CREATE	Open mode for creating a file. Creates the file if it doesn't exist. Truncates to '0' bytes if exists. Allows writing in append mode only (self seek to the end of file).

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

Note:
m2m_fs_last_error function returns the failure reason.

Examples: 17.1 File System

3.3. m2m_fs_close

M2M_API_RESULT m2m_fs_close(M2M_T_FS_HANDLE filehandle)

Description: the function 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: 17.1 File System

3.4. m2m_fs_delete

M2M_API_RESULT m2m_fs_delete(CHAR *filename)

Description: the function 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.

3.5. m2m_fs_clear

M2M_API_RESULT m2m_fs_clear(void)

Description: the function 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.

3.6. m2m_fs_copy

M2M_API_RESULT m2m_fs_copy(CHAR *srcfilename, CHAR *dstfilename)

Description: the function 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.

3.7. m2m_fs_find_first

M2M_API_RESULT m2m_fs_find_first(CHAR *filename_buffer, CHAR *file_spec)

Description: the function finds the first file having the name matching the provided pattern. To find the next file matching the provided 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. The buffer size must be not less than **M2M_FS_MAX_FILENAME_LEN**.

file_spec: pointer to the zero-terminated string containing the pattern you want to find.

Return value:

refer to **M2M_API_RESULT** enum

Output data:

if the file exists, the buffer contains the directory and the file name as shown 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

Note:

<file_spec> may include full path with drive and separators, example: "A:\MOD\m2mapz.bin".

Examples: 17.2 Listing all Files

3.8. m2m_fs_find_next

M2M_API_RESULT m2m_fs_find_next(CHAR *filename_buffer)

Description: the function finds the next file 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. The buffer size must be not less than **M2M_FS_MAX_FILENAME_LEN**.
Return value:
refer to **M2M_API_RESULT** enum

Output data:
if the file exists, the buffer contains the directory and the file name as shown 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

Examples: 17.2 Listing all Files

3.9. m2m_fs_rename

M2M_API_RESULT m2m_fs_rename(CHAR *oldfilename, CHAR *newfilename)

Description: the function 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.

3.10. m2m_fs_get_size

UINT32 m2m_fs_get_size(CHAR *filename)

Description: the function returns the size of the selected file identified by name.

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

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

Note:
m2m_fs_last_error function returns the failure reason.

3.11. m2m_fs_get_size_with_handle

UINT32 m2m_fs_get_size_with_handle(M2M_T_FS_HANDLE filehandle)

Description: the function returns the size of a file identified by the handle.

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.

3.12. m2m_fs_tell

UINT32 m2m_fs_tell(M2M_T_FS_HANDLE filehandle)

Description: the function returns the actual file data pointer.

Parameters:
filehandle: file handle

Return value:
on success: the actual position of the file data pointer
on failure: **M2M_FS_ERROR**

Note:
m2m_fs_last_error function returns the failure reason.

3.13. m2m_fs_seek

UINT32 m2m_fs_seek(M2M_T_FS_HANDLE filehandle, UINT32 offset)

Description: the function moves file data pointer to the given offset.

Parameters:
filehandle: file handle
offset: the offset to be moved, it must be less than the file size

Return value:
on success: the new position of the file data pointer
on failure: **M2M_FS_ERROR**

Note:
m2m_fs_last_error function returns the failure reason.

3.14. m2m_fs_truncate

M2M_T_FS_HANDLE m2m_fs_truncate(CHAR *filename, UINT32 new_size)

Description: the function truncates the file by discarding the last bytes. 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 fHandler = m2m_fs_truncate( filename, 10 );
if ( fHandler )
{
  m2m_fs_close( fHandler );          /* after the closing the file is effectively truncated  */
}
```

3.15. m2m_fs_getc

CHAR m2m_fs_getc(M2M_T_FS_HANDLE filehandle)

Description: the function reads the character from file.

Parameters:
filehandle: file handle

Return value:
on success: the character read
on failure: **EOF**

3.16. m2m_fs_gets

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

Description: the function reads a character string from an already opened file. It reads until finds an **EOF** or new line, as long as the length is not exceeded.

Parameters:
filehandle: file handle
buf: pointer to the allocated buffer that will be filled with the characters read from the file
length: number of characters to be read from the file. The number must be less than the file size

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

Output data:
on success: the allocated buffer filled with the characters string read from the file.

3.17. m2m_fs_read

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

Description: the function reads data from an already opened file in read-only 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 file
length: number of characters to be read from the file. The number must be less than the file size

Return value:
on success: number of bytes read
on failure: **M2M_FS_ERROR**

Output data:
on success: the allocated buffer filled with the characters read from the file.

Note:
m2m_fs_last_error function returns the failure reason.

3.18. m2m_fs_write

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

Description: the function writes data in to a file opened in write-only 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
length: number of characters to be written

Return value:
on success: number of characters written
on failure: **M2M_FS_ERROR**

Note:
m2m_fs_last_error function returns the failure reason.

Examples: 17.1 File System

3.19. m2m_fs_set_exec_permission

M2M_API_RESULT m2m_fs_set_exec_permission(CHAR *filename)

Description: the function has the same feature of the <permission> parameter of the AT#M2MWRITE AT command. By means of this function, 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

3.20. m2m_fs_set_run_permission

M2M_API_RESULT m2m_fs_set_run_permission(M2M_T_FS_RUN_PERM_MODE_TYPE mode,
CHAR *filename)

Description: the function has the same feature of the <mode> parameter of the AT#M2MRUN AT command. By means of this function, 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

3.21. m2m_fs_mk_dir

M2M_API_RESULT m2m_fs_mk_dir(CHAR *path)

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

Parameters:
path: pointer to the zero-terminated string containing the full path including the directory name to be created. The maximum length of path plus directory name is 124 characters.

Return value:
refer to **M2M_API_RESULT** enum

Note:
m2m_fs_last_error function returns the failure reason.

3.22. m2m_fs_rename_dir

M2M_API_RESULT m2m_fs_rename_dir(CHAR *oldpath, CHAR *newdirname)

Description: the function 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.

3.23. m2m_fs_rm_dir

M2M_API_RESULT m2m_fs_rm_dir(CHAR *path)

Description: the function 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.

3.24. m2m_fs_get_free_space

UINT32 m2m_fs_get_free_space(void)

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

3.25. m2m_fs_get_nof_files

INT32 m2m_fs_get_nof_files(void)

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

3.26. m2m_fs_last_error

M2M_T_FS_ERROR_TYPE m2m_fs_last_error(void)

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

Return value:
refer to **M2M_T_FS_ERROR_TYPE** enum

4. m2m_hw_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of the following devices:

- GPIO
- I2C
- HW timers
- UART
- USB

In accordance with the module that you are using, refer to document [2]/[5]/[7] to get hardware information, as for example GPIO port numbers.

4.1. m2m_hw_gpio_read

INT32 m2m_hw_gpio_read (INT32 io)

Description: the function reads the status of the selected GPIO port. Before using this API, it is mandatory to call the **m2m_hw_gpio_conf** function.

Parameters:
io: GPIO port number

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

Examples: 17.3 GPIO

4.2. m2m_hw_gpio_write

INT32 m2m_hw_gpio_write (INT32 io, INT32 val)

Description: the function sets automatically the selected GPIO port in output, and then the 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: 17.3 GPIO

4.3. m2m_hw_gpio_conf

INT32 m2m_hw_gpio_conf (INT32 io, INT32 dir)

Description: the function 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

4.4. m2m_hw_gpio_int_enable

void m2m_hw_gpio_int_enable(INT32 io)

Description: the function 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].

NOTICE: for HE910/UE910 families, the GPIO port number 5 and 10 cannot generate interrupts.

Parameters:
io: GPIO port number

4.5. m2m_hw_gpio_int_disable

void m2m_hw_gpio_int_disable(INT32 io)

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

Parameters:
io: GPIO port number

4.6. m2m_hw_gpio_int_enable_on_front

void m2m_hw_gpio_int_enable_on_front(INT32 io, M2M_INT_FRONT front);

Description: the function specifies which edge of the selected GPIO triggers the interrupt.

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

4.7. m2m_hw_timer_start

M2M_API_RESULT m2m_hw_timer_start(INT32 timer_id, UINT32 span)

Description: the function 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.

NOTICE: 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(timer_id).

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

Examples: 17.7 HW Timer

4.8. m2m_hw_timer_stop

void m2m_hw_timer_stop(INT32 timer_id)

Description: the function stops the selected HW timer.

Parameters:
timer_id: 1÷7

Examples: 17.7 HW Timer

4.9. m2m_hw_timer_state

UINT8 m2m_hw_timer_state(void)

Description: the function 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

4.10. m2m_hw_set_ms_count

M2M_API_RESULT m2m_hw_set_ms_count(INT32 on_off)

Description: the function starts/stops 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

4.11. m2m_hw_get_ms_count

M2M_API_RESULT m2m_hw_get_ms_count(UNT32 *m_secs)

Description: the function returns the current counter value.

Parameters:
m_secs: pointer to the allocated variable that will be filled with the current counter value.

Return value:
refer to **M2M_API_RESULT** enum

Output data:
on success: "m_secs" pointer points to the current counter value.

4.12. m2m_hw_uart_open

M2M_T_HW_UART_HANDLE m2m_hw_uart_open(void)

Description: the function opens the UART port (USIF0, see documents [2]/[5]/[7]).

NOTICE: the function returns always the same handle, any time it is called.

Return value:
on success: UART port handle
on failure: **M2M_HW_UART_HANDLE_INVALID**

4.13. m2m_hw_uart_close

M2M_T_HW_UART_RESULT m2m_hw_uart_close(M2M_T_HW_UART_HANDLE handle)

Description: the function closes the open UART port.

Parameters:
handle: UART port handle

Return value:
refer to **M2M_T_HW_UART_RESULT** enum

4.14. 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: the function receives data from UART. The receiving modes are set by the function **m2m_hw_uart_ioctl**.

Parameters:
handle: UART port handle
buffer: pointer to the allocated buffer that will be filled with received 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).

Return value:
refer to **M2M_T_HW_UART_RESULT** enum

Output data:
on success:
<buffer> points to the buffer filled with the read data.
<len_read> points to the number of characters in the buffer.

4.15. 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: the function sends data through UART. The sending modes are set by the function **m2m_hw_uart_ioctl**.

Parameters:
handle: UART 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).

Return value:
refer to **M2M_T_HW_UART_RESULT** enum

Output data:
on success: number of bytes that was sent

4.16. m2m_hw_uart_ioctl

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

Description: the function configures the UART port features using "arg" parameter (selector) and the "value" parameter (option).

Parameters:
handle: UART port handle
arg: selector of the UART port feature, refer to the tables below
value: option related to the selected UART feature, refer to the tables below

arg: sets the UART feature	**value**: sets the option related to the selected UART feature
M2M_HW_UART_IO_BLOCKING_SET valid for TX and RX	**M2M_HW_UART_IO_BLOCKING_ON**: sets blocking to ON, default mode. 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". **NOTICE**: 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). 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". Bytes exceeding "len" remain in the UART receiver buffer. You can delete them using the following function: **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 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 actually moved into UART transmitter buffer 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 activites 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 UART feature	**value**: sets the option related to the selected UART feature
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 UART feature	**value**: sets the option related to the selected UART feature
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 UART feature	**value**: sets the option related to the selected UART feature
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]/[8].
	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]/[8]. 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 UART feature	**value**: sets the option related to the selected UART feature
M2M_HW_UART_CLEAR_RX 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 UART feature	**value**: sets the option related to the selected UART feature
M2M_HW_UART_IO_RCV_FUNC 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 UART feature	**value**: sets the option related to the selected UART feature
M2M_HW_UART_IO_HW_OPTION_GET Get the UART setting	Get the UART setting: baudrate, databits, stop_bits, and parity. See **M2M_T_HW_UART_IO_HW_OPTIONS** structure
M2M_HW_UART_IO_HW_OPTION_SET Set the UART setting	Set the UART: baudrate, databits, stop_bits, and parity. 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

4.17. m2m_hw_uart_get_state

void m2m_hw_uart_get_state(USB_UART_STATE *ser_state)

Description: the function gets the current UART setting.

Parameters:
ser_state: pointer to the allocated buffer that will be filled with the current UART setting

Output data: ser_state points to the buffer filled with the current UART setting

4.18. m2m_hw_usb_open

M2M_API_RESULT m2m_hw_usb_open (M2M_USB_CH channel,
M2M_T_HW_USB_HANDLE *handle)

Description: the function opens the selected USB channel and returns the related handle. 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.

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

Return value:
refer to **M2M_API_RESULT** enum

Output data:
on success: handle of the specified USB channel
on failure: refer to **error codes for USB handle**

NOTICE: 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]/[8].

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

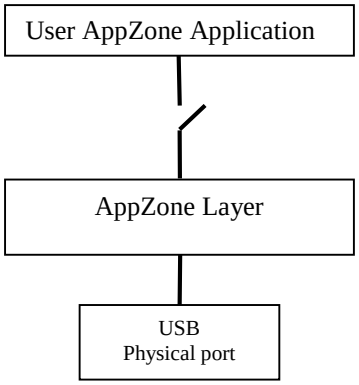
4.19. m2m_hw_usb_close

M2M_API_RESULT m2m_hw_usb_close (M2M_T_HW_USB_HANDLE handle)

Description: the function closes the USB channel identified by the handle. The function opens logically the switch shown in the figure below. The previous USB channel configuration (before AppZone Application execution, see document [4]/[8]) is not restored, AppZone Layer is still the owner of the resource even if the channel is closed, see also **m2m_usb_close_hwch**.

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

Return value:
refer to **M2M_API_RESULT** enum



4.20. 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: the function receives data from USB channel. The receiving modes are set by the function **m2m_hw_usb_ioctl**.

Parameters:
handle: USB channel handle
buffer: pointer to the allocated buffer that will be filled with received 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)

Return value:
refer to **M2M_API_RESULT** enum

Output data:
on success: "buffer" points to the buffer filled with the read data.
"len_read" points to the number of characters in the buffer.
on failure: len_sent = 0, buffer content unchanged

4.21. 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: the function sends data through USB channel. The sending modes are set by the function **m2m_hw_usb_ioctl**

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).

Return value:
refer to **M2M_API_RESULT** enum

Output data:
on success: number of bytes that was sent, see "len_sent" parameter
on failure: len_sent = 0, buffer content unchanged

4.22. 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: the function configures the USB channel feature using "arg" parameter (selector) and the "value" parameter (option).

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

arg: sets the USB feature	**value**: sets the option related to the selected USB feature
M2M_USB_BLOCKING_SET valid for TX and RX	**M2M_HW_USB_IO_BLOCKING_ON**: sets blocking to ON, default mode. 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". 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". Bytes exceeding "len" remain in the USB receiver buffer. You can delete them using the following function: **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 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 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 UART feature	**value**: sets the option related to the selected UART feature
M2M_USB_RX_BLOCKING_SET 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 UART feature	**value**: sets the option related to the selected UART feature
M2M_USB_TX_BLOCKING_SET 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 feature	**value**: sets the option related to the selected USB feature
M2M_USB_AT_MODE_SET 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]/[8].
	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]/[8]. 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 feature	**value**: sets the option related to the selected USB feature
M2M_USB_CLEAR_RX 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 feature	**value**: sets the option related to the selected USB feature
M2M_USB_RCV_FUNC Indicates that the **value** parameter is a callback to manage received data	static INT32 **hw_usb_read_cb** (M2M_T_HW_USB_HANDLE handle, CHAR *buffer, INT32 len)

Refer to the **M2M_HW_UART_IO_RCV_FUNC** example.

arg: sets the USB feature	**value**: sets the option related to the selected USB feature
M2M_USB_NO_ACTION For internal use only	/

Return value:

refer to **M2M_API_RESULT** enum

4.23. m2m_hw_usb_get_state

void m2m_hw_usb_get_state(M2M_USB_CH ch, USB_UART_STATE *usb_state)

Description: the function 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

4.24. 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: the function gets the USB channel.

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.

Return value:
refer to **M2M_API_RESULT**enum

Output data:
on success: USB channel, see **M2M_USB_CH** enum
on failure: **USB_CH**_NONE

4.25. m2m_hw_usb_get_instance

USER_USB_INSTANCE_T m2m_hw_usb_get_instance (M2M_USB_CH channel)

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

Parameters:
channel: USB channel

Return value:
refer to **USER_USB_INSTANCE_T** enum

NOTICE: referring **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**.

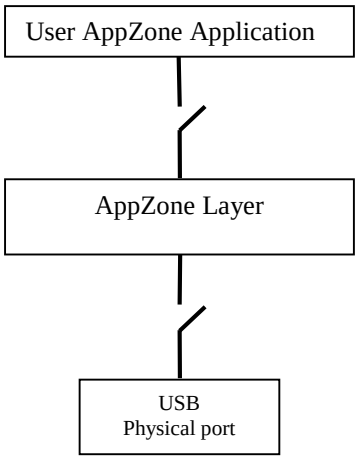
4.26. m2m_usb_close_hwch

M2M_API_RESULT m2m_usb_close_hwch (M2M_T_HW_USB_HANDLE *handle)

Description: the function closes the USB channel identified by the handle. The function opens logically the two switches shown in the figure below. The previous USB channel configuration (before AppZone Application execution, see document [4]/[8]) is restored, see also **m2m_hw_usb_close(...)** function.

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

Return value:
refer to **M2M_API_RESULT** enum



4.27. m2m_hw_sleep_mode

void m2m_hw_sleep_mode(UINT8 enter)

Description: the function carries out the same activities of the AT+CFUN=0/1 AT command, refer to documents [3]/[6]. It forces the module in/out power saving mode.

Parameters:
enter: 1 to enter power saving, 0 to exit, See the table below.

"enter" parameter	AT+CFUN	Module Mode
1	0	Power saving
0	1	Full features

Examples: 17.8 RTC

4.28. m2m_hw_power_down

void m2m_hw_power_down(void)

Description: the function forces 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 M2M application will start execution.

4.29. m2m_OTA_write_mem_data

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

Description: the function works on OTA section memory; it moves "len" bytes from the buffer pointed by "buffer" pointer 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:

Module Family	Offset Range
HE910	0 ÷ 0x280000 - 1
UE910	0 ÷ 0x280000 - 1
GE910	0 ÷ 0x140000 - 1

Return value:
on success: 1
on failure: 0

4.30. m2m_OTA_read_mem_data

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

Description: the function 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
len: number of bytes to read
offset: offset in the memory where data is read

Module Family	Offset Range
HE910	0 ÷ 0x280000 - 1
UE910	0 ÷ 0x280000 - 1
GE910	0 ÷ 0x140000 - 1

Return value:
on success: 1
on failure: 0

Output data:
on success: the allocated buffer is filled with data read.

4.31. m2m_OTA_erase_mem_data

INT32 m2m_OTA_erase_mem_data(void);

Description: the function 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

5. m2m_spi_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the following device:

- SPI port

5.1. m2m_spi_init

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

Description: the function initializes the SPI port, which uses the TX and RX lines provided by the USIFx 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. Refer to document [2]/[5]/[7] to have information on USIFx serial port.

Parameters:

usif_num: the SPI interface can be mapped onto the USIFx ports shown in the table below.

usif_num	USIFx	
1	0	Modem Serial Port 1 (AT Commands)
2	not used	/
3	1	Modem Serial Port 2 (Trace 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 speed.

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

5.2. m2m_spi_write

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

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

Parameters:
usif_num: select the already initialized SPI interface:

usif_num	USIFx
1	0
2	not used
3	1

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

Output data:
on success: the allocated buffer pointed by "bufferReceive" is filled with data read.

Return value:
refer to **M2M_T_SPI_RESULT**enum

5.3. m2m_spi_close

M2M_T_SPI_RESULT m2m_spi_close(INT16 usif_num);

Description: the function closes SPI port.

Parameters:
usif_num: select the SPI interface to close:

usif_num	USIFx
1	0
2	not used
3	1

Return value:
refer to **M2M_T_SPI_RESULT**enum

6. m2m_i2C_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the following device:

- i2C bus

6.1. m2m_hw_i2c_conf

M2M_T_HW_I2C_RESULT m2m_hw_i2c_conf(UINT8 i2c_sda, UINT8 i2c_scl)

Description: the function configures an I2C port; refer to documents [2]/[5]/[7] and [3]/[6] to get GPIO Pin information.

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=?" described in document [3]/[6].

Return value:
refer to **M2M_T_HW_I2C_RESULT** enum

6.2. m2m_hw_i2c_read

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

Description: the function reads data from an I2C device.

Parameters:
address: address of the I2C device, with the LSB used as read\write command. It does not matter if the
LSB is set to 0 or to 1. 10 bits address is supported
reg_addr: register address 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)
len: number of bytes to read, see **M2M_HW_I2C_MAX_BUF_LEN**.

Return value:
refer to **M2M_T_HW_I2C_RESULT** enum

Output data:
on success: "buffer" points to the buffer filled with the data read from the I2C device.

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 in accordance with the command.

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

6.3. m2m_hw_i2c_write

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

Description: the function writes data on an I2C device.

Parameters:
address: address of the I2C device, with the LSB used as read\write command. It does not matter if the
LSB is set to 0 or to 1. 10 bits address is supported
reg_addr: register address where the first byte is written
buffer: pointer to the buffer containing the data to write on I2C device (in hexadecimal format)
len: length of the buffer in bytes, see **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 in accordance with 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. m2m_network_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of some network information.

7.1. m2m_network_enable_registration_location_unsolicited

INT32 m2m_network_enable_registration_location_unsolicited(void)

Description: the function enables the notification of the location registration indication to the user M2M application. It means that every location registration update is routed to M2M_onRegStatusEvent(…) application callback function contained in the M2M_net.c file; refer to document [1].

Return value:

on success: 1
on failure: 0

7.2. m2m_network_disable_registration_location_unsolicited

INT32 m2m_network_disable_registration_location_unsolicited(void)

Description: the function disables the notification of the location registration indication to the user M2M application.

Return value:

on success: 1
on failure: 0

7.3. m2m_network_get_cell_information

INT32 m2m_network_get_cell_information(M2M_T_NETWORK_CELL_INFORMATION *cell_info)

Description: the function 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.

7.4. m2m_network_get_currently_selected_operator

INT32 m2m_network_get_currently_selected_operator (M2M_T_NETWORK_CURRENT_NETWORK *selected_op)

Description: the function gets the selected network operator. It returns the same information of the AT command "AT+COPS?" described in [3]/[6].

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.

7.5. m2m_network_get_reg_status

INT32 m2m_network_get_reg_status(M2M_T_NETWORK_REG_STATUS_INFO *reg_status_info)

Description: the function gets the registration status information.

Parameters:
reg_status_info: pointer to the allocated structure that will be filled with the registration status information, it must not be **NULL**

Return value:
on success: 1
on failure: 0

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

7.6. m2m_network_list_available_networks

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

Description: the function returns the list of available networks. The list contains the same information returned by the AT command "AT+COPS=?" described in document [3]/[6].

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

7.7. m2m_network_get_signal_strength

INT32 m2m_network_get_signal_strength(INT32 *rssi, INT32 *ber)

Description: the function returns the network signal strength, and bit error rate. The same information is returned by the AT command "AT+CSQ" described in document [3]/[6].

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. m2m_os_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of the Operating System features.

8.1. m2m_info_get_model

void m2m_info_get_model(CHAR *buf)

Description: the function returns the module model.

Parameters:
buf: pointer to the allocated buffer that will be filled with the module model. The buffer must have at least 128 bytes length.

Output data:
buffer filled with the module model string.

8.2. m2m_info_get_manufacturer

void m2m_info_get_manufacturer(CHAR *buf)

Description: the function returns the module manufacturer.

Parameters:
buf: pointer to the allocated buffer that will be filled with the module manufacturer. The buffer must have
at least 128 bytes length.

Output data:
buffer filled with the module manufacturer string.

8.3. m2m_info_get_factory_SN

void m2m_info_get_factory_SN(CHAR *buf)

Description: the function returns the module factory SN.

Parameters:
buf: pointer to the allocated buffer that will be filled with the factory SN. The buffer must have at least 128
bytes length.

Output data:
buffer filled with factory SN string.

8.4. m2m_info_get_serial_num

void m2m_info_get_serial_num(CHAR *buf)

Description: the function returns the module serial number.

Parameters:
buf: pointer to the allocated buffer that will be filled with the module serial number. The buffer must have
at least 128 bytes length.

Output data:
buffer filled with the module serial number string.

8.5. m2m_info_get_sw_version

void m2m_info_get_sw_version(CHAR *buf)

Description: the function 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. The buffer
must have at least 16 bytes length.

Output data:
buffer filled with the string of the Telit AppZone software version.

8.6. m2m_info_get_fw_version

void m2m_info_get_fw_version(CHAR *buf)

Description: the function returns the module software version. It is the same information returned by the AT command "AT+CGMR" described in document [3]/[6].

Parameters:
buf: pointer to the allocated buffer that will be filled with the module software version. The buffer must
have at least 128 bytes length.

Output data:
buffer filled with the string of the module software version.

8.7. m2m_info_get_MSISDN

void m2m_info_get_MSISDN(CHAR *buf)

Description: the function returns the module MSISDN.

Parameters:
buf: pointer to the allocated buffer that will be filled with the module MSISDN. The buffer must have at
least 1500 bytes length.

Output data:
buffer filled with the module MSISDN string.

8.8. m2m_info_get_IMSI

void m2m_info_get_IMSI(CHAR *buf)

Description: the function returns the module IMSI.

Parameters:
buf: pointer to the allocated buffer that will be filled with IMSI. The buffer must have at least 128 bytes
length.

Output data:
buffer filled with the module IMSI string.

8.9. m2m_os_set_version

INT32 m2m_os_set_version(CHAR *sw_version)

Description: the function 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

8.10. m2m_os_get_version

CHAR *m2m_os_get_version(void)

Description: the function 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.

8.11. m2m_os_get_current_task_id

UINT8 m2m_os_get_current_task_id(void)

Description: the function 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

8.12. 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: the function 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

NOTICE: 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: - M2M_msgProc1() - 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. In this configuration, each task calls a different M2M_msgProcX(…) callback. 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		

8.13. m2m_os_destroy_task

INT32 m2m_os_destroy_task(INT8 procId)

Description: the function 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

8.14. m2m_os_send_message_to_task

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

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

Parameters:
procId: task number at which is addressed the message. Task numbers (or 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

8.15. m2m_os_set_argc

INT8 m2m_os_set_argc(INT8 argc)

Description: the function 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

8.16. m2m_os_get_argc

INT8 m2m_os_get_argc(void)

Description: the function 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: /

8.17. m2m_os_set_argv

INT8 m2m_os_set_argv(UINT8 index, CHAR*arg)

Description: the function 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

8.18. m2m_os_get_argv

CHAR*m2m_os_get_argv(UINT8 index)

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

Parameters:
index: 0 ÷ 3, identifies 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**

8.19. m2m_os_iat_set_at_command_instance

INT32 m2m_os_iat_set_at_command_instance(UINT16 logPort, UINT16 atInstance);

Description: the function 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]/[8] 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

8.20. m2m_os_iat_send_at_command

INT32 m2m_os_iat_send_at_command(CHAR *atCmd, UINT16 logPort)

Description: the function 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 via; **m2m_os_iat_set_at_command_instance(…)** function.

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

8.21. m2m_os_iat_send_atdata_command

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

Description: the function 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 via **m2m_os_iat_set_at_command_instance(…)** function.

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

8.22. m2m_os_mem_pool

INT32 m2m_os_mem_pool(UINT32 pool_size)

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

Parameters:
pool_size: requested HEAP size expressed in bytes, max 2 Mbytes. If this function is not used, the pool memory space provided by the system is 8 Kbytes (default value).

NOTICE: 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

8.23. m2m_os_mem_alloc

void *m2m_os_mem_alloc(UINT32 size)

Description: the function 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.

8.24. m2m_os_mem_realloc

void *m2m_os_mem_realloc(void *ptr, UINT32 size)

Description: the function 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.

8.25. m2m_os_mem_free

void m2m_os_mem_free(void *mem)

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

Parameters:
mem: pointer to the memory to free.

8.26. m2m_os_get_mem_info

UINT32 m2m_os_get_mem_info(UINT32 *pool_frags)

Description: the function 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.

8.27. m2m_os_retrieve_clock

INT32 m2m_os_retrieve_clock(void);

Description: the function returns the system tick.

Return value:
system tick, 1 tick = 100 ms

8.28. m2m_os_sleep_ms

void m2m_os_sleep_ms(UINT32 ms)

Description: the function 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.

8.29. m2m_os_sys_reset

void m2m_os_sys_reset(INT32 id)

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

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

8.30. m2m_os_trace_out

void m2m_os_trace_out(CHAR *msg)

Description: the function sends user messages on the USIF1 serial port; refer to document [2]/[5]/[7]. The USIF1 port can be used to send out trace and user messages. The port can support both type of messages if it is suitably configured, the factory setting supports only user messages, see the table below.

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

Trace disabled	Trace enable	m2m_os_trace_out(…) working	Messages on terminal connected to USIF1
✓	/	✓	Only user messages are displayed on the terminal. They are issued in ASCII format; a hyper-terminal can display the user messages. 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.
/	✓	/	The trace messages are coded. A suitable tool is needed to display the trace messages on the terminal.

9. m2m_os_lock_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of the semaphores.

9.1. m2m_os_lock_init

M2M_T_OS_LOCK m2m_os_lock_init(INT32 val)

Description: the function returns a semaphore handle.

Parameters:
val: semaphore's counter value initialization. Use **M2M_OS_LOCK_CS** to initialize semaphore counter
to one or **M2M_OS_LOCK_IPC** to zero; the enum values are the counter values initialization in order to use the semaphore as a binary semaphore. Using these initializations, it does not mean that the semaphore is a binary semaphore. In any case, the semaphore is a counting semaphore; therefore, you can use a counter value initialization greater than one.

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

Examples: 17.4, 17.5

9.2. m2m_os_lock_lock

M2M_API_RESULT m2m_os_lock_lock(M2M_T_OS_LOCK lock)

Description: the function decreases by one the count of the semaphore identified by its handle. If the retrieved semaphore count – before the decreasing – is zero, the control is not returned to the calling task until another task calls the **m2m_os_lock_unlock(lock)** to unlock the semaphore.

Parameters:
lock: semaphore handle.

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

9.3. m2m_os_lock_wait

M2M_API_RESULT m2m_os_lock_wait(M2M_T_OS_LOCK lock, UINT32 timeout)

Description: the function decreases by one the count of the semaphore identified by its handle. If the retrieved semaphore count – before the decreasing – is zero, the control is not returned to the calling task until one of the two events happens:

- another task calls the **m2m_os_lock_unlock(lock)** to unlock the semaphore or
- the timeout is expired.

Parameters:
lock: semaphore handle;
timeout: is 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.

NOTICE: if the return value is:

- **M2M_API_RESULT_SUCCESS**: the calling task gets the control because another task unlocked the semaphore;
- **M2M_API_RESULT_FAIL**: the calling task gets the control because the timeout is expired.

9.4. m2m_os_lock_unlock

M2M_API_RESULT m2m_os_lock_unlock(M2M_T_OS_LOCK lock)

Description: the function increases by one the count of the semaphore identified by its handle.

Parameters:
lock: semaphore handle.

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

9.5. m2m_os_lock_destroy

M2M_API_RESULT m2m_os_lock_destroy(M2M_T_OS_LOCK lock)

Description: the function destroys a semaphore and releases all resources allocated for this semaphore.

Parameters:
lock: semaphore handle.

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

10. m2m_sms_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of SMS messages.

10.1. m2m_sms_enable_new_message_indication

INT32 m2m_sms_enable_new_message_indication(void)

Description: the function enables new message indication, see AT command "AT+CNMI" with <mt>=1, refer to document [3]/[6]. 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

10.2. m2m_sms_disable_new_message_indication

INT32 m2m_sms_disable_new_message_indication(void)

Description: the function disables a new message indication, see AT command "AT+CNMI" with <mt>=0, refer to document [3]/[6].

Return value:
on success: 1
on failure: 0

10.3. m2m_sms_get_all_messages

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

Description: the function 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

10.4. m2m_sms_get_text_message

INT32 m2m_sms_get_text_message(INT32 index, M2M_T_SMS_INFO *sms_info)

Description: the function 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.

10.5. m2m_sms_delete_message

INT32 m2m_sms_delete_message(INT32 index)

Description: the function deletes the selected SMS message.

Parameters:
index: identifies the message to be deleted.

Return value:
on success: 1
on failure: 0

10.6. m2m_sms_send_SMS

INT32 m2m_sms_send_SMS(CHAR *address, CHAR *message)

Description: the function 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

10.7. m2m_sms_set_PDU_mode_format

INT32 m2m_sms_set_PDU_mode_format(void)

Description: the function 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

10.8. m2m_sms_set_text_mode_format

INT32 m2m_sms_set_text_mode_format(void)

Description: the function 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

10.9. m2m_sms_set_preferred_message_storage

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

Description: the function selects the following memory storages as done by the AT command "AT+CPMS" described in the document [3]/[6]:

- <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]/[6]. 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.
GE910 modules do not support "ME" memory, see document [6].

Return value:

on success: 1
on failure: 0

Examples: 17.9 SMS Storage

10.10. m2m_sms_get_preferred_message_storage

INT32 m2m_sms_get_preferred_message_storage(M2M_T_SMS_MEM_STORAGE mem_storages[])

Description: the function gets the message storage status as done by the AT command "AT+CPMS?" described in the document [3]/[6]. <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: 17.9 SMS Storage

11. m2m_socket_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of sockets.

11.1. m2m_socket_bsd_socket

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

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

Parameters:
domain: **Socket_Address_Families**, for example: M2M_SOCKET_BSD_AF_INET, and so on;
type: **Socket_Types**, for example: M2M_SOCKET_BSD_SOCK_STREAM, and so on;
protocol: **Socket_Protocols**, for example: M2M_SOCKET_BSD_IPPROTO_IP, and so on.

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

Note:
m2m_socket_errno(…) function returns the failure reason.

Examples: 17.11 TCP-Client, 17.12 TCP-Server, 17.13 UDP-Client and 17.14 UDP-Server

11.2. m2m_socket_bsd_close

INT32 m2m_socket_bsd_close(M2M_SOCKET_BSD_SOCKET s)

Description: the function 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: 17.11 TCP-Client, 17.12 TCP-Server, 17.13 UDP-Client and 17.14 UDP-Server

11.3. m2m_socket_bsd_socket_state

INT32 m2m_socket_bsd_socket_state(M2M_SOCKET_BSD_SOCKET s)

Description: the function returns the state (open or closed) of the specified socket.

Parameters:
s: socket handle

Return value:
refer to **Socket_State**

Note:
m2m_socket_errno(…) function returns the failure reason.

11.4. 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: the function sets a particular socket option for the specified socket.

Parameters:

s:	socket handle;
level:	**Socket_Protocols**, for example: M2M_SOCKET_BSD_IPPROTO_IP, and so on; or **Level_number**: M2M_SOCKET_BSD_SOL_SOCKET;
optname:	**Socket_Option_Flags**, for example: M2M_SOCKET_BSD_SO_DEBUG, and so on;
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.

11.5. 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: the function returns the current value of a particular socket option for the specified socket.

Parameters:

s:	socket handle;
level:	**Socket_Protocols**, for example: M2M_SOCKET_BSD_IPPROTO_IP, and so on; or **Level_number**: M2M_SOCKET_BSD_SOL_SOCKET;
optname:	**Socket_Option_Flags**, for example: M2M_SOCKET_BSD_SO_DEBUG, and so on;
optval:	pointer to the allocated buffer that will be filled with the Socket Option value;
optlen:	pointer to the allocated variable that will be filled with the length of buffer pointed by "optval".

Return value:

on success: 0
on failure: < 0

Output data:

on success:

"optval" points to the buffer filled with Socket Option value of the specified socket;
"optlen" points to the variable filled with the size of the buffer.

Note:

m2m_socket_errno(…) function returns the failure reason.

11.6. m2m_socket_bsd_shutdown

INT32 m2m_socket_bsd_shutdown(M2M_SOCKET_BSD_SOCKET s, INT32 how)

Description: the function is a dummy function, no actions.

Parameters:

s:	not used;
how:	not used.

Return value: 0

11.7. 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: the function 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;

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.

Return value:

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

Output data:

on success:
"addr" points to the structure filled with address/port/protocol accepted;
"addrlen" points to the variable filled with the size of address/port/protocol accepted.

Note:

m2m_socket_errno(…) function returns the failure reason.

Example: 17.12 TCP-Server

11.8. m2m_socket_bsd_addr_str

CHAR *m2m_socket_bsd_addr_str(UINT32 ipAddr)

Description: the function converts the value of the ip address into a string.

Parameters:
ipAddr: ip address to be converted into a string.

Return value:
on success: pointer to the zero-terminated string containing the representation of ipAddr;
on failure: **NULL**.

Note:
m2m_socket_errno(…) function returns the failure reason.

11.9. m2m_socket_bsd_addr_str_ip6

CHAR *m2m_socket_bsd_addr_str_ip6(M2M_SOCKET_BSD_IPV6_ADDR *ipAddr)

Description: the function converts the value of the ip6 address into a string.

Parameters:
ipAddr: ip6 address to be converted into a string.

Return value:
on success: pointer to the zero-terminated string containing the representation of ipAddr;
on failure: **NULL**.

Note:
m2m_socket_errno(…) function returns the failure reason.

11.10. m2m_socket_bsd_inet_addr

UINT32 m2m_socket_bsd_inet_addr(const CHAR *ip_addr_str)

Description: the function converts the ip_addr_str string value into a UINT32 number.

Parameters:
ip_addr_str: pointer to a zero-terminated string containing the address string to be converted into UINT32.

Return value:
on success: address string converted into UNIT32;
on failure: 0.

Note:
m2m_socket_errno(…) function returns the failure reason.

Examples: 17.11 TCP-Client, 17.13 UDP-Client

11.11. 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: the function converts the ip_addr_str string value into a **M2M_SOCKET_BSD_ IPV6_ADDR** structure.

Parameters:
ip_addr_str: pointer to a zero-terminated string containing the 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 ip6 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 ip6 address.

Note:
m2m_socket_errno(…) function returns the failure reason.

11.12. m2m_socket_bsd_connect

INT32 m2m_socket_bsd_connect(M2M_SOCKET_BSD_SOCKET s, const M2M_SOCKET_BSD_SOCKADDR *name, INT32 namelen)

Description: the function 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:

m2m_socket_errno(…) function returns the failure reason.

Examples: 17.11 TCP-Client

11.13. m2m_socket_bsd_bind

INT32 m2m_socket_bsd_bind(M2M_SOCKET_BSD_SOCKET s,
M2M_SOCKET_BSD_SOCKADDR *name, INT32 namelen)

Description: the function 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: 17.12 TCP-Server, 17.14 UDP-Server

11.14. 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: the function 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, otherwise it 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. Range: seconds and microseconds; 0 for no timeout.

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.

11.15. m2m_socket_bsd_fd_set_func

void m2m_socket_bsd_fd_set_func(INT32 fd, M2M_SOCKET_BSD_FD_SET *set)

Description: the function 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.

11.16. m2m_socket_bsd_fd_clr_func

void m2m_socket_bsd_fd_clr_func(INT32 fd, M2M_SOCKET_BSD_FD_SET *set)

Description: the function 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.

11.17. m2m_socket_bsd_fd_isset_func

UINT8 m2m_socket_bsd_fd_isset_func(INT32 fd, M2M_SOCKET_BSD_FD_SET *set)

Description: the function 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.

11.18. m2m_socket_bsd_fd_zero_func

void m2m_socket_bsd_fd_zero_func(M2M_SOCKET_BSD_FD_SET *set)

Description: the function 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.

11.19. m2m_socket_bsd_get_host_by_name

UINT32 m2m_socket_bsd_get_host_by_name(const CHAR *domain_name)

Description: the function converts the domain name into host entry information.

Parameters:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into host entry information.

Return value:
on success: IP address in ipv4 format (32 bits inet address);
on failure: 0

Note:
m2m_socket_errno(…) function returns the failure reason.

11.20. 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: the function converts the domain name into host ip6 address.

Parameters:
domain_name: pointer to the zero-terminated string containing the domain name to be converted into host entry information;
ipAddr: pointer to the allocated **M2M_SOCKET_BSD_ IPV6_ADDR** structure that will be filled with host ip6 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 host ip6 address.

Note:
m2m_socket_errno(…) function returns the failure reason.

11.21. 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: the function 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.

11.22. 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: the function 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: 17.11 TCP-Client

11.23. m2m_socket_bsd_htonl

UINT32 m2m_socket_bsd_htonl(UINT32 x)

Description: the function converts the unsigned integer x from host byte order to network byte order.

11.24. m2m_socket_bsd_htons

UINT16 m2m_socket_bsd_htons(UINT16 x)

Description: the function converts the unsigned short integer x from host byte order to network byte order.

Examples: 17.11 TCP-Client, 17.12 TCP-Server, 17.13 UDP-Client and 17.14 UDP-Server

11.25. m2m_socket_bsd_ntohl

UINT32 m2m_socket_bsd_ntohl(UINT32 x)

Description: the function converts the unsigned integer x from network byte order to host byte order.

11.26. m2m_socket_bsd_ntohs

UINT16 m2m_socket_bsd_ntohs(UINT16 x)

Description: the function converts the unsigned short integer x from network byte order to host byte order.

11.27. m2m_socket_bsd_ioctl

INT32 m2m_socket_bsd_ioctl(M2M_SOCKET_BSD_SOCKET s, INT32 cmd, void *argp)

Description: the function is an IO control function. It can be used to:

1. set the selected socket to blocking or non-blocking mode.
2. get the size of data available on the given socket.

BSD sockets can operate in blocking or non-blocking mode. The application must check the return value to determine how many bytes have been sent or received. The blocking mode may cause problems if a socket continues to listen: a program hangs until some data arrives or internal timeout expires.

Parameters:
s: socket handle;
cmd: command to use. Currently are supported the following:

- **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.

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

Output data:
if cmd = **M2M_SOCKET_BSD_FIONREAD**:
"argp" is the pointer to the allocated variable filled with the number of the data to be read.

Note:
m2m_socket_errno(…) function returns the failure reason.

Examples: 17.10 Socket ioctl, 17.12 TCP-Server.

11.28. m2m_socket_bsd_listen

INT32 m2m_socket_bsd_listen(M2M_SOCKET_BSD_SOCKET s, INT32 backlog)

Description: the function 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: 17.12 TCP-Server.

11.29. m2m_socket_bsd_recv

INT32 m2m_socket_bsd_recv(M2M_SOCKET_BSD_SOCKET s, void *buf, INT32 len, INT32 flags)

Description: the function 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 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 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.
flags: not supported, ignored. Set it to 0.

Return value:
on success: number of bytes received.
on failure: < 0

Output data:
on success:
"buf" points to the buffer filled with received data.

Note:
m2m_socket_errno(…) function returns the failure reason.

Example: 17.11 TCP-Client

11.30. m2m_socket_bsd_recv_data_size

INT32 m2m_socket_bsd_recv_data_size(M2M_SOCKET_BSD_SOCKET s, UINT32 *len)

Description: the function 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.

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

Output data:
on success:
"len" points to the variable filled with the size of the data pending.

Note:
m2m_socket_errno(…) function returns the failure reason.

11.31. 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: the function 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 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.

Return value:

on success: number of bytes received.
on failure: < 0

Output data:

on success:
"buf" points to the buffer filled with received data.
"fromlen" points to the variable filled with the size of the socket address

Note:

m2m_socket_errno(…) function returns the failure reason.

Examples: 17.14 UDP-Server

11.32. m2m_socket_bsd_send

INT32 m2m_socket_bsd_send(M2M_SOCKET_BSD_SOCKET s, const void *buf, INT32 len, INT32 flags)

Description: the function 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: not supported, ignored. Set it to 0.

Return value:
on success: number of bytes sent.
on failure: < 0

Note:
m2m_socket_errno(…) function returns the failure reason.

Examples: 17.11 TCP-Client

11.33. m2m_socket_bsd_send_buf_size

UINT32 m2m_socket_bsd_send_buf_size(M2M_SOCKET_BSD_SOCKET s)

Description: the function 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.

Examples: 17.11 TCP-Client

11.34. 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: the function 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: not supported, ignored. Set it to 0.
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: 17.13 UDP-Client

11.35. m2m_socket_errno

INT32 m2m_socket_errno(void)

Description: the function returns the error code of the last "socket" operation.

Return value:

on success: 0.
on failure: refer to **Socket_Error_Types**.

11.36. m2m_pdp_activate

INT32 m2m_pdp_activate(CHAR *apn, CHAR *name, CHAR *pwd)

Description: the function activates a PDP context. Upon activation, **M2M_SOCKET_EVENT_PDP_ACTIVE** event is received via 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.
pwd: pointer to the User Password string.

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**

11.37. m2m_pdp_activate_ip6

INT32 m2m_pdp_activate_ip6(CHAR *apn, CHAR *name, CHAR *pwd)

Description: the function activates an IPV6 PDP context. Upon activation, **M2M_SOCKET_EVENT_PDP_IPV6_ACTIVE** event is received via 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.
pwd: pointer to the User Password string.

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**

11.38. m2m_pdp_deactive

INT32 m2m_pdp_deactive(void);

Description: the function deactivates a PDP context.

Return value (see **PDP_context_status** #defines):
on success: **M2M_PDP_STATE_SUCCESS**
on failure: **M2M_PDP_STATE_FAILURE**

11.39. m2m_pdp_get_status

INT32 m2m_pdp_get_status(void)

Description: the function gets the status of the PDP context.

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**

11.40. m2m_pdp_get_my_ip

UINT32 m2m_pdp_get_my_ip(void)

Description: the function gets the IP address of the PDP connection.

Return value:
on success: IP address in IPV4 format (32 bits inet address).
on failure: 0, if PDP is not active.

Here is an example using this function, and getting a string representation of the IP address:

```
CHAR  *ipAddr = m2m_socket_bsd_addr_str(m2m_pdp_get_my_ip());
```

11.41. m2m_pdp_get_my_ip6

INT32 m2m_pdp_get_my_ip6(M2M_SOCKET_BSD_IN6_ADDR *ipAddr)

Description: the function gets the IPV6 address of the 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 host ip6 address.

12. m2m_ipraw_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of ip6 raw mode.

12.1. m2m_ip6_raw

INT32 m2m_ip6_raw(UINT8 enable)

Description: the function enables/disables ip6 raw mode.

Parameters:
enable: 1 enable, 0 disable.

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

12.2. m2m_ip6_send

INT32 m2m_ip6_send(const void *buff, INT32 len)

Description: the function sends ip6 packets in raw mode.

Parameters:
buffer: pointer to the allocated buffer that contains the ip6 packet.
len: total number of characters written in the allocated buffer.

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

12.3. m2m_ip6_recv

INT32 m2m_ip6_raw(void *buff, INT32 len)

Description: the function returns the pending ip6 packets.

Parameters:
buffer: 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

12.4. 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: the function 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. - in output to specify the length of ip6 packet without payload.

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

13. m2m_ssl_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of the SSL. 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.

13.1. 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: the function creates an SSL service session using the certificates stored in a file. It is used for connection purposes, see **m2m_ssl_new_connection(…)** or **m2m_ssl_delete_connection(…)**. Each client application needs its own SSL service session.

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**.

13.2. m2m_ssl_delete_service

void m2m_ssl_delete_service(M2M_SSL_SERVICE_SESSION service_session)

Description: the function 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.

13.3. m2m_ssl_new_connection

INT32 m2m_ssl_new_connection(M2M_SSL_SERVICE_SESSION service_session, void *socket_fd,
INT32 flags, M2M_SSL_CONNECTION_CONTEXT *connection_ctx)

Description: the function creates a new client connection to the server. The connection, once closed, must be deleted using the **m2m_ssl_delete_connection(…)**.

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.

13.4. m2m_ssl_delete_connection

void m2m_ssl_delete_connection(M2M_SSL_CONNECTION_CONTEXT connection_ctx)

Description: the function deletes an existing connection.

Parameters:
connection_ctx: pointer to a valid, not **NULL**, connection session created with **m2m_ssl_new_connection (…)**.

13.5. m2m_ssl_encode_send

INT32 m2m_ssl_encode_send(M2M_SSL_CONNECTION_CONTEXT connection_ctx, UINT8 *buf, UINT32 len)

Description: the function 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_new_connection(…)**.
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.

13.6. m2m_ssl_decode

INT32 m2m_ssl_decode(M2M_SSL_CONNECTION_CONTEXT connection_ctx, UINT8 *buffer, INT32 length)

Description: the function 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_new_connection(…)**.
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.

14. m2m_timer_api.h

This header file provides the functions declarations (prototypes) of the set of APIs regarding the management of timers.

14.1. m2m_timer_create

M2M_T_TIMER_HANDLE m2m_timer_create(M2M_T_TIMER_TIMEOUT cb, void *arg)

Description: the function 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: 17.6 Timer

14.2. m2m_timer_start

void m2m_timer_start(M2M_T_TIMER_HANDLE timer, UINT32 msecs)

Description: the function starts the timer.

Parameters:
timer: timer handle.
msecs: timeout value in milliseconds.

Examples: 17.6 Timer

14.3. m2m_timer_stop

INT32 m2m_timer_stop(M2M_T_TIMER_HANDLE timer)

Description: the function 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.

14.4. m2m_timer_free

INT32 m2m_timer_free(M2M_T_TIMER_HANDLE timer_handle)

Description: the function 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.

15. Acronyms and Abbreviations

API	Application Programming Interface
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

16. Document History

Revision	Date	Product/SW Version	Changes
0	2015-02-16	/	First issue

17. Appendix: Examples

17.1. File System

```
#include "m2m_type.h"
#include "m2m_fs_api.h"

void create_file (void)
{
   CHAR  *file_name = "goofy";
   CHAR  buf[] = "hello world!";
   M2M_T_FS_HANDLE file_handle = NULL;

   if(M2M_API_RESULT_SUCCESS == m2m_fs_create(file_name))
    {
      file_handle = m2m_fs_open(file_name, M2M_FS_OPEN_APPEND);
      if(NULL != file_handle)
      {
         m2m_fs_write(file_handle, buf, sizeof(buf));
         m2m_fs_close(file_handle);
      }
   }
}
```

17.2. Listing all Files

```
CHAR item[128];

res = m2m_fs_find_first( item, "*" );
if ( res == M2M_API_RESULT_SUCCESS )
{
   PRINT( item );
   do
   {
      res = m2m_fs_find_next( item );
      if ( res == M2M_API_RESULT_SUCCESS )
      {
         PRINT( item );
       }
    }
    while ( res == M2M_API_RESULT_SUCCESS );
}
```

17.3. 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);
}
```

17.4. Semaphore CS

```
#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 behaviour: 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);
    .
    .
```

17.5. Semaphore IPC

```
#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);
    .
    .
```

17.6. 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  */
}
```

17.7. 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 */
}
```

17.8. RTC

```
#include <stdio.h>
#include <string.h>
#include "m2m_type.h"
#include "m2m_hw_api.h"
#include "m2m_clock_api.h"
#include "time.h"

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 method returns 0 on success, else returns -1 */

INT32 result = 0;
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;

result =  m2m_get_timeofday(&tv, &tz);
sec_cur = tv.tv_sec;
sec_prog = sec_cur + sec_prog;
restart = localtime(&sec_prog);

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 = (INT32)(restart->tm_year -100);

res_RTC = m2m_rtc_set_alarm(date_mem, time_mem);

if ( ( (INT32)res_RTC == 0 ) && (result == 0) )

   {result = 0;}

else

   {result = -1;}

return result;

}
```

17.9. SMS Storage

```
#include <stdio.h>
#include <string.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 )
  {
    PRINT("Error\n");
    return;
  }

 /* Get SMS storage status */
 Res = m2m_sms_get_preferred_message_storage(memory);
 if ( Res != 1 )
  {
    PRINT("Error\n");
    return;
  }

 for ( i = 0; i < 3; i ++ )
  {
    PRINT( "mem[%d] = %s", i, memory[i].mem );
    PRINT( "used[%d] = %d", i, memory[i].nUsed );
    PRINT( "tot[%d] = %d", i, memory[i].nTotal );
  }
}
```

17.10. 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 */
}
```

17.11. TCP-Client

```
void tcp_client()

{
	struct M2M_SOCKET_BSD_SOCKADDR_IN stSockAddr;
	CHAR  buf[21] = "Hello from AppZone!";
	INT32 Res;
	UINT32 addr = 0;
	UINT16 port = 0;
	INT32 namelen = 0;
	M2M_SOCKET_BSD_SOCKET SocketFD ;
	M2M_SOCKET_BSD_FD_SET  set;

	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)
	   {
	   return;
	   }

	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(xxxxx);
	stSockAddr.sin_addr.s_addr = m2m_socket_bsd_inet_addr("xxx.xxx.xxx.xxx");

	if (M2M_SOCKET_BSD_INVALID_SOCKET ==
	   m2m_socket_bsd_connect(SocketFD, (M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                                    M2M_SOCKET_BSD_SOCKADDR_IN)))
	   {
	   m2m_socket_bsd_close(SocketFD);
	   return;
	   }

	Res = m2m_socket_bsd_get_sock_name(SocketFD, ( M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, &namelen
);

	if (Res == 0)
	   {
	   /*  IP address and Port */
	   PRINT ("IP: %s",m2m_socket_bsd_addr_str(stSockAddr.sin_addr.s_addr));
	   PRINT ("PORT: %d",stSockAddr.sin_port);
	   }

	Res = m2m_socket_bsd_send_buf_size(SocketFD);

	if (Res > sizeof(buf))
	   {
	   Res = m2m_socket_bsd_send(SocketFD, buf, sizeof(buf), 0);
	   }

	m2m_socket_bsd_close(SocketFD);
	return;

}
```

17.12. TCP-Server

```
#include "m2m_type.h"
#include "m2m_socket_api.h"

void TCPServer()
{
   struct M2M_SOCKET_BSD_SOCKADDR_IN stSockAddr;
   M2M_SOCKET_BSD_SOCKET ConnectFD;

   M2M_SOCKET_BSD_SOCKET 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)
   {
      return;
   }

   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(6500);
   stSockAddr.sin_addr.s_addr = M2M_SOCKET_BSD_INADDR_ANY;

   if(M2M_SOCKET_BSD_INVALID_SOCKET ==
      m2m_socket_bsd_bind(SocketFD,(const struct M2M_SOCKET_BSD_SOCKADDR *)&stSockAddr, sizeof(struct
                          M2M_SOCKET_BSD_SOCKADDR_IN)))
   {
      m2m_socket_bsd_close(SocketFD);
      return;
   }

   if(M2M_SOCKET_BSD_INVALID_SOCKET == m2m_socket_bsd_listen(SocketFD, 2))
   {
      m2m_socket_bsd_close(SocketFD);
      return;
   }

   for(;;)
   {
     ConnectFD = m2m_socket_bsd_accept(SocketFD, NULL, NULL);

     if(0 > ConnectFD)
     {
        m2m_socket_bsd_close(SocketFD);
        return;
     }

     /* perform read/write operations */

     m2m_socket_bsd_close(ConnectFD);
   }

   return;
}
```

17.13. UDP-Client

```
#include "m2m_type.h"
#include "m2m_socket_api.h"

void UDPClient()
{
    M2M_SOCKET_BSD_SOCKET sock;
    struct M2M_SOCKET_BSD_SOCKADDR_IN sa;
    INT32 bytes_sent, buffer_length;
    CHAR  buffer[30];

    buffer_length = sprintf(buffer, sizeof buffer, "Hello World!");

    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)
    {
        return;
    }

    memset(&sa, 0, sizeof(sa));
    sa.sin_family = M2M_SOCKET_BSD_AF_INET;
    sa.sin_addr.s_addr = m2m_socket_bsd_inet_addr((_CHAR_*)"xxx.xxx.xxx.xxx");
    sa.sin_port = m2m_socket_bsd_htons(7501);

    bytes_sent = m2m_socket_bsd_send_to(sock, buffer, buffer_length, 0,(struct M2M_SOCKET_BSD_SOCKADDR*)&sa,
sizeof
                                                (struct M2M_SOCKET_BSD_SOCKADDR_IN));

    if (bytes_sent < 0)
        return;

    m2m_socket_bsd_close(sock);    /* close the socket */
    return;
}
```

17.14. UDP-Server

```
#include "m2m_type.h"
#include "m2m_socket_api.h"

void UDPServer(void)
{
    struct M2M_SOCKET_BSD_SOCKADDR_IN sa;
    CHAR  buffer[1024];
    INT32 fromlen, recsize;
    M2M_SOCKET_BSD_SOCKET sock =
    m2m_socket_bsd_socket(M2M_SOCKET_BSD_PF_INET, M2M_SOCKET_BSD_SOCK_DGRAM,
                                    M2M_SOCKET_BSD_IPPROTO_UDP);

    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(6500);

    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;
    }

    for (;;)
    {
        recsize = m2m_socket_bsd_recv_from(sock, (void *)buffer, 1024, 0, (struct M2M_SOCKET_BSD_SOCKADDR
*)&sa,
                                                    &fromlen);
        if (recsize < 0)
          return;
        m2m_os_sleep_ms(1000);
    }
}
```

18. Appendix: Declarations of C identifiers

This section collects the declarations of the C identifiers used by the function prototypes described in the present User Guide.

18.1. m2m_cb_app_func.h

```
/* 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
```

18.2. m2m_clock_api.h

```
/* 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 alarm are set */
        M2M_RTC_INVALID_ARG,                    /* Invalid argument */
        M2M_RTC_FAILURE                         /* Failure */
}M2M_T_RTC_RESULT;
```

18.3. m2m_fs_api.h

```
/* 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;
```

18.4. m2m_hw_api.h

```
/* 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_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 even, 1 odd. */
} M2M_T_HW_UART_IO_HW_OPTIONS;
```

18.5. m2m_spi_api.h

```
/* M2M_SPI_BUFFER_LEN: Max buffers length (in bytes) */
#define M2M_SPI_BUFFER_LEN 128

/* M2M_T_SPI_RESULT: SPI result definition */
typedef enum {
        M2M_SPI_SUCCESS = 0,                    /* Succes */
        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;
```

18.6. m2m_i2C_api.h

```
/* 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;
```

18.7. m2m_network_api.h

```
/* 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 neighbour cell in GSM mode */
    CELL_TYPE_UMTS_RANKED,                  /* Cell is a UMTS neighbour cell (all states but CELL_DCH) */
    CELL_TYPE_UMTS_NOT_RANKED,              /* Cell is a UMTS neighbour 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;
```

18.8. m2m_os_api.h

```
/* 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;
```

18.9. m2m_os_lock_api.h

```
/* M2M_T_OS_LOCK: Lock handle */
typedef void *M2M_T_OS_LOCK;
```

18.10. m2m_sms_api.h

```
/* 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_CHAR     20

/* 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;
```

18.11. m2m_socket_api.h

```
/* M2M_SOCKET_BSD_SOCKET: Socket identifier */
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      1       /* Stream socket type used for TCP */
#define M2M_SOCKET_BSD_SOCK_DGRAM               2       /* Datagram socket type used for UDP */
#define M2M_SOCKET_BSD_SOCK_RAW         3       /* Raw socket type */
#define M2M_SOCKET_BSD_SOCK_TUN         4  /* TUN (tunneling done at the ip layer) socket type, 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         0x0001      /* Turn on debugging info recording, Not supported */
#define M2M_SOCKET_BSD_SO_ACCEPTCONN    0x0002      /* Socket has had listen(), Not supported */
#define M2M_SOCKET_BSD_SO_REUSEADDR     0x0004      /* Allow local address reuse, always set */
#define M2M_SOCKET_BSD_SO_KEEPALIVE     0x0008      /* Keep connections alive, not enabled by default */
#define M2M_SOCKET_BSD_SO_DONTROUTE     0x0010      /* Just use interface addresses, Not supported */
#define M2M_SOCKET_BSD_SO_BROADCAST     0x0020      /* Permit sending of broadcast msgs, Not supported */
#define M2M_SOCKET_BSD_SO_USELOOPBACK   0x0040      /* Bypass hardware when possible, Not supported */
#define M2M_SOCKET_BSD_SO_LINGER        0x0080      /* Linger on close if data present, Not supported */
#define M2M_SOCKET_BSD_SO_OOBINLINE     0x0100      /* Leave received OOB data in line, Not supported */
#define M2M_SOCKET_BSD_SO_DONTLINGER   (INT32)(~M2M_SOCKET_BSD_SO_LINGER) /* Don't Linger, Not
supported */
#define M2M_SOCKET_BSD_SO_SNDBUF        0x1001      /* Send buffer size, supported */
#define M2M_SOCKET_BSD_SO_RCVBUF        0x1002      /* Receive buffer size, supported */
#define M2M_SOCKET_BSD_SO_SNDLOWAT      0x1003      /* Send low-water mark,          Not supported */
#define M2M_SOCKET_BSD_SO_RCVLOWAT      0x1004      /* Receive low-water mark, Not supported */
#define M2M_SOCKET_BSD_SO_SNDTIMEO      0x1005      /* Send timeout, Not supported */
#define M2M_SOCKET_BSD_SO_RCVTIMEO      0x1006      /* Receive timeout, supported */
#define M2M_SOCKET_BSD_SO_ERROR         0x1007      /* Get error status and clear  */
#define M2M_SOCKET_BSD_SO_TYPE          0x1008      /* Get socket type, supported */
```


```
#define M2M_SOCKET_BSD_TCP_NODELAY          0x01          /* Don't delay send to coalesce packets, supported  */

/* ========================================================================== */



/* 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;

/* M2M_SOCKET_BSD_SOCKADDR: 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      32


/* 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;
```


```
/* 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;


/* FD_SETs used for m2m_socket_bsd_select(). */
void m2m_socket_bsd_fd_zero_func(M2M_SOCKET_BSD_FD_SET* set);

/* FD_SETs used for m2m_socket_bsd_select(). */
void m2m_socket_bsd_fd_set_func(INT32 fd, M2M_SOCKET_BSD_FD_SET* set);

/* FD_SETs used for m2m_socket_bsd_select(). */
void m2m_socket_bsd_fd_clr_func(INT32 fd, M2M_SOCKET_BSD_FD_SET* set);

/* FD_SETs used for m2m_socket_bsd_select(). */
UINT8 m2m_socket_bsd_fd_isset_func(INT32 fd, M2M_SOCKET_BSD_FD_SET* set);

/* Network byte order <-> Host byte order conversion functions. */
UINT16  m2m_socket_bsd_htons(UINT16 x);             /* Host to network byte order (short) */
UINT16  m2m_socket_bsd_ntohs(UINT16 x);             /* Network to host byte order (short) */
UINT32  m2m_socket_bsd_htonl(UINT32 x);         /* Host to network byte order (long) */
UINT32  m2m_socket_bsd_ntohl(UINT32 x);         /* Network to host byte order (long) */


/* ======================================================================== */

/* 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, typicaly 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_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 opened. */
#define M2M_SOCKET_STATE_OPEN       1

/* socket is closed. */
#define M2M_SOCKET_STATE_CLOSED     0


/* ================================================================================ */
```


```
/* 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


/* ========================================================================== */
```

18.12. m2m_ssl_api.h

```
/* 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_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 occured while encoding on socket */
#define M2M_SSL_READ_ERROR              -16             /* An error occured while decoding from socket */
#define M2M_SSL_END_OF_FILE             -17             /* Ther's 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_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 */
```

18.13. m2m_timer_api.h

```
/* M2M_T_TIMER_HANDLE: timer handle provided by the m2m_timer_create() function */
typedef void *M2M_T_TIMER_HANDLE;                    /* Timer handle */

/* M2M_T_TIMER_TIMEOUT: timer callback function prototype */
typedef void (*M2M_T_TIMER_TIMEOUT)(void *);
```

18.14. m2m_type.h

```
/* APIs results definitions */
typedef enum
{
  M2M_API_RESULT_INVALID_ARG = -1,
  M2M_API_RESULT_FAIL = 0,
  M2M_API_RESULT_SUCCESS = 1

} M2M_API_RESULT;
```