GSM-AT Lib

Tilen MAJERLE

Mar 25, 2020

CONTENTS

Welcome to the documentation for version latest-develop.

GSM-AT Lib is generic, platform independent, library to control *SIM800/SIM900* or *SIM7000/SIM7020 (NB-Iot LTE)* devices from *SIMCom*. Its objective is to run on master system, while SIMCom device runs official AT commands firmware developed and maintained by *SIMCom*.

Download library · *Getting started* · Open Github

CHAPTER

ONE

FEATURES

- Supports `SIM800/SIM900 (2G)` and `SIM7000/SIM7020 (NB-Iot LTE)` modules
- Platform independent and very easy to port
  - Development of library under Win32 platform
  - Provided examples for ARM Cortex-M or Win32 platforms
- Written in C language (C99)
- Allows different configurations to optimize user requirements
- Supports implementation with operating systems with advanced inter-thread communications
  - Currently only OS mode is supported
  - 2 different threads handling user data and received data
    - First (producer) thread (collects user commands from user threads and starts the command processing)
    - Second (process) thread reads the data from GSM device and does the job accordingly
- Allows sequential API for connections in client and server mode
- Includes several applications built on top of library
  - MQTT client for MQTT connection
- User friendly MIT license

CHAPTER

TWO

REQUIREMENTS

- C compiler
- Supported GSM Physical device

CHAPTER

THREE

CONTRIBUTE

Fresh contributions are always welcome. Simple instructions to proceed:

1. Fork Github repository
2. Respect C style & coding rules used by the library
3. Create a pull request to `develop` branch with new features or bug fixes

Alternatively you may:

1. Report a bug
2. Ask for a feature request

CHAPTER

FOUR

LICENSE

```
MIT License

Copyright (c) 2020 Tilen MAJERLE

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Getting started

5.1.1 Download library

Library is primarly hosted on Github.

- Download latest release from releases area on Github
- Clone *develop* branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

- Download and install `git` if not already
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available `3` options
  - Run `git clone --recurse-submodules https://github.com/MaJerle/gsm-at-lib` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/gsm-at-lib` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch master https://github.com/MaJerle/gsm-at-lib` to clone *latest stable* branch, including submodules
- Navigate to `examples` directory and run favourite example

Update cloned to latest version

- Open console and navigate to path in the system where your resources repository is. Use command `cd your_path`
- Run `git pull origin master --recurse-submodules` command to pull latest changes and to fetch latest changes from submodules
- Run `git submodule foreach git pull origin master` to update & merge all submodules

Note: This is preferred option to use when you want to evaluate library and run prepared examples. Repository consists of multiple submodules which can be automatically downloaded when cloning and pulling changes from root repository.

5.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page.

- Copy `gsm_at_lib` folder to your project
- Add `gsm_at_lib/src/include` folder to *include path* of your toolchain
- Add port architecture `gsm_at_lib/src/include/system/port/_arch_` folder to *include path* of your toolchain
- Add source files from `gsm_at_lib/src/` folder to toolchain build
- Add source files from `gsm_at_lib/src/system/` folder to toolchain build for arch port
- Copy `gsm_at_lib/src/include/gsm/gsm_config_template.h` to project folder and rename it to `gsm_config.h`
- Build the project

5.1.3 Configuration file

Library comes with template config file, which can be modified according to needs. This file shall be named `gsm_config.h` and its default template looks like the one below:

Tip: Check *GSM Configuration* section for possible configuration settings

Listing 1: Config file template

```
1  /**
2   * \file            gsm_config_template.h
3   * \brief           Template config file
4   */
5
6  /*
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
```

(continues on next page)

(continued from previous page)

```
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef GSM_HDR_CONFIG_H
#define GSM_HDR_CONFIG_H

/* Rename this file to "gsm_config.h" for your application */

/*
 * Open "include/gsm/gsm_config_default.h" and
 * copy & replace here settings you want to change values
 */

/* After user configuration, call default config to merge config together */
#include "gsm/gsm_config_default.h"

#endif /* GSM_HDR_CONFIG_H */
```

5.2 User manual

5.2.1 Overview

IoT activity is embedded in today's application. Almost every device uses some type of communication, from WiFi, BLE, Sub-GHz or NB-IoT/LTE/2G/3G.

GSM-AT Lib has been developed to allow customers to:

- Develop on single (host MCU) architecture at the same time and do not care about device arch
- Shorten time to market

Customers using *GSM-AT Lib* do not need to take care about proper command for specific task, they can call API functions to execute the task. Library will take the necessary steps in order to send right command to device via low-level driver (usually UART) and process incoming response from device before it will notify application layer if it was successfuly or not.

To summarize:

- *GSM* device runs official *AT* firmware, provided by device vendor
- Host MCU runs custom application, together with *GSM-AT Lib* library
- Host MCU communicates with *GSM* device with UART or similar interface.

5.2.2 Architecture

Architecture of the library consists of 4 layers.

Fig. 1: GSM-AT layer architecture overview

Application layer

User layer is the highest layer of the final application. This is the part where API functions are called to execute some command.

Middleware layer

Middleware part is actively developed and shall not be modified by customer by any means. If there is a necessity to do it, often it means that developer of the application uses it wrongly. This part is platform independent and does not use any specific compiler features for proper operation.

Note: There is no compiler specific features implemented in this layer.

System & low-level layer

Application needs to fully implement this part and resolve it with care. Functions are related to actual implementation with *GSM* device and are highly architecture oriented. Some examples for *WIN32* and *ARM Cortex-M* are included with library.

Tip: Check *Porting guide* for detailed instructions and examples.

System functions

System functions are bridge between operating system running on embedded system and GSM-AT Library. Functions need to provide:

- Thread management
- Binary semaphore management
- Recursive mutex management
- Message queue management
- Current time status information

Tip: System function prototypes are available in *System functions* section.

Low-level implementation

Low-Level, or *GSM_LL*, is part, dedicated for communication between *GSM-AT* middleware and *GSM* physical device. Application needs to implement output function to send necessary *AT command* instruction aswell as implement *input module* to send received data from *GSM* device to *GSM-AT* middleware.

Application must also assure memory assignment for *Memory manager* when default allocation is used.

Tip: Low level, input module & memory function prototypes are available in *Low-Level functions*, *Input module* and *Memory manager* respectfully.

GSM physical device

5.2.3 Inter thread communication

GSM-AT Library is only available with operating system. For successful resources management, it uses `2` threads within library and allows multiple application threads to post new command to be processed.

Fig. 2: Inter-thread architecture block diagram

Producing and *Processing* threads are part of library, its implementation is in `gsm_threads.c` file.

Processing thread

Processing thread is in charge of processing each and every received character from *GSM* device. It can process *URC* messages which are received from *GSM* device without any command request. Some of them are:

- *+RECEIVE* indicating new data packet received from remote side on active connection
- *RING* indicating new call to be processed by *GSM*
- and more others

Note: Received messages without any command (URC messages) are sent to application layer using events, where they can be processed and used in further steps

This thread also checks and processes specific received messages based on active command. As an example, when application tries to make a new connection to remote server, it starts command with `AT+CIPSTART` message. Thread understands that active command is to connect to remote side and will wait for potential `0, CONNECT OK` message, indicating connection status. it will also wait for `OK` or `ERROR`, indicating *command finished* status before it unlocks **sync_sem** to unblock *producing thread*.

Tip: When thread tries to unlock **sync_sem**, it first checks if it has been locked by *producing thread*.

Producing thread

Producing thread waits for command messages posted from application thread. When new message has been received, it sends initial *AT message* over AT port.

- It checks if command is valid and if it has corresponding initial AT sequence, such as `AT+CIPSTART`
- It locks **sync_sem** semaphore and waits for processing thread to unlock it
  - *Processing thread* is in charge to read respone from *GSM* and react accordingly. See previous section for details.
- If application uses *blocking mode*, it unlocks command **sem** semaphore and returns response
- If application uses *non-blocking mode*, it frees memory for message and sends event with response message

Application thread

Application thread is considered any thread which calls API functions and therefore writes new messages to *producing message queue*, later processed by *producing thread*.

A new message memory is allocated in this thread and type of command is assigned to it, together with required input data for command. It also sets *blocking* or *non-blocking* mode, how command shall be executed.

When application tries to execute command in *blocking mode*, it creates new sync semaphore **sem**, locks it, writes message to *producing queue* and waits for **sem** to get unlocked. This effectively puts thread to blocked state by operating system and removes it from scheduler until semaphore is unlocked again. Semaphore **sem** gets unlocked in *producing thread* when response has been received for specific command.

Tip: **sem** semaphore is unlocked in *producing* thread after **sync_sem** is unlocked in *processing* thread

Note: Every command message uses its own **sem** semaphore to sync multiple *application* threads at the same time.

If message is to be executed in *non-blocking* mode, **sem** is not created as there is no need to block application thread. When this is the case, application thread will only write message command to *producing queue* and return status of writing to application.

5.2.4 Events and callback functions

Library uses events to notify application layer for (possible, but not limited to) unexpected events. This concept is used aswell for commands with longer executing time, such as *scanning access points* or when application starts new connection as client mode.

There are `3` types of events/callbacks available:

- *Global event* callback function, assigned when initializing library
- *Connection specific event* callback function, to process only events related to connection, such as *connection error*, *data send*, *data receive*, *connection closed*
- *API function* call based event callback function

Every callback is always called from protected area of middleware (when exclusing access is granted to single thread only), and it can be called from one of these 3 threads:

- *Producing thread*
- *Processing thread*
- *Input thread*, when `GSM_CFG_INPUT_USE_PROCESS` is enabled and `gsm_input_process()` function is called

Tip: Check *Inter thread communication* for more details about *Producing* and *Processing* thread.

Global event callback

Global event callback function is assigned at library initialization. It is used by the application to receive any kind of event, except the one related to connection:

- GSM station successfully connected to access point
- GSM physical device reset has been detected
- Restore operation finished
- New station has connected to access point
- and many more..

Tip: Check *Event management* section for different kind of events

By default, global event function is single function. If the application tries to split different events with different callback functions, it is possible to do so by using `gsm_evt_register()` function to register a new, custom, event function.

Tip: Implementation of *Netconn API* leverages `gsm_evt_register()` to receive event when station disconnected from wifi access point. Check its source file for actual implementation.

Listing 2: Netconn API module actual implementation

```
/**
 * \file            gsm_netconn.c
 * \brief           API functions for sequential calls
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
```

(continues on next page)

(continued from previous page)

```
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "gsm/gsm_netconn.h"
#include "gsm/gsm_private.h"
#include "gsm/gsm_conn.h"
#include "gsm/gsm_mem.h"

#if GSM_CFG_NETCONN || __DOXYGEN__

/* Check conditions */
#if !GSM_CFG_CONN
#error "GSM_CFG_CONN must be enabled for NETCONN API!"
#endif /* !GSM_CFG_CONN */

#if GSM_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2
#error "GSM_CFG_NETCONN_RECEIVE_QUEUE_LEN must be greater or equal to 2"
#endif /* GSM_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2 */

/**
 * \brief           Sequential API structure
 */
typedef struct gsm_netconn {
    struct gsm_netconn* next;                   /*!< Linked list entry */

    gsm_netconn_type_t type;                    /*!< Netconn type */

    size_t rcv_packets;                         /*!< Number of received packets so far on this connection */
    gsm_conn_p conn;                            /*!< Pointer to actual connection */

    gsm_sys_mbox_t mbox_receive;                /*!< Message queue for receive mbox */

    gsm_linbuff_t buff;                         /*!< Linear buffer structure */

    uint16_t conn_timeout;                      /*!< Connection timeout in units of seconds when
                                                     netconn is in server (listen) mode.
                                                     Connection will be automatically closed if there is no
                                                     data exchange in time. Set to `0` when timeout feature is disabled. */
```

(continues on next page)

(continued from previous page)

```
69
70  #if GSM_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__
71      uint32_t rcv_timeout;                       /*!< Receive timeout in unit of
    ↪milliseconds */
72  #endif
73  } gsm_netconn_t;
74
75  static uint8_t recv_closed = 0xFF;
76  static gsm_netconn_t* netconn_list;             /*!< Linked list of netconn entries */
77
78  /**
79   * \brief           Flush all mboxes and clear possible used memories
80   * \param[in]       nc: Pointer to netconn to flush
81   * \param[in]       protect: Set to 1 to protect against multi-thread access
82   */
83  static void
84  flush_mboxes(gsm_netconn_t* nc, uint8_t protect) {
85      gsm_pbuf_p pbuf;
86      if (protect) {
87          gsm_core_lock();
88      }
89      if (gsm_sys_mbox_isvalid(&nc->mbox_receive)) {
90          while (gsm_sys_mbox_getnow(&nc->mbox_receive, (void **)&pbuf)) {
91              if (pbuf != NULL && (uint8_t *)pbuf != (uint8_t *)&recv_closed) {
92                  gsm_pbuf_free(pbuf);            /* Free received data buffers */
93              }
94          }
95          gsm_sys_mbox_delete(&nc->mbox_receive); /* Delete message queue */
96          gsm_sys_mbox_invalid(&nc->mbox_receive);/* Invalid handle */
97      }
98      if (protect) {
99          gsm_core_unlock();
100     }
101 }
102
103 /**
104  * \brief           Callback function for every server connection
105  * \param[in]       evt: Pointer to callback structure
106  * \return          Member of \ref gsmr_t enumeration
107  */
108 static gsmr_t
109 netconn_evt(gsm_evt_t* evt) {
110     gsm_conn_p conn;
111     gsm_netconn_t* nc = NULL;
112     uint8_t close = 0;
113
114     conn = gsm_conn_get_from_evt(evt);          /* Get connection from event */
115     switch (gsm_evt_get_type(evt)) {
116         /*
117          * A new connection has been active
118          * and should be handled by netconn API
119          */
120         case GSM_EVT_CONN_ACTIVE: {             /* A new connection active is active
    ↪*/
121             if (gsm_conn_is_client(conn)) {     /* Was connection started by us? */
122                 nc = gsm_conn_get_arg(conn);    /* Argument should be already set */
123                 if (nc != NULL) {
```

(continues on next page)

(continued from previous page)

```
                    nc->conn = conn;            /* Save actual connection */
                } else {
                    close = 1;                  /* Close this connection, invalid netconn */
                }
            } else {
                GSM_DEBUGF(GSM_CFG_DBG_NETCONN | GSM_DBG_TYPE_TRACE | GSM_DBG_LVL_WARNING,
                    "[NETCONN] Closing connection, it is not in client mode!\r\n");
                close = 1;                      /* Close the connection at this point */
            }

            /* Decide if some events want to close the connection */
            if (close) {
                if (nc != NULL) {
                    gsm_conn_set_arg(conn, NULL);   /* Reset argument */
                    gsm_netconn_delete(nc);     /* Free memory for API */
                }
                gsm_conn_close(conn, 0);        /* Close the connection */
                close = 0;
            }
            break;
        }

        /*
         * We have a new data received which
         * should have netconn structure as argument
         */
        case GSM_EVT_CONN_RECV: {
            gsm_pbuf_p pbuf;

            nc = gsm_conn_get_arg(conn);        /* Get API from connection */
            pbuf = gsm_evt_conn_recv_get_buff(evt);/* Get received buff */

            gsm_conn_recved(conn, pbuf);        /* Notify stack about received data */

            gsm_pbuf_ref(pbuf);                 /* Increase reference counter */
            if (nc == NULL || !gsm_sys_mbox_isvalid(&nc->mbox_receive)
                || !gsm_sys_mbox_putnow(&nc->mbox_receive, pbuf)) {
                GSM_DEBUGF(GSM_CFG_DBG_NETCONN,
                    "[NETCONN] Ignoring more data for receive!\r\n");
                gsm_pbuf_free(pbuf);            /* Free pbuf */
                return gsmOKIGNOREMORE;         /* Return OK to free the memory and ignore further data */
            }
            ++nc->rcv_packets;                  /* Increase number of received packets */
            GSM_DEBUGF(GSM_CFG_DBG_NETCONN | GSM_DBG_TYPE_TRACE,
                "[NETCONN] Received pbuf contains %d bytes. Handle written to receive mbox\r\n",
                (int)gsm_pbuf_length(pbuf, 0));
            break;
        }

        /* Connection was just closed */
        case GSM_EVT_CONN_CLOSE: {
```

(continues on next page)

(continued from previous page)

```
            nc = gsm_conn_get_arg(conn);        /* Get API from connection */

            /*
             * In case we have a netconn available,
             * simply write pointer to received variable to indicate closed state
             */
            if (nc != NULL && gsm_sys_mbox_isvalid(&nc->mbox_receive)) {
                gsm_sys_mbox_putnow(&nc->mbox_receive, (void *)&recv_closed);
            }

            break;
        }
        default:
            return gsmERR;
    }
    return gsmOK;
}

/**
 * \brief           Global event callback function
 * \param[in]       evt: Callback information and data
 * \return          \ref gsmOK on success, member of \ref gsmr_t otherwise
 */
static gsmr_t
gsm_evt(gsm_evt_t* evt) {
    switch (gsm_evt_get_type(evt)) {
        default: break;
    }
    return gsmOK;
}

/**
 * \brief           Create new netconn connection
 * \param[in]       type: Netconn connection type
 * \return          New netconn connection on success, `NULL` otherwise
 */
gsm_netconn_p
gsm_netconn_new(gsm_netconn_type_t type) {
    gsm_netconn_t* a;
    static uint8_t first = 1;

    /* Register only once! */
    gsm_core_lock();
    if (first) {
        first = 0;
        gsm_evt_register(gsm_evt);              /* Register global event function */
    }
    gsm_core_unlock();
    a = gsm_mem_calloc(1, sizeof(*a));          /* Allocate memory for core object */
    if (a != NULL) {
        a->type = type;                         /* Save netconn type */
        a->conn_timeout = 0;                    /* Default connection timeout */
        if (!gsm_sys_mbox_create(&a->mbox_receive, GSM_CFG_NETCONN_RECEIVE_QUEUE_
↪LEN)) {    /* Allocate memory for receiving message box */
            GSM_DEBUGF(GSM_CFG_DBG_NETCONN | GSM_DBG_TYPE_TRACE | GSM_DBG_LVL_DANGER,
                "[NETCONN] Cannot create receive MBOX\r\n");
            goto free_ret;
```

(continues on next page)

(continued from previous page)

```
        }
        gsm_core_lock();
        if (netconn_list == NULL) {             /* Add new netconn to the existing list */
            netconn_list = a;
        } else {
            a->next = netconn_list;             /* Add it to beginning of the list */
            netconn_list = a;
        }
        gsm_core_unlock();
    }
    return a;
free_ret:
    if (gsm_sys_mbox_isvalid(&a->mbox_receive)) {
        gsm_sys_mbox_delete(&a->mbox_receive);
        gsm_sys_mbox_invalid(&a->mbox_receive);
    }
    if (a != NULL) {
        gsm_mem_free_s((void **)&a);
    }
    return NULL;
}

/**
 * \brief           Delete netconn connection
 * \param[in]       nc: Netconn handle
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_netconn_delete(gsm_netconn_p nc) {
    GSM_ASSERT("netconn != NULL", nc != NULL);

    gsm_core_lock();
    flush_mboxes(nc, 0);                        /* Clear mboxes */

    /* Remove netconn from linkedlist */
    if (netconn_list == nc) {
        netconn_list = netconn_list->next;      /* Remove first from linked list */
    } else if (netconn_list != NULL) {
        gsm_netconn_p tmp, prev;
        /* Find element on the list */
        for (prev = netconn_list, tmp = netconn_list->next;
            tmp != NULL; prev = tmp, tmp = tmp->next) {
            if (nc == tmp) {
                prev->next = tmp->next;         /* Remove tmp from linked list */
                break;
            }
        }
    }
    gsm_core_unlock();

    gsm_mem_free_s((void **)&nc);
    return gsmOK;
}

/**
 * \brief           Connect to server as client
```

(continues on next page)

(continued from previous page)

```
 * \param[in]       nc: Netconn handle
 * \param[in]       host: Pointer to host, such as domain name or IP address in string format
 * \param[in]       port: Target port to use
 * \return          \ref gsmOK if successfully connected, member of \ref gsmr_t otherwise
 */
gsmr_t
gsm_netconn_connect(gsm_netconn_p nc, const char* host, gsm_port_t port) {
    gsmr_t res;

    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("host != NULL", host != NULL);
    GSM_ASSERT("port > 0", port > 0);

    /*
     * Start a new connection as client and:
     *
     *  - Set current netconn structure as argument
     *  - Set netconn callback function for connection management
     *  - Start connection in blocking mode
     */
    res = gsm_conn_start(NULL, (gsm_conn_type_t)nc->type, host, port, nc, netconn_evt, 1);
    return res;
}

/**
 * \brief           Write data to connection output buffers
 * \note            This function may only be used on TCP or SSL connections
 * \param[in]       nc: Netconn handle used to write data to
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_netconn_write(gsm_netconn_p nc, const void* data, size_t btw) {
    size_t len, sent;
    const uint8_t* d = data;
    gsmr_t res;

    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("nc->type must be TCP or SSL", nc->type == GSM_NETCONN_TYPE_TCP || nc->type == GSM_NETCONN_TYPE_SSL);
    GSM_ASSERT("nc->conn must be active", gsm_conn_is_active(nc->conn));

    /*
     * Several steps are done in write process
     *
     * 1. Check if buffer is set and check if there is something to write to it.
     *    1. In case buffer will be full after copy, send it and free memory.
     * 2. Check how many bytes we can write directly without needed to copy
     * 3. Try to allocate a new buffer and copy remaining input data to it
     * 4. In case buffer allocation fails, send data directly (may affect on speed and effectivenes)
     */

```

(continues on next page)

(continued from previous page)

```
    /* Step 1 */
    if (nc->buff.buff != NULL) {                /* Is there a write buffer ready to accept more data? */
        len = GSM_MIN(nc->buff.len - nc->buff.ptr, btw);    /* Get number of bytes we can write to buffer */
        if (len > 0) {
            GSM_MEMCPY(&nc->buff.buff[nc->buff.ptr], data, len);/* Copy memory to temporary write buffer */
            d += len;
            nc->buff.ptr += len;
            btw -= len;
        }

        /* Step 1.1 */
        if (nc->buff.ptr == nc->buff.len) {
            res = gsm_conn_send(nc->conn, nc->buff.buff, nc->buff.len, &sent, 1);

            gsm_mem_free_s((void **)&nc->buff.buff);
            if (res != gsmOK) {
                return res;
            }
        } else {
            return gsmOK;                       /* Buffer is not yet full yet */
        }
    }

    /* Step 2 */
    if (btw >= GSM_CFG_CONN_MAX_DATA_LEN) {
        size_t rem;
        rem = btw % GSM_CFG_CONN_MAX_DATA_LEN;  /* Get remaining bytes for max data length */
        res = gsm_conn_send(nc->conn, d, btw - rem, &sent, 1);  /* Write data directly */
        if (res != gsmOK) {
            return res;
        }
        d += sent;                              /* Advance in data pointer */
        btw -= sent;                            /* Decrease remaining data to send */
    }

    if (btw == 0) {                             /* Sent everything? */
        return gsmOK;
    }

    /* Step 3 */
    if (nc->buff.buff == NULL) {                /* Check if we should allocate a new buffer */
        nc->buff.buff = gsm_mem_malloc(sizeof(*nc->buff.buff) * GSM_CFG_CONN_MAX_DATA_LEN);
        nc->buff.len = GSM_CFG_CONN_MAX_DATA_LEN;   /* Save buffer length */
        nc->buff.ptr = 0;                       /* Save buffer pointer */
    }

    /* Step 4 */
    if (nc->buff.buff != NULL) {                /* Memory available? */
        GSM_MEMCPY(&nc->buff.buff[nc->buff.ptr], d, btw);   /* Copy data to buffer */
        nc->buff.ptr += btw;
```

(continues on next page)

(continued from previous page)

```
    } else {                                    /* Still no memory available? */
        return gsm_conn_send(nc->conn, data, btw, NULL, 1); /* Simply send directly blocking */
    }
    return gsmOK;
}

/**
 * \brief           Flush buffered data on netconn \e TCP/SSL connection
 * \note            This function may only be used on \e TCP/SSL connection
 * \param[in]       nc: Netconn handle to flush data
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_netconn_flush(gsm_netconn_p nc) {
    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("nc->type must be TCP or SSL", nc->type == GSM_NETCONN_TYPE_TCP || nc->type == GSM_NETCONN_TYPE_SSL);
    GSM_ASSERT("nc->conn must be active", gsm_conn_is_active(nc->conn));

    /*
     * In case we have data in write buffer,
     * flush them out to network
     */
    if (nc->buff.buff != NULL) {                /* Check remaining data */
        if (nc->buff.ptr > 0) {                 /* Do we have data in current buffer? */
            gsm_conn_send(nc->conn, nc->buff.buff, nc->buff.ptr, NULL, 1);  /* Send data */
        }
        gsm_mem_free_s((void **)&nc->buff.buff);
    }
    return gsmOK;
}

/**
 * \brief           Send data on \e UDP connection to default IP and port
 * \param[in]       nc: Netconn handle used to send
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_netconn_send(gsm_netconn_p nc, const void* data, size_t btw) {
    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("nc->type must be UDP", nc->type == GSM_NETCONN_TYPE_UDP);
    GSM_ASSERT("nc->conn must be active", gsm_conn_is_active(nc->conn));

    return gsm_conn_send(nc->conn, data, btw, NULL, 1);
}

/**
 * \brief           Send data on \e UDP connection to specific IP and port
 * \note            Use this function in case of UDP type netconn
 * \param[in]       nc: Netconn handle used to send
 * \param[in]       ip: Pointer to IP address
 * \param[in]       port: Port number used to send data
```

(continues on next page)

(continued from previous page)

```
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_netconn_sendto(gsm_netconn_p nc, const gsm_ip_t* ip, gsm_port_t port, const void* data, size_t btw) {
    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("nc->type must be UDP", nc->type == GSM_NETCONN_TYPE_UDP);
    GSM_ASSERT("nc->conn must be active", gsm_conn_is_active(nc->conn));

    return gsm_conn_sendto(nc->conn, ip, port, data, btw, NULL, 1);
}

/**
 * \brief           Receive data from connection
 * \param[in]       nc: Netconn handle used to receive from
 * \param[in]       pbuf: Pointer to pointer to save new receive buffer to.
 *                     When function returns, user must check for valid pbuf value `pbuf != NULL`
 * \return          \ref gsmOK when new data ready,
 * \return          \ref gsmCLOSED when connection closed by remote side,
 * \return          \ref gsmTIMEOUT when receive timeout occurs
 * \return          Any other member of \ref gsmr_t otherwise
 */
gsmr_t
gsm_netconn_receive(gsm_netconn_p nc, gsm_pbuf_p* pbuf) {
    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("pbuf != NULL", pbuf != NULL);

    *pbuf = NULL;
#if GSM_CFG_NETCONN_RECEIVE_TIMEOUT
    /*
     * Wait for new received data for up to specific timeout
     * or throw error for timeout notification
     */
    if (gsm_sys_mbox_get(&nc->mbox_receive, (void **)pbuf, nc->rcv_timeout) == GSM_SYS_TIMEOUT) {
        return gsmTIMEOUT;
    }
#else /* GSM_CFG_NETCONN_RECEIVE_TIMEOUT */
    /* Forever wait for new receive packet */
    gsm_sys_mbox_get(&nc->mbox_receive, (void **)pbuf, 0);
#endif /* !GSM_CFG_NETCONN_RECEIVE_TIMEOUT */

    /* Check if connection closed */
    if ((uint8_t *)(*pbuf) == (uint8_t *)&recv_closed) {
        *pbuf = NULL;                           /* Reset pbuf */
        return gsmCLOSED;
    }
    return gsmOK;                               /* We have data available */
}

/**
 * \brief           Close a netconn connection
 * \param[in]       nc: Netconn handle to close
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
```

(continues on next page)

(continued from previous page)

```
 */
gsmr_t
gsm_netconn_close(gsm_netconn_p nc) {
    gsm_conn_p conn;

    GSM_ASSERT("nc != NULL", nc != NULL);
    GSM_ASSERT("nc->conn != NULL", nc->conn != NULL);
    GSM_ASSERT("nc->conn must be active", gsm_conn_is_active(nc->conn));

    gsm_netconn_flush(nc);                      /* Flush data and ignore result */
    conn = nc->conn;
    nc->conn = NULL;

    gsm_conn_set_arg(conn, NULL);               /* Reset argument */
    gsm_conn_close(conn, 1);                    /* Close the connection */
    flush_mboxes(nc, 1);                        /* Flush message queues */
    return gsmOK;
}

/**
 * \brief           Get connection number used for netconn
 * \param[in]       nc: Netconn handle
 * \return          `-1` on failure, connection number between `0` and \ref GSM_CFG_
↪MAX_CONNS otherwise
 */
int8_t
gsm_netconn_getconnnum(gsm_netconn_p nc) {
    if (nc != NULL && nc->conn != NULL) {
        return gsm_conn_getnum(nc->conn);
    }
    return -1;
}

#if GSM_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__

/**
 * \brief           Set timeout value for receiving data.
 *
 * When enabled, \ref gsm_netconn_receive will only block for up to
 * \e timeout value and will return if no new data within this time
 *
 * \param[in]       nc: Netconn handle
 * \param[in]       timeout: Timeout in units of milliseconds.
 *                  Set to `0` to disable timeout for \ref gsm_netconn_receive␣
↪function
 */
void
gsm_netconn_set_receive_timeout(gsm_netconn_p nc, uint32_t timeout) {
    nc->rcv_timeout = timeout;
}

/**
 * \brief           Get netconn receive timeout value
 * \param[in]       nc: Netconn handle
 * \return          Timeout in units of milliseconds.
 *                  If value is `0`, timeout is disabled (wait forever)
 */
```

(continues on next page)

(continued from previous page)

```
uint32_t
gsm_netconn_get_receive_timeout(gsm_netconn_p nc) {
    return nc->rcv_timeout;                     /* Return receive timeout */
}

#endif /* GSM_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__ */

#endif /* GSM_CFG_NETCONN || __DOXYGEN__ */
```

Connection specific event

This events are subset of global event callback. They work exactly the same way as global, but only receive events related to connections.

Tip: Connection related events start with `GSM_EVT_CONN_*`, such as `GSM_EVT_CONN_RECV`. Check *Event management* for list of all connection events.

Connection events callback function is set when client (application starts connection) starts a new connection with *`gsm_conn_start()`* function

Listing 3: An example of client with its dedicated event callback function

```
#include "client.h"
#include "gsm/gsm.h"
#include "gsm/gsm_network_api.h"

/* Host parameter */
#define CONN_HOST           "example.com"
#define CONN_PORT           80

static gsmr_t   conn_callback_func(gsm_evt_t* evt);

/**
 * \brief           Request data for connection
 */
static const
uint8_t req_data[] = ""
"GET / HTTP/1.1\r\n"
"Host: " CONN_HOST "\r\n"
"Connection: close\r\n"
"\r\n";

/**
 * \brief           Start a new connection(s) as client
 */
void
client_connect(void) {
    gsmr_t res;

    /* Attach to GSM network */
    gsm_network_request_attach();

    /* Start a new connection as client in non-blocking mode */
```

(continues on next page)

(continued from previous page)

```
    if ((res = gsm_conn_start(NULL, GSM_CONN_TYPE_TCP, "example.com", 80, NULL, conn_callback_func, 0)) == gsmOK) {
        printf("Connection to " CONN_HOST " started...\r\n");
    } else {
        printf("Cannot start connection to " CONN_HOST "!\r\n");
    }
}

/**
 * \brief           Event callback function for connection-only
 * \param[in]       evt: Event information with data
 * \return          \ref gsmOK on success, member of \ref gsmr_t otherwise
 */
static gsmr_t
conn_callback_func(gsm_evt_t* evt) {
    gsm_conn_p conn;
    gsmr_t res;
    uint8_t conn_num;

    conn = gsm_conn_get_from_evt(evt);          /* Get connection handle from event */
    if (conn == NULL) {
        return gsmERR;
    }
    conn_num = gsm_conn_getnum(conn);           /* Get connection number for identification */
    switch (gsm_evt_get_type(evt)) {
        case GSM_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = gsm_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /* Start sending data in non-blocking mode */
            if (res == gsmOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection manually...\r\n");
                gsm_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case GSM_EVT_CONN_CLOSE: {              /* Connection closed */
            if (gsm_evt_conn_close_is_forced(evt)) {
                printf("Connection %d closed by client!\r\n", (int)conn_num);
            } else {
                printf("Connection %d closed by remote side!\r\n", (int)conn_num);
            }
            break;
        }
        case GSM_EVT_CONN_SEND: {               /* Data send event */
            gsmr_t res = gsm_evt_conn_send_get_result(evt);
            if (res == gsmOK) {
                printf("Data sent successfully on connection %d...waiting to receive data from remote side...\r\n", (int)conn_num);
            } else {
                printf("Error while sending data on connection %d!\r\n", (int)conn_num);
            }
            break;
```

(continues on next page)

(continued from previous page)

```
        }
        case GSM_EVT_CONN_RECV: {               /* Data received from remote side */
            gsm_pbuf_p pbuf = gsm_evt_conn_recv_get_buff(evt);
            gsm_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)gsm_pbuf_
↪length(pbuf, 1), (int)conn_num);
            break;
        }
        case GSM_EVT_CONN_ERROR: {              /* Error connecting to server */
            const char* host = gsm_evt_conn_error_get_host(evt);
            gsm_port_t port = gsm_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default: break;
    }
    return gsmOK;
}
```

API call event

API function call event function is special type of event and is linked to command execution. It is especially useful when dealing with non-blocking commands to understand when specific command execution finished and when next operation could start.

Every API function, which directly operates with AT command on physical device layer, has optional `2` parameters for API call event:

- Callback function, called when command finished
- Custom user parameter for callback function

Below is an example code for SMS send. It uses custom API callback function to notify application when command has been executed successfully

Listing 4: Simple example for API call event, using DNS module

```
/* Somewhere in thread function */

/* Get device hostname in blocking mode */
/* Function returns actual result */
if (gsm_sms_send("+0123456789", "text", NULL, NULL, 1 /* 1 means blocking call */) ==␣
↪gsmOK) {
    /* At this point we have valid result from device */
    printf("SMS sent successfully\r\n");
} else {
    printf("Error trying to send SMS..\r\n");
}
```

5.2.5 Blocking or non-blocking API calls

API functions often allow application to set `blocking` parameter indicating if function shall be blocking or non-blocking.

Blocking mode

When the function is called in blocking mode `blocking = 1`, application thread gets suspended until response from *GSM* device is received. If there is a queue of multiple commands, thread may wait a while before receiving data.

When API function returns, application has valid response data and can react immediately.

- Linear programming model may be used
- Application may use multiple threads for real-time execution to prevent system stalling when running function call

Warning: Due to internal architecture, it is not allowed to call API functions in *blocking mode* from events or callbacks. Any attempt to do so will result in function returning error.

Example code:

Listing 5: Blocking command example

```
/* Somewhere in thread function */

/* Get device hostname in blocking mode */
/* Function returns actual result */
if (gsm_sms_send("+0123456789", "text", NULL, NULL, 1 /* 1 means blocking call */) ==␣
↪gsmOK) {
    /* At this point we have valid result from device */
    printf("SMS sent successfully\r\n");
} else {
    printf("Error trying to send SMS..\r\n");
}
```

Non-blocking mode

If the API function is called in non-blocking mode, function will return immediately with status indicating if command request has been successfully sent to internal command queue. Response has to be processed in event callback function.

> **Warning:** Due to internal architecture, it is only allowed to call API functions in *non-blocking mode* from events or callbacks. Any attempt to do so will result in function returning error.

Example code:

Listing 6: Non-blocking command example

```
/* Hostname event function, called when gsm_sms_send() function finishes */
void
sms_send_fn(gsmr_t res, void* arg) {
    /* Check actual result from device */
    if (res == gsmOK) {
        printf("SMS sent successfully\r\n");
    } else {
        printf("Error trying to send SMS\r\n");
    }
}

/* Somewhere in thread and/or other GSM event function */

/* Send SMS in non-blocking mode */
/* Function now returns if command has been sent to internal message queue */
if (gsm_sms_send("number", "text message", sms_send_fn, NULL, 0 /* 0 means non-
↪blocking call */) == gsmOK) {
    /* At this point we only know that command has been sent to queue */
    printf("SMS send message command sent to queue.\r\n");
} else {
    /* Error writing message to queue */
    printf("Cannot send SMS send message command to queue. Maybe out of memory? Check␣
↪result from function\r\n");
}
```

> **Warning:** When using non-blocking API calls, do not use local variables as parameter. This may introduce *undefined behavior* and *memory corruption* if application function returns before command is executed.

Example of a bad code:

Listing 7: Example of bad usage of non-blocking command

```
/* Hostname event function, called when gsm_sms_send() function finishes */
void
sms_send_fn(gsmr_t res, void* arg) {
    /* Check actual result from device */
    if (res == gsmOK) {
        printf("SMS sent successfully\r\n");
    } else {
        printf("Error trying to send SMS\r\n");
```

(continues on next page)

(continued from previous page)

```
    }
}

/* Check hostname */
void
check_hostname(void) {
    char message[] = "text message";

    /* Send SMS in non-blocking mode */
    /* Function now returns if command has been sent to internal message queue */
    /* It uses pointer to local data but w/o blocking command */
    if (gsm_sms_send("number", message, sms_send_fn, NULL, 0 /* 0 means non-blocking call */) == gsmOK) {
        /* At this point we only know that command has been sent to queue */
        printf("SMS send message command sent to queue.\r\n");
    } else {
        /* Error writing message to queue */
        printf("Cannot send SMS send message command to queue. Maybe out of memory? Check result from function\r\n");
    }
}
```

5.2.6 Porting guide

High level of *GSM-AT* library is platform independent, written in ANSI C99, however there is an important part where middleware needs to communicate with target *GSM* device and it must work under different optional operating systems selected by final customer.

Porting consists of:

- Implementation of *low-level* part, for actual communication between host device and *GSM* device
- Implementation of system functions, link between target operating system and middleware functions
- Assignment of memory for allocation manager

Implement low-level driver

To successfully implement all parts of *low-level* driver, application must take care of:

- Implementing `gsm_ll_init()` and `gsm_ll_deinit()` callback functions
- Implement and assign *send data* and optional *hardware reset* functions callbacks
- Assign memory for allocation manager when using default allocator
- Process received data from *GSM* device and send them to input module for further processing

Tip: Port examples are available for STM32 and WIN32 architectures. Both actual working and up-to-date implementations are available within the library.

Note: Check *Input module* for more information about direct & indirect input processing.

Implement system functions

System functions are bridge between operating system calls and *GSM* middleware. *GSM* library relies on stable operating system features and its implementation and does not require any special features which do not normally come with operating systems.

Operating system must support:

- Thread management functions
- Mutex management functions
- Binary semaphores only functions, no need for counting semaphores
- Message queue management functions

Warning: If any of the features are not available within targeted operating system, customer needs to resolve it with care. As an example, message queue is not available in WIN32 OS API therefore custom message queue has been implemented using binary semaphores

Application needs to implement all system call functions, starting with `gsm_sys_`. It must also prepare header file for standard types in order to support OS types within *GSM* middleware.

An example code is provided latter section of this page for WIN32 and STM32.

Note: Check *System functions* for function prototypes.

Steps to follow

- Copy `gsm_at_lib/src/system/gsm_sys_template.c` to the same folder and rename it to application port, eg. `gsm_sys_win32.c`
- Open newly created file and implement all system functions
- Copy folder `gsm_at_lib/src/include/system/port/template/*` to the same folder and rename *folder name* to application port, eg. `cmsis_os`
- Open `gsm_sys_port.h` file from newly created folder and implement all *typedefs* and *macros* for specific target
- Add source file to compiler sources and add path to header file to include paths in compiler options

Note: Check *System functions* for function prototypes.

Example: Low-level driver for WIN32

Example code for low-level porting on *WIN32* platform. It uses native *Windows* features to open *COM* port and read/write from/to it.

Notes:

- It uses separate thread for received data processing. It uses *gsm_input_process()* or *gsm_input()* functions, based on application configuration of *GSM_CFG_INPUT_USE_PROCESS* parameter.
  - When *GSM_CFG_INPUT_USE_PROCESS* is disabled, dedicated receive buffer is created by *GSM-AT* library and *gsm_input()* function just writes data to it and does not process received characters immediately. This is handled by *Processing* thread at later stage instead.
  - When *GSM_CFG_INPUT_USE_PROCESS* is enabled, *gsm_input_process()* is used, which directly processes input data and sends potential callback/event functions to application layer.
- Memory manager has been assigned to `1` region of `GSM_MEM_SIZE` size
- It sets *send* and *reset* callback functions for *GSM-AT* library

Listing 8: Actual implementation of low-level driver for WIN32

```
/**
 * \file            gsm_ll_win32.c
 * \brief           Low-level communication with GSM device for WIN32
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "system/gsm_ll.h"
#include "gsm/gsm.h"
#include "gsm/gsm_mem.h"
```

(continues on next page)

(continued from previous page)

```
#include "gsm/gsm_input.h"

#if !__DOXYGEN__

static uint8_t initialized = 0;
static HANDLE thread_handle;
static volatile HANDLE com_port;                /*!< COM port handle */
static uint8_t data_buffer[0x1000];             /*!< Received data array */

static void uart_thread(void* param);

/**
 * \brief           Send data to GSM device, function called from GSM stack when we have data to send
 * \param[in]       data: Pointer to data to send
 * \param[in]       len: Number of bytes to send
 * \return          Number of bytes sent
 */
static size_t
send_data(const void* data, size_t len) {
    DWORD written;
    if (com_port != NULL) {
#if !GSM_CFG_AT_ECHO
        const uint8_t* d = data;
        HANDLE hConsole;

        hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
        for (DWORD i = 0; i < len; ++i) {
            printf("%c", d[i]);
        }
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
#endif /* !GSM_CFG_AT_ECHO */

        /* Write data to AT port */
        WriteFile(com_port, data, len, &written, NULL);
        FlushFileBuffers(com_port);
        return written;
    }
    return 0;
}

/**
 * \brief           Configure UART (USB to UART)
 */
static void
configure_uart(uint32_t baudrate) {
    DCB dcb = { 0 };
    dcb.DCBlength = sizeof(dcb);

    /*
     * On first call,
     * create virtual file on selected COM port and open it
     * as generic read and write
     */
    if (!initialized) {
```

(continues on next page)

(continued from previous page)

```
        static const LPCWSTR com_ports[] = {
            L"\\\\.\\COM23",
            L"\\\\.\\COM12",
            L"\\\\.\\COM9",
            L"\\\\.\\COM8",
            L"\\\\.\\COM4"
        };
        for (size_t i = 0; i < sizeof(com_ports) / sizeof(com_ports[0]); ++i) {
            com_port = CreateFile(com_ports[i],
                GENERIC_READ | GENERIC_WRITE,
                0,
                0,
                OPEN_EXISTING,
                0,
                NULL
            );
            if (GetCommState(com_port, &dcb)) {
                printf("COM PORT %s opened!\r\n", (const char *)com_ports[i]);
                break;
            }
        }
    }

    /* Configure COM port parameters */
    if (GetCommState(com_port, &dcb)) {
        COMMTIMEOUTS timeouts;

        dcb.BaudRate = baudrate;
        dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;
        dcb.StopBits = ONESTOPBIT;

        if (!SetCommState(com_port, &dcb)) {
            printf("Cannot set COM PORT info\r\n");
        }
        if (GetCommTimeouts(com_port, &timeouts)) {
            /* Set timeout to return immediately from ReadFile function */
            timeouts.ReadIntervalTimeout = MAXDWORD;
            timeouts.ReadTotalTimeoutConstant = 0;
            timeouts.ReadTotalTimeoutMultiplier = 0;
            if (!SetCommTimeouts(com_port, &timeouts)) {
                printf("Cannot set COM PORT timeouts\r\n");
            }
            GetCommTimeouts(com_port, &timeouts);
        } else {
            printf("Cannot get COM PORT timeouts\r\n");
        }
    } else {
        printf("Cannot get COM PORT info\r\n");
    }

    /* On first function call, create a thread to read data from COM port */
    if (!initialized) {
        thread_handle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)uart_thread, NULL,␣
↪0, 0);
    }
}
```

(continues on next page)

(continued from previous page)

```
148
149 /**
150  * \brief            UART thread
151  */
152 static void
153 uart_thread(void* param) {
154     DWORD bytes_read;
155     gsm_sys_sem_t sem;
156     FILE* file = NULL;
157
158     gsm_sys_sem_create(&sem, 0);                /* Create semaphore for delay␣
    ↪functions */
159
160     while (com_port == NULL) {
161         gsm_sys_sem_wait(&sem, 1);              /* Add some delay with yield */
162     }
163
164     fopen_s(&file, "log_file.txt", "w+");       /* Open debug file in write mode */
165     while (1) {
166         /*
167          * Try to read data from COM port
168          * and send it to upper layer for processing
169          */
170         do {
171             ReadFile(com_port, data_buffer, sizeof(data_buffer), &bytes_read, NULL);
172             if (bytes_read > 0) {
173                 HANDLE hConsole;
174                 hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
175                 SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
176                 for (DWORD i = 0; i < bytes_read; ++i) {
177                     printf("%c", data_buffer[i]);
178                 }
179                 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |␣
    ↪FOREGROUND_BLUE);
180
181                 /* Send received data to input processing module */
182 #if GSM_CFG_INPUT_USE_PROCESS
183                 gsm_input_process(data_buffer, (size_t)bytes_read);
184 #else /* GSM_CFG_INPUT_USE_PROCESS */
185                 gsm_input(data_buffer, (size_t)bytes_read);
186 #endif /* !GSM_CFG_INPUT_USE_PROCESS */
187
188                 /* Write received data to output debug file */
189                 if (file != NULL) {
190                     fwrite(data_buffer, 1, bytes_read, file);
191                     fflush(file);
192                 }
193             }
194         } while (bytes_read == (DWORD)sizeof(data_buffer));
195
196         /* Implement delay to allow other tasks processing */
197         gsm_sys_sem_wait(&sem, 1);
198     }
199 }
200
201 /**
202  * \brief            Callback function called from initialization process
```

(continues on next page)

(continued from previous page)

```
 *
 * \note            This function may be called multiple times if AT baudrate is changed from application.
 *                  It is important that every configuration except AT baudrate is configured only once!
 *
 * \note            This function may be called from different threads in GSM stack when using OS.
 *                  When \ref GSM_CFG_INPUT_USE_PROCESS is set to 1, this function may be called from user UART thread.
 *
 * \param[in,out]   ll: Pointer to \ref gsm_ll_t structure to fill data for communication functions
 * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
 */
gsmr_t
gsm_ll_init(gsm_ll_t* ll) {
#if !GSM_CFG_MEM_CUSTOM
    /* Step 1: Configure memory for dynamic allocations */
    static uint8_t memory[0x10000];             /* Create memory for dynamic allocations with specific size */

    /*
     * Create memory region(s) of memory.
     * If device has internal/external memory available,
     * multiple memories may be used
     */
    gsm_mem_region_t mem_regions[] = {
        { memory, sizeof(memory) }
    };
    if (!initialized) {
        gsm_mem_assignmemory(mem_regions, GSM_ARRAYSIZE(mem_regions));  /* Assign memory for allocations to GSM library */
    }
#endif /* !GSM_CFG_MEM_CUSTOM */

    /* Step 2: Set AT port send function to use when we have data to transmit */
    if (!initialized) {
        ll->send_fn = send_data;                /* Set callback function to send data */
    }

    /* Step 3: Configure AT port to be able to send/receive data to/from GSM device */
    configure_uart(ll->uart.baudrate);          /* Initialize UART for communication */
    initialized = 1;
    return gsmOK;
}

#endif /* !__DOXYGEN__ */
```

Example: Low-level driver for STM32

Example code for low-level porting on *STM32* platform. It uses *CMSIS-OS* based application layer functions for implementing threads & other OS dependent features.

Notes:

- It uses separate thread for received data processing. It uses `gsm_input_process()` function to directly process received data without using intermediate receive buffer
- Memory manager has been assigned to `1` region of `GSM_MEM_SIZE` size
- It sets *send* and *reset* callback functions for *GSM-AT* library

Listing 9: Actual implementation of low-level driver for STM32

```
/**
 * \file            gsm_ll_stm32.c
 * \brief           Generic STM32 driver, included in various STM32 driver variants
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */

/*
 * How it works
 *
 * On first call to \ref gsm_ll_init, new thread is created and processed in usart_ll_
↪thread function.
 * USART is configured in RX DMA mode and any incoming bytes are processed inside␣
↪thread function.
 * DMA and USART implement interrupt handlers to notify main thread about new data␣
↪ready to send to upper layer.
```

(continues on next page)

(continued from previous page)

```
 *
 * More about UART + RX DMA: https://github.com/MaJerle/stm32-usart-dma-rx-tx
 *
 * \ref GSM_CFG_INPUT_USE_PROCESS must be enabled in `gsm_config.h` to use this driver.
 */
#include "gsm/gsm.h"
#include "gsm/gsm_mem.h"
#include "gsm/gsm_input.h"
#include "system/gsm_ll.h"

#if !__DOXYGEN__

#if !GSM_CFG_INPUT_USE_PROCESS
#error "GSM_CFG_INPUT_USE_PROCESS must be enabled in `gsm_config.h` to use this driver."
#endif /* GSM_CFG_INPUT_USE_PROCESS */

#if !defined(GSM_USART_DMA_RX_BUFF_SIZE)
#define GSM_USART_DMA_RX_BUFF_SIZE      0x1000
#endif /* !defined(GSM_USART_DMA_RX_BUFF_SIZE) */

#if !defined(GSM_MEM_SIZE)
#define GSM_MEM_SIZE                    0x1000
#endif /* !defined(GSM_MEM_SIZE) */

#if !defined(GSM_USART_RDR_NAME)
#define GSM_USART_RDR_NAME              RDR
#endif /* !defined(GSM_USART_RDR_NAME) */

/* USART memory */
static uint8_t      usart_mem[GSM_USART_DMA_RX_BUFF_SIZE];
static uint8_t      is_running, initialized;
static size_t       old_pos;

/* USART thread */
static void usart_ll_thread(void* arg);
static osThreadId_t usart_ll_thread_id;

/* Message queue */
static osMessageQueueId_t usart_ll_mbox_id;

/**
 * \brief           USART data processing
 */
static void
usart_ll_thread(void* arg) {
    size_t pos;

    GSM_UNUSED(arg);

    while (1) {
        void* d;
        /* Wait for the event message from DMA or USART */
        osMessageQueueGet(usart_ll_mbox_id, &d, NULL, osWaitForever);

        /* Read data */
```

(continues on next page)

(continued from previous page)

```
#if defined(GSM_USART_DMA_RX_STREAM)
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(GSM_USART_DMA, GSM_USART_DMA_
↪RX_STREAM);
#else
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(GSM_USART_DMA, GSM_USART_DMA_
↪RX_CH);
#endif /* defined(GSM_USART_DMA_RX_STREAM) */
        if (pos != old_pos && is_running) {
            if (pos > old_pos) {
                gsm_input_process(&usart_mem[old_pos], pos - old_pos);
            } else {
                gsm_input_process(&usart_mem[old_pos], sizeof(usart_mem) - old_pos);
                if (pos > 0) {
                    gsm_input_process(&usart_mem[0], pos);
                }
            }
            old_pos = pos;
            if (old_pos == sizeof(usart_mem)) {
                old_pos = 0;
            }
        }
    }
}

/**
 * \brief           Configure UART using DMA for receive in double buffer mode and␣
↪IDLE line detection
 */
static void
configure_uart(uint32_t baudrate) {
    static LL_USART_InitTypeDef usart_init;
    static LL_DMA_InitTypeDef dma_init;
    LL_GPIO_InitTypeDef gpio_init;

    if (!initialized) {
        /* Enable peripheral clocks */
        GSM_USART_CLK;
        GSM_USART_DMA_CLK;
        GSM_USART_TX_PORT_CLK;
        GSM_USART_RX_PORT_CLK;

#if defined(GSM_RESET_PIN)
        GSM_RESET_PORT_CLK;
#endif /* defined(GSM_RESET_PIN) */

        /* Global pin configuration */
        LL_GPIO_StructInit(&gpio_init);
        gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
        gpio_init.Pull = LL_GPIO_PULL_UP;
        gpio_init.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
        gpio_init.Mode = LL_GPIO_MODE_OUTPUT;

#if defined(GSM_RESET_PIN)
        /* Configure RESET pin */
        gpio_init.Pin = GSM_RESET_PIN;
        LL_GPIO_Init(GSM_RESET_PORT, &gpio_init);
#endif /* defined(GSM_RESET_PIN) */
```

(continues on next page)

(continued from previous page)

```
150
151         /* Configure USART pins */
152         gpio_init.Mode = LL_GPIO_MODE_ALTERNATE;
153
154         /* TX PIN */
155         gpio_init.Alternate = GSM_USART_TX_PIN_AF;
156         gpio_init.Pin = GSM_USART_TX_PIN;
157         LL_GPIO_Init(GSM_USART_TX_PORT, &gpio_init);
158
159         /* RX PIN */
160         gpio_init.Alternate = GSM_USART_RX_PIN_AF;
161         gpio_init.Pin = GSM_USART_RX_PIN;
162         LL_GPIO_Init(GSM_USART_RX_PORT, &gpio_init);
163
164         /* Configure UART */
165         LL_USART_DeInit(GSM_USART);
166         LL_USART_StructInit(&usart_init);
167         usart_init.BaudRate = baudrate;
168         usart_init.DataWidth = LL_USART_DATAWIDTH_8B;
169         usart_init.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
170         usart_init.OverSampling = LL_USART_OVERSAMPLING_16;
171         usart_init.Parity = LL_USART_PARITY_NONE;
172         usart_init.StopBits = LL_USART_STOPBITS_1;
173         usart_init.TransferDirection = LL_USART_DIRECTION_TX_RX;
174         LL_USART_Init(GSM_USART, &usart_init);
175
176         /* Enable USART interrupts and DMA request */
177         LL_USART_EnableIT_IDLE(GSM_USART);
178         LL_USART_EnableIT_PE(GSM_USART);
179         LL_USART_EnableIT_ERROR(GSM_USART);
180         LL_USART_EnableDMAReq_RX(GSM_USART);
181
182         /* Enable USART interrupts */
183         NVIC_SetPriority(GSM_USART_IRQ, NVIC_EncodePriority(NVIC_
    ↪GetPriorityGrouping(), 0x07, 0x00));
184         NVIC_EnableIRQ(GSM_USART_IRQ);
185
186         /* Configure DMA */
187         is_running = 0;
188 #if defined(GSM_USART_DMA_RX_STREAM)
189         LL_DMA_DeInit(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
190         dma_init.Channel = GSM_USART_DMA_RX_CH;
191 #else
192         LL_DMA_DeInit(GSM_USART_DMA, GSM_USART_DMA_RX_CH);
193         dma_init.PeriphRequest = GSM_USART_DMA_RX_REQ_NUM;
194 #endif /* defined(GSM_USART_DMA_RX_STREAM) */
195         dma_init.PeriphOrM2MSrcAddress = (uint32_t)&GSM_USART->GSM_USART_RDR_NAME;
196         dma_init.MemoryOrM2MDstAddress = (uint32_t)usart_mem;
197         dma_init.Direction = LL_DMA_DIRECTION_PERIPH_TO_MEMORY;
198         dma_init.Mode = LL_DMA_MODE_CIRCULAR;
199         dma_init.PeriphOrM2MSrcIncMode = LL_DMA_PERIPH_NOINCREMENT;
200         dma_init.MemoryOrM2MDstIncMode = LL_DMA_MEMORY_INCREMENT;
201         dma_init.PeriphOrM2MSrcDataSize = LL_DMA_PDATAALIGN_BYTE;
202         dma_init.MemoryOrM2MDstDataSize = LL_DMA_MDATAALIGN_BYTE;
203         dma_init.NbData = sizeof(usart_mem);
204         dma_init.Priority = LL_DMA_PRIORITY_MEDIUM;
205 #if defined(GSM_USART_DMA_RX_STREAM)
```

(continues on next page)

(continued from previous page)

```
        LL_DMA_Init(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM, &dma_init);
#else
        LL_DMA_Init(GSM_USART_DMA, GSM_USART_DMA_RX_CH, &dma_init);
#endif /* defined(GSM_USART_DMA_RX_STREAM) */

        /* Enable DMA interrupts */
#if defined(GSM_USART_DMA_RX_STREAM)
        LL_DMA_EnableIT_HT(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TC(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TE(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_FE(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_DME(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableIT_HT(GSM_USART_DMA, GSM_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TC(GSM_USART_DMA, GSM_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TE(GSM_USART_DMA, GSM_USART_DMA_RX_CH);
#endif /* defined(GSM_USART_DMA_RX_STREAM) */

        /* Enable DMA interrupts */
        NVIC_SetPriority(GSM_USART_DMA_RX_IRQ, NVIC_EncodePriority(NVIC_
↪GetPriorityGrouping(), 0x07, 0x00));
        NVIC_EnableIRQ(GSM_USART_DMA_RX_IRQ);

        old_pos = 0;
        is_running = 1;

        /* Start DMA and USART */
#if defined(GSM_USART_DMA_RX_STREAM)
        LL_DMA_EnableStream(GSM_USART_DMA, GSM_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableChannel(GSM_USART_DMA, GSM_USART_DMA_RX_CH);
#endif /* defined(GSM_USART_DMA_RX_STREAM) */
        LL_USART_Enable(GSM_USART);
    } else {
        osDelay(10);
        LL_USART_Disable(GSM_USART);
        usart_init.BaudRate = baudrate;
        LL_USART_Init(GSM_USART, &usart_init);
        LL_USART_Enable(GSM_USART);
    }

    /* Create mbox and start thread */
    if (usart_ll_mbox_id == NULL) {
        usart_ll_mbox_id = osMessageQueueNew(10, sizeof(void *), NULL);
    }
    if (usart_ll_thread_id == NULL) {
        const osThreadAttr_t attr = {
            .stack_size = 1024
        };
        usart_ll_thread_id = osThreadNew(usart_ll_thread, usart_ll_mbox_id, &attr);
    }
}

#if defined(GSM_RESET_PIN)
/**
 * \brief           Hardware reset callback
 */
```

(continues on next page)

(continued from previous page)

```
static uint8_t
reset_device(uint8_t state) {
    if (state) {                                /* Activate reset line */
        LL_GPIO_ResetOutputPin(GSM_RESET_PORT, GSM_RESET_PIN);
    } else {
        LL_GPIO_SetOutputPin(GSM_RESET_PORT, GSM_RESET_PIN);
    }
    return 1;
}
#endif /* defined(GSM_RESET_PIN) */

/**
 * \brief           Send data to GSM device
 * \param[in]       data: Pointer to data to send
 * \param[in]       len: Number of bytes to send
 * \return          Number of bytes sent
 */
static size_t
send_data(const void* data, size_t len) {
    const uint8_t* d = data;

    for (size_t i = 0; i < len; ++i, ++d) {
        LL_USART_TransmitData8(GSM_USART, *d);
        while (!LL_USART_IsActiveFlag_TXE(GSM_USART)) {}
    }
    return len;
}

/**
 * \brief           Callback function called from initialization process
 * \note            This function may be called multiple times if AT baudrate is changed from application
 * \param[in,out]   ll: Pointer to \ref gsm_ll_t structure to fill data for communication functions
 * \param[in]       baudrate: Baudrate to use on AT port
 * \return          Member of \ref gsmr_t enumeration
 */
gsmr_t
gsm_ll_init(gsm_ll_t* ll) {
#if !GSM_CFG_MEM_CUSTOM
    static uint8_t memory[GSM_MEM_SIZE];
    gsm_mem_region_t mem_regions[] = {
        { memory, sizeof(memory) }
    };

    if (!initialized) {
        gsm_mem_assignmemory(mem_regions, GSM_ARRAYSIZE(mem_regions));  /* Assign memory for allocations */
    }
#endif /* !GSM_CFG_MEM_CUSTOM */

    if (!initialized) {
        ll->send_fn = send_data;                /* Set callback function to send data */
#if defined(GSM_RESET_PIN)
        ll->reset_fn = reset_device;            /* Set callback for hardware reset */
#endif /* defined(GSM_RESET_PIN) */
```

(continues on next page)

(continued from previous page)

```
315     }
316
317     configure_uart(ll->uart.baudrate);          /* Initialize UART for communication */
318     initialized = 1;
319     return gsmOK;
320 }
321
322 /**
323  * \brief           Callback function to de-init low-level communication part
324  * \param[in,out]   ll: Pointer to \ref gsm_ll_t structure to fill data for communication functions
325  * \return          \ref gsmOK on success, member of \ref gsmr_t enumeration otherwise
326  */
327 gsmr_t
328 gsm_ll_deinit(gsm_ll_t* ll) {
329     if (usart_ll_mbox_id != NULL) {
330         osMessageQueueId_t tmp = usart_ll_mbox_id;
331         usart_ll_mbox_id = NULL;
332         osMessageQueueDelete(tmp);
333     }
334     if (usart_ll_thread_id != NULL) {
335         osThreadId_t tmp = usart_ll_thread_id;
336         usart_ll_thread_id = NULL;
337         osThreadTerminate(tmp);
338     }
339     initialized = 0;
340     GSM_UNUSED(ll);
341     return gsmOK;
342 }
343
344 /**
345  * \brief           UART global interrupt handler
346  */
347 void
348 GSM_USART_IRQHANDLER(void) {
349     LL_USART_ClearFlag_IDLE(GSM_USART);
350     LL_USART_ClearFlag_PE(GSM_USART);
351     LL_USART_ClearFlag_FE(GSM_USART);
352     LL_USART_ClearFlag_ORE(GSM_USART);
353     LL_USART_ClearFlag_NE(GSM_USART);
354
355     if (usart_ll_mbox_id != NULL) {
356         void* d = (void *)1;
357         osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
358     }
359 }
360
361 /**
362  * \brief           UART DMA stream/channel handler
363  */
364 void
365 GSM_USART_DMA_RX_IRQHANDLER(void) {
366     GSM_USART_DMA_RX_CLEAR_TC;
367     GSM_USART_DMA_RX_CLEAR_HT;
368
369     if (usart_ll_mbox_id != NULL) {
```

(continues on next page)

(continued from previous page)

```
        void* d = (void *)1;
        osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
    }
}

#endif /* !__DOXYGEN__ */
```

Example: System functions for WIN32

Listing 10: Actual header implementation of system functions for WIN32

```
/**
 * \file            gsm_sys_port.h
 * \brief           WIN32 based system file implementation
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef GSM_HDR_SYSTEM_PORT_H
#define GSM_HDR_SYSTEM_PORT_H

#include <stdint.h>
#include <stdlib.h>
#include "gsm_config.h"
#include "windows.h"

#ifdef __cplusplus
extern "C" {
```

(continues on next page)

(continued from previous page)

```
#endif /* __cplusplus */

#if GSM_CFG_OS && !__DOXYGEN__

typedef HANDLE                      gsm_sys_mutex_t;
typedef HANDLE                      gsm_sys_sem_t;
typedef HANDLE                      gsm_sys_mbox_t;
typedef HANDLE                      gsm_sys_thread_t;
typedef int                         gsm_sys_thread_prio_t;

#define GSM_SYS_MUTEX_NULL          ((HANDLE)0)
#define GSM_SYS_SEM_NULL            ((HANDLE)0)
#define GSM_SYS_MBOX_NULL           ((HANDLE)0)
#define GSM_SYS_TIMEOUT             (INFINITE)
#define GSM_SYS_THREAD_PRIO         (0)
#define GSM_SYS_THREAD_SS           (4096)

#endif /* GSM_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* GSM_HDR_SYSTEM_PORT_H */
```

Listing 11: Actual implementation of system functions for WIN32

```
/**
 * \file            gsm_sys_win32.c
 * \brief           System dependant functions for WIN32
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
```

(continues on next page)

(continued from previous page)

```
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include <string.h>
#include <stdlib.h>
#include "system/gsm_sys.h"

#if !__DOXYGEN__

/**
 * \brief           Custom message queue implementation for WIN32
 */
typedef struct {
    gsm_sys_sem_t sem_not_empty;                /*!< Semaphore indicates not empty */
    gsm_sys_sem_t sem_not_full;                 /*!< Semaphore indicates not full */
    gsm_sys_sem_t sem;                          /*!< Semaphore to lock access */
    size_t in, out, size;
    void* entries[1];
} win32_mbox_t;

static LARGE_INTEGER freq, sys_start_time;
static gsm_sys_mutex_t sys_mutex;               /* Mutex ID for main protection */

static uint8_t
mbox_is_full(win32_mbox_t* m) {
    size_t size = 0;
    if (m->in > m->out) {
        size = (m->in - m->out);
    } else if (m->out > m->in) {
        size = m->size - m->out + m->in;
    }
    return size == m->size - 1;
}

static uint8_t
mbox_is_empty(win32_mbox_t* m) {
    return m->in == m->out;
}

static uint32_t
osKernelSysTick(void) {
    LONGLONG ret;
    LARGE_INTEGER now;

    QueryPerformanceFrequency(&freq);           /* Get frequency */
    QueryPerformanceCounter(&now);              /* Get current time */
    ret = now.QuadPart - sys_start_time.QuadPart;
    return (uint32_t)(((ret) * 1000) / freq.QuadPart);
}

uint8_t
gsm_sys_init(void) {
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&sys_start_time);

    gsm_sys_mutex_create(&sys_mutex);
```

(continues on next page)

(continued from previous page)

```
    return 1;
}

uint32_t
gsm_sys_now(void) {
    return osKernelSysTick();
}

uint8_t
gsm_sys_protect(void) {
    gsm_sys_mutex_lock(&sys_mutex);
    return 1;
}

uint8_t
gsm_sys_unprotect(void) {
    gsm_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
gsm_sys_mutex_create(gsm_sys_mutex_t* p) {
    *p = CreateMutex(NULL, FALSE, NULL);
    return *p != NULL;
}

uint8_t
gsm_sys_mutex_delete(gsm_sys_mutex_t* p) {
    return CloseHandle(*p);
}

uint8_t
gsm_sys_mutex_lock(gsm_sys_mutex_t* p) {
    DWORD ret;
    ret = WaitForSingleObject(*p, INFINITE);
    if (ret != WAIT_OBJECT_0) {
        return 0;
    }
    return 1;
}

uint8_t
gsm_sys_mutex_unlock(gsm_sys_mutex_t* p) {
    return (uint8_t)ReleaseMutex(*p);
}

uint8_t
gsm_sys_mutex_isvalid(gsm_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
gsm_sys_mutex_invalid(gsm_sys_mutex_t* p) {
    *p = GSM_SYS_MUTEX_NULL;
    return 1;
}

```

(continues on next page)

(continued from previous page)

```
uint8_t
gsm_sys_sem_create(gsm_sys_sem_t* p, uint8_t cnt) {
    HANDLE h;
    h = CreateSemaphore(NULL, !!cnt, 1, NULL);
    *p = h;
    return *p != NULL;
}

uint8_t
gsm_sys_sem_delete(gsm_sys_sem_t* p) {
    return CloseHandle(*p);
}

uint32_t
gsm_sys_sem_wait(gsm_sys_sem_t* p, uint32_t timeout) {
    DWORD ret;
    uint32_t tick = osKernelSysTick();

    if (timeout == 0) {
        ret = WaitForSingleObject(*p, INFINITE);
        return 1;
    } else {
        ret = WaitForSingleObject(*p, timeout);
        if (ret == WAIT_OBJECT_0) {
            return 1;
        } else {
            return GSM_SYS_TIMEOUT;
        }
    }
}

uint8_t
gsm_sys_sem_release(gsm_sys_sem_t* p) {
    return ReleaseSemaphore(*p, 1, NULL);
}

uint8_t
gsm_sys_sem_isvalid(gsm_sys_sem_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
gsm_sys_sem_invalid(gsm_sys_sem_t* p) {
    *p = GSM_SYS_SEM_NULL;
    return 1;
}

uint8_t
gsm_sys_mbox_create(gsm_sys_mbox_t* b, size_t size) {
    win32_mbox_t* mbox;

    *b = NULL;

    mbox = malloc(sizeof(*mbox) + size * sizeof(void *));
    if (mbox != NULL) {
        memset(mbox, 0x00, sizeof(*mbox));
        mbox->size = size + 1;                  /* Set it to 1 more as cyclic buffer
↪has only one less than size */
```

(continues on next page)

(continued from previous page)

```
        gsm_sys_sem_create(&mbox->sem, 1);
        gsm_sys_sem_create(&mbox->sem_not_empty, 0);
        gsm_sys_sem_create(&mbox->sem_not_full, 0);
        *b = mbox;
    }
    return *b != NULL;
}

uint8_t
gsm_sys_mbox_delete(gsm_sys_mbox_t* b) {
    win32_mbox_t* mbox = *b;
    gsm_sys_sem_delete(&mbox->sem);
    gsm_sys_sem_delete(&mbox->sem_not_full);
    gsm_sys_sem_delete(&mbox->sem_not_empty);
    free(mbox);
    return 1;
}

uint32_t
gsm_sys_mbox_put(gsm_sys_mbox_t* b, void* m) {
    win32_mbox_t* mbox = *b;
    uint32_t time = osKernelSysTick();          /* Get start time */

    gsm_sys_sem_wait(&mbox->sem, 0);            /* Wait for access */

    /*
     * Since function is blocking until ready to write something to queue,
     * wait and release the semaphores to allow other threads
     * to process the queue before we can write new value.
     */
    while (mbox_is_full(mbox)) {
        gsm_sys_sem_release(&mbox->sem);        /* Release semaphore */
        gsm_sys_sem_wait(&mbox->sem_not_full, 0);   /* Wait for semaphore indicating
↪not full */
        gsm_sys_sem_wait(&mbox->sem, 0);        /* Wait availability again */
    }
    mbox->entries[mbox->in] = m;
    if (++mbox->in >= mbox->size) {
        mbox->in = 0;
    }
    gsm_sys_sem_release(&mbox->sem_not_empty);  /* Signal non-empty state */
    gsm_sys_sem_release(&mbox->sem);            /* Release access for other threads */
    return osKernelSysTick() - time;
}

uint32_t
gsm_sys_mbox_get(gsm_sys_mbox_t* b, void** m, uint32_t timeout) {
    win32_mbox_t* mbox = *b;
    uint32_t time;

    time = osKernelSysTick();

    /* Get exclusive access to message queue */
    if (gsm_sys_sem_wait(&mbox->sem, timeout) == GSM_SYS_TIMEOUT) {
        return GSM_SYS_TIMEOUT;
    }
    while (mbox_is_empty(mbox)) {
```

(continues on next page)

(continued from previous page)

```
        gsm_sys_sem_release(&mbox->sem);
        if (gsm_sys_sem_wait(&mbox->sem_not_empty, timeout) == GSM_SYS_TIMEOUT) {
            return GSM_SYS_TIMEOUT;
        }
        gsm_sys_sem_wait(&mbox->sem, timeout);
    }
    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
    }
    gsm_sys_sem_release(&mbox->sem_not_full);
    gsm_sys_sem_release(&mbox->sem);

    return osKernelSysTick() - time;
}

uint8_t
gsm_sys_mbox_putnow(gsm_sys_mbox_t* b, void* m) {
    win32_mbox_t* mbox = *b;

    gsm_sys_sem_wait(&mbox->sem, 0);
    if (mbox_is_full(mbox)) {
        gsm_sys_sem_release(&mbox->sem);
        return 0;
    }
    mbox->entries[mbox->in] = m;
    if (mbox->in == mbox->out) {
        gsm_sys_sem_release(&mbox->sem_not_empty);
    }
    if (++mbox->in >= mbox->size) {
        mbox->in = 0;
    }
    gsm_sys_sem_release(&mbox->sem);
    return 1;
}

uint8_t
gsm_sys_mbox_getnow(gsm_sys_mbox_t* b, void** m) {
    win32_mbox_t* mbox = *b;

    gsm_sys_sem_wait(&mbox->sem, 0);            /* Wait exclusive access */
    if (mbox->in == mbox->out) {
        gsm_sys_sem_release(&mbox->sem);        /* Release access */
        return 0;
    }

    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
    }
    gsm_sys_sem_release(&mbox->sem_not_full);   /* Queue not full anymore */
    gsm_sys_sem_release(&mbox->sem);            /* Release semaphore */
    return 1;
}

uint8_t
gsm_sys_mbox_isvalid(gsm_sys_mbox_t* b) {
```

(continues on next page)

(continued from previous page)

```
    return b != NULL && *b != NULL;             /* Return status if message box is valid */
}

uint8_t
gsm_sys_mbox_invalid(gsm_sys_mbox_t* b) {
    *b = GSM_SYS_MBOX_NULL;                     /* Invalidate message box */
    return 1;
}

uint8_t
gsm_sys_thread_create(gsm_sys_thread_t* t, const char* name, gsm_sys_thread_fn thread_func, void* const arg, size_t stack_size, gsm_sys_thread_prio_t prio) {
    HANDLE h;
    DWORD id;
    h = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)thread_func, arg, 0, &id);
    if (t != NULL) {
        *t = h;
    }
    return h != NULL;
}

uint8_t
gsm_sys_thread_terminate(gsm_sys_thread_t* t) {
    HANDLE h = NULL;

    if (t == NULL) {                            /* Shall we terminate ourself? */
        h = GetCurrentThread();                 /* Get current thread handle */
    } else {                                    /* We have known thread, find handle by looking at ID */
        h = *t;
    }
    TerminateThread(h, 0);
    return 1;
}

uint8_t
gsm_sys_thread_yield(void) {
    /* Not implemented */
    return 1;
}

#endif /* !__DOXYGEN__ */
```

Example: System functions for CMSIS-OS

Listing 12: Actual header implementation of system functions for CMSIS-OS based operating systems

```
/**
 * \file            gsm_sys_port.h
 * \brief           System dependent functions for CMSIS-OS based operating system
 */

/*
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef GSM_HDR_SYSTEM_PORT_H
#define GSM_HDR_SYSTEM_PORT_H

#include <stdint.h>
#include <stdlib.h>
#include "gsm_config.h"
#include "cmsis_os.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#if GSM_CFG_OS && !__DOXYGEN__

typedef osMutexId_t                 gsm_sys_mutex_t;
typedef osSemaphoreId_t             gsm_sys_sem_t;
typedef osMessageQueueId_t          gsm_sys_mbox_t;
typedef osThreadId_t                gsm_sys_thread_t;
typedef osPriority_t                gsm_sys_thread_prio_t;

#define GSM_SYS_MUTEX_NULL          ((gsm_sys_mutex_t)0)
#define GSM_SYS_SEM_NULL            ((gsm_sys_sem_t)0)
#define GSM_SYS_MBOX_NULL           ((gsm_sys_mbox_t)0)
#define GSM_SYS_TIMEOUT             ((uint32_t)osWaitForever)
#define GSM_SYS_THREAD_PRIO         (osPriorityNormal)
#define GSM_SYS_THREAD_SS           (512)

#endif /* GSM_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
```

(continues on next page)

(continued from previous page)

```
}
#endif /* __cplusplus */

#endif /* GSM_HDR_SYSTEM_PORT_H */
```

Listing 13: Actual implementation of system functions for CMSIS-OS based operating systems

```
/**
 * \file            gsm_sys_cmsis_os.c
 * \brief           System dependent functions for CMSIS-OS based operating system
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of GSM-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "system/gsm_sys.h"
#include "cmsis_os.h"

#if !__DOXYGEN__

static osMutexId_t sys_mutex;

uint8_t
gsm_sys_init(void) {
    gsm_sys_mutex_create(&sys_mutex);
    return 1;
}

uint32_t
gsm_sys_now(void) {
```

(continues on next page)

(continued from previous page)

```
    return osKernelSysTick();
}

uint8_t
gsm_sys_protect(void) {
    gsm_sys_mutex_lock(&sys_mutex);
    return 1;
}

uint8_t
gsm_sys_unprotect(void) {
    gsm_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
gsm_sys_mutex_create(gsm_sys_mutex_t* p) {
    const osMutexAttr_t attr = {
            .attr_bits = osMutexRecursive
    };
    *p = osMutexNew(&attr);
    return *p != NULL;
}

uint8_t
gsm_sys_mutex_delete(gsm_sys_mutex_t* p) {
    return osMutexDelete(*p) == osOK;
}

uint8_t
gsm_sys_mutex_lock(gsm_sys_mutex_t* p) {
    return osMutexAcquire(*p, osWaitForever) == osOK;
}

uint8_t
gsm_sys_mutex_unlock(gsm_sys_mutex_t* p) {
    return osMutexRelease(*p) == osOK;
}

uint8_t
gsm_sys_mutex_isvalid(gsm_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
gsm_sys_mutex_invalid(gsm_sys_mutex_t* p) {
    *p = GSM_SYS_MUTEX_NULL;
    return 1;
}

uint8_t
gsm_sys_sem_create(gsm_sys_sem_t* p, uint8_t cnt) {
    return (*p = osSemaphoreNew(1, cnt > 0 ? 1 : 0, NULL)) != NULL;
}

uint8_t
gsm_sys_sem_delete(gsm_sys_sem_t* p) {
```

(continues on next page)

(continued from previous page)

```
106     return osSemaphoreDelete(*p) == osOK;
107 }
108 
109 uint32_t
110 gsm_sys_sem_wait(gsm_sys_sem_t* p, uint32_t timeout) {
111     uint32_t tick = osKernelSysTick();
112     return (osSemaphoreAcquire(*p, timeout == 0 ? osWaitForever : timeout) == osOK) ?␣
    ↪(osKernelSysTick() - tick) : GSM_SYS_TIMEOUT;
113 }
114 
115 uint8_t
116 gsm_sys_sem_release(gsm_sys_sem_t* p) {
117     return osSemaphoreRelease(*p) == osOK;
118 }
119 
120 uint8_t
121 gsm_sys_sem_isvalid(gsm_sys_sem_t* p) {
122     return p != NULL && *p != NULL;
123 }
124 
125 uint8_t
126 gsm_sys_sem_invalid(gsm_sys_sem_t* p) {
127     *p = GSM_SYS_SEM_NULL;
128     return 1;
129 }
130 
131 uint8_t
132 gsm_sys_mbox_create(gsm_sys_mbox_t* b, size_t size) {
133     return (*b = osMessageQueueNew(size, sizeof(void *), NULL)) != NULL;
134 }
135 
136 uint8_t
137 gsm_sys_mbox_delete(gsm_sys_mbox_t* b) {
138     if (osMessageQueueGetCount(*b) > 0) {
139         return 0;
140     }
141     return osMessageQueueDelete(*b) == osOK;
142 }
143 
144 uint32_t
145 gsm_sys_mbox_put(gsm_sys_mbox_t* b, void* m) {
146     uint32_t tick = osKernelSysTick();
147     return osMessageQueuePut(*b, &m, 0, osWaitForever) == osOK ? (osKernelSysTick() -␣
    ↪tick) : GSM_SYS_TIMEOUT;
148 }
149 
150 uint32_t
151 gsm_sys_mbox_get(gsm_sys_mbox_t* b, void** m, uint32_t timeout) {
152     uint32_t tick = osKernelSysTick();
153     return osMessageQueueGet(*b, m, NULL, timeout == 0 ? osWaitForever : timeout) ==␣
    ↪osOK ? (osKernelSysTick() - tick) : GSM_SYS_TIMEOUT;
154 }
155 
156 uint8_t
157 gsm_sys_mbox_putnow(gsm_sys_mbox_t* b, void* m) {
158     return osMessageQueuePut(*b, &m, 0, 0) == osOK;
159 }
```

(continues on next page)

(continued from previous page)

```
160
161 uint8_t
162 gsm_sys_mbox_getnow(gsm_sys_mbox_t* b, void** m) {
163     return osMessageQueueGet(*b, m, NULL, 0) == osOK;
164 }
165
166 uint8_t
167 gsm_sys_mbox_isvalid(gsm_sys_mbox_t* b) {
168     return b != NULL && *b != NULL;
169 }
170
171 uint8_t
172 gsm_sys_mbox_invalid(gsm_sys_mbox_t* b) {
173     *b = GSM_SYS_MBOX_NULL;
174     return 1;
175 }
176
177 uint8_t
178 gsm_sys_thread_create(gsm_sys_thread_t* t, const char* name, gsm_sys_thread_fn thread_
    ↪func, void* const arg, size_t stack_size, gsm_sys_thread_prio_t prio) {
179     gsm_sys_thread_t id;
180     const osThreadAttr_t thread_attr = {
181             .name = (char *)name,
182             .priority = (osPriority)prio,
183             .stack_size = stack_size > 0 ? stack_size : GSM_SYS_THREAD_SS
184     };
185
186     id = osThreadNew(thread_func, arg, &thread_attr);
187     if (t != NULL) {
188         *t = id;
189     }
190     return id != NULL;
191 }
192
193 uint8_t
194 gsm_sys_thread_terminate(gsm_sys_thread_t* t) {
195     if (t != NULL) {
196         osThreadTerminate(*t);
197     } else {
198         osThreadExit();
199     }
200     return 1;
201 }
202
203 uint8_t
204 gsm_sys_thread_yield(void) {
205     osThreadYield();
206     return 1;
207 }
208
209 #endif /* !__DOXYGEN__ */
```

5.3 API reference

List of all the modules:

5.3.1 GSM AT Lib

Ring buffer

group **GSM_BUFF**
Generic ring buffer.

Defines

BUF_PREF(x)
Buffer function/typedef prefix string.

It is used to change function names in zero time to easily re-use same library between applications. Use `#define BUF_PREF(x) my_prefix_ ## x` to change all function names to (for example) `my_prefix_buff_init`

Note Modification of this macro must be done in header and source file aswell

Functions

uint8_t **gsm_buff_init**(*gsm_buff_t* *buff, size_t *size*)
Initialize buffer.

Return `1` on success, `0` otherwise

Parameters

- `[in] buff`: Pointer to buffer structure
- `[in] size`: Size of buffer in units of bytes

void **gsm_buff_free**(*gsm_buff_t* *buff)
Free dynamic allocation if used on memory.

Parameters

- `[in] buff`: Pointer to buffer structure

void **gsm_buff_reset**(*gsm_buff_t* *buff)
Resets buffer to default values. Buffer size is not modified.

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_write**(*gsm_buff_t* *buff, **const** void *data, size_t *btw*)
Write data to buffer Copies data from `data` array to buffer and marks buffer as full for maximum `count` number of bytes.

Return Number of bytes written to buffer. When returned value is less than `btw`, there was no enough memory available to copy full data array

Parameters

- `[in] buff`: Buffer handle
- `[in] data`: Pointer to data to write into buffer
- `[in] btw`: Number of bytes to write

size_t **gsm_buff_read**(*gsm_buff_t *buff*, void *data*, size_t *btr*)
Read data from buffer Copies data from buffer to `data` array and marks buffer as free for maximum `btr` number of bytes.

Return Number of bytes read and copied to data array

Parameters

- `[in] buff`: Buffer handle
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btr`: Number of bytes to read

size_t **gsm_buff_peek**(*gsm_buff_t *buff*, size_t *skip_count*, void *data*, size_t *btp*)
Read from buffer without changing read pointer (peek only)

Return Number of bytes peeked and written to output array

Parameters

- `[in] buff`: Buffer handle
- `[in] skip_count`: Number of bytes to skip before reading data
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btp`: Number of bytes to peek

size_t **gsm_buff_get_free**(*gsm_buff_t *buff*)
Get number of bytes in buffer available to write.

Return Number of free bytes in memory

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_get_full**(*gsm_buff_t *buff*)
Get number of bytes in buffer available to read.

Return Number of bytes ready to be read

Parameters

- `[in] buff`: Buffer handle

void *__gsm_buff_get_linear_block_read_address__(*gsm_buff_t *buff*)
Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_get_linear_block_read_length** (*gsm_buff_t *buff*)
Get length of linear block address before it overflows for read operation.

Return Linear buffer size in units of bytes for read operation

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_skip** (*gsm_buff_t *buff*, size_t *len*)
Skip (ignore; advance read pointer) buffer data Marks data as read in the buffer and increases free memory for up to `len` bytes.

Note Useful at the end of streaming transfer such as DMA

Return Number of bytes skipped

Parameters

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to skip and mark as read

void ***gsm_buff_get_linear_block_write_address** (*gsm_buff_t *buff*)
Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_get_linear_block_write_length** (*gsm_buff_t *buff*)
Get length of linear block address before it overflows for write operation.

Return Linear buffer size in units of bytes for write operation

Parameters

- `[in] buff`: Buffer handle

size_t **gsm_buff_advance** (*gsm_buff_t *buff*, size_t *len*)
Advance write pointer in the buffer. Similar to skip function but modifies write pointer instead of read.

Note Useful when hardware is writing to buffer and application needs to increase number of bytes written to buffer by hardware

Return Number of bytes advanced for write operation

Parameters

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to advance

struct gsm_buff_t
#include <*gsm_typedefs.h*> Buffer structure.

Public Members

uint8_t *__buff__
: Pointer to buffer data. Buffer is considered initialized when `buff != NULL`

size_t __size__
: Size of buffer data. Size of actual buffer is `1` byte less than this value

size_t __r__
: Next read pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

size_t __w__
: Next write pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

Connections

Connections are essential feature of WiFi device and middleware. It is developed with strong focus on its performance and since it may interact with huge amount of data, it tries to use zero-copy (when available) feature, to decrease processing time.

GSM AT Firmware by default supports up to `5` connections being active at the same time and supports:

- Up to `5` TCP connections active at the same time
- Up to `5` UDP connections active at the same time
- Up to `1` SSL connection active at a time

Note: Client or server connections are available. Same API function call are used to send/receive data or close connection.

Architecture of the connection API is using callback event functions. This allows maximal optimization in terms of responsiveness on different kind of events.

Example below shows *bare minimum* implementation to:

- Start a new connection to remote host
- Send *HTTP GET* request to remote host
- Process received data in event and print number of received bytes

Listing 14: Client connection minimum example

```
#include "client.h"
#include "gsm/gsm.h"
#include "gsm/gsm_network_api.h"

/* Host parameter */
#define CONN_HOST           "example.com"
#define CONN_PORT           80

static gsmr_t   conn_callback_func(gsm_evt_t* evt);

/**
 * \brief           Request data for connection
 */
static const
```

(continues on next page)

(continued from previous page)

```
uint8_t req_data[] = ""
"GET / HTTP/1.1\r\n"
"Host: " CONN_HOST "\r\n"
"Connection: close\r\n"
"\r\n";

/**
 * \brief           Start a new connection(s) as client
 */
void
client_connect(void) {
    gsmr_t res;

    /* Attach to GSM network */
    gsm_network_request_attach();

    /* Start a new connection as client in non-blocking mode */
    if ((res = gsm_conn_start(NULL, GSM_CONN_TYPE_TCP, "example.com", 80, NULL, conn_
↪callback_func, 0)) == gsmOK) {
        printf("Connection to " CONN_HOST " started...\r\n");
    } else {
        printf("Cannot start connection to " CONN_HOST "!\r\n");
    }
}

/**
 * \brief           Event callback function for connection-only
 * \param[in]       evt: Event information with data
 * \return          \ref gsmOK on success, member of \ref gsmr_t otherwise
 */
static gsmr_t
conn_callback_func(gsm_evt_t* evt) {
    gsm_conn_p conn;
    gsmr_t res;
    uint8_t conn_num;

    conn = gsm_conn_get_from_evt(evt);          /* Get connection handle from event */
    if (conn == NULL) {
        return gsmERR;
    }
    conn_num = gsm_conn_getnum(conn);           /* Get connection number for␣
↪identification */
    switch (gsm_evt_get_type(evt)) {
        case GSM_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = gsm_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /*␣
↪Start sending data in non-blocking mode */
            if (res == gsmOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection␣
↪manually...\r\n");
                gsm_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case GSM_EVT_CONN_CLOSE: {              /* Connection closed */
```

(continues on next page)

(continued from previous page)

```
            if (gsm_evt_conn_close_is_forced(evt)) {
                printf("Connection %d closed by client!\r\n", (int)conn_num);
            } else {
                printf("Connection %d closed by remote side!\r\n", (int)conn_num);
            }
            break;
        }
        case GSM_EVT_CONN_SEND: {               /* Data send event */
            gsmr_t res = gsm_evt_conn_send_get_result(evt);
            if (res == gsmOK) {
                printf("Data sent successfully on connection %d...waiting to receive␣
↪data from remote side...\r\n", (int)conn_num);
            } else {
                printf("Error while sending data on connection %d!\r\n", (int)conn_
↪num);
            }
            break;
        }
        case GSM_EVT_CONN_RECV: {               /* Data received from remote side */
            gsm_pbuf_p pbuf = gsm_evt_conn_recv_get_buff(evt);
            gsm_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)gsm_pbuf_
↪length(pbuf, 1), (int)conn_num);
            break;
        }
        case GSM_EVT_CONN_ERROR: {              /* Error connecting to server */
            const char* host = gsm_evt_conn_error_get_host(evt);
            gsm_port_t port = gsm_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default: break;
    }
    return gsmOK;
}
```

Sending data

Receiving data flow is always the same. Whenever new data packet arrives, corresponding event is called to notify application layer. When it comes to sending data, application may decide between 2 options (*this is valid only for non-UDP connections):

- Write data to temporary transmit buffer
- Execute *send command* for every API function call

Temporary transmit buffer

By calling `gsm_conn_write()` on active connection, temporary buffer is allocated and input data are copied to it. There is always up to `1` internal buffer active. When it is full (or if input data length is longer than maximal size), data are immediately send out and are not written to buffer.

GSM AT Firmware allows (current revision) to transmit up to `2048` bytes at a time with single command. When trying to send more than this, application would need to issue multiple *send commands* on *AT commands level*.

Write option is used mostly when application needs to write many different small chunks of data. Temporary buffer hence prevents many *send command* instructions as it is faster to send single command with big buffer, than many of them with smaller chunks of bytes.

Transmit packet manually

In some cases it is not possible to use temporary buffers, mostly because of memory constraints. Application can directly start *send data* instructions on *AT* level by using `gsm_conn_send()` or `gsm_conn_sendto()` functions.

group **GSM_CONN**
Connection API functions.

Typedefs

typedef struct gsm_conn ***gsm_conn_p**
Pointer to *gsm_conn_t* structure.

Enums

enum gsm_conn_type_t
List of possible connection types.

Values:

GSM_CONN_TYPE_TCP
Connection type is TCP

GSM_CONN_TYPE_UDP
Connection type is UDP

GSM_CONN_TYPE_SSL
Connection type is TCP over SSL

Functions

gsmr_t **gsm_conn_start**(*gsm_conn_p* **conn*, gsm_conn_type_t *type*, **const** char ***const** *host*, *gsm_port_t* *port*, void ***const** *arg*, *gsm_evt_fn* *conn_evt_fn*, **const** uint32_t *blocking*)
Start a new connection of specific type.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[out] conn`: Pointer to connection handle to set new connection reference in case of successful connection

- [in] type: Connection type. This parameter can be a value of *gsm_conn_type_t* enumeration
- [in] host: Connection host. In case of IP, write it as string, ex. "192.168.1.1"
- [in] port: Connection port
- [in] arg: Pointer to user argument passed to connection if successfully connected
- [in] conn_evt_fn: Callback function for this connection
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_conn_close** (*gsm_conn_p conn*, **const** uint32_t *blocking*)

Close specific or all connections.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- [in] conn: Connection handle to close. Set to NULL if you want to close all connections.
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_conn_send** (*gsm_conn_p conn*, **const** void **data*, size_t *btw*, size_t ***const** *bw*, **const** uint32_t *blocking*)

Send data on already active connection either as client or server.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- [in] conn: Connection handle to send data
- [in] data: Data to send
- [in] btw: Number of bytes to send
- [out] bw: Pointer to output variable to save number of sent data when successfully sent. Parameter value might not be accurate if you combine *gsm_conn_write* and *gsm_conn_send* functions
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_conn_sendto** (*gsm_conn_p conn*, **const** *gsm_ip_t* ***const** *ip*, *gsm_port_t port*, **const** void **data*, size_t *btw*, size_t **bw*, **const** uint32_t *blocking*)

Send data on active connection of type UDP to specific remote IP and port.

Note In case IP and port values are not set, it will behave as normal send function (suitable for TCP too)

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- [in] conn: Connection handle to send data
- [in] ip: Remote IP address for UDP connection
- [in] port: Remote port connection
- [in] data: Pointer to data to send
- [in] btw: Number of bytes to send
- [out] bw: Pointer to output variable to save number of sent data when successfully sent
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_conn_set_arg**(*gsm_conn_p conn*, void ***const** *arg*)
Set argument variable for connection.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

See *gsm_conn_get_arg*

Parameters

- `[in] conn`: Connection handle to set argument
- `[in] arg`: Pointer to argument

void ***gsm_conn_get_arg**(*gsm_conn_p conn*)
Get user defined connection argument.

Return User argument

See *gsm_conn_set_arg*

Parameters

- `[in] conn`: Connection handle to get argument

uint8_t **gsm_conn_is_client**(*gsm_conn_p conn*)
Check if connection type is client.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

uint8_t **gsm_conn_is_active**(*gsm_conn_p conn*)
Check if connection is active.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

uint8_t **gsm_conn_is_closed**(*gsm_conn_p conn*)
Check if connection is closed.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

int8_t **gsm_conn_getnum**(*gsm_conn_p conn*)
Get the number from connection.

Return Connection number in case of success or -1 on failure

Parameters

- `[in] conn`: Connection pointer

gsmr_t **gsm_get_conns_status**(**const** uint32_t *blocking*)
Gets connections status.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] blocking`: Status whether command should be blocking or not

gsm_conn_p **gsm_conn_get_from_evt**(*gsm_evt_t* **evt*)
Get connection from connection based event.

Return Connection pointer on success, `NULL` otherwise

Parameters

- `[in] evt`: Event which happened for connection

gsmr_t **gsm_conn_write**(*gsm_conn_p conn*, **const** void **data*, size_t *btw*, uint8_t *flush*, size_t ***const** *mem_available*)
Write data to connection buffer and if it is full, send it non-blocking way.

Note This function may only be called from core (connection callbacks)

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection to write
- `[in] data`: Data to copy to write buffer
- `[in] btw`: Number of bytes to write
- `[in] flush`: Flush flag. Set to `1` if you want to send data immediately after copying
- `[out] mem_available`: Available memory size available in current write buffer. When the buffer length is reached, current one is sent and a new one is automatically created. If function returns *gsmOK* and `*mem_available = 0`, there was a problem allocating a new buffer for next operation

gsmr_t **gsm_conn_recved**(*gsm_conn_p conn*, *gsm_pbuf_p pbuf*)
Notify connection about received data which means connection is ready to accept more data.

Once data reception is confirmed, stack will try to send more data to user.

Note Since this feature is not supported yet by AT commands, function is only prototype and should be used in connection callback when data are received

Note Function is not thread safe and may only be called from connection event function

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection handle
- `[in] pbuf`: Packet buffer received on connection

size_t **gsm_conn_get_total_recved_count**(*gsm_conn_p conn*)
Get total number of bytes ever received on connection and sent to user.

Return Count of received bytes on connection

Parameters

- `[in] conn`: Connection handle

uint8_t **gsm_conn_get_remote_ip** (*gsm_conn_p conn*, *gsm_ip_t *ip*)
Get connection remote IP address.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle
- `[out] ip`: Pointer to IP output handle

gsm_port_t **gsm_conn_get_remote_port** (*gsm_conn_p conn*)
Get connection remote port number.

Return Port number on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle

gsm_port_t **gsm_conn_get_local_port** (*gsm_conn_p conn*)
Get connection local port number.

Return Port number on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle

Debug support

Middleware has extended debugging capabilities. These consist of different debugging levels and types of debug messages, allowing to track and catch different types of warnings, severe problems or simply output messages program flow messages (trace messages).

Module is highly configurable using library configuration methods. Application must enable some options to decide what type of messages and for which modules it would like to output messages.

With default configuration, `printf` is used as output function. This behavior can be changed with `GSM_CF_DBG_OUT` configuration.

For successful debugging, application must:

- Enable global debugging by setting *GSM_CFG_DBG* to *GSM_DBG_ON*
- Configure which types of messages to output
- Configure debugging level, from all messages to severe only
- Enable specific modules to debug, by setting its configuration value to *GSM_DBG_ON*

Tip: Check *GSM Configuration* for all modules with debug implementation.

An example code with config and latter usage:

Listing 15: Debug configuration setup

```
/* Modifications of gsm_config.h file for configuration */

/* Enable global debug */
#define GSM_CFG_DBG                 GSM_DBG_ON

/*
 * Enable debug types.
 * Application may use bitwise OR | to use multiple types:
 *    GSM_DBG_TYPE_TRACE | GSM_DBG_TYPE_STATE
 */
#define GSM_CFG_DBG_TYPES_ON        GSM_DBG_TYPE_TRACE

/* Enable debug on custom module */
#define MY_DBG_MODULE               GSM_DBG_ON
```

Listing 16: Debug usage within middleware

```
#include "gsm/gsm_debug.h"

/*
 * Print debug message to the screen
 * Trace message will be printed as it is enabled in types
 * while state message will not be printed.
 */
GSM_DEBUGF(MY_DBG_MODULE | GSM_DBG_TYPE_TRACE, "This is trace message on my program\r\
↪n");
GSM_DEBUGF(MY_DBG_MODULE | GSM_DBG_TYPE_STATE, "This is state message on my program\r\
↪n");
```

group **GSM_DEBUG**

Debugging support module to track stack.

Debug levels

List of debug levels

GSM_DBG_LVL_ALL

Print all messages of all types

GSM_DBG_LVL_WARNING

Print warning and upper messages

GSM_DBG_LVL_DANGER

Print danger errors

GSM_DBG_LVL_SEVERE

Print severe problems affecting program flow

GSM_DBG_LVL_MASK

Mask for getting debug level

Debug types

List of possible debugging types

GSM_DBG_TYPE_TRACE
Debug trace messages for program flow

GSM_DBG_TYPE_STATE
Debug state messages (such as state machines)

GSM_DBG_TYPE_ALL
All debug types

Defines

GSM_DBG_ON
Indicates debug is enabled

GSM_DBG_OFF
Indicates debug is disabled

GSM_DEBUGF(c, fmt, ...)
Print message to the debug "window" if enabled.

Parameters

- `[in]` `c`: Condition if debug of specific type is enabled
- `[in]` `fmt`: Formatted string for debug
- `[in]` `...`: Variable parameters for formatted string

GSM_DEBUGW(c, cond, fmt, ...)
Print message to the debug "window" if enabled when specific condition is met.

Parameters

- `[in]` `c`: Condition if debug of specific type is enabled
- `[in]` `cond`: Debug only if this condition is true
- `[in]` `fmt`: Formatted string for debug
- `[in]` `...`: Variable parameters for formatted string

Device info

group **GSM_DEVICE_INFO**
Basic device information.

Functions

gsmr_t **gsm_device_get_manufacturer**(char *manuf, size_t len, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Get device manufacturer.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] manuf: Pointer to output string array to save manufacturer info
- [in] len: Length of string array including NULL termination
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_device_get_model**(char *model, size_t len, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Get device model name.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] model: Pointer to output string array to save model info
- [in] len: Length of string array including NULL termination
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_device_get_revision**(char *rev, size_t len, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Get device revision.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] rev: Pointer to output string array to save revision info
- [in] len: Length of string array including NULL termination
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_device_get_serial_number**(char *serial, size_t len, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Get device serial number.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] serial`: Pointer to output string array to save serial number info
- `[in] len`: Length of string array including `NULL` termination
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Event management

group **GSM_EVT**

Event helper functions.

Reset event

Event helper functions for GSM_EVT_RESET event

gsmr_t **gsm_evt_reset_get_result** (*gsm_evt_t* *cc)

Get reset sequence operation status.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event data

Restore event

Event helper functions for GSM_EVT_RESTORE event

gsmr_t **gsm_evt_restore_get_result** (*gsm_evt_t* *cc)

Get restore sequence operation status.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event data

Current network operator

Event helper functions for GSM_EVT_NETWORK_OPERATOR_CURRENT event

const *gsm_operator_curr_t* ***gsm_evt_network_operator_get_current** (*gsm_evt_t* *cc)

Get current operator data from event.

Return Current operator handle

Parameters

- `[in] cc`: Event data

Connection data received

Event helper functions for GSM_EVT_CONN_RECV event

gsm_pbuf_p **gsm_evt_conn_recv_get_buff** (*gsm_evt_t* *cc)
Get buffer from received data.

Return Buffer handle

Parameters

- `[in] cc`: Event handle

gsm_conn_p **gsm_evt_conn_recv_get_conn** (*gsm_evt_t* *cc)
Get connection handle for receive.

Return Connection handle

Parameters

- `[in] cc`: Event handle

Connection data send

Event helper functions for GSM_EVT_CONN_SEND event

gsm_conn_p **gsm_evt_conn_send_get_conn** (*gsm_evt_t* *cc)
Get connection handle for data sent event.

Return Connection handle

Parameters

- `[in] cc`: Event handle

size_t **gsm_evt_conn_send_get_length** (*gsm_evt_t* *cc)
Get number of bytes sent on connection.

Return Number of bytes sent

Parameters

- `[in] cc`: Event handle

gsmr_t **gsm_evt_conn_send_get_result** (*gsm_evt_t* *cc)
Check if connection send was successful.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

Connection active

Event helper functions for GSM_EVT_CONN_ACTIVE event

gsm_conn_p **gsm_evt_conn_active_get_conn** (*gsm_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in] cc`: Event handle

uint8_t **gsm_evt_conn_active_is_client** (*gsm_evt_t* *cc)
Check if new connection is client.

Return `1` if client, `0` otherwise

Parameters

- `[in] cc`: Event handle

Connection close event

Event helper functions for GSM_EVT_CONN_CLOSE event

gsm_conn_p **gsm_evt_conn_close_get_conn** (*gsm_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in] cc`: Event handle

uint8_t **gsm_evt_conn_close_is_client** (*gsm_evt_t* *cc)
Check if close connection was client.

Return `1` if client, `0` otherwise

Parameters

- `[in] cc`: Event handle

uint8_t **gsm_evt_conn_close_is_forced** (*gsm_evt_t* *cc)
Check if connection close even was forced by user.

Return `1` if forced, `0` otherwise

Parameters

- `[in] cc`: Event handle

gsmr_t **gsm_evt_conn_close_get_result** (*gsm_evt_t* *cc)
Get connection close event result.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

Connection poll

Event helper functions for GSM_EVT_CONN_POLL event

gsm_conn_p **gsm_evt_conn_poll_get_conn** (*gsm_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in] cc`: Event handle

Connection error

Event helper functions for GSM_EVT_CONN_ERROR event

gsmr_t **gsm_evt_conn_error_get_error** (*gsm_evt_t* *cc)
Get connection error type.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

gsm_conn_type_t **gsm_evt_conn_error_get_type** (*gsm_evt_t* *cc)
Get connection type.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

const char ***gsm_evt_conn_error_get_host** (*gsm_evt_t* *cc)
Get connection host.

Return Host name for connection

Parameters

- `[in] cc`: Event handle

gsm_port_t **gsm_evt_conn_error_get_port** (*gsm_evt_t* *cc)
Get connection port.

Return Host port number

Parameters

- `[in] cc`: Event handle

void ***gsm_evt_conn_error_get_arg** (*gsm_evt_t* *cc)
Get user argument.

Return User argument

Parameters

- `[in] cc`: Event handle

Signal strength

Event helper functions for GSM_EVT_CONN_RECV event

int16_t **gsm_evt_signal_strength_get_rssi**(*gsm_evt_t* *cc)
Get RSSi from CSQ command.

Return RSSI value in units of dBm

Parameters

- `[in] cc`: Event data

SMS received

Event helper functions for GSM_EVT_SMS_RECV event

size_t **gsm_evt_sms_recv_get_pos**(*gsm_evt_t* *cc)
Get SMS position in memory which has been saved on receive.

Return SMS position in memory

Parameters

- `[in] cc`: Event handle

gsm_mem_t **gsm_evt_sms_recv_get_mem**(*gsm_evt_t* *cc)
Get SMS memory used to save SMS on receive.

Return SMS memory location

Parameters

- `[in] cc`: Event handle

SMS content read

Event helper functions for GSM_EVT_SMS_READ event

gsm_sms_entry_t ***gsm_evt_sms_read_get_entry**(*gsm_evt_t* *cc)
Get SMS entry after successful read.

Return SMS entry

Parameters

- `[in] cc`: Event handle

gsmr_t **gsm_evt_sms_read_get_result**(*gsm_evt_t* *cc)
Get SMS read operation result.

Return SMS entry

Parameters

- `[in] cc`: Event handle

SMS send

Event helper functions for GSM_EVT_SMS_SEND event

gsmr_t **gsm_evt_sms_send_get_result** (*gsm_evt_t* *cc)
Get SMS send result status.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

size_t **gsm_evt_sms_send_get_pos** (*gsm_evt_t* *cc)
Get SMS send position in memory.

Note Use only if SMS sent successfully

Return Position in memory

Parameters

- `[in] cc`: Event handle

SMS delete

Event helper functions for GSM_EVT_SMS_DELETE event

gsmr_t **gsm_evt_sms_delete_get_result** (*gsm_evt_t* *cc)
Get SMS delete result status.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event handle

size_t **gsm_evt_sms_delete_get_pos** (*gsm_evt_t* *cc)
Get SMS delete memory position.

Return Deleted position in memory

Parameters

- `[in] cc`: Event handle

gsm_mem_t **gsm_evt_sms_delete_get_mem** (*gsm_evt_t* *cc)
Get SMS delete memory.

Return SMS memory for delete operation

Parameters

- `[in] cc`: Event handle

Call status changed

Event helper functions for GSM_EVT_CALL_CHANGED event

const gsm_call_t ***gsm_evt_call_changed_get_call** (*gsm_evt_t* *cc*)
Get call information from changed event.

Return Position in memory

Parameters

- `[in] cc`: Event handle

Operator scan

Event helper functions for GSM_EVT_OPERATOR_SCAN event

gsmr_t **gsm_evt_operator_scan_get_result** (*gsm_evt_t* *cc*)
Get operator scan operation status.

Return Member of *gsmr_t* enumeration

Parameters

- `[in] cc`: Event data

gsm_operator_t ***gsm_evt_operator_scan_get_entries** (*gsm_evt_t* *cc*)
Get operator entries from scan.

Return Pointer to array of operator entries

Parameters

- `[in] cc`: Event data

size_t **gsm_evt_operator_scan_get_length** (*gsm_evt_t* *cc*)
Get length of operators scanned.

Return Number of operators scanned

Parameters

- `[in] cc`: Event data

Typedefs

typedef gsmr_t (***gsm_evt_fn**) (**struct** gsm_evt *evt)
Event function prototype.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] evt`: Callback event data

Enums

enum gsm_evt_type_t
List of possible callback types received to user.

Values:

GSM_EVT_INIT_FINISH
Initialization has been finished at this point

GSM_EVT_RESET
Device reset operation finished

GSM_EVT_RESTORE
Device restore operation finished

GSM_EVT_CMD_TIMEOUT
Timeout on command. When application receives this event, it may reset system as there was (maybe) a problem in device

GSM_EVT_DEVICE_PRESENT
Notification when device present status changes

GSM_EVT_DEVICE_IDENTIFIED
Device identified event

GSM_EVT_SIGNAL_STRENGTH
Signal strength event

GSM_EVT_SIM_STATE_CHANGED
SIM card state changed

GSM_EVT_OPERATOR_SCAN
Operator scan finished event

GSM_EVT_NETWORK_OPERATOR_CURRENT
Current operator event

GSM_EVT_NETWORK_REG_CHANGED
Network registration changed. Available even when *GSM_CFG_NETWORK* is disabled

GSM_EVT_NETWORK_ATTACHED
Attached to network, PDP context active and ready for TCP/IP application

GSM_EVT_NETWORK_DETACHED
Detached from network, PDP context not active anymore

GSM_EVT_CONN_RECV
Connection data received

GSM_EVT_CONN_SEND
Connection data send

GSM_EVT_CONN_ACTIVE
Connection just became active

GSM_EVT_CONN_ERROR
Client connection start was not successful

GSM_EVT_CONN_CLOSE
Connection close event. Check status if successful

GSM_EVT_CONN_POLL
Poll for connection if there are any changes

GSM_EVT_SMS_ENABLE
SMS enable event

GSM_EVT_SMS_READY
SMS ready event

GSM_EVT_SMS_SEND
SMS send event

GSM_EVT_SMS_RECV
SMS received

GSM_EVT_SMS_READ
SMS read

GSM_EVT_SMS_DELETE
SMS delete

GSM_EVT_SMS_LIST
SMS list

GSM_EVT_CALL_ENABLE
Call enable event

GSM_EVT_CALL_READY
Call ready event

GSM_EVT_CALL_CHANGED
Call info changed, `+CLCK` statement received

GSM_EVT_CALL_RING
Call is ringing event

GSM_EVT_CALL_BUSY
Call is busy

GSM_EVT_CALL_NO_CARRIER
No carrier to make a call

GSM_EVT_PB_ENABLE
Phonebook enable event

GSM_EVT_PB_LIST
Phonebook list event

GSM_EVT_PB_SEARCH
Phonebook search event

Functions

gsmr_t **gsm_evt_register**(*gsm_evt_fn fn*)
Register callback function for global (non-connection based) events.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to call on specific event

gsmr_t **gsm_evt_unregister**(*gsm_evt_fn fn*)
Unregister callback function for global (non-connection based) events.

Note Function must be first registered using *gsm_evt_register*

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to remove from event list

gsm_evt_type_t **gsm_evt_get_type**(*gsm_evt_t* *cc)
Get event type.

Return Event type. Member of *gsm_evt_type_t* enumeration

Parameters

- `[in] cc`: Event handle

struct gsm_evt_t
#include <gsm_typedefs.h> Global callback structure to pass as parameter to callback function.

Public Members

gsm_evt_type_t **type**
Callback type

gsmr_t **res**
Reset operation result

Restore operation result

Scan operation result

Send data result

Result of close event. Set to *gsmOK* on success.

SMS send result information

SMS read result information

Operation success

Result on command

Enable status

struct *gsm_evt_t*::**[anonymous]**::**[anonymous] reset**
Reset sequence finish. Use with GSM_EVT_RESET event

struct *gsm_evt_t*::**[anonymous]**::**[anonymous] restore**
Restore sequence finish. Use with GSM_EVT_RESTORE event

gsm_sim_state_t **state**
SIM state

struct *gsm_evt_t*::**[anonymous]**::**[anonymous] cpin**
CPIN event

const *gsm_operator_curr_t* ***operator_current**
Current operator info

struct *gsm_evt_t*::**[anonymous]**::**[anonymous] operator_current**
Current operator event. Use with GSM_EVT_NETWORK_OPERATOR_CURRENT event

gsm_operator_t ***ops**
Pointer to operators

size_t **opf**
Number of operators found

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **operator_scan**
Operator scan event. Use with GSM_EVT_OPERATOR_SCAN event

int16_t **rssi**
Strength in units of dBm

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **rssi**
Signal strength event. Use with GSM_EVT_SIGNAL_STRENGTH event

gsm_conn_p **conn**
Connection where data were received

Connection where data were sent

Pointer to connection

Set connection pointer

gsm_pbuf_p **buff**
Pointer to received data

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **conn_data_recv**
Network data received. Use with GSM_EVT_CONN_RECV event

size_t **sent**
Number of bytes sent on connection

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **conn_data_send**
Data successfully sent. Use with GSM_EVT_CONN_SEND event

const char ***host**
Host to use for connection

gsm_port_t **port**
Remote port used for connection

gsm_conn_type_t **type**
Connection type

void ***arg**
Connection argument used on connection

gsmr_t **err**
Error value

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **conn_error**
Client connection start error. Use with GSM_EVT_CONN_ERROR event

uint8_t **client**
Set to `1` if connection is/was client mode

uint8_t **forced**
Set to `1` if connection action was forced (when active: 1 = CLIENT, 0 = SERVER: when closed, 1 = CMD, 0 = REMOTE)

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **conn_active_close**
Process active and closed statuses at the same time. Use with GSM_EVT_CONN_ACTIVE or GSM_EVT_CONN_CLOSE events

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **conn_poll**
Polling active connection to check for timeouts. Use with GSM_EVT_CONN_POLL event

gsmr_t **status**
Enable status

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_enable**
SMS enable event. Use with *GSM_EVT_SMS_ENABLE* event

size_t **pos**
Position in memory

Received position in memory for sent SMS

Deleted position in memory for sent SMS

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_send**
SMS sent info. Use with GSM_EVT_SMS_SEND event

gsm_mem_t **mem**
Memory of received message

Memory of deleted message

Memory used for scan

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_recv**
SMS received info. Use with GSM_EVT_SMS_RECV event

gsm_sms_entry_t ***entry**
SMS entry

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_read**
SMS read. Use with GSM_EVT_SMS_READ event

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_delete**
SMS delete. Use with GSM_EVT_SMS_DELETE event

gsm_sms_entry_t ***entries**
Pointer to entries

size_t **size**
Number of valid entries

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **sms_list**
SMS list. Use with *GSM_EVT_SMS_LIST* event

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **call_enable**
Call enable event. Use with *GSM_EVT_CALL_ENABLE* event

const gsm_call_t ***call**
Call information

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **call_changed**
Call changed info. Use with GSM_EVT_CALL_CHANGED event

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **pb_enable**
Phonebook enable event. Use with *GSM_EVT_PB_ENABLE* event

gsm_pb_entry_t ***entries**
Pointer to entries

struct *gsm_evt_t*::**[anonymous]**::**[anonymous]** **pb_list**
Phonebok list. Use with *GSM_EVT_PB_LIST* event

const char ***search**
Search string

struct *gsm_evt_t*::**[anonymous]**::**[anonymous] pb_search**
Phonebok search list. Use with *GSM_EVT_PB_SEARCH* event

union *gsm_evt_t*::**[anonymous] evt**
Callback event union

File Transfer Protocol

group **GSM_FTP**
File Transfer Protocol (FTP) manager.

Currently it is under development

HTTP

group **GSM_HTTP**
Hyper Text Transfer Protocol (HTTP) manager.

Currently it is under development

Input module

Input module is used to input received data from *GSM* device to *GSM-AT-Lib* middleware part. 2 processing options are possible:

- Indirect processing with *gsm_input()* (default mode)
- Direct processing with *gsm_input_process()*

Tip: Direct or indirect processing mode is select by setting *GSM_CFG_INPUT_USE_PROCESS* configuration value.

Indirect processing

With indirect processing mode, every received character from *GSM* physical device is written to intermediate buffer between low-level driver and *processing* thread.

Function *gsm_input()* is used to write data to buffer, which is later processed by *processing* thread.

Indirect processing mode allows embedded systems to write received data to buffer from interrupt context (outside threads). As a drawback, its performance is decreased as it involves copying every receive character to intermediate buffer, and may also introduce RAM memory footprint increase.

Direct processing

Direct processing is targeting more advanced host controllers, like STM32 or WIN32 implementation use. It is developed with DMA support in mind, allowing low-level drivers to skip intermediate data buffer and process input bytes directly.

Note: When using this mode, function *gsm_input_process()* must be used and it may only be called from thread context. Processing of input bytes is done in low-level input thread, started by application.

Tip: Check *Porting guide* for implementation examples.

---

group **GSM_INPUT**

Input function for received data.

Functions

gsmr_t **gsm_input**(**const** void **data*, size_t *len*)

Write data to input buffer.

Note *GSM_CFG_INPUT_USE_PROCESS* must be disabled to use this function

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] data`: Pointer to data to write
- `[in] len`: Number of data elements in units of bytes

gsmr_t **gsm_input_process**(**const** void **data*, size_t *len*)

Process input data directly without writing it to input buffer.

Note This function may only be used when in OS mode, where single thread is dedicated for input read of AT receive

Note *GSM_CFG_INPUT_USE_PROCESS* must be enabled to use this function

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] data`: Pointer to received data to be processed
- `[in] len`: Length of data to process in units of bytes

Memory manager

group **GSM_MEM**
Dynamic memory manager.

Functions

uint8_t **gsm_mem_assignmemory**(**const** *gsm_mem_region_t* *regions, size_t *size*)
Assign memory region(s) for allocation functions.

Note You can allocate multiple regions by assigning start address and region size in units of bytes

Return `1` on success, `0` otherwise

Note Function is not available when *GSM_CFG_MEM_CUSTOM* is `1`

Parameters

- `[in] regions`: Pointer to list of regions to use for allocations
- `[in] len`: Number of regions to use

void *gsm_mem_malloc(size_t *size*)
Allocate memory of specific size.

Return Memory address on success, `NULL` otherwise

Note Function is not available when *GSM_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] size`: Number of bytes to allocate

void *gsm_mem_realloc(void *ptr, size_t *size*)
Reallocate memory to specific size.

Note After new memory is allocated, content of old one is copied to new memory

Return Memory address on success, `NULL` otherwise

Note Function is not available when *GSM_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] ptr`: Pointer to current allocated memory to resize, returned using *gsm_mem_malloc*, *gsm_mem_calloc* or *gsm_mem_realloc* functions
- `[in] size`: Number of bytes to allocate on new memory

void *gsm_mem_calloc(size_t *num*, size_t *size*)
Allocate memory of specific size and set memory to zero.

Return Memory address on success, `NULL` otherwise

Note Function is not available when *GSM_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] num`: Number of elements to allocate
- `[in] size`: Size of each element

void **gsm_mem_free**(void *_ptr_)
Free memory.

Note Function is not available when *GSM_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] ptr`: Pointer to memory previously returned using *gsm_mem_malloc*, *gsm_mem_calloc* or *gsm_mem_realloc* functions

uint8_t **gsm_mem_free_s**(void **_ptr_)
Free memory in safe way by invalidating pointer after freeing.

Return `1` on success, `0` otherwise

Parameters

- `[in] ptr`: Pointer to pointer to allocated memory to free

struct gsm_mem_region_t
#include <*gsm_mem.h*> Single memory region descriptor.

Public Members

void ***start_addr**
Start address of region

size_t **size**
Size in units of bytes of region

Network

group **GSM_NETWORK**
Network manager.

Enums

enum gsm_network_reg_status_t
Network Registration status.

Values:

GSM_NETWORK_REG_STATUS_SIM_ERR = 0x00
SIM card error

GSM_NETWORK_REG_STATUS_CONNECTED = 0x01
Device is connected to network

GSM_NETWORK_REG_STATUS_SEARCHING = 0x02
Network search is in progress

GSM_NETWORK_REG_STATUS_DENIED = 0x03
Registration denied

GSM_NETWORK_REG_STATUS_CONNECTED_ROAMING = 0x05
Device is connected and is roaming

GSM_NETWORK_REG_STATUS_CONNECTED_SMS_ONLY = 0x06
Device is connected to home network in SMS-only mode

GSM_NETWORK_REG_STATUS_CONNECTED_ROAMING_SMS_ONLY = 0x07
Device is roaming in SMS-only mode

Functions

gsmr_t **gsm_network_rssi**(int16_t *rssi, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)
Read RSSI signal from network operator.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- `[out] rssi`: RSSI output variable. When set to `0`, RSSI is not valid
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsm_network_reg_status_t **gsm_network_get_reg_status**(void)
Get network registration status.

Return Member of gsm_network_reg_status_t enumeration

gsmr_t **gsm_network_attach**(**const** char *apn, **const** char *user, **const** char *pass, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)
Attach to network and active PDP context.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- `[in] apn`: APN name
- `[in] user`: User name to attach. Set to `NULL` if not used
- `[in] pass`: User password to attach. Set to `NULL` if not used
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_network_detach**(**const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)
Detach from network.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

uint8_t **gsm_network_is_attached**(void)
Check if device is attached to network and PDP context is active.

Return `1` on success, `0` otherwise

gsmr_t **gsm_network_copy_ip**(*gsm_ip_t* **ip*)
Copy IP address from internal value to user variable.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[out] ip`: Pointer to output IP variable

gsmr_t **gsm_network_check_status**(**const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Check network PDP status.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Network API

Network API provides functions for multi-thread application network management. It allows multiple threads to request to join to network (internet access).

Network API module controls when network connection shall be active or can be closed.

group **GSM_NETWORK_API**
Network API functions for multi-thread operations.

Functions

gsmr_t **gsm_network_set_credentials**(**const** char **apn*, **const** char **user*, **const** char **pass*)
Set system network credentials before asking for attach.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] apn`: APN domain. Set to `NULL` if not used
- `[in] user`: APN username. Set to `NULL` if not used

- `[in] pass`: APN password. Set to `NULL` if not used

gsmr_t **gsm_network_request_attach**(void)
Request manager to attach to network.

Note This function is blocking and cannot be called from event functions

Return *gsmOK* on success (when attached), member of *gsmr_t* otherwise

gsmr_t **gsm_network_request_detach**(void)
Request manager to detach from network.

If other threads use network, manager will not disconnect from network otherwise it will disable network access

Note This function is blocking and cannot be called from event functions

Return *gsmOK* on success (when attached), member of *gsmr_t* otherwise

Network operator

group **GSM_OPERATOR**
network operator API

Enums

enum gsm_operator_status_t
Operator status value.

Values:

GSM_OPERATOR_STATUS_UNKNOWN = 0x00
Unknown operator

GSM_OPERATOR_STATUS_AVAILABLE
Operator is available

GSM_OPERATOR_STATUS_CURRENT
Operator is currently active

GSM_OPERATOR_STATUS_FORBIDDEN
Operator is forbidden

enum gsm_operator_mode_t
Operator selection mode.

Values:

GSM_OPERATOR_MODE_AUTO = 0x00
Operator automatic mode

GSM_OPERATOR_MODE_MANUAL = 0x01
Operator manual mode

GSM_OPERATOR_MODE_DEREGISTER = 0x02
Operator deregistered from network

GSM_OPERATOR_MODE_MANUAL_AUTO = 0x04
Operator manual mode first. If fails, auto mode enabled

enum gsm_operator_format_t
Operator data format.

Values:

GSM_OPERATOR_FORMAT_LONG_NAME = 0x00
COPS command returned long name

GSM_OPERATOR_FORMAT_SHORT_NAME
COPS command returned short name

GSM_OPERATOR_FORMAT_NUMBER
COPS command returned number

GSM_OPERATOR_FORMAT_INVALID
Unknown format

Functions

gsmr_t **gsm_operator_get**(*gsm_operator_curr_t* *curr, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Get current operator.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[out] curr`: Pointer to output variable to save info about current operator
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_operator_set**(gsm_operator_mode_t *mode*, gsm_operator_format_t *format*, **const** char *name, uint32_t *num*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Set current operator.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] mode`: Operator mode. This parameter can be a value of *gsm_operator_mode_t* enumeration
- `[in] format`: Operator data format. This parameter can be a value of *gsm_operator_format_t* enumeration
- `[in] name`: Operator name. This parameter must be valid according to `format` parameter
- `[in] num`: Operator number. This parameter must be valid according to `format` parameter
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_operator_scan**(*gsm_operator_t* *ops, size_t opsl, size_t *opf, **const** *gsm_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Scan for available operators.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] ops`: Pointer to array to write found operators
- `[in] opsl`: Length of input array in units of elements
- `[out] opf`: Pointer to ouput variable to save number of operators found
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

struct gsm_operator_t

#include <gsm_typedefs.h> Operator details for scan.

Public Members

gsm_operator_status_t **stat**

Operator status

char **long_name**[20]

Operator long name

char **short_name**[20]

Operator short name

uint32_t **num**

Operator numeric value

struct gsm_operator_curr_t

#include <gsm_typedefs.h> Current operator info.

Public Members

gsm_operator_mode_t **mode**

Operator mode

gsm_operator_format_t **format**

Data format

char **long_name**[20]

Long name format

char **short_name**[20]

Short name format

uint32_t **num**

Number format

union *gsm_operator_curr_t*::**[anonymous]** **data**

Operator data union

Packet buffer

Packet buffer (or *pbuf*) is buffer manager to handle received data from any connection. It is optimized to construct big buffer of smaller chunks of fragmented data as received bytes are not always coming as single packet.

Pbuf block diagram

Fig. 3: Block diagram of pbuf chain

Image above shows structure of *pbuf* chain. Each *pbuf* consists of:

- Pointer to next *pbuf*, or `NULL` when it is last in chain
- Length of current packet length
- Length of current packet and all next in chain
  - If *pbuf* is last in chain, total length is the same as current packet length
- Reference counter, indicating how many pointers point to current *pbuf*
- Actual buffer data

Top image shows `3` pbufs connected to single chain. There are `2` custom pointer variables to point at different *pbuf* structures. Second *pbuf* has reference counter set to `2`, as `2` variables point to it:

- *next* of *pbuf 1* is the first one
- *User variable 2* is the second one

Table 1: Block structure

Block number	Next pbuf	Block size	Total size in chain	Reference counter
Block 1	*Block 2*	`150`	`550`	`1`
Block 2	*Block 3*	`130`	`400`	`2`
Block 3	`NULL`	`270`	`270`	`1`

Reference counter

Reference counter holds number of references (or variables) pointing to this block. It is used to properly handle memory free operation, especially when *pbuf* is used by lib core and application layer.

Note: If there would be no reference counter information and application would free memory while another part of library still uses its reference, application would invoke *undefined behavior* and system could crash instantly.

When application tries to free pbuf chain as on first image, it would normally call *`gsm_pbuf_free()`* function. That would:

- Decrease reference counter by `1`
- If reference counter `==` `0`, it removes it from chain list and frees packet buffer memory
- If reference counter `!=` `0` after decrease, it stops free procedure
- Go to next pbuf in chain and repeat steps

As per first example, result of freeing from *user variable 1* would look similar to image and table below. First block (blue) had reference counter set to `1` prior freeing operation. It was successfully removed as *user variable 1* was the only one pointing to it, while second (green) block had reference counter set to `2`, preventing free operation.

Fig. 4: Block diagram of pbuf chain after free from *user variable 1*

Table 2: Block diagram of pbuf chain after free from *user variable 1*

Block number	Next pbuf	Block size	Total size in chain	Reference counter
Block 2	*Block 3*	`130`	`400`	`1`
Block 3	`NULL`	`270`	`270`	`1`

Note: *Block 1* has been successfully freed, but since *block 2* had reference counter set to `2` before, it was only decreased by `1` to a new value `1` and free operation stopped instead. *User variable 2* is still using *pbuf* starting at *block 2* and must manually call *gsm_pbuf_free()* to free it.

Concatenating vs chaining

This section will explain difference between *concat* and *chain* operations. Both operations link `2` pbufs together in a chain of pbufs, difference is that *chain* operation increases *reference counter* to linked pbuf, while *concat* keeps *reference counter* at its current status.

Fig. 5: Different pbufs, each pointed to by its own variable

Concat operation

Concat operation shall be used when `2` pbufs are linked together and reference to *second* is no longer used.

Fig. 6: Structure after pbuf concat

After concating *2 pbufs* together, reference counter of second is still set to `1`, however we can see that `2` pointers point to *second pbuf*.

Note: After application calls *gsm_pbuf_cat()*, it must not use pointer which points to *second pbuf*. This would invoke *undefined behavior* if one pointer tries to free memory while second still points to it.

An example code showing proper usage of concat operation:

Listing 17: Packet buffer concat example

```
1 gsm_pbuf_p a, b;
2
3 /* Create 2 pbufs of different sizes */
4 a = gsm_pbuf_new(10);
5 b = gsm_pbuf_new(20);
```

(continues on next page)

(continued from previous page)

```

/* Link them together with concat operation */
/* Reference on b will stay as is, won't be increased */
gsm_pbuf_cat(a, b);

/*
 * Operating with b variable has from now on undefined behavior,
 * application shall stop using variable b to access pbuf.
 *
 * The best way would be to set b reference to NULL
 */
b = NULL;

/*
 * When application doesn't need pbufs anymore,
 * free a and it will also free b
 */
gsm_pbuf_free(a);
```

Chain operation

Chain operation shall be used when `2` pbufs are linked together and reference to *second* is still required.

Fig. 7: Structure after pbuf chain

After chainin *2 pbufs* together, reference counter of second is increased by `1`, which allows application to reference second *pbuf* separatelly.

Note: After application calls *gsm_pbuf_chain()*, it also has to manually free its reference using *gsm_pbuf_free()* function. Forgetting to free pbuf invokes memory leak

---

An example code showing proper usage of chain operation:

Listing 18: Packet buffer chain example

```
gsm_pbuf_p a, b;

/* Create 2 pbufs of different sizes */
a = gsm_pbuf_new(10);
b = gsm_pbuf_new(20);

/* Chain both pbufs together */
/* This will increase reference on b as 2 variables now point to it */
gsm_pbuf_chain(a, b);

/*
 * When application does not need a anymore, it may free it

 * This will free only pbuf a, as pbuf b has now 2 references:
 *  - one from pbuf a
 *  - one from variable b
 */
```

(continues on next page)

(continued from previous page)

```

/* If application calls this, it will free only first pbuf */
/* As there is link to b pbuf somewhere */
gsm_pbuf_free(a);

/* Reset a variable, not used anymore */
a = NULL;

/*
 * At this point, b is still valid memory block,
 * but when application doesn't need it anymore,
 * it should free it, otherwise memory leak appears
 */
gsm_pbuf_free(b);

/* Reset b variable */
b = NULL;
```

Extract pbuf data

Each *pbuf* holds some amount of data bytes. When multiple *pbufs* are linked together (either chained or concated), blocks of raw data are not linked to contiguous memory block. It is necessary to process block by block manually.

An example code showing proper reading of any *pbuf*:

Listing 19: Packet buffer data extraction

```
const void* data;
size_t pos, len;
gsm_pbuf_p a, b, c;

const char str_a[] = "This is one long";
const char str_a[] = "string. We want to save";
const char str_a[] = "chain of pbufs to file";

/* Create pbufs to hold these strings */
a = gsm_pbuf_new(strlen(str_a));
b = gsm_pbuf_new(strlen(str_b));
c = gsm_pbuf_new(strlen(str_c));

/* Write data to pbufs */
gsm_pbuf_take(a, str_a, strlen(str_a), 0);
gsm_pbuf_take(b, str_b, strlen(str_b), 0);
gsm_pbuf_take(c, str_c, strlen(str_c), 0);

/* Connect pbufs together */
gsm_pbuf_chain(a, b);
gsm_pbuf_chain(a, c);

/*
 * pbuf a now contains chain of b and c together
 * and at this point application wants to print (or save) data from chained pbuf
 *
 * Process pbuf by pbuf with code below
 */
```

(continues on next page)

(continued from previous page)

```

/*
 * Get linear address of current pbuf at specific offset
 * Function will return pointer to memory address at specific position
 * and `len` will hold length of data block
 */
pos = 0;
while ((data = gsm_pbuf_get_linear_addr(a, pos, &len)) != NULL) {
    /* Custom process function... */
    /* Process data with data pointer and block length */
    process_data(data, len);
    printf("Str: %.*s", len, data);

    /* Increase offset position for next block */
    pos += len;
}

/* Call free only on a pbuf. Since it is chained, b and c will be freed too */
gsm_pbuf_free(a);
```

group **GSM_PBUF**

Packet buffer manager.

Typedefs

typedef struct gsm_pbuf ***gsm_pbuf_p**

Pointer to *gsm_pbuf_t* structure.

Functions

gsm_pbuf_p **gsm_pbuf_new**(size_t *len*)

Allocate packet buffer for network data of specific size.

Return Pointer to allocated memory, `NULL` otherwise

Parameters

- `[in] len`: Length of payload memory to allocate

size_t **gsm_pbuf_free**(*gsm_pbuf_p pbuf*)

Free previously allocated packet buffer.

Return Number of freed pbufs from head

Parameters

- `[in] pbuf`: Packet buffer to free

void ***gsm_pbuf_data**(**const** *gsm_pbuf_p pbuf*)

Get data pointer from packet buffer.

Return Pointer to data buffer on success, `NULL` otherwise

Parameters

- `[in] pbuf`: Packet buffer

size_t **gsm_pbuf_length**(**const** *gsm_pbuf_p pbuf*, uint8_t *tot*)
Get length of packet buffer.

Return Length of data in units of bytes

Parameters

- `[in] pbuf`: Packet buffer to get length for
- `[in] tot`: Set to `1` to return total packet chain length or `0` to get only first packet length

gsmr_t **gsm_pbuf_take**(*gsm_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *offset*)
Copy user data to chain of pbufs.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] pbuf`: First pbuf in chain to start copying to
- `[in] data`: Input data to copy to pbuf memory
- `[in] len`: Length of input data to copy
- `[in] offset`: Start offset in pbuf where to start copying

size_t **gsm_pbuf_copy**(*gsm_pbuf_p pbuf*, void **data*, size_t *len*, size_t *offset*)
Copy memory from pbuf to user linear memory.

Return Number of bytes copied

Parameters

- `[in] pbuf`: Pbuf to copy from
- `[out] data`: User linear memory to copy to
- `[in] len`: Length of data in units of bytes
- `[in] offset`: Possible start offset in pbuf

gsmr_t **gsm_pbuf_cat**(*gsm_pbuf_p head*, **const** *gsm_pbuf_p tail*)
Concatenate `2` packet buffers together to one big packet.

Note After `tail` pbuf has been added to `head` pbuf chain, it must not be referenced by user anymore as it is now completelly controlled by `head` pbuf. In simple words, when user calls this function, it should not call *gsm_pbuf_free* function anymore, as it might make memory undefined for `head` pbuf.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

See *gsm_pbuf_chain*

Parameters

- `[in] head`: Head packet buffer to append new pbuf to
- `[in] tail`: Tail packet buffer to append to head pbuf

gsmr_t **gsm_pbuf_chain**(*gsm_pbuf_p head*, *gsm_pbuf_p tail*)
Chain 2 pbufs together. Similar to *gsm_pbuf_cat* but now new reference is done from head pbuf to tail pbuf.

Note After this function call, user must call *gsm_pbuf_free* to remove its reference to tail pbuf and allow control to head pbuf: gsm_pbuf_free(tail)

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

See *gsm_pbuf_cat*

Parameters

- `[in] head`: Head packet buffer to append new pbuf to
- `[in] tail`: Tail packet buffer to append to head pbuf

gsmr_t **gsm_pbuf_ref**(*gsm_pbuf_p pbuf*)
Increment reference count on pbuf.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] pbuf`: pbuf to increase reference

uint8_t **gsm_pbuf_get_at**(**const** *gsm_pbuf_p pbuf*, size_t *pos*, uint8_t **el*)
Get value from pbuf at specific position.

Return `1` on success, `0` otherwise

Parameters

- `[in] pbuf`: Pbuf used to get data from
- `[in] pos`: Position at which to get element
- `[out] el`: Output variable to save element value at desired position

size_t **gsm_pbuf_memcmp**(**const** *gsm_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *offset*)
Compare pbuf memory with memory from data.

Note Compare is done on entire pbuf chain

Return `0` if equal, `GSM_SIZET_MAX` if memory/offset too big or anything between if not equal

See *gsm_pbuf_strcmp*

Parameters

- `[in] pbuf`: Pbuf used to compare with data memory
- `[in] data`: Actual data to compare with
- `[in] len`: Length of input data in units of bytes
- `[in] offset`: Start offset to use when comparing data

size_t **gsm_pbuf_strcmp**(**const** *gsm_pbuf_p pbuf*, **const** char **str*, size_t *offset*)
Compare pbuf memory with input string.

Note Compare is done on entire pbuf chain

Return `0` if equal, `GSM_SIZET_MAX` if memory/offset too big or anything between if not equal

See *gsm_pbuf_memcmp*

Parameters

- [in] pbuf: Pbuf used to compare with data memory
- [in] str: String to be compared with pbuf
- [in] offset: Start memory offset in pbuf

size_t **gsm_pbuf_memfind**(**const** *gsm_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *off*)
Find desired needle in a haystack.

Return GSM_SIZET_MAX if no match or position where in pbuf we have a match

See *gsm_pbuf_strfind*

Parameters

- [in] pbuf: Pbuf used as haystack
- [in] needle: Data memory used as needle
- [in] len: Length of needle memory
- [in] off: Starting offset in pbuf memory

size_t **gsm_pbuf_strfind**(**const** *gsm_pbuf_p pbuf*, **const** char **str*, size_t *off*)
Find desired needle (str) in a haystack (pbuf)

Return GSM_SIZET_MAX if no match or position where in pbuf we have a match

See *gsm_pbuf_memfind*

Parameters

- [in] pbuf: Pbuf used as haystack
- [in] str: String to search for in pbuf
- [in] off: Starting offset in pbuf memory

uint8_t **gsm_pbuf_advance**(*gsm_pbuf_p pbuf*, int *len*)
Advance pbuf payload pointer by number of len bytes. It can only advance single pbuf in a chain.

Note When other pbufs are referencing current one, they are not adjusted in length and total length

Return 1 on success, 0 otherwise

Parameters

- [in] pbuf: Pbuf to advance
- [in] len: Number of bytes to advance. when negative is used, buffer size is increased only if it was decreased before

gsm_pbuf_p **gsm_pbuf_skip**(*gsm_pbuf_p pbuf*, size_t *offset*, size_t **new_offset*)
Skip a list of pbufs for desired offset.

Note Reference is not changed after return and user must not free the memory of new pbuf directly

Return New pbuf on success, NULL otherwise

Parameters

- [in] pbuf: Start of pbuf chain
- [in] offset: Offset in units of bytes to skip

- `[out] new_offset`: Pointer to output variable to save new offset in returned pbuf

void *__gsm_pbuf_get_linear_addr__(__const__ *gsm_pbuf_p* *pbuf*, size_t *offset*, size_t **new_len*)
Get linear offset address for pbuf from specific offset.

Note Since pbuf memory can be fragmented in chain, you may need to call function multiple times to get memory for entire pbuf chain

Return Pointer to memory on success, `NULL` otherwise

Parameters

- `[in] pbuf`: Pbuf to get linear address
- `[in] offset`: Start offset from where to start
- `[out] new_len`: Length of memory returned by function

void __gsm_pbuf_set_ip__(*gsm_pbuf_p* *pbuf*, __const__ *gsm_ip_t* **ip*, *gsm_port_t* *port*)
Set IP address and port number for received data.

Parameters

- `[in] pbuf`: Packet buffer
- `[in] ip`: IP to assing to packet buffer
- `[in] port`: Port number to assign to packet buffer

__struct gsm_pbuf_t__
#include <gsm_private.h> Packet buffer structure.

Public Members

__struct__ gsm_pbuf ***next**
Next pbuf in chain list

size_t **tot_len**
Total length of pbuf chain

size_t **len**
Length of payload

size_t **ref**
Number of references to this structure

uint8_t ***payload**
Pointer to payload memory

gsm_ip_t **ip**
Remote address for received IPD data

gsm_port_t **port**
Remote port for received IPD data

Phonebook

group **GSM_PHONEBOOK**
Phonebook manager.

Functions

gsmr_t **gsm_pb_enable**(**const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Enable phonebook functionality.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_disable**(**const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Disable phonebook functionality.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_add**(gsm_mem_t *mem*, **const** char **name*, **const** char **num*, gsm_number_type_t *type*, **const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Add new phonebook entry to desired memory.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] name`: Entry name
- `[in] num`: Entry phone number
- `[in] type`: Entry phone number type
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_edit**(gsm_mem_t *mem*, size_t *pos*, **const** char **name*, **const** char **num*, gsm_number_type_t *type*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Edit or overwrite phonebook entry at desired memory and position.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] pos`: Entry position in memory to edit
- `[in] name`: New entry name
- `[in] num`: New entry phone number
- `[in] type`: New entry phone number type
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_delete**(gsm_mem_t *mem*, size_t *pos*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Delete phonebook entry at desired memory and position.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] pos`: Entry position in memory to delete
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_read**(gsm_mem_t *mem*, size_t *pos*, gsm_pb_entry_t **entry*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Read single phonebook entry.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] pos`: Entry position in memory to read
- `[out] entry`: Pointer to entry variable to save data
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_list**(gsm_mem_t *mem*, size_t *start_index*, gsm_pb_entry_t **entries*, size_t *etr*, size_t **er*, **const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

List entires from specific memory.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] start_index`: Start position in memory to list
- `[out] entries`: Pointer to array to save entries
- `[in] etr`: Number of entries to read
- `[out] er`: Pointer to output variable to save entries listed
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_pb_search**(gsm_mem_t *mem*, **const** char **search*, gsm_pb_entry_t **entries*, size_t *etr*, size_t **er*, **const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Search for entires with specific name from specific memory.

Note Search works by entry name only. Phone number search is not available

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to use to save entry. Use *GSM_MEM_CURRENT* to use current memory
- `[in] search`: String to search for
- `[out] entries`: Pointer to array to save entries
- `[in] etr`: Number of entries to read
- `[out] er`: Pointer to output variable to save entries found
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Ping support

group **GSM_PING**
PING manager.

SIM card

group **GSM_SIM**
SIM card manager.

Enums

enum gsm_sim_state_t
SIM state.

Values:

GSM_SIM_STATE_NOT_INSERTED
SIM is not inserted in socket

GSM_SIM_STATE_READY
SIM is ready for operations

GSM_SIM_STATE_NOT_READY
SIM is not ready for any operation

GSM_SIM_STATE_PIN
SIM is waiting for SIM to be given

GSM_SIM_STATE_PUK
SIM is waiting for PUT to be given

GSM_SIM_STATE_PH_PIN

GSM_SIM_STATE_PH_PUK

Functions

gsm_sim_state_t **gsm_sim_get_current_state**(void)
Get current cached SIM state from stack.

Note Information is always valid, starting after successful device reset using *gsm_reset* function call

Return Member of *gsm_sim_state_t* enumeration

gsmr_t **gsm_sim_pin_enter**(**const** char *pin, **const** *gsm_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t blocking)
Enter pin code to unlock SIM.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] pin`: Pin code in string format
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used

- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_sim_pin_add**(**const** char *pin, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Add pin number to open SIM card.

Note Use this function only if your SIM card doesn't have PIN code. If you wish to change current pin, use gsm_sim_pin_change instead

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] pin: Current SIM pin code
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_sim_pin_remove**(**const** char *pin, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Remove pin code from SIM.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] pin: Current pin code
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_sim_pin_change**(**const** char *pin, **const** char *new_pin, **const** gsm_api_cmd_evt_fn evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Change current pin code.

Return gsmOK on success, member of gsmr_t enumeration otherwise

Parameters

- [in] pin: Current pin code
- [in] new_pin: New pin code
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

gsmr_t **gsm_sim_puk_enter**(**const** char *puk, **const** char *new_pin, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Enter PUK code and new PIN to unlock SIM card.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] puk`: PUK code associated with SIM card
- `[in] new_pin`: New PIN code to use
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

SMS

group **GSM_SMS**

SMS manager.

Enums

enum gsm_sms_status_t

SMS status in current memory.

Values:

GSM_SMS_STATUS_ALL
Process all SMS, used for mass delete or SMS list

GSM_SMS_STATUS_READ
SMS status is read

GSM_SMS_STATUS_UNREAD
SMS status is unread

GSM_SMS_STATUS_SENT
SMS status is sent

GSM_SMS_STATUS_UNSENT
SMS status is unsent

GSM_SMS_STATUS_INBOX
SMS status, used only for mass delete operation

Functions

gsmr_t **gsm_sms_enable**(**const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)
Enable SMS functionality.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_disable**(**const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)
Disable SMS functionality.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_send**(**const** char **num*, **const** char **text*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)
Send SMS text to phone number.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] num`: String number
- `[in] text`: Text to send. Maximal `160` characters
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_read**(gsm_mem_t *mem*, size_t *pos*, *gsm_sms_entry_t* **entry*, uint8_t *update*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)
Read SMS entry at specific memory and position.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory used to read message from
- `[in] pos`: Position number in memory to read
- `[out] entry`: Pointer to SMS entry structure to fill data to

- `[in] update`: Flag indicates update. Set to `1` to change `UNREAD` messages to `READ` or `0` to leave as is
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_delete**(gsm_mem_t *mem*, size_t *pos*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void *__const__ *evt_arg*, **const** uint32_t *blocking*)

Delete SMS entry at specific memory and position.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory used to read message from
- `[in] pos`: Position number in memory to read
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_delete_all**(gsm_sms_status_t *status*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void *__const__ *evt_arg*, **const** uint32_t *blocking*)

Delete all SMS entries with specific status.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] status`: SMS status. This parameter can be one of all possible types in *gsm_sms_status_t* enumeration
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_list**(gsm_mem_t *mem*, gsm_sms_status_t *stat*, *gsm_sms_entry_t* **entries*, size_t *etr*, size_t **er*, uint8_t *update*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void *__const__ *evt_arg*, **const** uint32_t *blocking*)

List SMS from SMS memory.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem`: Memory to read entries from. Use *GSM_MEM_CURRENT* to read from current memory
- `[in] stat`: SMS status to read, either `read`, `unread`, `sent`, `unsent` or `all`
- `[out] entries`: Pointer to array to save SMS entries
- `[in] etr`: Number of entries to read

- `[out] er`: Pointer to output variable to save number of entries in array
- `[in] update`: Flag indicates update. Set to `1` to change `UNREAD` messages to `READ` or `0` to leave as is
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_sms_set_preferred_storage**(gsm_mem_t *mem1*, gsm_mem_t *mem2*, gsm_mem_t *mem3*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Set preferred storage for SMS.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] mem1`: Preferred memory for read/delete SMS operations. Use *GSM_MEM_CURRENT* to keep it as is
- `[in] mem2`: Preferred memory for sent/write SMS operations. Use *GSM_MEM_CURRENT* to keep it as is
- `[in] mem3`: Preferred memory for received SMS entries. Use *GSM_MEM_CURRENT* to keep it as is
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

struct gsm_sms_mem_t

#include <gsm_private.h> SMS memory information.

Public Members

uint32_t **mem_available**
Bit field of available memories

gsm_mem_t **current**
Current memory choice

size_t **total**
Size of memory in units of entries

size_t **used**
Number of used entries

struct gsm_sms_t

#include <gsm_private.h> SMS structure.

Public Members

uint8_t **ready**
Flag indicating feature ready by device

uint8_t **enabled**
Flag indicating feature enabled

gsm_sms_mem_t **mem**[3]
3 memory info for operation,receive,sent storage

struct gsm_pb_mem_t
#include <*gsm_private.h*> SMS memory information.

Public Members

uint32_t **mem_available**
Bit field of available memories

gsm_mem_t **current**
Current memory choice

size_t **total**
Size of memory in units of entries

size_t **used**
Number of used entries

struct gsm_sms_entry_t
#include <*gsm_typedefs.h*> SMS entry structure.

Public Members

gsm_mem_t **mem**
Memory storage

size_t **pos**
Memory position

gsm_datetime_t **datetime**
Date and time

gsm_sms_status_t **status**
Message status

char **number**[26]
Phone number

char **name**[20]
Name in phonebook if exists

char **data**[161]
Data memory

size_t **length**
Length of SMS data

Timeout manager

Timeout manager allows application to call specific function at desired time. It is used in middleware (and can be used by application too) to poll active connections.

Note: Callback function is called from *processing* thread. It is not allowed to call any blocking API function from it.

When application registers timeout, it needs to set timeout, callback function and optional user argument. When timeout elapses, GSM middleware will call timeout callback.

This feature can be considered as single-shot software timer.

group **GSM_TIMEOUT**
Timeout manager.

Typedefs

typedef void (***gsm_timeout_fn**) (void *arg)
Timeout callback function prototype.

Parameters

- `[in] arg`: Custom user argument

Functions

gsmr_t **gsm_timeout_add** (uint32_t *time*, *gsm_timeout_fn* *fn*, void **arg*)
Add new timeout to processing list.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] time`: Time in units of milliseconds for timeout execution
- `[in] fn`: Callback function to call when timeout expires
- `[in] arg`: Pointer to user specific argument to call when timeout callback function is executed

gsmr_t **gsm_timeout_remove** (*gsm_timeout_fn* *fn*)
Remove callback from timeout list.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to identify timeout to remove

struct gsm_timeout_t
#include <gsm_typedefs.h> Timeout structure.

Public Members

struct gsm_timeout ***next**
Pointer to next timeout entry

uint32_t **time**
Time difference from previous entry

void ***arg**
Argument to pass to callback function

gsm_timeout_fn **fn**
Callback function for timeout

Structures and enumerations

group **GSM_TYPEDEFS**
List of core structures and enumerations.

Typedefs

typedef uint16_t **gsm_port_t**
Port variable.

typedef void (***gsm_api_cmd_evt_fn**)(gsmr_t res, void *arg)
Function declaration for API function command event callback function.

Parameters

- `[in] res`: Operation result, member of *gsmr_t* enumeration
- `[in] arg`: Custom user argument

Enums

enum gsm_cmd_t
List of possible messages.

Values:

GSM_CMD_IDLE = 0
IDLE mode

GSM_CMD_RESET
Reset device

GSM_CMD_RESET_DEVICE_FIRST_CMD
Reset device first driver specific command

GSM_CMD_ATE0
Disable ECHO mode on AT commands

GSM_CMD_ATE1
Enable ECHO mode on AT commands

GSM_CMD_GSLP
Set GSM to sleep mode

GSM_CMD_RESTORE
Restore GSM internal settings to default values

GSM_CMD_UART

GSM_CMD_CGACT_SET_0

GSM_CMD_CGACT_SET_1

GSM_CMD_CGATT_SET_0

GSM_CMD_CGATT_SET_1

GSM_CMD_NETWORK_ATTACH
Attach to a network

GSM_CMD_NETWORK_DETACH
Detach from network

GSM_CMD_CIPMUX_SET

GSM_CMD_CIPRXGET_SET

GSM_CMD_CSTT_SET

GSM_CMD_CALL_ENABLE

GSM_CMD_A
Re-issues the Last Command Given

GSM_CMD_ATA
Answer an Incoming Call

GSM_CMD_ATD
Mobile Originated Call to Dial A Number

GSM_CMD_ATD_N
Originate Call to Phone Number in Current Memory: ATD<n>

GSM_CMD_ATD_STR
Originate Call to Phone Number in Memory Which Corresponds to Field “str”: ATD>str

GSM_CMD_ATDL
Redial Last Telephone Number Used

GSM_CMD_ATE
Set Command Echo Mode

GSM_CMD_ATH
Disconnect Existing

GSM_CMD_ATI
Display Product Identification Information

GSM_CMD_ATL
Set Monitor speaker

GSM_CMD_ATM
Set Monitor Speaker Mode

GSM_CMD_PPP
Switch from Data Mode or PPP Online Mode to Command Mode, “+++” originally

GSM_CMD_ATO
Switch from Command Mode to Data Mode

GSM_CMD_ATP
Select Pulse Dialing

GSM_CMD_ATQ
Set Result Code Presentation Mode

GSM_CMD_ATS0
Set Number of Rings before Automatically Answering the Call

GSM_CMD_ATS3
Set Command Line Termination Character

GSM_CMD_ATS4
Set Response Formatting Character

GSM_CMD_ATS5
Set Command Line Editing Character

GSM_CMD_ATS6
Pause Before Blind

GSM_CMD_ATS7
Set Number of Seconds to Wait for Connection Completion

GSM_CMD_ATS8
Set Number of Seconds to Wait for Comma Dial Modifier Encountered in Dial String of D Command

GSM_CMD_ATS10
Set Disconnect Delay after Indicating the Absence of Data Carrier

GSM_CMD_ATT
Select Tone Dialing

GSM_CMD_ATV
TA Response Format

GSM_CMD_ATX
Set CONNECT Result Code Format and Monitor Call Progress

GSM_CMD_ATZ
Reset Default Configuration

GSM_CMD_AT_C
Set DCD Function Mode, AT&C

GSM_CMD_AT_D
Set DTR Function, AT&D

GSM_CMD_AT_F
Factory Defined Configuration, AT&F

GSM_CMD_AT_V
Display Current Configuration, AT&V

GSM_CMD_AT_W
Store Active Profile, AT&W

GSM_CMD_GCAP
Request Complete TA Capabilities List

GSM_CMD_GMI
Request Manufacturer Identification

GSM_CMD_GMM
Request TA Model Identification

GSM_CMD_GMR
Request TA Revision Identification of Software Release

GSM_CMD_GOI
Request Global Object Identification

GSM_CMD_GSN
Request TA Serial Number Identification (IMEI)

GSM_CMD_ICF
Set TE-TA Control Character Framing

GSM_CMD_IFC
Set TE-TA Local Data Flow Control

GSM_CMD_IPR
Set TE-TA Fixed Local Rate

GSM_CMD_HVOIC
Disconnect Voice Call Only

GSM_CMD_COPS_SET
Set operator

GSM_CMD_COPS_GET
Get current operator

GSM_CMD_COPS_GET_OPT
Get a list of available operators

GSM_CMD_CPAS
Phone Activity Status

GSM_CMD_CGMI_GET
Request Manufacturer Identification

GSM_CMD_CGMM_GET
Request Model Identification

GSM_CMD_CGMR_GET
Request TA Revision Identification of Software Release

GSM_CMD_CGSN_GET
Request Product Serial Number Identification (Identical with +GSN)

GSM_CMD_CLCC_SET
List Current Calls of ME

GSM_CMD_CLCK
Facility Lock

GSM_CMD_CACM
Accumulated Call Meter (ACM) Reset or Query

GSM_CMD_CAMM
Accumulated Call Meter Maximum (ACM max) Set or Query

GSM_CMD_CAOC
Advice of Charge

GSM_CMD_CBST
Select Bearer Service Type

GSM_CMD_CCFC
Call Forwarding Number and Conditions Control

GSM_CMD_CCWA
Call Waiting Control

GSM_CMD_CEER
Extended Error Report

GSM_CMD_CSCS
Select TE Character Set

GSM_CMD_CSTA
Select Type of Address

GSM_CMD_CHLD
Call Hold and Multiparty

GSM_CMD_CIMI
Request International Mobile Subscriber Identity

GSM_CMD_CLIP
Calling Line Identification Presentation

GSM_CMD_CLIR
Calling Line Identification Restriction

GSM_CMD_CMEE_SET
Report Mobile Equipment Error

GSM_CMD_COLP
Connected Line Identification Presentation

GSM_CMD_PHONEBOOK_ENABLE

GSM_CMD_CPBF
Find Phonebook Entries

GSM_CMD_CPBR
Read Current Phonebook Entries

GSM_CMD_CPBS_SET
Select Phonebook Memory Storage

GSM_CMD_CPBS_GET
Get current Phonebook Memory Storage

GSM_CMD_CPBS_GET_OPT
Get available Phonebook Memory Storages

GSM_CMD_CPBW_SET
Write Phonebook Entry

GSM_CMD_CPBW_GET_OPT
Get options for write Phonebook Entry

GSM_CMD_SIM_PROCESS_BASIC_CMDS
Command setup, executed when SIM is in READY state

GSM_CMD_CPIN_SET
Enter PIN

GSM_CMD_CPIN_GET
Read current SIM status

GSM_CMD_CPIN_ADD
Add new PIN to SIM if pin is not set

GSM_CMD_CPIN_CHANGE
Change already active SIM

GSM_CMD_CPIN_REMOVE
Remove current PIN

GMM_CMD_CPUK_SET
Enter PUK and set new PIN

GSM_CMD_CSQ_GET
Signal Quality Report

GSM_CMD_CFUN_SET
Set Phone Functionality

GSM_CMD_CFUN_GET
Get Phone Functionality

GSM_CMD_CREG_SET
Network Registration set output

GSM_CMD_CREG_GET
Get current network registration status

GSM_CMD_CBC
Battery Charge

GSM_CMD_CNUM
Subscriber Number

GSM_CMD_CPWD
Change Password

GSM_CMD_CR
Service Reporting Control

GSM_CMD_CRC
Set Cellular Result Codes for Incoming Call Indication

GSM_CMD_CRLP
Select Radio Link Protocol Parameters

GSM_CMD_CRSM
Restricted SIM Access

GSM_CMD_VTD
Tone Duration

GSM_CMD_VTS
DTMF and Tone Generation

GSM_CMD_CMUX
Multiplexer Control

GSM_CMD_CPOL
Preferred Operator List

GSM_CMD_COPN
Read Operator Names

GSM_CMD_CCLK
Clock

GSM_CMD_CSIM
Generic SIM Access

GSM_CMD_CALM
Alert Sound Mode

GSM_CMD_CALS
Alert Sound Select

GSM_CMD_CRSL
Ringer Sound Level

GSM_CMD_CLVL
Loud Speaker Volume Level

GSM_CMD_CMUT
Mute Control

GSM_CMD_CPUC
Price Per Unit and Currency Table

GSM_CMD_CCWE
Call Meter Maximum Event

GSM_CMD_CUSD_SET
Unstructured Supplementary Service Data, Set command

GSM_CMD_CUSD_GET
Unstructured Supplementary Service Data, Get command

GSM_CMD_CUSD
Unstructured Supplementary Service Data, Execute command

GSM_CMD_CSSN
Supplementary Services Notification

GSM_CMD_CIPMUX
Start Up Multi-IP Connection

GSM_CMD_CIPSTART
Start Up TCP or UDP Connection

GSM_CMD_CIPSEND
Send Data Through TCP or UDP Connection

GSM_CMD_CIPQSEND
Select Data Transmitting Mode

GSM_CMD_CIPACK
Query Previous Connection Data Transmitting State

GSM_CMD_CIPCLOSE
Close TCP or UDP Connection

GSM_CMD_CIPSHUT
Deactivate GPRS PDP Context

GSM_CMD_CLPORT
Set Local Port

GSM_CMD_CSTT
Start Task and Set APN, username, password

GSM_CMD_CIICR
Bring Up Wireless Connection with GPRS or CSD

GSM_CMD_CIFSR
Get Local IP Address

GSM_CMD_CIPSTATUS
Query Current Connection Status

GSM_CMD_CDNSCFG
Configure Domain Name Server

GSM_CMD_CDNSGIP
Query the IP Address of Given Domain Name

GSM_CMD_CIPHEAD
Add an IP Head at the Beginning of a Package Received

GSM_CMD_CIPATS
Set Auto Sending Timer

GSM_CMD_CIPSPRT
Set Prompt of greater than sign When Module Sends Data

GSM_CMD_CIPSERVER
Configure Module as Server

GSM_CMD_CIPCSGP
Set CSD or GPRS for Connection Mode

GSM_CMD_CIPSRIP
Show Remote IP Address and Port When Received Data

GSM_CMD_CIPDPDP
Set Whether to Check State of GPRS Network Timing

GSM_CMD_CIPMODE
Select TCPIP Application Mode

GSM_CMD_CIPCCFG
Configure Transparent Transfer Mode

GSM_CMD_CIPSHOWTP
Display Transfer Protocol in IP Head When Received Data

GSM_CMD_CIPUDPMODE
UDP Extended Mode

GSM_CMD_CIPRXGET
Get Data from Network Manually

GSM_CMD_CIPSCONT
Save TCPIP Application Context

GSM_CMD_CIPRDTIMER
Set Remote Delay Timer

GSM_CMD_CIPSGTXT
Select GPRS PDP context

GSM_CMD_CIPTKA
Set TCP Keepalive Parameters

GSM_CMD_CIPSSL
Connection SSL function

GSM_CMD_SMS_ENABLE

GSM_CMD_CMGD
Delete SMS Message

GSM_CMD_CMGF
Select SMS Message Format

GSM_CMD_CMGL
List SMS Messages from Preferred Store

GSM_CMD_CMGR
Read SMS Message

GSM_CMD_CMGS
Send SMS Message

GSM_CMD_CMGW
Write SMS Message to Memory

GSM_CMD_CMSS
Send SMS Message from Storage

GSM_CMD_CMGDA
MASS SMS delete

GSM_CMD_CNMI
New SMS Message Indications

GSM_CMD_CPMS_SET
Set preferred SMS Message Storage

GSM_CMD_CPMS_GET
Get preferred SMS Message Storage

GSM_CMD_CPMS_GET_OPT
Get optional SMS message storages

GSM_CMD_CRES
Restore SMS Settings

GSM_CMD_CSAS
Save SMS Settings

GSM_CMD_CSCA
SMS Service Center Address

GSM_CMD_CSCB
Select Cell Broadcast SMS Messages

GSM_CMD_CSDH
Show SMS Text Mode Parameters

GSM_CMD_CSMP
Set SMS Text Mode Parameters

GSM_CMD_CSMS
Select Message Service

GSM_CMD_END
Last CMD entry

enum gsm_conn_connect_res_t
Connection result on connect command.

Values:

GSM_CONN_CONNECT_UNKNOWN
No valid result

GSM_CONN_CONNECT_OK
Connected OK

GSM_CONN_CONNECT_ERROR
Error on connection

GSM_CONN_CONNECT_ALREADY
Already connected

enum gsmr_t
Result enumeration used across application functions.

Values:

gsmOK = 0
Function returned OK

gsmOKIGNOREMORE
Function succedded, should continue as gsmOK but ignore sending more data. This result is possible on connection data receive callback

gsmERR
Generic error

gsmPARERR
Wrong parameters on function call

gsmERRMEM
Memory error occurred

gsmTIMEOUT
Timeout occurred on command

gsmCONT
There is still some command to be processed in current command

gsmCLOSED
Connection just closed

gsmINPROG
Operation is in progress

gsmERRNOTENABLED
Feature not enabled error

gsmERRNOIP
Station does not have IP address

gsmERRNOFREECONN
There is no free connection available to start

gsmERRCONNTIMEOUT
Timeout received when connection to access point

gsmERRPASS
Invalid password for access point

gsmERRNOAP
No access point found with specific SSID and MAC address

gsmERRCONNFAIL
Connection failed to access point

gsmERRWIFINOTCONNECTED
Wifi not connected to access point

gsmERRNODEVICE
Device is not present

gsmERRBLOCKING
Blocking mode command is not allowed

enum gsm_device_model_t
GSM device model type.

Values:

GSM_DEVICE_MODEL_END
End of device model

GSM_DEVICE_MODEL_UNKNOWN
Unknown device model

enum gsm_mem_t
Available device memories.

Values:

GSM_MEM_END
End of memory list

GSM_MEM_CURRENT
Use current memory for read/delete operation

GSM_MEM_UNKNOWN = 0x1F
Unknown memory

enum gsm_number_type_t
GSM number type.

Values:

GSM_NUMBER_TYPE_NATIONAL = 129
Number is national

GSM_NUMBER_TYPE_INTERNATIONAL = 145
Number is international

struct gsm_conn_t
#include <gsm_private.h> Connection structure.

Public Members

gsm_conn_type_t **type**
Connection type

uint8_t **num**
Connection number

gsm_ip_t **remote_ip**
Remote IP address

gsm_port_t **remote_port**
Remote port number

gsm_port_t **local_port**
Local IP address

gsm_evt_fn **evt_func**
Callback function for connection

void ***arg**
User custom argument

uint8_t **val_id**
Validation ID number. It is increased each time a new connection is established. It protects sending data to wrong connection in case we have data in send queue, and connection was closed and active again in between.

gsm_linbuff_t **buff**
Linear buffer structure

size_t **total_recved**
Total number of bytes received

uint8_t **active** : 1
Status whether connection is active

uint8_t **client** : 1
Status whether connection is in client mode

uint8_t **data_received** : 1
Status whether first data were received on connection

uint8_t **in_closing** : 1
Status if connection is in closing mode. When in closing mode, ignore any possible received data from function

uint8_t **bearer** : 1
Bearer used. Can be `1` or `0`

struct *gsm_conn_t*::**[anonymous]**::**[anonymous] f**
Connection flags

union *gsm_conn_t*::**[anonymous] status**
Connection status union with flag bits

struct gsm_pbuf_t
#include <gsm_private.h> Packet buffer structure.

Public Members

struct gsm_pbuf ***next**
Next pbuf in chain list

size_t **tot_len**
Total length of pbuf chain

size_t **len**
Length of payload

size_t **ref**
Number of references to this structure

uint8_t ***payload**
Pointer to payload memory

gsm_ip_t **ip**
Remote address for received IPD data

gsm_port_t **port**
Remote port for received IPD data

struct gsm_ipd_t
#include <gsm_private.h> Incoming network data read structure.

Public Members

uint8_t **read**
Set to 1 when we should process input data as connection data

size_t **tot_len**
Total length of packet

size_t **rem_len**
Remaining bytes to read in current +IPD statement

gsm_conn_p **conn**
Pointer to connection for network data

size_t **buff_ptr**
Buffer pointer to save data to. When set to `NULL` while `read = 1`, reading should ignore incoming data

gsm_pbuf_p **buff**
Pointer to data buffer used for receiving data

struct gsm_msg_t
#include <gsm_private.h> Message queue structure to share between threads.

Public Members

gsm_cmd_t **cmd_def**
Default message type received from queue

gsm_cmd_t **cmd**
Since some commands can have different subcommands, sub command is used here

uint8_t **i**
Variable to indicate order number of subcommands

gsm_sys_sem_t **sem**
Semaphore for the message

uint8_t **is_blocking**
Status if command is blocking

uint32_t **block_time**
Maximal blocking time in units of milliseconds. Use 0 to for non-blocking call

gsmr_t **res**
Result of message operation

gsmr_t (***fn**) (**struct** gsm_msg *)
Processing callback function to process packet

uint32_t **delay**
Delay to use before sending first reset AT command

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **reset**
Reset device

uint32_t **baudrate**
Baudrate for AT port

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **uart**
UART configuration

uint8_t **mode**
Functionality mode

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cfun**
Set phone functionality

const char ***pin**
Pin code to write

New pin code

Current pin code

New PIN code

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cpin_enter**
Enter pin code

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cpin_add**
Add pin code if previously wasn't set

const char ***current_pin**
Current pin code

const char ***new_pin**
New pin code

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cpin_change**
Change current pin code

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cpin_remove**
Remove PIN code

const char ***puk**
PUK code

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cpuk_enter**
Enter PUK and new PIN

size_t **cnum_tries**
Number of tries

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sim_info**
Get information for SIM card

char ***str**
Pointer to output string array

size_t **len**
Length of output string array including trailing zero memory

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **device_info**
All kind of device info, serial number, model, manufacturer, revision

int16_t ***rssi**
Pointer to RSSI variable

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **csq**
Signal strength

uint8_t **read**
Flag indicating we can read the COPS actual data

Read the data flag

gsm_operator_t ***ops**
Pointer to operators array

size_t **opsl**
Length of operators array

size_t **opsi**
Current operator index array

size_t ***opf**
Pointer to number of operators found

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cops_scan**
Scan operators

gsm_operator_curr_t ***curr**
Pointer to output current operator

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cops_get**
Get current operator info

gsm_operator_mode_t **mode**
COPS mode

gsm_operator_format_t **format**
Operator format to print

const char ***name**
Short or long name, according to format

Entry name

uint32_t **num**
Number in case format is number

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **cops_set**
Set operator settings

gsm_conn_t ****conn**
Pointer to pointer to save connection used

const char ***host**
Host to use for connection

gsm_port_t **port**
Remote port used for connection

gsm_conn_type_t **type**
Connection type

void ***arg**
Connection custom argument

gsm_evt_fn **evt_func**
Callback function to use on connection

uint8_t **num**
Connection number used for start

gsm_conn_connect_res_t **conn_res**
Connection result status

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **conn_start**
Structure for starting new connection

gsm_conn_t ***conn**
Pointer to connection to close

Pointer to connection to send data

uint8_t **val_id**
Connection current validation ID when command was sent to queue

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **conn_close**
Close connection

size_t **btw**
Number of remaining bytes to write

size_t **ptr**
Current write pointer for data

const uint8_t ***data**
Data to send

size_t **sent**
Number of bytes sent in last packet

size_t **sent_all**
Number of bytes sent all together

uint8_t **tries**
Number of tries used for last packet

uint8_t **wait_send_ok_err**
Set to 1 when we wait for SEND OK or SEND ERROR

const *gsm_ip_t* ***remote_ip**
Remote IP address for UDP connection

gsm_port_t **remote_port**
Remote port address for UDP connection

uint8_t **fau**
Free after use flag to free memory after data are sent (or not)

size_t ***bw**
Number of bytes written so far

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **conn_send**
Structure to send data on connection

const char ***num**
Phone number

Entry number

const char ***text**
SMS content to send

uint8_t **format**
SMS format, `0 = PDU`, `1 = text`

size_t **pos**
Set on +CMGS response if command is OK

SMS position in memory

Memory position. Set to 0 to use new one or SIZE_T MAX to delete entry

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_send**
Send SMS

gsm_mem_t **mem**
Memory to read from

Memory to delete from

Memory to use for read

Array of memories

Memory to use

gsm_sms_entry_t ***entry**
Pointer to entry to write info

uint8_t **update**
Update SMS status after read operation

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_read**
Read single SMS

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_delete**
Delete SMS message

gsm_sms_status_t **status**
SMS status to delete

SMS entries status

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_delete_all**
Mass delete SMS messages

gsm_sms_entry_t ***entries**
Pointer to entries

size_t **etr**
Entries to read (array length)

NUmber of entries to read

size_t **ei**
Current entry index in array

Current entry index

size_t ***er**
Final entries read pointer for user

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_list**
List SMS messages

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **sms_memory**
Set preferred memories

const char ***number**
Phone number to dial

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **call_start**
Start a new call

gsm_number_type_t **type**
Entry phone number type

uint8_t **del**
Flag indicates delete

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **pb_write**
Write/Edit/Delete entry

size_t **start_index**
Start index in phonebook to read

gsm_pb_entry_t ***entries**
Pointer to entries array

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **pb_list**
List phonebook entries

const char ***search**
Search string

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **pb_search**
Search phonebook entries

const char ***code**
Code to send

char ***resp**
Response array

size_t **resp_len**
Length of response array

size_t **resp_write_ptr**
Write pointer for response

uint8_t **quote_det**
Information if quote has been detected

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **ussd**
Execute USSD command

const char ***apn**
APN address

const char ***user**
APN username

const char ***pass**
APN password

struct *gsm_msg_t*::**[anonymous]**::**[anonymous]** **network_attach**
Settings for network attach

union *gsm_msg_t*::**[anonymous]** **msg**
Group of different possible message contents

struct gsm_ip_mac_t
#include <*gsm_private.h*> IP and MAC structure with netmask and gateway addresses.

Public Members

gsm_ip_t **ip**
IP address

gsm_ip_t **gw**
Gateway address

gsm_ip_t **nm**
Netmask address

gsm_mac_t **mac**
MAC address

struct gsm_link_conn_t
#include <*gsm_private.h*> Link connection active info.

Public Members

uint8_t **failed**
Status if connection successful

uint8_t **num**
Connection number

uint8_t **is_server**
Status if connection is client or server

gsm_conn_type_t **type**
Connection type

gsm_ip_t **remote_ip**
Remote IP address

gsm_port_t **remote_port**
Remote port

gsm_port_t **local_port**
Local port number

struct gsm_evt_func_t
#include <gsm_private.h> Callback function linked list prototype.

Public Members

struct gsm_evt_func ***next**
Next function in the list

gsm_evt_fn **fn**
Function pointer itself

struct gsm_sms_mem_t
#include <gsm_private.h> SMS memory information.

Public Members

uint32_t **mem_available**
Bit field of available memories

gsm_mem_t **current**
Current memory choice

size_t **total**
Size of memory in units of entries

size_t **used**
Number of used entries

struct gsm_sms_t
#include <gsm_private.h> SMS structure.

Public Members

uint8_t **ready**
Flag indicating feature ready by device

uint8_t **enabled**
Flag indicating feature enabled

gsm_sms_mem_t **mem**[3]
3 memory info for operation,receive,sent storage

struct gsm_pb_mem_t
#include <gsm_private.h> SMS memory information.

Public Members

uint32_t **mem_available**
Bit field of available memories

gsm_mem_t **current**
Current memory choice

size_t **total**
Size of memory in units of entries

size_t **used**
Number of used entries

struct gsm_pb_t
#include <*gsm_private.h*> Phonebook structure.

Public Members

uint8_t **ready**
Flag indicating feature ready by device

uint8_t **enabled**
Flag indicating feature enabled

gsm_pb_mem_t **mem**
Memory information

struct gsm_sim_t
#include <*gsm_private.h*> SIM structure.

Public Members

gsm_sim_state_t **state**
Current SIM status

struct gsm_network_t
#include <*gsm_private.h*> Network info.

Public Members

gsm_network_reg_status_t **status**
Network registration status

gsm_operator_curr_t **curr_operator**
Current operator information

uint8_t **is_attached**
Flag indicating device is attached and PDP context is active

gsm_ip_t **ip_addr**
Device IP address when network PDP context is enabled

struct gsm_modules_t
#include <*gsm_private.h*> GSM modules structure.

Public Members

char **model_manufacturer**[20]
Device manufacturer

char **model_number**[20]
Device model number

char **model_serial_number**[20]
Device serial number

char **model_revision**[20]
Device revision

gsm_device_model_t **model**
Device model

gsm_sim_t **sim**
SIM data

gsm_network_t **network**
Network status

int16_t **rssi**
RSSI signal strength. `0` = invalid, `-53 % -113` = valid

uint8_t **active_conns_cur_parse_num**
Current connection number used for parsing

gsm_conn_t **conns**[**GSM_CFG_MAX_CONNS**]
Array of all connection structures

gsm_ipd_t **ipd**
Connection incoming data structure

uint8_t **conn_val_id**
Validation ID increased each time device connects to network

gsm_sms_t **sms**
SMS information

gsm_pb_t **pb**
Phonebook information

gsm_call_t **call**
Call information

struct gsm_t
#include <gsm_private.h> GSM global structure.

Public Members

size_t **locked_cnt**
Counter how many times (recursive) stack is currently locked

gsm_sys_sem_t **sem_sync**
Synchronization semaphore between threads

gsm_sys_mbox_t **mbox_producer**
Producer message queue handle

gsm_sys_mbox_t **mbox_process**
Consumer message queue handle

gsm_sys_thread_t **thread_produce**
Producer thread handle

gsm_sys_thread_t **thread_process**
Processing thread handle

gsm_buff_t **buff**
Input processing buffer

gsm_ll_t **ll**
Low level functions

gsm_msg_t ***msg**
Pointer to current user message being executed

gsm_evt_t **evt**
Callback processing structure

gsm_evt_func_t ***evt_func**
Callback function linked list

gsm_modules_t **m**
All modules. When resetting, reset structure

uint8_t **initialized** : 1
Flag indicating GSM library is initialized

uint8_t **dev_present** : 1
Flag indicating GSM device is present

struct *gsm_t*::**[anonymous]**::**[anonymous]** **f**
Flags structure

union *gsm_t*::**[anonymous]** **status**
Status structure

struct gsm_dev_mem_map_t
#include <gsm_private.h> Memory mapping structure between string and value in app.

Public Members

gsm_mem_t **mem**
Mem indication

const char ***mem_str**
Memory string

struct gsm_dev_model_map_t
#include <gsm_private.h> Device models map between model and other information.

Public Members

gsm_device_model_t **model**
Device model

const char ***id_str**
Model string identification

uint8_t **is_2g**
Status if modem is 2G

uint8_t **is_lte**
Status if modem is LTE

struct gsm_unicode_t
#include <*gsm_private.h*> Unicode support structure.

Public Members

uint8_t **ch**[4]
UTF-8 max characters

uint8_t **t**
Total expected length in UTF-8 sequence

uint8_t **r**
Remaining bytes in UTF-8 sequence

gsmr_t **res**
Current result of processing

struct gsm_ip_t
#include <*gsm_typedefs.h*> IP structure.

Public Members

uint8_t **ip**[4]
IPv4 address

struct gsm_mac_t
#include <*gsm_typedefs.h*> MAC address.

Public Members

uint8_t **mac**[6]
MAC address

struct gsm_datetime_t
#include <*gsm_typedefs.h*> Date and time structure.

Public Members

uint8_t **date**
Day in a month, from 1 to up to 31

uint8_t **month**
Month in a year, from 1 to 12

uint16_t **year**
Year

uint8_t **day**
Day in a week, from 1 to 7, 0 = invalid

uint8_t **hours**
Hours in a day, from 0 to 23

uint8_t **minutes**
Minutes in a hour, from 0 to 59

uint8_t **seconds**
Seconds in a minute, from 0 to 59

struct gsm_linbuff_t
#include <*gsm_typedefs.h*> Linear buffer structure.

Public Members

uint8_t ***buff**
Pointer to buffer data array

size_t **len**
Length of buffer array

size_t **ptr**
Current buffer pointer

Unicode

Unicode decoder block. It can decode sequence of *UTF-8* characters, between 1 and 4 bytes long.

Note: This is simple implementation and does not support string encoding.

group **GSM_UNICODE**
Unicode support manager.

Functions

gsmr_t **gsmi_unicode_decode**(*gsm_unicode_t* *uni, uint8_t *ch*)
Decode single character for unicode (UTF-8 only) format.

Parameters

- `[inout] s`: Pointer to unicode decode control structure
- `[in] c`: UTF-8 character sequence to test for device

Return Value

- `gsmOK`: Function succedded, there is a valid UTF-8 sequence
- `gsmINPROG`: Function continues well but expects some more data to finish sequence
- `gsmERR`: Error in UTF-8 sequence

struct gsm_unicode_t
#include <gsm_private.h> Unicode support structure.

Public Members

uint8_t **ch**[4]
UTF-8 max characters

uint8_t **t**
Total expected length in UTF-8 sequence

uint8_t **r**
Remaining bytes in UTF-8 sequence

gsmr_t **res**
Current result of processing

Unstructured Supplementary Service Data

group **GSM_USSD**
Unstructured Supplementary Service Data.

Functions

gsmr_t **gsm_ussd_run**(**const** char *code, char *resp, size_t *resp_len*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Run USSD command, such as `*123#` to get balance on SIM card.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] code`: Code to run, such as `*123#`
- `[out] resp`: Pointer to array to save response
- `[in] resp_len`: Length of array, including string `NULL` termination
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used

- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Utilities

Utility functions for various cases. These function are used across entire middleware and can also be used by application.

group **GSM_UTILS**
Utilities.

Defines

GSM_ASSERT(msg, c)
Assert an input parameter if in valid range.

Note Since this is a macro, it may only be used on a functions where return status is of type *gsmr_t* enumeration

Parameters

- `[in] msg`: message to print to debug if test fails
- `[in] c`: Condition to test

GSM_MEM_ALIGN(x)
Align `x` value to specific number of bytes, provided by *GSM_CFG_MEM_ALIGNMENT* configuration.

Return Input value aligned to specific number of bytes

Parameters

- `[in] x`: Input value to align

GSM_MIN(x, y)
Get minimal value between `x` and `y` inputs.

Return Minimal value between `x` and `y` parameters

Parameters

- `[in] x`: First input to test
- `[in] y`: Second input to test

GSM_MAX(x, y)
Get maximal value between `x` and `y` inputs.

Return Maximal value between `x` and `y` parameters

Parameters

- `[in] x`: First input to test
- `[in] y`: Second input to test

GSM_ARRAYSIZE(x)
Get size of statically declared array.

Return Number of array elements

Parameters

- `[in] x`: Input array

GSM_UNUSED(x)

Unused argument in a function call.

Note Use this on all parameters in a function which are not used to prevent compiler warnings complaining about "unused variables"

Parameters

- `[in] x`: Variable which is not used

GSM_U32(x)

Get input value casted to `unsigned 32-bit` value.

Parameters

- `[in] x`: Input value

GSM_U16(x)

Get input value casted to `unsigned 16-bit` value.

Parameters

- `[in] x`: Input value

GSM_U8(x)

Get input value casted to `unsigned 8-bit` value.

Parameters

- `[in] x`: Input value

GSM_I32(x)

Get input value casted to `signed 32-bit` value.

Parameters

- `[in] x`: Input value

GSM_I16(x)

Get input value casted to `signed 16-bit` value.

Parameters

- `[in] x`: Input value

GSM_I8(x)

Get input value casted to `signed 8-bit` value.

Parameters

- `[in] x`: Input value

GSM_SZ(x)
Get input value casted to `size_t` value.

Parameters

- `[in] x`: Input value

gsm_u32_to_str(num, out)
Convert `unsigned 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

gsm_u32_to_hex_str(num, out, w)
Convert `unsigned 32-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply.

gsm_i32_to_str(num, out)
Convert `signed 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

gsm_u16_to_str(num, out)
Convert `unsigned 16-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

gsm_u16_to_hex_str(num, out, w)
Convert `unsigned 16-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert

- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply.

gsm_i16_to_str(num, out)
Convert `signed 16-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

gsm_u8_to_str(num, out)
Convert `unsigned 8-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

gsm_u8_to_hex_str(num, out, w)
Convert `unsigned 16-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply.

gsm_i8_to_str(num, out)
Convert `signed 8-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

Functions

char *__gsm_u32_to_gen_str__(uint32_t *num*, char **out*, uint8_t *is_hex*, uint8_t *padding*)
Convert `unsigned 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] is_hex`: Set to `1` to output hex, 0 otherwise
- `[in] width`: Width of output string. When number is shorter than width, leading `0` characters will apply. This parameter is valid only when formatting hex numbers

char *__gsm_i32_to_gen_str__(int32_t *num*, char **out*)
Convert `signed 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

group **GSM**
GSM stack.

Functions

gsmr_t __gsm_init__(*gsm_evt_fn evt_func*, **const** uint32_t *blocking*)
Init and prepare GSM stack for device operation.

Note Function must be called from operating system thread context. It creates necessary threads and waits them to start, thus running operating system is important.

- When *GSM_CFG_RESET_ON_INIT* is enabled, reset sequence will be sent to device otherwise manual call to *gsm_reset* is required to setup device

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] evt_func`: Global event callback function for all major events
- `[in] blocking`: Status whether command should be blocking or not. Used when *GSM_CFG_RESET_ON_INIT* is enabled.

gsmr_t __gsm_reset__(**const** *gsm_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Execute reset and send default commands.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command is finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_reset_with_delay**(uint32_t *delay*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Execute reset and send default commands with delay.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] delay`: Number of milliseconds to wait before initiating first command to device
- `[in] evt_fn`: Callback function called when command is finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_set_func_mode**(uint8_t *mode*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Set modem function mode.

Note Use this function to set modem to normal or low-power mode

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] mode`: Mode status. Set to `1` for full functionality or `0` for low-power mode (no functionality)
- `[in] evt_fn`: Callback function called when command is finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

gsmr_t **gsm_core_lock**(void)

Lock stack from multi-thread access, enable atomic access to core.

If lock was `0` prior funcion call, lock is enabled and increased

Note Function may be called multiple times to increase locks. Application must take care to call *gsm_core_unlock* the same amount of time to make sure lock gets back to `0`

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

gsmr_t **gsm_core_unlock**(void)

Unlock stack for multi-thread access.

Used in conjunction with *gsm_core_lock* function

If lock was non-zero before function call, lock is decreased. When `lock == 0`, protection is disabled and other threads may access to core

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

gsmr_t **gsm_device_set_present**(uint8_t *present*, **const** *gsm_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Notify stack if device is present or not.

Use this function to notify stack that device is not physically connected and not ready to communicate with host device

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] present`: Flag indicating device is present
- `[in] evt_fn`: Callback function called when command is finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

uint8_t **gsm_device_is_present**(void)

Check if device is present.

Return `1` on success, `0` otherwise

uint8_t **gsm_delay**(uint32_t *ms*)

Delay for amount of milliseconds.

Delay is based on operating system semaphores. It locks semaphore and waits for timeout in `ms` time. Based on operating system, thread may be put to *blocked* list during delay and may improve execution speed

Return `1` on success, `0` otherwise

Parameters

- `[in] ms`: Milliseconds to delay

5.3.2 GSM Configuration

This is the default configuration of the middleware. When any of the settings shall be modified, it shall be done in dedicated application config `gsm_config.h` file.

Note: Check *Getting started* to create configuration file.

group **GSM_CONFIG**

Configuration parameters.

Defines

GSM_CFG_OS
Enables 1 or disables 0 operating system support for GSM library.

Note Value must be set to 1 in the current revision

Note Check *OS configuration* group for more configuration related to operating system

GSM_CFG_MEM_CUSTOM
Enables 1 or disables 0 custom memory management functions.

When set to 1, *Memory manager* block must be provided manually. This includes implementation of functions *gsm_mem_malloc*, *gsm_mem_calloc*, *gsm_mem_realloc* and *gsm_mem_free*

Note Function declaration follows standard C functions `malloc, calloc, realloc, free`. Declaration is available in `gsm/gsm_mem.h` file. Include this file to final implementation file

Note When implementing custom memory allocation, it is necessary to take care of multiple threads accessing same resource for custom allocator

GSM_CFG_MEM_ALIGNMENT
Memory alignment for dynamic memory allocations.

Note Some CPUs can work faster if memory is aligned, usually to 4 or 8 bytes. To speed up this possibilities, you can set memory alignment and library will try to allocate memory on aligned boundaries.

Note Some CPUs such ARM Cortex-M0 don't support unaligned memory access. This CPUs must have set correct memory alignment value.

Note This value must be power of 2

GSM_CFG_USE_API_FUNC_EVT
Enables 1 or disables 0 callback function and custom parameter for API functions.

When enabled, 2 additional parameters are available in API functions. When command is executed, callback function with its parameter could be called when not set to `NULL`.

GSM_CFG_MAX_CONNS
Maximal number of connections AT software can support on GSM device.

GSM_CFG_CONN_MAX_DATA_LEN
Maximal number of bytes we can send at single command to GSM.

Note Value can not exceed 1460 bytes or no data will be ever send

Note This is limitation of GSM AT commands and on systems where RAM is not an issue, it should be set to maximal value (1460) to optimize data transfer speed performance

GSM_CFG_MAX_SEND_RETRIES
Set number of retries for send data command.

Sometimes it may happen that AT+SEND command fails due to different problems. Trying to send the same data multiple times can raise chances we are successful.

GSM_CFG_IPD_MAX_BUFF_SIZE
Maximum single buffer size for network receive data (TCP/UDP connections)

Note When GSM sends buffer buffer than maximal, multiple buffers are created

GSM_CFG_AT_PORT_BAUDRATE
Default baudrate used for AT port.

Note Later, user may call API function to change to desired baudrate if necessary

GSM_CFG_RCV_BUFF_SIZE
Buffer size for received data waiting to be processed.

Note When server mode is active and a lot of connections are in queue this should be set high otherwise your buffer may overflow

Note Buffer size also depends on TX user driver if it uses DMA or blocking mode In case of DMA (CPU can work other tasks), buffer may be smaller as CPU will have more time to process all the incoming bytes

Note This parameter has no meaning when *GSM_CFG_INPUT_USE_PROCESS* is enabled

GSM_CFG_RESET_ON_INIT
Enables `1` or disables `0` reset sequence after *gsm_init* call.

Note When this functionality is disabled, user must manually call *gsm_reset* to send reset sequence to GSM device.

GSM_CFG_RESET_ON_DEVICE_PRESENT
Enables `1` or disables `0` reset sequence after *gsm_device_set_present* call.

Note When this functionality is disabled, user must manually call *gsm_reset* to send reset sequence to GSM device.

GSM_CFG_RESET_DELAY_DEFAULT
Default delay (milliseconds unit) before sending first AT command on reset sequence.

GSM_CFG_CONN_POLL_INTERVAL
Poll interval for connections in units of milliseconds.

Value indicates interval time to call poll event on active connections.

Note Single poll interval applies for all connections

group **GSM_CONFIG_DBG**
Debugging configurations.

Defines

GSM_CFG_DBG
Set global debug support.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

Note Set to *GSM_DBG_OFF* to globally disable all debugs

GSM_CFG_DBG_OUT(fmt, ...)
Debugging output function.

Called with format and optional parameters for printf style debug

GSM_CFG_DBG_LVL_MIN
Minimal debug level.

Check GSM_DBG_LVL for possible values

GSM_CFG_DBG_TYPES_ON
Enabled debug types.

When debug is globally enabled with *GSM_CFG_DBG* parameter, user must enable debug types such as TRACE or STATE messages.

Check GSM_DBG_TYPE for possible options. Separate values with `bitwise OR` operator

GSM_CFG_DBG_INIT
Set debug level for init function.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_MEM
Set debug level for memory manager.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_INPUT
Set debug level for input module.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_THREAD
Set debug level for GSM threads.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_ASSERT
Set debug level for asserting of input variables.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_IPD
Set debug level for incoming data received from device.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_PBUF
Set debug level for packet buffer manager.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_CONN
Set debug level for connections.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_VAR
Set debug level for dynamic variable allocations.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_NETCONN
Set debug level for netconn sequential API.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_AT_ECHO
Enables `1` or disables `0` echo mode on AT commands sent to GSM device.

Note This mode is useful when debugging GSM communication

group **GSM_CONFIG_OS**
Operating system dependant configuration.

Defines

GSM_CFG_THREAD_PRODUCER_MBOX_SIZE
Set number of message queue entries for procuder thread.

Message queue is used for storing memory address to command data

GSM_CFG_THREAD_PROCESS_MBOX_SIZE
Set number of message queue entries for processing thread.

Message queue is used to notify processing thread about new received data on AT port

GSM_CFG_INPUT_USE_PROCESS
Enables `1` or disables `0` direct support for processing input data.

When this mode is enabled, no overhead is included for copying data to receive buffer because bytes are processed directly.

Note This mode can only be used when *GSM_CFG_OS* is enabled

Note When using this mode, separate thread must be dedicated only for reading data on AT port

Note Best case for using this mode is if DMA receive is supported by host device

GSM_THREAD_PRODUCER_HOOK()
Producer thread hook, called each time thread wakes-up and does the processing.

It can be used to check if thread is alive.

GSM_THREAD_PROCESS_HOOK()
Process thread hook, called each time thread wakes-up and does the processing.

It can be used to check if thread is alive.

group **GSM_CONFIG_STD_LIB**
Standard C library configuration.

Configuration allows you to overwrite default C language function in case of better implementation with hardware (for example DMA for data copy).

Defines

GSM_MEMCPY(dst, src, len)
Memory copy function declaration.

User is able to change the memory function, in case hardware supports copy operation, it may implement its own

Function prototype must be similar to:

```
void *  my_memcpy(void* dst, const void* src, size_t len);
```

Return Destination memory start address

Parameters

- `[in] dst`: Destination memory start address
- `[in] src`: Source memory start address
- `[in] len`: Number of bytes to copy

GSM_MEMSET(dst, b, len)
Memory set function declaration.

Function prototype must be similar to:

```
void * my_memset(void* dst, int b, size_t len);
```

Return Destination memory start address

Parameters

- `[in] dst`: Destination memory start address
- `[in] b`: Value (byte) to set in memory
- `[in] len`: Number of bytes to set

group **GSM_CONFIG_MODULES**
Configuration of specific modules.

Defines

GSM_CFG_NETWORK
Enables `1` or disables `0` network functionality used for TCP/IP communication.

Network must be enabled to use all GPRS/LTE functions such as connection API, FTP, HTTP, etc.

GSM_CFG_NETWORK_IGNORE_CGACT_RESULT
Ignores `1` or not `0` result from `AT+CGACT` command.

Note This may be used for data-only SIM cards where command might fail when trying to attach to network for data transfer

GSM_CFG_CONN
Enables `1` or disables `0` connection API.

Note *GSM_CFG_NETWORK* must be enabled to use connection feature

GSM_CFG_SMS
Enables `1` or disables `0` SMS API.

GSM_CFG_CALL
Enables `1` or disables `0` call API.

GSM_CFG_PHONEBOOK
Enables `1` or disables `0` phonebook API.

GSM_CFG_HTTP
Enables `1` or disables `0` HTTP API.

Note *GSM_CFG_NETWORK* must be enabled to use connection feature

GSM_CFG_FTP
Enables 1 or disables 0 FTP API.

Note *GSM_CFG_NETWORK* must be enabled to use connection feature

GSM_CFG_PING
Enables 1 or disables 0 PING API.

Note *GSM_CFG_NETWORK* must be enabled to use connection feature

GSM_CFG_USSD
Enables 1 or disables 0 USSD API.

group **GSM_CONFIG_MODULES_NETCONN**
Configuration of netconn API module.

Defines

GSM_CFG_NETCONN
Enables 1 or disables 0 NETCONN sequential API support for OS systems.

Note To use this feature, OS support is mandatory.

See *GSM_CFG_OS*

GSM_CFG_NETCONN_RECEIVE_TIMEOUT
Enables 1 or disables 0 receive timeout feature.

When this option is enabled, user will get an option to set timeout value for receive data on netconn, before function returns timeout error.

Note Even if this option is enabled, user must still manually set timeout, by default time will be set to 0 which means no timeout.

GSM_CFG_NETCONN_ACCEPT_QUEUE_LEN
Accept queue length for new client when netconn server is used.

Defines number of maximal clients waiting in accept queue of server connection

GSM_CFG_NETCONN_RECEIVE_QUEUE_LEN
Receive queue length for pbuf entries.

Defines maximal number of pbuf data packet references for receive

group **GSM_CONFIG_MODULES_MQTT**
Configuration of MQTT and MQTT API client modules.

Defines

GSM_CFG_MQTT_MAX_REQUESTS
Maximal number of open MQTT requests at a time.

GSM_CFG_DBG_MQTT
Set debug level for MQTT client module.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

GSM_CFG_DBG_MQTT_API
Set debug level for MQTT API client module.

Possible values are *GSM_DBG_ON* or *GSM_DBG_OFF*

5.3.3 Platform specific

List of all the modules:

Low-Level functions

Low-level module consists of callback-only functions, which are called by middleware and must be implemented by final application.

Tip: Check *Porting guide* for actual implementation

group **GSM_LL**
Low-level communication functions.

Typedefs

typedef size_t (***gsm_ll_send_fn**) (**const** void *data, size_t len)
Function prototype for AT output data.

Return Number of bytes sent

Parameters

- `[in] data`: Pointer to data to send. This parameter can be set to `NULL`
- `[in] len`: Number of bytes to send. This parameter can be set to `0` to indicate that internal buffer can be flushed to stream. This is implementation defined and feature might be ignored

typedef uint8_t (***gsm_ll_reset_fn**) (uint8_t state)
Function prototype for hardware reset of GSM device.

Return `1` on successful action, `0` otherwise

Parameters

- `[in] state`: State indicating reset. When set to `1`, reset must be active (usually pin active low), or set to `0` when reset is cleared

Functions

gsmr_t **gsm_ll_init** (*gsm_ll_t *ll*)
Callback function called from initialization process.

Note This function may be called multiple times if AT baudrate is changed from application. It is important that every configuration except AT baudrate is configured only once!

Note This function may be called from different threads in GSM stack when using OS. When *GSM_CFG_INPUT_USE_PROCESS* is set to 1, this function may be called from user UART thread.

Return gsmOK on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[inout] ll`: Pointer to *gsm_ll_t* structure to fill data for communication functions

gsmr_t **gsm_ll_deinit** (*gsm_ll_t *ll*)
Callback function to de-init low-level communication part.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[inout] ll`: Pointer to *gsm_ll_t* structure to fill data for communication functions

struct gsm_ll_t
#include <gsm_typedefs.h> Low level user specific functions.

Public Members

gsm_ll_send_fn **send_fn**
Callback function to transmit data

gsm_ll_reset_fn **reset_fn**
Reset callback function

uint32_t **baudrate**
UART baudrate value

struct *gsm_ll_t*::**[anonymous]** **uart**
UART communication parameters

System functions

System functions are bridge between operating system system calls and middleware system calls. Middleware is tightly coupled with operating system features hence it is important to include OS features directly.

It includes support for:

- Thread management, to start/stop threads
- Mutex management for recursive mutexes
- Semaphore management for binary-only semaphores
- Message queues for thread-safe data exchange between threads
- Core system protection for mutual exclusion to access shared resources

Tip: Check *Porting guide* for actual implementation guidelines.

group **GSM_SYS**
System based function for OS management, timings, etc.

Main

uint8_t **gsm_sys_init**(void)
Init system dependant parameters.

After this function is called, all other system functions must be fully ready.

Return `1` on success, `0` otherwise

uint32_t **gsm_sys_now**(void)
Get current time in units of milliseconds.

Return Current time in units of milliseconds

uint8_t **gsm_sys_protect**(void)
Protect middleware core.

Stack protection must support recursive mode. This function may be called multiple times, even if access has been granted before.

Note Most operating systems support recursive mutexes.

Return `1` on success, `0` otherwise

uint8_t **gsm_sys_unprotect**(void)
Unprotect middleware core.

This function must follow number of calls of *gsm_sys_protect* and unlock access only when counter reached back zero.

Note Most operating systems support recursive mutexes.

Return `1` on success, `0` otherwise

Mutex

uint8_t **gsm_sys_mutex_create**(*gsm_sys_mutex_t* **p*)
Create new recursive mutex.

Note Recursive mutex has to be created as it may be locked multiple times before unlocked

Return `1` on success, `0` otherwise

Parameters

- `[out] p`: Pointer to mutex structure to allocate

uint8_t **gsm_sys_mutex_delete**(*gsm_sys_mutex_t* **p*)
Delete recursive mutex from system.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to mutex structure

uint8_t **gsm_sys_mutex_lock**(*gsm_sys_mutex_t* *p)
Lock recursive mutex, wait forever to lock.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to mutex structure

uint8_t **gsm_sys_mutex_unlock**(*gsm_sys_mutex_t* *p)
Unlock recursive mutex.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to mutex structure

uint8_t **gsm_sys_mutex_isvalid**(*gsm_sys_mutex_t* *p)
Check if mutex structure is valid system.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to mutex structure

uint8_t **gsm_sys_mutex_invalid**(*gsm_sys_mutex_t* *p)
Set recursive mutex structure as invalid.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to mutex structure

Semaphores

uint8_t **gsm_sys_sem_create**(*gsm_sys_sem_t* *p, uint8_t *cnt*)
Create a new binary semaphore and set initial state.

Note Semaphore may only have `1` token available

Return `1` on success, `0` otherwise

Parameters

- `[out] p`: Pointer to semaphore structure to fill with result
- `[in] cnt`: Count indicating default semaphore state: `0`: Take semaphore token immediately `1`: Keep token available

uint8_t **gsm_sys_sem_delete**(*gsm_sys_sem_t* *p)
Delete binary semaphore.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to semaphore structure

uint32_t **gsm_sys_sem_wait**(*gsm_sys_sem_t* *p, uint32_t *timeout*)
Wait for semaphore to be available.

Return Number of milliseconds waited for semaphore to become available or *GSM_SYS_TIMEOUT* if not available within given time

Parameters

- `[in] p`: Pointer to semaphore structure
- `[in] timeout`: Timeout to wait in milliseconds. When `0` is applied, wait forever

uint8_t **gsm_sys_sem_release**(*gsm_sys_sem_t* *p)
Release semaphore.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to semaphore structure

uint8_t **gsm_sys_sem_isvalid**(*gsm_sys_sem_t* *p)
Check if semaphore is valid.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to semaphore structure

uint8_t **gsm_sys_sem_invalid**(*gsm_sys_sem_t* *p)
Invalid semaphore.

Return `1` on success, `0` otherwise

Parameters

- `[in] p`: Pointer to semaphore structure

Message queues

uint8_t **gsm_sys_mbox_create**(*gsm_sys_mbox_t* *b, size_t *size*)
Create a new message queue with entry type of `void *`

Return `1` on success, `0` otherwise

Parameters

- `[out] b`: Pointer to message queue structure
- `[in] size`: Number of entries for message queue to hold

uint8_t **gsm_sys_mbox_delete**(*gsm_sys_mbox_t* *b)
Delete message queue.

Return `1` on success, `0` otherwise

Parameters

- `[in]` `b`: Pointer to message queue structure

uint32_t **gsm_sys_mbox_put** (*gsm_sys_mbox_t* *b, void *m)
Put a new entry to message queue and wait until memory available.

Return Time in units of milliseconds needed to put a message to queue

Parameters

- `[in]` `b`: Pointer to message queue structure
- `[in]` `m`: Pointer to entry to insert to message queue

uint32_t **gsm_sys_mbox_get** (*gsm_sys_mbox_t* *b, void **m, uint32_t *timeout*)
Get a new entry from message queue with timeout.

Return Time in units of milliseconds needed to put a message to queue or *GSM_SYS_TIMEOUT* if it was not successful

Parameters

- `[in]` `b`: Pointer to message queue structure
- `[in]` `m`: Pointer to pointer to result to save value from message queue to
- `[in]` `timeout`: Maximal timeout to wait for new message. When `0` is applied, wait for unlimited time

uint8_t **gsm_sys_mbox_putnow** (*gsm_sys_mbox_t* *b, void *m)
Put a new entry to message queue without timeout (now or fail)

Return `1` on success, `0` otherwise

Parameters

- `[in]` `b`: Pointer to message queue structure
- `[in]` `m`: Pointer to message to save to queue

uint8_t **gsm_sys_mbox_getnow** (*gsm_sys_mbox_t* *b, void **m)
Get an entry from message queue immediately.

Return `1` on success, `0` otherwise

Parameters

- `[in]` `b`: Pointer to message queue structure
- `[in]` `m`: Pointer to pointer to result to save value from message queue to

uint8_t **gsm_sys_mbox_isvalid** (*gsm_sys_mbox_t* *b)
Check if message queue is valid.

Return `1` on success, `0` otherwise

Parameters

- `[in]` `b`: Pointer to message queue structure

uint8_t **gsm_sys_mbox_invalid**(*gsm_sys_mbox_t* *b)
Invalid message queue.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

Threads

uint8_t **gsm_sys_thread_create**(*gsm_sys_thread_t* *t, **const** char *name, *gsm_sys_thread_fn thread_func*, void ***const** *arg*, size_t *stack_size*, *gsm_sys_thread_prio_t prio*)
Create a new thread.

Return 1 on success, 0 otherwise

Parameters

- [out] t: Pointer to thread identifier if create was successful. It may be set to NULL
- [in] name: Name of a new thread
- [in] thread_func: Thread function to use as thread body
- [in] arg: Thread function argument
- [in] stack_size: Size of thread stack in uints of bytes. If set to 0, reserve default stack size
- [in] prio: Thread priority

uint8_t **gsm_sys_thread_terminate**(*gsm_sys_thread_t* *t)
Terminate thread (shut it down and remove)

Return 1 on success, 0 otherwise

Parameters

- [in] t: Pointer to thread handle to terminate. If set to NULL, terminate current thread (thread from where function is called)

uint8_t **gsm_sys_thread_yield**(void)
Yield current thread.

Return 1 on success, 0 otherwise

Defines

GSM_SYS_MUTEX_NULL
Mutex invalid value.

Value assigned to *gsm_sys_mutex_t* type when it is not valid.

GSM_SYS_SEM_NULL
Semaphore invalid value.

Value assigned to *gsm_sys_sem_t* type when it is not valid.

GSM_SYS_MBOX_NULL
Message box invalid value.

Value assigned to *gsm_sys_mbox_t* type when it is not valid.

GSM_SYS_TIMEOUT
OS timeout value.

Value returned by operating system functions (mutex wait, sem wait, mbox wait) when it returns timeout and does not give valid value to application

GSM_SYS_THREAD_PRIO
Default thread priority value used by middleware to start built-in threads.

Threads can well operate with normal (default) priority and do not require any special feature in terms of priority for prioer operation.

GSM_SYS_THREAD_SS
Stack size in units of bytes for system threads.

It is used as default stack size for all built-in threads.

Typedefs

typedef void (***gsm_sys_thread_fn**) (void *)
Thread function prototype.

typedef osMutexId_t **gsm_sys_mutex_t**
System mutex type.

It is used by middleware as base type of mutex.

typedef osSemaphoreId_t **gsm_sys_sem_t**
System semaphore type.

It is used by middleware as base type of mutex.

typedef osMessageQueueId_t **gsm_sys_mbox_t**
System message queue type.

It is used by middleware as base type of mutex.

typedef osThreadId_t **gsm_sys_thread_t**
System thread ID type.

typedef osPriority **gsm_sys_thread_prio_t**
System thread priority type.

It is used as priority type for system function, to start new threads by middleware.

5.3.4 Applications

MQTT Client

MQTT client v3.1.1 implementation, based on callback (non-netconn) connection API.

group **GSM_APP_MQTT_CLIENT**
MQTT client.

Typedefs

typedef struct gsm_mqtt_client ***gsm_mqtt_client_p**
Pointer to gsm_mqtt_client_t structure.

typedef void (***gsm_mqtt_evt_fn**) (*gsm_mqtt_client_p* client, *gsm_mqtt_evt_t* *evt)
MQTT event callback function.

Parameters

- `[in] client`: MQTT client
- `[in] evt`: MQTT event with type and related data

Enums

enum gsm_mqtt_qos_t
Quality of service enumeration.

Values:

GSM_MQTT_QOS_AT_MOST_ONCE = 0x00
Delivery is not guaranteed to arrive, but can arrive `up to 1 time` = non-critical packets where losses are allowed

GSM_MQTT_QOS_AT_LEAST_ONCE = 0x01
Delivery is quaranteed `at least once`, but it may be delivered multiple times with the same content

GSM_MQTT_QOS_EXACTLY_ONCE = 0x02
Delivery is quaranteed `exactly once` = very critical packets such as billing informations or similar

enum gsm_mqtt_state_t
State of MQTT client.

Values:

GSM_MQTT_CONN_DISCONNECTED = 0x00
Connection with server is not established

GSM_MQTT_CONN_CONNECTING
Client is connecting to server

GSM_MQTT_CONN_DISCONNECTING
Client connection is disconnecting from server

GSM_MQTT_CONNECTING
MQTT client is connecting... CONNECT command has been sent to server

GSM_MQTT_CONNECTED
MQTT is fully connected and ready to send data on topics

enum gsm_mqtt_evt_type_t
MQTT event types.

Values:

GSM_MQTT_EVT_CONNECT
MQTT client connect event

GSM_MQTT_EVT_SUBSCRIBE
MQTT client subscribed to specific topic

GSM_MQTT_EVT_UNSUBSCRIBE
MQTT client unsubscribed from specific topic

GSM_MQTT_EVT_PUBLISH
MQTT client publish message to server event.
Note When publishing packet with quality of service *GSM_MQTT_QOS_AT_MOST_ONCE*, you may not receive event, even if packet was successfully sent, thus do not rely on this event for packet with `qos = GSM_MQTT_QOS_AT_MOST_ONCE`

GSM_MQTT_EVT_PUBLISH_RECV
MQTT client received a publish message from server

GSM_MQTT_EVT_DISCONNECT
MQTT client disconnected from MQTT server

GSM_MQTT_EVT_KEEP_ALIVE
MQTT keep-alive sent to server and reply received

enum gsm_mqtt_conn_status_t
List of possible results from MQTT server when executing connect command.

Values:

GSM_MQTT_CONN_STATUS_ACCEPTED = 0x00
Connection accepted and ready to use

GSM_MQTT_CONN_STATUS_REFUSED_PROTOCOL_VERSION = 0x01
Connection Refused, unacceptable protocol version

GSM_MQTT_CONN_STATUS_REFUSED_ID = 0x02
Connection refused, identifier rejected

GSM_MQTT_CONN_STATUS_REFUSED_SERVER = 0x03
Connection refused, server unavailable

GSM_MQTT_CONN_STATUS_REFUSED_USER_PASS = 0x04
Connection refused, bad user name or password

GSM_MQTT_CONN_STATUS_REFUSED_NOT_AUTHORIZED = 0x05
Connection refused, not authorized

GSM_MQTT_CONN_STATUS_TCP_FAILED = 0x100
TCP connection to server was not successful

Functions

gsm_mqtt_client_p **gsm_mqtt_client_new**(size_t *tx_buff_len*, size_t *rx_buff_len*)
Allocate a new MQTT client structure.

Return Pointer to new allocated MQTT client structure or `NULL` on failure

Parameters

- `[in] tx_buff_len`: Length of raw data output buffer
- `[in] rx_buff_len`: Length of raw data input buffer

void **gsm_mqtt_client_delete**(*gsm_mqtt_client_p client*)
Delete MQTT client structure.

Note MQTT client must be disconnected first

Parameters

- `[in] client`: MQTT client

gsmr_t **gsm_mqtt_client_connect**(*gsm_mqtt_client_p client*, **const** char **host*, *gsm_port_t port*, *gsm_mqtt_evt_fn evt_fn*, **const** *gsm_mqtt_client_info_t *info*)

Connect to MQTT server.

Note After TCP connection is established, CONNECT packet is automatically sent to server

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] host`: Host address for server
- `[in] port`: Host port number
- `[in] evt_fn`: Callback function for all events on this MQTT client
- `[in] info`: Information structure for connection

gsmr_t **gsm_mqtt_client_disconnect**(*gsm_mqtt_client_p client*)

Disconnect from MQTT server.

Return *gsmOK* if request sent to queue or member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT client

uint8_t **gsm_mqtt_client_is_connected**(*gsm_mqtt_client_p client*)

Test if client is connected to server and accepted to MQTT protocol.

Note Function will return error if TCP is connected but MQTT not accepted

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT client

gsmr_t **gsm_mqtt_client_subscribe**(*gsm_mqtt_client_p client*, **const** char **topic*, gsm_mqtt_qos_t *qos*, void **arg*)

Subscribe to MQTT topic.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic name to subscribe to
- `[in] qos`: Quality of service. This parameter can be a value of *gsm_mqtt_qos_t*
- `[in] arg`: User custom argument used in callback

gsmr_t **gsm_mqtt_client_unsubscribe**(*gsm_mqtt_client_p client*, **const** char **topic*, void **arg*)

Unsubscribe from MQTT topic.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic name to unsubscribe from
- `[in] arg`: User custom argument used in callback

gsmr_t **gsm_mqtt_client_publish**(*gsm_mqtt_client_p client*, **const** char **topic*, **const** void **payload*, uint16_t *len*, gsm_mqtt_qos_t *qos*, uint8_t *retain*, void **arg*)

Publish a new message on specific topic.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic to send message to
- `[in] payload`: Message data
- `[in] payload_len`: Length of payload data
- `[in] qos`: Quality of service. This parameter can be a value of *gsm_mqtt_qos_t* enumeration
- `[in] retain`: Retian parameter value
- `[in] arg`: User custom argument used in callback

void ***gsm_mqtt_client_get_arg**(*gsm_mqtt_client_p client*)

Get user argument on client.

Return User argument

Parameters

- `[in] client`: MQTT client handle

void **gsm_mqtt_client_set_arg**(*gsm_mqtt_client_p client*, void **arg*)

Set user argument on client.

Parameters

- `[in] client`: MQTT client handle
- `[in] arg`: User argument

struct gsm_mqtt_client_info_t

#include <*gsm_mqtt_client.h*> MQTT client information structure.

Public Members

const char ***id**
Client unique identifier. It is required and must be set by user

const char ***user**
Authentication username. Set to `NULL` if not required

const char ***pass**
Authentication password, set to `NULL` if not required

uint16_t **keep_alive**
Keep-alive parameter in units of seconds. When set to `0`, functionality is disabled (not recommended)

const char ***will_topic**
Will topic

const char ***will_message**
Will message

gsm_mqtt_qos_t **will_qos**
Will topic quality of service

struct gsm_mqtt_request_t
#include <gsm_mqtt_client.h> MQTT request object.

Public Members

uint8_t **status**
Entry status flag for in use or pending bit

uint16_t **packet_id**
Packet ID generated by client on publish

void ***arg**
User defined argument

uint32_t **expected_sent_len**
Number of total bytes which must be sent on connection before we can say "packet was sent".

uint32_t **timeout_start_time**
Timeout start time in units of milliseconds

struct gsm_mqtt_evt_t
#include <gsm_mqtt_client.h> MQTT event structure for callback function.

Public Members

gsm_mqtt_evt_type_t **type**
Event type

gsm_mqtt_conn_status_t **status**
Connection status with MQTT

struct *gsm_mqtt_evt_t*::**[anonymous]**::**[anonymous] connect**
Event for connecting to server

uint8_t **is_accepted**
Status if client was accepted to MQTT prior disconnect event

struct *gsm_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **disconnect**
Event for disconnecting from server

void ***arg**
User argument for callback function

gsmr_t **res**
Rgsmonse status

struct *gsm_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **sub_unsub_scribed**
Event for (un)subscribe to/from topics

struct *gsm_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **publish**
Published event

const uint8_t ***topic**
Pointer to topic identifier

size_t **topic_len**
Length of topic

const void ***payload**
Topic payload

size_t **payload_len**
Length of topic payload

uint8_t **dup**
Duplicate flag if message was sent again

gsm_mqtt_qos_t **qos**
Received packet quality of service

struct *gsm_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **publish_recv**
Publish received event

union *gsm_mqtt_evt_t*::**[anonymous]** **evt**
Event data parameters

group **GSM_APP_MQTT_CLIENT_EVT**
Event helper functions.

Connect event

Note Use these functions on *GSM_MQTT_EVT_CONNECT* event

gsm_mqtt_client_evt_connect_get_status (client, evt)
Get connection status.

Return Connection status. Member of *gsm_mqtt_conn_status_t*

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Disconnect event

Note Use these functions on *GSM_MQTT_EVT_DISCONNECT* event

gsm_mqtt_client_evt_disconnect_is_accepted(client, evt)
Check if MQTT client was accepted by server when disconnect event occurred.

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Subscribe/unsubscribe event

Note Use these functions on *GSM_MQTT_EVT_SUBSCRIBE* or *GSM_MQTT_EVT_UNSUBSCRIBE* events

gsm_mqtt_client_evt_subscribe_get_argument(client, evt)
Get user argument used on *gsm_mqtt_client_subscribe*.

Return User argument

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_subscribe_get_result(client, evt)
Get result of subscribe event.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_unsubscribe_get_argument(client, evt)
Get user argument used on *gsm_mqtt_client_unsubscribe*.

Return User argument

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_unsubscribe_get_result(client, evt)
Get result of unsubscribe event.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT client

- `[in] evt`: Event handle

Publish receive event

Note Use these functions on *GSM_MQTT_EVT_PUBLISH_RECV* event

gsm_mqtt_client_evt_publish_recv_get_topic(client, evt)
Get topic from received publish packet.

Return Topic name

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_recv_get_topic_len(client, evt)
Get topic length from received publish packet.

Return Topic length

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_recv_get_payload(client, evt)
Get payload from received publish packet.

Return Packet payload

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_recv_get_payload_len(client, evt)
Get payload length from received publish packet.

Return Payload length

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_recv_is_duplicate(client, evt)
Check if packet is duplicated.

Return `1` if duplicated, `0` otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_recv_get_qos (client, evt)
Get received quality of service.

Return Member of *gsm_mqtt_qos_t* enumeration

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Publish event

Note Use these functions on *GSM_MQTT_EVT_PUBLISH* event

gsm_mqtt_client_evt_publish_get_argument (client, evt)
Get user argument used on *gsm_mqtt_client_publish*.

Return User argument

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

gsm_mqtt_client_evt_publish_get_result (client, evt)
Get result of publish event.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Defines

gsm_mqtt_client_evt_get_type (client, evt)
Get MQTT event type.

Return MQTT Event type, value of *gsm_mqtt_evt_type_t* enumeration

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

MQTT Client API

MQTT Client API provides sequential API built on top of *MQTT Client*.

Listing 20: MQTT API application example code

```
/*
 * MQTT client API example with GSM device.
 *
 * Once device is connected to network,
 * it will try to connect to mosquitto test server and start the MQTT.
 *
 * If successfully connected, it will publish data to "gsm_mqtt_topic" topic every x seconds.
 *
 * To check if data are sent, you can use mqtt-spy PC software to inspect
 * test.mosquitto.org server and subscribe to publishing topic
 */

#include "gsm/apps/gsm_mqtt_client_api.h"
#include "mqtt_client_api.h"
#include "gsm/gsm_mem.h"
#include "gsm/gsm_network_api.h"

/**
 * \brief           Connection information for MQTT CONNECT packet
 */
static const gsm_mqtt_client_info_t
mqtt_client_info = {
    .keep_alive = 10,

    /* Server login data */
    .user = "8a215f70-a644-11e8-ac49-e932ed599553",
    .pass = "26aa943f702e5e780f015cd048a91e8fb54cca28",

    /* Device identifier address */
    .id = "2c3573a0-0176-11e9-a056-c5cffe7f75f9",
};

/**
 * \brief           Memory for temporary topic
 */
static char
mqtt_topic_str[256];

/**
 * \brief           Generate random number and write it to string
 * \param[out]      str: Output string with new number
 */
void
generate_random(char* str) {
    static uint32_t random_beg = 0x8916;
    random_beg = random_beg * 0x00123455 + 0x85654321;
    sprintf(str, "%u", (unsigned)((random_beg >> 8) & 0xFFFF));
}

/**
 * \brief           MQTT client API thread
```

(continues on next page)

(continued from previous page)

```
 */
void
mqtt_client_api_thread(void const* arg) {
    gsm_mqtt_client_api_p client;
    gsm_mqtt_conn_status_t conn_status;
    gsm_mqtt_client_api_buf_p buf;
    gsmr_t res;
    char random_str[10];

    /* Request network attach */
    while (gsm_network_request_attach() != gsmOK) {
        gsm_delay(1000);
    }

    /* Create new MQTT API */
    client = gsm_mqtt_client_api_new(256, 128);
    if (client == NULL) {
        goto terminate;
    }

    while (1) {
        /* Make a connection */
        printf("Joining MQTT server\r\n");

        /* Try to join */
        conn_status = gsm_mqtt_client_api_connect(client, "mqtt.mydevices.com", 1883,␣
↪&mqtt_client_info);
        if (conn_status == GSM_MQTT_CONN_STATUS_ACCEPTED) {
            printf("Connected and accepted!\r\n");
            printf("Client is ready to subscribe and publish to new messages\r\n");
        } else {
            printf("Connect API response: %d\r\n", (int)conn_status);
            gsm_delay(5000);
            continue;
        }

        /* Subscribe to topics */
        sprintf(mqtt_topic_str, "v1/%s/things/%s/cmd/#", mqtt_client_info.user, mqtt_
↪client_info.id);
        if (gsm_mqtt_client_api_subscribe(client, mqtt_topic_str, GSM_MQTT_QOS_AT_
↪LEAST_ONCE) == gsmOK) {
            printf("Subscribed to topic\r\n");
        } else {
            printf("Problem subscribing to topic!\r\n");
        }

        while (1) {
            /* Receive MQTT packet with 1000ms timeout */
            res = gsm_mqtt_client_api_receive(client, &buf, 5000);
            if (res == gsmOK) {
                if (buf != NULL) {
                    printf("Publish received!\r\n");
                    printf("Topic: %s, payload: %s\r\n", buf->topic, buf->payload);
                    gsm_mqtt_client_api_buf_free(buf);
                    buf = NULL;
                }
            } else if (res == gsmCLOSED) {
```

(continues on next page)

(continued from previous page)

```
                printf("MQTT connection closed!\r\n");
                break;
            } else if (res == gsmTIMEOUT) {
                printf("Timeout on MQTT receive function. Manually publishing.\r\n");

                /* Publish data on channel 1 */
                generate_random(random_str);
                sprintf(mqtt_topic_str, "v1/%s/things/%s/data/1", mqtt_client_info.user, mqtt_client_info.id);
                gsm_mqtt_client_api_publish(client, mqtt_topic_str, random_str, strlen(random_str), GSM_MQTT_QOS_AT_LEAST_ONCE, 0);
            }
        }
        goto terminate;
    }

terminate:
    gsm_mqtt_client_api_delete(client);
    gsm_network_request_detach();
    printf("MQTT client thread terminate\r\n");
    gsm_sys_thread_terminate(NULL);
}
```

group **GSM_APP_MQTT_CLIENT_API**

Sequential, single thread MQTT client API.

Typedefs

typedef struct gsm_mqtt_client_api_buf ***gsm_mqtt_client_api_buf_p**

Pointer to *gsm_mqtt_client_api_buf_t* structure.

Functions

gsm_mqtt_client_api_p **gsm_mqtt_client_api_new**(size_t *tx_buff_len*, size_t *rx_buff_len*)

Create new MQTT client API.

Return Client handle on success, `NULL` otherwise

Parameters

- `[in] tx_buff_len`: Maximal TX buffer for maximal packet length
- `[in] rx_buff_len`: Maximal RX buffer

void **gsm_mqtt_client_api_delete**(gsm_mqtt_client_api_p *client*)

Delete client from memory.

Parameters

- `[in] client`: MQTT API client handle

gsm_mqtt_conn_status_t **gsm_mqtt_client_api_connect**(gsm_mqtt_client_api_p *client*, **const** char **host*, *gsm_port_t port*, **const** *gsm_mqtt_client_info_t* **info*)

Connect to MQTT broker.

Return *GSM_MQTT_CONN_STATUS_ACCEPTED* on success, member of *gsm_mqtt_conn_status_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] host`: TCP host
- `[in] port`: TCP port
- `[in] info`: MQTT client info

gsmr_t **gsm_mqtt_client_api_close**(gsm_mqtt_client_api_p *client*)

Close MQTT connection.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle

gsmr_t **gsm_mqtt_client_api_subscribe**(gsm_mqtt_client_api_p *client*, **const** char **topic*, gsm_mqtt_qos_t *qos*)

Subscribe to topic.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to subscribe on
- `[in] qos`: Quality of service. This parameter can be a value of *gsm_mqtt_qos_t*

gsmr_t **gsm_mqtt_client_api_unsubscribe**(gsm_mqtt_client_api_p *client*, **const** char **topic*)

Unsubscribe from topic.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to unsubscribe from

gsmr_t **gsm_mqtt_client_api_publish**(gsm_mqtt_client_api_p *client*, **const** char **topic*, **const** void **data*, size_t *btw*, gsm_mqtt_qos_t *qos*, uint8_t *retain*)

Publish new packet to MQTT network.

Return *gsmOK* on success, member of *gsmr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to publish on
- `[in] data`: Data to send
- `[in] btw`: Number of bytes to send for data parameter
- `[in] qos`: Quality of service. This parameter can be a value of *gsm_mqtt_qos_t*
- `[in] retain`: Set to `1` for retain flag, `0` otherwise

uint8_t **gsm_mqtt_client_api_is_connected**(gsm_mqtt_client_api_p *client*)
Check if client MQTT connection is active.

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT API client handle

gsmr_t **gsm_mqtt_client_api_receive**(gsm_mqtt_client_api_p *client*, *gsm_mqtt_client_api_buf_p* **p*, uint32_t *timeout*)
Receive next packet in specific timeout time.

Note This function can be called from separate thread than the rest of API function, which allows you to handle receive data separated with custom timeout

Return *gsmOK* on success, *gsmCLOSED* if MQTT is closed, *gsmTIMEOUT* on timeout

Parameters

- `[in] client`: MQTT API client handle
- `[in] p`: Pointer to output buffer
- `[in] timeout`: Maximal time to wait before function returns timeout

void **gsm_mqtt_client_api_buf_free**(*gsm_mqtt_client_api_buf_p* *p*)
Free buffer memory after usage.

Parameters

- `[in] p`: Buffer to free

struct gsm_mqtt_client_api_buf_t
#include <*gsm_mqtt_client_api.h*> MQTT API RX buffer.

Public Members

char ***topic**
Topic data

size_t **topic_len**
Topic length

uint8_t ***payload**
Payload data

size_t **payload_len**
Payload length

gsm_mqtt_qos_t **qos**
Quality of service

Netconn API

Netconn API is addon on top of existing connection module and allows sending and receiving data with sequential API calls, similar to *POSIX socket* API.

It can operate in client mode and uses operating system features, such as message queues and semaphore to link non-blocking callback API for connections with sequential API for application thread.

Note: Connection API does not directly allow receiving data with sequential and linear code execution. All is based on connection event system. Netconn adds this functionality as it is implemented on top of regular connection API.

Warning: Netconn API are designed to be called from application threads ONLY. It is not allowed to call any of *netconn API* functions from within interrupt or callback event functions.

Netconn client

Fig. 8: Netconn API client block diagram

Above block diagram shows basic architecture of netconn client application. There is always one application thread (in green) which calls *netconn API* functions to interact with connection API in synchronous mode.

Every netconn connection uses dedicated structure to handle message queue for data received packet buffers. Each time new packet is received (red block, *data received event*), reference to it is written to message queue of netconn structure, while application thread reads new entries from the same queue to get packets.

Listing 21: Netconn client example

```
#include "netconn_client.h"
#include "gsm/gsm.h"
#include "gsm/gsm_network_api.h"

#if GSM_CFG_NETCONN

/**
 * \brief           Host and port settings
 */
#define NETCONN_HOST        "example.com"
#define NETCONN_PORT        80

/**
 * \brief           Request header to send on successful connection
 */
static const char
request_header[] = ""
"GET / HTTP/1.1\r\n"
"Host: " NETCONN_HOST "\r\n"
"Connection: close\r\n"
```

(continues on next page)

(continued from previous page)

```
"\r\n";

/**
 * \brief           Netconn client thread implementation
 * \param[in]       arg: User argument
 */
void
netconn_client_thread(void const* arg) {
    gsmr_t res;
    gsm_pbuf_p pbuf;
    gsm_netconn_p client;
    gsm_sys_sem_t* sem = (void *)arg;

    /* Request attach to network */
    while (gsm_network_request_attach() != gsmOK) {
        gsm_delay(1000);
    }

    /*
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept received packet buffers
     */
    client = gsm_netconn_new(GSM_NETCONN_TYPE_TCP);
    if (client != NULL) {
        /*
         * Connect to external server as client
         * with custom NETCONN_CONN_HOST and CONN_PORT values
         *
         * Function will block thread until we are successfully connected (or not) to server
         */
        res = gsm_netconn_connect(client, NETCONN_HOST, NETCONN_PORT);
        if (res == gsmOK) {                     /* Are we successfully connected? */
            printf("Connected to " NETCONN_HOST "\r\n");
            res = gsm_netconn_write(client, request_header, sizeof(request_header) - 1);    /* Send data to server */
            if (res == gsmOK) {
                res = gsm_netconn_flush(client);/* Flush data to output */
            }
            if (res == gsmOK) {                 /* Were data sent? */
                printf("Data were successfully sent to server\r\n");

                /*
                 * Since we sent HTTP request,
                 * we are expecting some data from server
                 * or at least forced connection close from remote side
                 */
                do {
                    /*
                     * Receive single packet of data
                     *
                     * Function will block thread until new packet
                     * is ready to be read from remote side
                     *
                     * After function returns, don't forgot the check value.
                     * Returned status will give you info in case connection
```

(continues on next page)

(continued from previous page)

```
                     * was closed too early from remote side
                     */
                    res = gsm_netconn_receive(client, &pbuf);
                    if (res == gsmCLOSED) {     /* Was the connection closed? This can be checked by return status of receive function */
                        printf("Connection closed by remote side...\r\n");
                        break;
                    } else if (res == gsmTIMEOUT) {
                        printf("Netconn timeout while receiving data. You may try multiple readings before deciding to close manually\r\n");
                    }

                    if (res == gsmOK && pbuf != NULL) { /* Make sure we have valid packet buffer */
                        /*
                         * At this point read and manipulate
                         * with received buffer and check if you expect more data
                         *
                         * After you are done using it, it is important
                         * you free the memory otherwise memory leaks will appear
                         */
                        printf("Received new data packet of %d bytes\r\n", (int)gsm_pbuf_length(pbuf, 1));
                        gsm_pbuf_free(pbuf);    /* Free the memory after usage */
                        pbuf = NULL;
                    }
                } while (1);
            } else {
                printf("Error writing data to remote host!\r\n");
            }

            /*
             * Check if connection was closed by remote server
             * and in case it wasn't, close it manually
             */
            if (res != gsmCLOSED) {
                gsm_netconn_close(client);
            }
        } else {
            printf("Cannot connect to remote host %s:%d!\r\n", NETCONN_HOST, NETCONN_PORT);
        }
        gsm_netconn_delete(client);             /* Delete netconn structure */
    }
    gsm_network_request_detach();               /* Detach from network */

    if (gsm_sys_sem_isvalid(sem)) {
        gsm_sys_sem_release(sem);
    }
    gsm_sys_thread_terminate(NULL);             /* Terminate current thread */
}

#endif /* GSM_CFG_NETCONN */
```

Non-blocking receive

By default, netconn API is written to only work in separate application thread, dedicated for network connection processing. Because of that, by default every function is fully blocking. It will wait until result is ready to be used by application.

It is, however, possible to enable timeout feature for receiving data only. When this feature is enabled, *gsm_netconn_receive()* will block for maximal timeout set with *gsm_netconn_set_receive_timeout()* function.

When enabled, if there is no received data for timeout amount of time, function will return with timeout status and application needs to process it accordingly.

Tip: *GSM_CFG_NETCONN_RECEIVE_TIMEOUT* must be set to `1` to use this feature.

---

group **GSM_NETCONN**
Network connection.

Typedefs

typedef struct gsm_netconn ***gsm_netconn_p**
Netconn object structure.

Enums

enum gsm_netconn_type_t
Netconn connection type.

Values:

GSM_NETCONN_TYPE_TCP = GSM_CONN_TYPE_TCP
TCP connection

GSM_NETCONN_TYPE_UDP = GSM_CONN_TYPE_UDP
UDP connection

GSM_NETCONN_TYPE_SSL = GSM_CONN_TYPE_SSL
TCP connection over SSL

Functions

gsm_netconn_p **gsm_netconn_new**(gsm_netconn_type_t *type*)
Create new netconn connection.

Return New netconn connection on success, `NULL` otherwise

Parameters

- `[in] type`: Netconn connection type

gsmr_t **gsm_netconn_delete**(*gsm_netconn_p nc*)
Delete netconn connection.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle

gsmr_t **gsm_netconn_connect** (*gsm_netconn_p nc*, **const** char **host*, *gsm_port_t port*)
Connect to server as client.

Return *gsmOK* if successfully connected, member of *gsmr_t* otherwise

Parameters

- `[in] nc`: Netconn handle
- `[in] host`: Pointer to host, such as domain name or IP address in string format
- `[in] port`: Target port to use

gsmr_t **gsm_netconn_receive** (*gsm_netconn_p nc*, *gsm_pbuf_p *pbuf*)
Receive data from connection.

Return *gsmOK* when new data ready,

Return *gsmCLOSED* when connection closed by remote side,

Return *gsmTIMEOUT* when receive timeout occurs

Return Any other member of *gsmr_t* otherwise

Parameters

- `[in] nc`: Netconn handle used to receive from
- `[in] pbuf`: Pointer to pointer to save new receive buffer to. When function returns, user must check for valid pbuf value `pbuf != NULL`

gsmr_t **gsm_netconn_close** (*gsm_netconn_p nc*)
Close a netconn connection.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle to close

int8_t **gsm_netconn_getconnnum** (*gsm_netconn_p nc*)
Get connection number used for netconn.

Return `-1` on failure, connection number between `0` and *GSM_CFG_MAX_CONNS* otherwise

Parameters

- `[in] nc`: Netconn handle

void **gsm_netconn_set_receive_timeout** (*gsm_netconn_p nc*, uint32_t *timeout*)
Set timeout value for receiving data.

When enabled, *gsm_netconn_receive* will only block for up to *timeout* value and will return if no new data within this time

Parameters

- `[in] nc`: Netconn handle

- `[in] timeout`: Timeout in units of milliseconds. Set to `0` to disable timeout for *gsm_netconn_receive* function

uint32_t **gsm_netconn_get_receive_timeout** (*gsm_netconn_p nc*)
Get netconn receive timeout value.

Return Timeout in units of milliseconds. If value is `0`, timeout is disabled (wait forever)

Parameters

- `[in] nc`: Netconn handle

gsmr_t **gsm_netconn_write** (*gsm_netconn_p nc*, **const** void **data*, size_t *btw*)
Write data to connection output buffers.

Note This function may only be used on TCP or SSL connections

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to write data to
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

gsmr_t **gsm_netconn_flush** (*gsm_netconn_p nc*)
Flush buffered data on netconn *TCP/SSL* connection.

Note This function may only be used on *TCP/SSL* connection

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle to flush data

gsmr_t **gsm_netconn_send** (*gsm_netconn_p nc*, **const** void **data*, size_t *btw*)
Send data on *UDP* connection to default IP and port.

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to send
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

gsmr_t **gsm_netconn_sendto** (*gsm_netconn_p nc*, **const** *gsm_ip_t *ip*, *gsm_port_t port*, **const** void **data*, size_t *btw*)
Send data on *UDP* connection to specific IP and port.

Note Use this function in case of UDP type netconn

Return *gsmOK* on success, member of *gsmr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to send

- `[in] ip`: Pointer to IP address
- `[in] port`: Port number used to send data
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

5.4 Examples and demos

Various examples are provided for fast library evaluation on embedded systems. These are optimized prepared and maintained for `2` platforms, but could be easily extended to more platforms:

- WIN32 examples, prepared as Visual Studio Community projects
- ARM Cortex-M examples for STM32, prepared as STM32CubeIDE GCC projects

Warning: Library is platform independent and can be used on any platform.

5.4.1 Example architectures

There are many platforms available today on a market, however supporting them all would be tough task for single person. Therefore it has been decided to support (for purpose of examples) `2` platforms only, *WIN32* and *STM32*.

WIN32

Examples for *WIN32* are prepared as Visual Studio Community projects. You can directly open project in the IDE, compile & debug.

Application opens *COM* port, set in the low-level driver. External USB to UART converter (FTDI-like device) is necessary in order to connect to *GSM* device.

Note: *GSM* device is connected with *USB to UART converter* only by *RX* and *TX* pins.

Device driver is located in `/gsm_at_lib/src/system/gsm_ll_win32.c`

STM32

Embedded market is supported by many vendors and STMicroelectronics is, with their STM32 series of microcontrollers, one of the most important players. There are numerous amount of examples and topics related to this architecture.

Examples for *STM32* are natively supported with STM32CubeIDE, an official development IDE from STMicroelectronics.

You can run examples on one of official development boards, available in repository examples.

Table 3: Supported development boards

Board name	GSM settings				Debug settings		
	UART	MTX	MRX	RST	UART	MDTX	MDRX
STM32F429ZI-Nucleo	USART6	PC6	PC7	PC5	USART3	PD8	PD9

Pins to connect with GSM device:

- *MTX*: MCU TX pin, connected to GSM RX pin
- *MRX*: MCU RX pin, connected to GSM TX pin
- *RST*: MCU output pin to control reset state of GSM device

Other pins are for your information and are used for debugging purposes on board.

- MDTX: MCU Debug TX pin, connected via on-board ST-Link to PC
- MDRX: MCU Debug RX pin, connected via on-board ST-Link to PC
- Baudrate is always set to `921600` bauds

5.4.2 Examples list

Here is a list of all examples coming with this library.

Tip: Examples are located in `/examples/` folder in downloaded package. Check *Download library* section to get your package.

Tip: Do not forget to set PIN & PUK codes of your SIM card before running any of examples. Open `/snippets/sim_manager.c` and update `pin_code` and `puk_code` variables.

Device info

Simple example which prints basic device information:

- Device Manufacturer
- Device Model
- Device serial number
- Device revision number

MQTT Client API

Similar to *MQTT Client* examples, but it uses separate thread to process events in blocking mode. Application does not use events to process data, rather it uses blocking API to receive packets

Netconn client

Netconn client is based on sequential API. It starts connection to server, sends initial request and then waits to receive data.

Processing is in separate thread and fully sequential, no callbacks or events.

Call

Call example answers received call. If GSM device supports calls and has microphone/speaker connected to module itself, it can simply communicate over voice.

Call & SMS

This example shows how to receive a call and send reply with SMS. When application receives call, it hangs-up immediately and sends back SMS asking caller to send SMS instead.

When application receives SMS, it will send same SMS content back to the sender's number.

SMS Send receive

It demonstrates sending and receiving SMS either in events or using thread processing.

INDEX

G

H

I

K

L

M

N

O

P

Q