LibGSM

Version 1.0

User Manual

eitech.systems

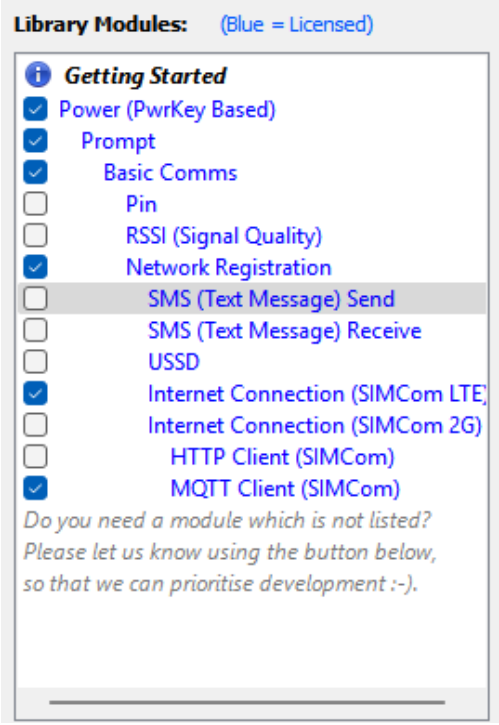
Contents

Start

Quick Start

This section can be used to get a LibGSM project up and running as quick as possible.

First select the required modules from the "Library Modules" list. Note that dependant modules will be selected automatically when the "Generate Code" button is clicked. Also note that some modules require a license to have been purchased before they can be used.



Now select whether you would like to generate code for a C or C++ application by selecting the relevant project type.

Once the correct modules have been selected and the project type has been selected, click "Save As" to select the output directory and save the current settings. The output directory would normally be the root directory of your firmware project (Arduino, STM32, ESP32, etc). The firmware project can be created before or after LibGSM code generation, using your usual tool for that platform.

At this point you are able to generate your code and begin your project, however it is recommended to at least read until the end of this section as well as the Library Architecture and Library Requirements sections below. In general, the following further steps are required:

- Provide a 1ms timer source to the gsm__SysTime_ms() function (e.g. "millis()") in gsm_manager.c[pp] (you will usually need to add a header to the "includes" section at the top of gsm_manager.c[pp] as well, e.g. #include <Arduino.h>).

- Fill in the HAL (Hardware Abstraction Layer) functions in gsm_manager.c[pp] (UART, GPIOs) in order to connect LibGSM to the modem hardware.
- Call gsmmgrPoll() regularly from your main program loop (you will also need to add an include for gsm_manager.h to your main project file, e.g. #include "gsm_manager.h" at the top of your main.c[pp] file).
- Handle events in the relevant functions in gsm_manager.c[pp] (e.g. gsmSmsReadEventHandler for the SMS Read module), and add calls from your application code to initiate actions in the GSM Library (e.g. gsmmgrSmsSend for the SMS Send module).

Each module (when clicked on in the "Library Modules" list) provides a "Module Info" tab which provides extra information regarding that module and its use case. Please refer to this information when help is required.

There are a number of modules that provide additional settings which allow the user to customise their project based on their requirements. For example, when clicking on the "Prompt" module, a settings tab will be available which allows the user to define the size of the Prompt buffer (among other settings).

If guidance is required to get your project up and running, please refer to the example projects. Example projects are provided for user interactable modules. To download these examples, select the required module from the "Library Modules" list and click on the "Example Projects" tab and then the "Download Example" button to begin the download.

Library Architecture

The architecture of LibGSM is modular, which offers a number of advantages:

- Library features (modules) which are not required in a specific project can be left out for greater efficiency.
- Modules can be slotted in and out; for example, some modules may work with all GSM modems whilst others may be modem-specific and only included if a particular modem is used.

Many modules are dependent on other modules – for example, most library modules are dependent on a Power module to confirm that the modem is "on" before they will try to communicate with the modem.

The various library modules are usually tied together in a central controller file, normally called gsm_manager.c or gsm_manager.cpp. Library modules may require external routines to interface with hardware (this is called a "Hardware Abstraction Layer") and/or to alert the host application to "events" (such as when an MQTT message is received).

Library Requirements

In order to ensure that the LibGSM library runs successfully, the user must ensure the following:

- A 1ms timer must be available for the gsm__SysTime_ms() function. This function outline can be seen below *(this user code section will be auto-generated when using the LibGSM Code Generator)*.
- The gsmmgrPoll() routine must be called regularly. This can be done by including this polling routine in your main loop.
- Avoid "blocking" code. This means that blocking delays should preferably not be used in your code, as LibGSM will not be able to run during the blocking delay. Instead, a state machine or other non-blocking coding technique (such as an RTOS) can be used. gsm__SysTime_ms() (or it's timekeeping source) can be used in a state machine format to ensure that no code is blocked while waiting for a time period to pass. Traditional delays can also be replaced by an alternative method which calls gsmmgrPoll() regularly, such as in the example code provided below.
- Traditional "blocking" UART communication should be avoided. Note that the LibGSM Code Generator can be used to enable (enabled by default) a non-blocking UART debug output. This code can be adapted as required. The LibGSM Code Generator can also be used to setup non-blocking Tx and Rx buffers for communication with the GSM modem (see the "Prompt" module); alternatively, you can set up DMA buffers depending on your hardware.
- Certain modules will require a HAL ("Hardware Abstraction Layer") interface to operate successfully. For example, the "Power" module will need to have access to the GSM modem power pin (normally referred to as "PWRKEY" by the modem documentation) in order to start the GSM modem *(user code sections for these HAL interfaces are auto-generated when using the LibGSM Code Generator)*.

gsm__SysTime_ms() Routine

```
uint32_t gsm__SysTime_ms() {
    // USER CODE BEGIN SysTime_ms

    // ToDo: Return system time in milliseconds. E.g.:
    // return millis();

    // USER CODE END SysTime_ms
}
```

Non-blocking delay example code

```
void delay_ms_ex(uint32_t milliseconds) {
    uint32_t u32StartTS;
    u32StartTS = gsm__SysTime_ms();
    while (gsm__SysTime_ms() - u32StartTS < milliseconds) {
        gsmmgrPoll();
    }
}
```

Module Information

As stated in the "Quick Start" section, each module has an information tab. To access this tab, the module can be selected in the "Library Module" list. This will display the "Module Info" in the main view of the application.

The information displayed will be specific to that module only and allows the user to easily find all the relevant information.

This tab will provide a module overview and detailed explanations regarding module dependencies, module initialisation, module polling, callable functions, event handlers, and in some cases a workflow summary is also provided.

Module Dependencies

In short, module dependencies are library modules that are required to ensure that the selected module can operate correctly.

For example, if you would like to use the "SMS Send" module, the GSM modem must be powered on, communication needs to be established with this modem and the modem must be registered on a network. Therefore, the relevant modules (Power, Prompt, Basic Comms and Network Registration) are required.

A list of all the module dependencies can be found in the "Module Info" tab for each module. Alternatively, you can select the required module(s) in the "Library Module" list (ensure modules not required are not selected) and click on the "Check Module Dependencies" button. This will automatically select all the dependant modules.

Note that when clicking "Generate Code", the LibGSM Code Generator application will automatically include any dependant modules, assuming your license includes these modules.

User Code Sections

The code generated for the user will include "USER CODE BEGIN" and "USER CODE END" sections. For example, the "USER CODE BEGIN Includes" and "USER CODE END Includes" sections.

Any code that is between the "BEGIN" and "END" sections will remain intact if the LibGSM Code Generator is used to generate new code again (assuming the same project directory is used).

User code that is not between these sections will be lost if new code is generated again. Please note that if these sections are removed or edited, then the user code cannot be included if the LibGSM Code Generator is used to generate new code again.

Debug Output (UART)

The LibGSM Code Generator can be used to enable a non-blocking debug UART output. This setting can be found by selecting "Getting Started" in the "Library Modules" list and

clicking on the “Settings [Getting Started]” tab where you can enable the debug output buffer.

This setting will include the buffers and routines required to use a non-blocking UART debug output. Note that the routines include commented out example code which can be implemented to finalise the debug output.

The example projects (more information in the section below) that are provided with the library include the use of this debug UART output and can be used as a reference when setting up your project.

Note that the debug UART output is enabled by default.

Example Projects

Example projects are provided for user interactable modules. These examples can be found by selecting the required module from the “Library Modules” list and then clicking on the “Example Projects” tab. Please click the “Download Example” button to begin the download process.

Each one of these modules provide an example for both an Arduino project as well as an STM32CubeIDE project, but can easily be adapted for any C++ or C application from these examples.

Module Settings

When scrolling through the modules in the “Library Modules” list, you may notice that certain modules will have a “Settings” tab available. These settings can be updated to suit your application.

The details for each of these settings are too extensive to be covered in this section. For more information regarding each setting, please refer to the “Settings” and “Module Info” tabs for the respective module.

Note that module settings can be saved and reused when required. More on this below.

Save Settings

In order to minimise the setup time required each time new code needs to be generated, an option to save your module settings is built into the LibGSM Code Generator. This allows you to save your settings and pick up where you left off.

You can save the current settings by clicking “Save As” and selecting your directory of choice. Note that if a directory has already been selected, and is displayed in the directory bar, you can simply click the “Save” button.

Note that settings will automatically be saved when generating code (“Generate Code” button).

Load Settings

To pick up where you have left off, or to use settings from a previous project, you can open the project directory in question by clicking the "Open" button. This will load all the settings that were previously saved for this project.

Project Type

The LibGSM Code Generator allows you to select between a C or C++ project type. This will ensure that the generated files are compatible with your C or C++ project.

Note that when selecting a C++ project type, a check box will appear to allow the user to generate an Arduino compatible project structure by including the library modules in a "src" folder. If the "src" folder is not required, simply unselect the "Arduino Project" check box.

The library files will be included in a LibGSM folder regardless of the project type selection.

Licensing

Some modules require a license to have been purchased before they can be used. A module will be shown in blue if that module is included in the current license. Note that example projects (where applicable) are included in the module license.

The "SMS (Text Message) Receive" module is provided as a "free-to-use" module which allows you to test library functionality. Any dependant module required to use the "SMS (Text Message) Receive" module will be included when generating your code.

Power Module (PwrKey Based)

Summary

The Power module is responsible for monitoring and controlling the power-on state of the GSM modem. It can power the modem on, power it off, or restart it.

The "PwrKey Based" Power module works with most modems (SIMCom, uBlox, Quectel) and uses 2 GPIO lines: "PwrKey" and "Status". An optional "Reset" line can also be connected as a last-resort failsafe. Some modems may not use the "PwrKey" technique, and a different library module is required to work with these.

Hardware Abstraction Layer (HAL)

The "PwrKey" GPIO is a modem input and acts as a "button" which is used to power the modem on or off.

The "Status" GPIO is a modem output which indicates whether the modem is powered on or off.

The "Reset" GPIO (optional) is a modem input which hard-resets the modem (without proper shutdown).

The GPIOs are interfaced with via Callback functions; the functions are auto-generated if using the LibGSM Code Generator, however the implementation inside the function must be completed depending on the specific MCU / compiler.

Module Dependencies

Power module dependencies are listed below:

- None

Additional Information

By default, the Power module will power the modem "on" at startup.

Other library modules may request the Power module to restart the modem if they believe that the modem may have hung / gotten stuck.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_Power_Instance GSM_Power_Instance;
struct __GSM_Power_Instance {
    void    (*SetPowerKey)(GSM_Power_Instance* inst, bool activated);
    void    (*SetReset)(GSM_Power_Instance* inst, bool activated);
    bool    (*ReadStatus)(GSM_Power_Instance* inst);
};
```

Set the "SetPowerKey" and "ReadStatus" callbacks, as well as optionally the "SetReset" callback (or set it to 0 otherwise), then call the initialisation routine.

```
void gsmPowerInit(GSM_Power_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmPowerPoll(GSM_Power_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmPowerGet

Get the power status of the modem (on or off).

```
bool gsmPowerGet(GSM_Power_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrPowerGet();
```

gsmPowerSet

Set the power status of the modem (on or off).

```
void gsmPowerSet(GSM_Power_Instance* inst, bool powered_on);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrPowerSet(bool powered_on);
```

gsmPowerSetPowerKeyPressDurations

gsmPowerSetPowerKeyPressDurations can optionally be called to customise the duration of the "PwrKey" button presses for turning the modem on/off, if a particular modem requires specific timing.

```
void gsmPowerSetPowerKeyPressDurations(GSM_Power_Instance* inst, uint16_t
                                       powerOnPressTime_ms, uint16_t
                                       powerOffPressTime_ms);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrPowerSetPowerKeyPressDurations(uint16_t powerOnPressTime_ms,
                                          uint16_t powerOffPressTime_ms);
```

Prompt Module

Summary

The Prompt module is responsible for facilitating serial communication with the modem.

Internally, the Prompt module delegates dependant modules' access to the modem, to ensure that only one module attempts to communicate with the modem at any given time.

Hardware Abstraction Layer (HAL)

The Prompt module needs to be able to send and receive serial UART data. In order to do this in a platform agnostic way (in a way which can work with a variety of compilers / hardware), it uses function pointers which the user must hook up to callback functions implementing the relevant code for their compiler / hardware.

The "Write" function is used to write a single byte.

The "WriteCompleted" function is used to confirm whether there is still serial data being transmitted on the hardware, or if the Tx is idle.

The "CanRead" function is used to confirm whether any serial data has been received and is available to be read.

The "Read" function is used to read a single byte of received serial data.

If using the LibGSM Code Generator then the functions will be auto-generated, however the implementation inside the function must be manually completed depending on the specific hardware / compiler. Commented-out example implementations are included in the auto-generated functions, and the example programs can also be checked for further example implementations.

The Prompt module also requires a data buffer ("DataBuffer") to store incoming data before it can be processed. It is up to the user to declare this buffer and decide it's size ("DataBufferSize"), depending on their use case and the available memory in their hardware. If, for example, the library is being used only to send and receive short text messages then a small buffer (e.g. 128 bytes) will suffice; if larger HTML or MQTT transactions are being performed however then a larger buffer may be desired (note however that most modules processing larger data are capable of "chunking" it, so the buffer does not necessarily need to be large enough to hold the full transaction in one go).

If "echo" is turned on in the modem (see the "Basic Comms" module), then an optional echo cancellation buffer ("EchoCancellationBuffer") can also be supplied, to enable the Prompt module to recognise and ignore data echoed back from the modem. The size of this buffer ("EchoCancellationBufferSize") is also up to the user, depending on their use case and the available memory in their hardware. If used, then the echo cancellation buffer should ideally be at least as large as the largest amount of data sent to the modem in a single transaction (the largest amount of data which the modem will echo back), however

note that modems generally do not echo back data payloads submitted to be sent via e.g. HTTP or MQTT, so the echo cancellation buffer generally does not need to be very large.

Module Dependencies

Prompt module dependencies are listed below:

- Power module.

Additional Information

Depending on your hardware, you may wish to implement additional buffers and/or DMA on your serial UART Rx and/or Tx. The LibGSM Code Generator can optionally implement additional circular buffers on both the Rx and Tx channels if desired. It is important that serial communication with the modem is reliable, and that data is not lost due to buffer overflow. Please see the example programs for example implementations.

The Prompt module may trigger a restart using the Power module interface if it detects that a library module has become hung (this should never happen, but it is a failsafe contingency).

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_Prompt_Instance GSM_Prompt_Instance;
struct __GSM_Prompt_Instance {
    GSM_Power_Interface* Power;
    uint8_t*    DataBuffer;
    uint32_t    DataBufferSize;
    void        (*Write)(GSM_Prompt_Instance* inst, uint8_t _data);
    bool        (*WriteCompleted)(GSM_Prompt_Instance* inst);
    bool        (*CanRead)(GSM_Prompt_Instance* inst);
    uint8_t     (*Read)(GSM_Prompt_Instance* inst);
    uint8_t*    EchoCancellationBuffer; // Optional
    uint32_t    EchoCancellationBufferSize;
};
```

Set the Power module interface reference (pointer).

Set the “DataBuffer” reference (pointer) as well as “DataBufferSize”.

Set the “Write”, “WriteCompleted”, “CanRead”, and “Read” callbacks.

If echo is turned on in the modem (see the “Basic Comms” module) then set the “EchoCancellationBuffer” reference (pointer) as well as “EchoCancellationBufferSize”.

Call the initialisation routine.

```
void gsmPromptInit(GSM_Prompt_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmPromptPoll(GSM_Prompt_Instance* inst);
```

Callable Functions

The Prompt module does not have any functions which can be optionally called by the host application.

Basic Comms Module

Summary

The Basic Comms module is responsible for confirming that communication with the modem is working. Other library modules will not attempt to communicate with the modem unless the Basic Comms module confirms that communication with the modem is working and that the modem is responsive.

Module Dependencies

Basic Comms module dependencies are listed below:

- Power module.
- Prompt module.

Additional Information

The Basic Comms module will setup the echo state of the modem to be either on or off, depending on the user requirement. Please refer to initialisation section below. Having the echo state on is useful for debugging, since only the modem Tx pin needs to be probed in this case to see full communications.

The Basic Comms module will trigger a restart using the Power module interface if communication with the modem cannot be established, or if communication to the modem is lost at any point (since the Basic Comms module will periodically check to confirm that the modem is still responsive).

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_BasicComms_Instance GSM_BasicComms_Instance;
struct __GSM_BasicComms_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    uint8_t EchoOn : 1;
};
```

Set the Power and Prompt module interface references (pointers), and define whether echoing of commands from the modem is wanted or not ("EchoOn"- 1 is equivalent to turning echo on), then call the initialisation routine.

```
void gsmBasicCommsInit(GSM_BasicComms_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmBasicCommsPoll(GSM_BasicComms_Instance* inst);
```

Callable Functions

The following function may optionally be called by the host application:

gsmBasicCommsCommunicationEstablished

Get the status of communication with the modem (established or not).

```
bool gsmBasicCommsCommunicationEstablished(GSM_BasicComms_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrBasicCommsCommunicationEstablished();
```

PIN Module

Summary

The PIN module is responsible for "unlocking" the SIM card that is inserted in the modem, if required.

Module Dependencies

PIN module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.

Additional Information

The PIN module will not enter the PIN for the SIM being used (even if the callable function is used) if the module has determined that the SIM is already "unlocked".

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_PIN_Instance GSM_PIN_Instance;
struct __GSM_PIN_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_BasicComms_Interface* BasicComms;
    void    (*EventHandler)(GSM_PIN_Instance* inst, GSM_PIN_EventType
                            event_type, void* event_data);
};
```

Set the Power, Prompt and Basic Comms module interface references (pointers).

Optionally set the "EventHandler" callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmPinInit(GSM_PIN_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmPinPoll(GSM_PIN_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmPinEnter

Trigger the process to enter the PIN code for the SIM card being used. Note that the PIN code to be used is passed as a parameter to this function.

```
void gsmPinEnter(GSM_PIN_Instance* inst, uint16_t sim_pin);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrPinEnter(uint16_t sim_pin);
```

gsmPinStatus

This function can optionally be used to determine the current status of the SIM card with regard to pin code entry requirements. The returned variable is an Enum (GSM_PIN_SimStatus), the details of which are below.

```
GSM_PIN_SimStatus gsmPinStatus(GSM_PIN_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
GSM_PIN_SimStatus gsmmgrPinStatus();
```

```
typedef enum {
    GSM_PIN_SimStatus_UNKNOWN, /* Status not yet acquired from modem */
    GSM_PIN_SimStatus_READY, /* PIN already entered successfully,
                                or not required to be entered */
    GSM_PIN_SimStatus_SIM_PIN, /* PIN is required to be entered */
    GSM_PIN_SimStatus_SIM_PUK, /* PUK is required to be entered */
    GSM_PIN_SimStatus_PH_SIM_PIN, /* Phone-to-SIM password is required */
    GSM_PIN_SimStatus_SIM_PIN2, /* PIN2 is required to be entered */
    GSM_PIN_SimStatus_SIM_PUK2, /* PUK2 is required to be entered */
    GSM_PIN_SimStatus_PH_NET_PIN, /* Network Personalisation Password
                                     is required to be entered */
    GSM_PIN_SimStatus_OTHER /* Unrecognised response from modem */
} GSM_PIN_SimStatus;
```

Please note that further information regarding these statuses can be obtained from your GSM modem datasheet / command manual.

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the PIN module:

```
typedef enum
{
    GSM_PIN_EventType_SimStatus,
    GSM_PIN_EventType_PinSuccess,
    GSM_PIN_EventType_PinFailed,
} GSM_PIN_EventType;
```

The event data ("event_data"), if set, differs per event (details below).

GSM_PIN_EventType_SimStatus

This event will be triggered once the status of the SIM card has been determined. The event data will be a pointer to the following struct:

```
typedef struct __GSM_PIN_Event_SimStatus GSM_PIN_Event_SimStatus;
struct __GSM_PIN_Event_SimStatus {
    GSM_PIN_SimStatus SimStatus;
};
```

GSM_PIN_EventType_PinSuccess

This event will be triggered if the correct PIN code was entered successfully. No event data is set with this event.

GSM_PIN_EventType_PinFailed

This event will be triggered if the incorrect PIN was entered or if the SIM card is expecting a PUK code or any other alternative code (refer to GSM_PIN_SimStatus Enum). No event data is set with this event.

Signal Quality Module

Summary

The Signal Quality module can be used to query the signal strength indication (RSSI) and the bit error rate (BER) from the modem.

Module Dependencies

Signal Quality module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.

Additional Information

Note that the RSSI value returned is not in dBm, nor the BER value in percent; the values returned are the raw results output by the modem - the calibration of which differs per modem model, and thus require different conversion calculations per modem model to obtain the RSSI dBm and BER percentage equivalent values. Please consult your modem documentation for more details.

The Signal Quality module does provide an optional function (gsmSignalQualityRssiTo_dBm) to complete the RSSI conversion, based on an average value observed between modems tested. Please refer to the section labelled "Callable Functions" for more information.

Note that a value of "99" will be returned by the modem if the RSSI value is not known or detectable. This result may be seen if the modem has not yet successfully registered on the network.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_SignalQuality_Instance GSM_SignalQuality_Instance;
struct __GSM_SignalQuality_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_BasicComms_Interface* BasicComms;
    void    (*EventHandler)(GSM_SignalQuality_Instance* inst,
                            GSM_SignalQuality_EventType event_type,
                            void* event_data);
};
```

Set the Power, Prompt and BasicComms module interface references (pointers).

Optionally set the "EventHandler" callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmSignalQualityInit(GSM_SignalQuality_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmSignalQualityPoll(GSM_SignalQuality_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmSignalQualityQuery

Trigger the process to query the signal quality from the modem. The result will be output using an event handler (see Event Handler section below).

```
bool gsmSignalQualityQuery(GSM_SignalQuality_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrSignalQualityQuery();
```

gsmSignalQualityRssiTo_dBm

This optional function can be used to convert the raw RSSI value returned by the modem to a rough dBm value. The calculation and value returned are based on an average across a variety of modems; for a more accurate value please consult your modem data sheet / command manual and implement your own calculation accordingly.

```
int8_t gsmSignalQualityRssiTo_dBm(uint8_t rawRssiValue);
```

The following convenience function is available when interfacing with the Signal Quality module:

```
int8_t gsmmgrSignalQualityRssiTo_dBm(uint8_t rawRssiValue);
```

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the Signal Quality module:

```
typedef enum
{
    GSM_SignalQuality_EventType_QuerySuccess,
} GSM_SignalQuality_EventType;
```

The event data ("event_data"), if set, differs per event (details below).

GSM_SignalQuality_EventType_QuerySuccess

This event will be triggered once the modem has returned the result of the signal quality query. The event data will be a pointer to the following struct (note that Quality is equivalent to BER):

```
typedef struct __GSM_SignalQuality_Event_QuerySuccess
GSM_SignalQuality_Event_QuerySuccess;
struct __GSM_SignalQuality_Event_QuerySuccess {
    uint8_t Rssi;
    uint8_t Quality;
};
```

Please see the section "Additional Information" above for information regarding the units and calibration of the values.

Network Registration Module

Summary

The Network Registration module is responsible for monitoring whether the modem is registered on the cellular network.

Module Dependencies

Network Registration module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.

Additional Information

Other library modules depend on this module to inform them regarding whether they can proceed to attempt operations which require the modem to have registered on the network first (such as sending an SMS, making a call, etc).

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_NetworkRegistration_Instance
GSM_NetworkRegistration_Instance;
struct __GSM_NetworkRegistration_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_BasicComms_Interface* BasicComms;
};
```

Set the Power, Prompt and Basic Comms module interface references (pointers), then call the initialisation routine.

```
void gsmNetworkRegistrationInit(GSM_NetworkRegistration_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmNetworkRegistrationPoll(GSM_NetworkRegistration_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmNetworkRegistrationIsRegistered

Get the network registration status of the modem (registered or not).

```
bool gsmNetworkRegistrationIsRegistered(GSM_NetworkRegistration_Instance*
                                        inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrNetworkRegistrationIsRegistered();
```

SMS Send Module

Summary

The SMS Send module can be used to send text messages.

Module Dependencies

SMS Send module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_SmsSend_Instance GSM_SmsSend_Instance;
struct __GSM_SmsSend_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_NetworkRegistration_Interface* NetworkRegistration;
    GSM_BasicComms_Interface* BasicComms;
    void     (*EventHandler)(GSM_SmsSend_Instance* inst,
                             GSM_SmsSend_EventType event_type,
                             void* event_data);
};
```

Set the Power, Prompt, Network Registration, and Basic Comms module interface references (pointers).

Optionally set the "EventHandler" callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmSmsSendInit(GSM_SmsSend_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmSmsSendPoll(GSM_SmsSend_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmSmsSend

Trigger the process to send an SMS with the specified text to the specified phone number.

The function will return **true** if the SMS module is not currently in the process of sending an SMS and the new SMS has been successfully queued to be sent, and will return **false** if the module is already in the process of sending an SMS and the new SMS was not queued to be sent (note that "gsmSmsSendCancel" can be used to cancel a currently queued SMS if necessary).

If you would like an SMS Send Queue module to be developed then please submit your request to us so that development can be prioritised.

The result of the attempt to send an SMS will be output using the SMS event handler (see "Event Handler" section below). Note that you can easily "Daisy-Chain" messages to be sent by monitoring the "SendSuccess" / "SendFailed" events and queueing the next message as soon as the present one has been sent (or failed).

```
bool gsmSmsSend(GSM_SmsSend_Instance* inst, char* phoneNumber, char*
                textBody);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrSmsSend(char* phoneNumber, char* textBody);
```

gsmSmsSendCancel

Cancel and remove the currently queued SMS (if any).

The function will return **true** if there was an SMS queued to be sent which was successfully removed, or **false** if there was no SMS already queued which could be cancelled.

```
bool gsmSmsSendCancel(GSM_SmsSend_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrSmsSendCancel();
```

gsmSmsSendGetDeliveryStatusReportType

This function can be used to get an Enum type that provides more information for the "MessageStatus" field returned in the "GSM_SmsSend_EventType_DeliveryStatusReport" delivery report event (refer to section titled "Event Handler for more information).

Please note that this function only provides a summary of the possible statuses that are referenced in the GSM 03.40 specification. Refer to this specification to obtain a detailed definition for each message status that can be obtained from your modem. See https://www.etsi.org/deliver/etsi_gts/03/0340/05.03.00_60/gsmts_0340v050300p.pdf for the specification (page 51).

```
GSM_SmsSend_DeliveryStatusReportType
                gsmSmsSendGetDeliveryStatusReportType(uint8_t messageStatus);
```

The following convenience function is available when interfacing with the SMS Send module:

```
GSM_SmsSend_DeliveryStatusReportType
             gsmmgrSmsSendGetDeliveryStatusReportType(uint8_t messageStatus);
```

```
typedef enum
{
    GSM_SmsSend_DeliveryStatusReportType_TransactionComplete,
    GSM_SmsSend_DeliveryStatusReportType_MessageReceived,
    GSM_SmsSend_DeliveryStatusReportType_MessageForwarded,
    GSM_SmsSend_DeliveryStatusReportType_TempError_Retrying,
    GSM_SmsSend_DeliveryStatusReportType_PermanentError,
    GSM_SmsSend_DeliveryStatusReportType_TempError_NoRetriesRemaining,
    GSM_SmsSend_DeliveryStatusReportType_ServiceRejected,
    GSM_SmsSend_DeliveryStatusReportType_UnrecognisedError,
} GSM_SmsSend_DeliveryStatusReportType;
```

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the SMS Send module:

```
typedef enum
{
    GSM_SmsSend_EventType_SendSuccess,
    GSM_SmsSend_EventType_SendFailed,
    GSM_SmsSend_EventType_DeliveryStatusReport,
} GSM_SmsSend_EventType;
```

The event data ("event_data"), if set, differs per event (details below).

GSM_SmsSend_EventType_SendSuccess

This event will be triggered once the modem has successfully sent the queued SMS (successfully submitted it to the network). Note that "SMS Sent" is different from "SMS Delivered", which is a separate event (see GSM_SmsSend_EventType_DeliveryStatusReport below). The event data will be a pointer to the following struct:

```
typedef struct __GSM_SmsSend_Event_SendSuccess GSM_SmsSend_Event_SendSuccess;
struct __GSM_SmsSend_Event_SendSuccess {
    uint8_t MessageReference;
};
```

The struct fields are detailed below:

- MessageReference – This is a reference number that is assigned to the SMS that was sent. The reference number can be used to match Delivery Status Report events (which also have a MessageReference field) to the original SMS sent.

GSM_SmsSend_EventType_SendFailed

This event will be triggered if the modem fails to send the queued SMS (fails to submit it to the network).

GSM_SmsSend_EventType_DeliveryStatusReport

This event will be triggered once the modem has received a delivery report for any SMS that has been sent. The event data will be a pointer to the following struct:

```
typedef struct __GSM_SmsSend_Event_DeliveryStatusReport
GSM_SmsSend_Event_DeliveryStatusReport;
struct __GSM_SmsSend_Event_DeliveryStatusReport{
    uint16_t MessageReference;
    uint8_t  MessageStatus;
    uint32_t ServiceCentreEpoch;
    int32_t  ServiceCentreTimeZoneOffsetInSeconds;
    uint32_t DischargeEpoch;
    int32_t  DischargeTimeZoneOffsetInSeconds;
    char     PhoneNumber[32];
};
```

The struct fields are detailed below:

- MessageReference – This is a reference number that was assigned to the SMS that was sent (provided in the "GSM_SmsSend_EventType_SendSuccess" event).
- MessageStatus – This is the delivery status of the SMS. The callable function gsmSmsSendGetDeliveryStatusReportType (detailed above) can be used to convert the MessageStatus code to a more readable enum type. Please refer to the GSM 03.40 specification to obtain a detailed definition for each message status. See https://www.etsi.org/deliver/etsi_gts/03/0340/05.03.00_60/gsmts_0340v050300p.pdf for the specification (page 51).
- ServiceCentreEpoch – This is the UTC Epoch timestamp for when the SMS was received by the cellular provider.
- ServiceCentreTimeZoneOffsetInSeconds – This value represents the number of seconds to be added to the UTC Epoch timestamp (ServiceCentreEpoch) to convert it to the local time zone. By adding this offset to the UTC timestamp, the local time for the event can be accurately determined.
- DischargeEpoch – This is the UTC Epoch timestamp for when the SMS was sent to the specified number by the cellular provider.
- DischargeTimeZoneOffsetInSeconds – This value represents the number of seconds to be added to the UTC Epoch timestamp (DischargeEpoch) to convert it

to the local time zone. By adding this offset to the UTC timestamp, the local time for the event can be accurately determined.

- PhoneNumber – This is the phone number that the SMS was sent to.

SMS Read Module

Summary

The SMS Read module is used to read text messages received by the modem. The module will read and output the SMS text when an SMS is received; the module will also routinely poll the SIM card to check for any unread text messages.

Module Dependencies

SMS Read module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.

Additional Information

The SMS Read module will delete each message after it has been read and output to the user, ensuring that there is always sufficient space on the SIM card for new messages to be received.

All read messages will be output using the SMS Read event handler. Please refer to the section titled "Event Handler" (below) for more information.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_SmsRead_Instance GSM_SmsRead_Instance;
struct __GSM_SmsRead_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_BasicComms_Interface* BasicComms;
    void     (*EventHandler)(GSM_SmsRead_Instance* inst,
                             GSM_SmsRead_EventType event_type,
                             void* event_data);
};
```

Set the Power, Prompt, and Basic Comms module interface references (pointers).

Set the "EventHandler" callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmSmsReadInit(GSM_SmsRead_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmSmsReadPoll(GSM_SmsRead_Instance* inst);
```

Callable Functions

The SMS Read module does not have any functions which can be optionally called by the host application.

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following event may be raised by the SMS Read module:

```
typedef enum
{
    GSM_SmsRead_EventType_ReadSuccess,
} GSM_SmsRead_EventType;
```

The event data ("event_data"), if set, differs per event (details below).

GSM_SmsRead_EventType_ReadSuccess

This event will be triggered when the SMS Read module has successfully read an SMS that has been received. The event data will be a pointer to the following struct:

```
typedef struct __GSM_SmsRead_Event_ReadSuccess GSM_SmsRead_Event_ReadSuccess;
struct __GSM_SmsRead_Event_ReadSuccess {
    GSM_SmsRead_SmsStatus SmsStatus;
    uint32_t EpochTime;
    int32_t  TimeZoneCompensation;
    char     PhoneNumber[32];
    char     SmsData[161];
};
```

The GSM_SmsRead_SmsStatus enum is as follows:

```
typedef enum
{
    GSM_SmsRead_SmsStatus_REC_UNREAD, /* Received, unread */
    GSM_SmsRead_SmsStatus_REC_READ, /* Received, read already */
    GSM_SmsRead_SmsStatus_STO_UNSENT, /* Drafted (stored), unsent */
    GSM_SmsRead_SmsStatus_STO_SENT, /* Drafted (stored), sent already */
    GSM_SmsRead_SmsStatus_ERROR, /* Unrecognised result from modem */
} GSM_SmsRead_SmsStatus;
```

The struct fields are detailed below:

- SmsStatus – This variable is an Enum (detailed above) that provides more information regarding the "read" status of the message.

- EpochTime – This is the UTC Epoch timestamp for when the message was received.
- TimeZoneCompensation – This value represents the number of seconds to be added to the UTC Epoch timestamp (EpochTime) to convert it to the local time zone. By adding this offset to the UTC timestamp, the local time for the event can be accurately determined.
- PhoneNumber – This is the phone number that the SMS was received from.
- SmsData – This is the text of the SMS that was received.

USSD Module

Summary

The USSD module can be used to perform USSD sessions (both initiating a session and handling responses).

Module Dependencies

USSD module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.

Additional Information

Depending on the modem being used, the process of starting a USSD session may terminate any active internet connection for the duration of the session. This will affect any active HTTP, MQTT, or other internet-based-module sessions.

If you are using the HTTP or MQTT modules simultaneously with the USSD module, then please be aware that HTTP GET and POST requests will need to be attempted again in the case that a USSD session is started during an HTTP request, and any MQTT topic subscriptions will need to be subscribed to again if you are using a clean session to connect to the MQTT broker.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_USSD_Instance GSM_USSD_Instance;
struct __GSM_USSD_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_NetworkRegistration_Interface* NetworkRegistration;
    void (*EventHandler)(GSM_USSD_Instance* inst,
                         GSM_USSD_EventType event_type,
                         void* event_data);
};
```

Setup the Power, Prompt, and Network Registration module interface references (pointers).

Optionally set the “EventHandler” callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmUssdInit(GSM_USSD_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmUssdPoll(GSM_USSD_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmUssdSend

This function serves a dual purpose. It can be used to start a USSD session (if there is not currently an active session), or to respond to a currently active USSD session (upon a "GSM_USSD_EventType_UssdResponse" event being received, as detailed in the "Event Handler" section below). The "ussd_payload" variable must be the code to be used to start the session (when a new session is required), or the code to respond with (when there is already an active session).

The function will return **true** if the USSD module is not currently in the process of starting a USSD session or replying to a currently active session, and will return **false** if the module is already in the process of starting a USSD session or replying to a currently active session.

The result of this function will be output using the USSD event handler (see "Event Handler" section below).

```
bool gsmUssdSend(GSM_USSD_Instance* inst, char* ussd_payload);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrUssdSend(char* ussd_payload);
```

gsmUssdCancelSession

This function will cancel/stop any currently active USSD session. It will also stop a new session from being started if a requested session (initiated via "gsmUssdSend") has not yet been started.

The function will return **true** if the USSD module is not already attempting to cancel a USSD session, and will return **false** if the module is already in the process of attempting to cancel a USSD session.

The result of the cancellation attempt will be output using the USSD event handler (see "Event Handler" section below, "GSM_USSD_EventType_UssdCancelled" event).

```
bool gsmUssdCancelSession(GSM_USSD_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrUssdCancelSession();
```

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the USSD module:

```
typedef enum
{
    GSM_USSD_EventType_UssdSent,
    GSM_USSD_EventType_UssdResponse,
    GSM_USSD_EventType_UssdCancelled,
} GSM_USSD_EventType;
```

The event data ("event_data"), if set, differs per event (details below).

GSM_USSD_EventType_UssdSent

This event will be triggered when the USSD request/response has successfully been sent. This will be the case when starting a new session or when responding to a currently active session.

GSM_USSD_EventType_UssdResponse

This event will be triggered when the USSD module has received a USSD session response from the modem. The event data will be a pointer to the following struct:

```
typedef struct __GSM_USSD_Event_UssdResponse GSM_USSD_Event_UssdResponse;
struct __GSM_USSD_Event_UssdResponse {
    GSM_USSD_ResponseCode ResponseCode;
    char* DataResult;
};
```

```
typedef enum
{
    GSM_USSD_ResponseCode_NoActionRequired = 0, /* No response required */
    GSM_USSD_ResponseCode_ActionRequired = 1, /* Response required */
    GSM_USSD_ResponseCode_SessionTerminated = 2, /* Session terminated
                                                    by network */
    GSM_USSD_ResponseCode_LocalClientResponse = 3, /* Operation still
                                                      in progress */
    GSM_USSD_ResponseCode_OperationNotSupported = 4, /* USSD code rejected
                                                        by network */
    GSM_USSD_ResponseCode_NetworkTimeout = 5, /* USSD request/response
                                                 timed out */
```

```
} GSM_USSD_ResponseCode;
```

The struct fields are detailed below:

- ResponseCode – This field is an Enum that provides information regarding the status of the USSD session (such as whether a response is required or not).
- DataResult – This field is the text (string) content of the USSD response, if any; otherwise, if the modem has not included any text output with the response code, then the field will be set to 0 (null).

GSM_USSD_EventType_UssdCancelled

This event will be triggered when the USSD module has successfully cancelled/stopped a USSD session.

Internet Connection (SIMCom 2G) Module

Summary

The Internet Connection (SIMCom 2G) module is responsible for managing the internet connection (connecting to / disconnecting from the internet) on SIMCom 2G modems.

Module Dependencies

The Internet Connection (SIMCom 2G) module dependencies are list below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.

Additional Information

This module makes use of the AT+CGATT and AT+SAPBR commands to manage the internet connection, and has been tested on SIMCom SIM800 series modems.

Both home network as well as roaming connections are supported.

An APN may be specified if required. Please see section titled "Initialisation" below for more information.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_InternetConnectionSimcom2G_Instance
GSM_InternetConnectionSimcom2G_Instance;
struct __GSM_InternetConnectionSimcom2G_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_NetworkRegistration_Interface* NetworkRegistration;
    char* APN_IfAny;
};
```

Set the Power, Prompt, and Network Registration modules.

Optionally set the "APN_IfAny" field if a specific APN needs to be used, or otherwise ensure that it is set to 0 (null).

Call the initialisation routine.

```
void
gsmInternetConnectionSimcom2GInit(GSM_InternetConnectionSimcom2G_Instance*
inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void
gsmInternetConnectionSimcom2GPoll(GSM_InternetConnectionSimcom2G_Instance*
inst);
```

Callable Functions

The following function may optionally be called by the host application:

gsmInternetConnectionSimcom2GIsConnected

Get the internet connection status of the modem (connected or not).

```
bool
gsmInternetConnectionSimcom2GIsConnected(GSM_InternetConnectionSimcom2G_Instan
ce* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrInternetConnectionSimcom2GIsConnected();
```

Internet Connection (SIMCom LTE) Module

Summary

The Internet Connection (SIMCom LTE) module is responsible for managing the internet connection (connecting to / disconnecting from the internet) on SIMCom LTE modems.

Module Dependencies

The Internet Connection (SIMCom LTE) module dependencies are list below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.

Additional Information

This module makes use of the AT+CGATT command to manage the internet connection, and has been tested on SIMCom A7600 series modems.

Both home network as well as roaming connections are supported.

An APN may be specified if required. Please see section titled "Initialisation" below for more information.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_InternetConnectionSimcom_Instance
GSM_InternetConnectionSimcom_Instance;
struct __GSM_InternetConnectionSimcom_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_NetworkRegistration_Interface* NetworkRegistration;
    char* APN_IfAny;
};
```

Set the Power, Prompt, and Network Registration modules.

Optionally set the "APN_IfAny" field if a specific APN needs to be used, or otherwise ensure that it is set to 0 (null).

Call the initialisation routine.

```
void gsmInternetConnectionSimcomInit(GSM_InternetConnectionSimcom_Instance*
                                     inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmInternetConnectionSimcomPoll(GSM_InternetConnectionSimcom_Instance*
                                     inst);
```

Callable Functions

The following function may optionally be called by the host application:

gsmInternetConnectionSimcomIsConnected

Get the internet connection status of the modem (connected or not).

```
bool
gsmInternetConnectionSimcomIsConnected(GSM_InternetConnectionSimcom_Instance*
inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrInternetConnectionSimcomIsConnected();
```

HTTP Client (SIMCom) Module

Summary

The HTTP Client (SIMCom) module can be used to perform HTTP(S) operations (GET and POST requests).

Module Dependencies

HTTP Client (SIMCom) module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.
- Internet Connection (SIMCom LTE) module.

Additional Information

The HTTP module can be used for both HTTP and HTTPS requests and it can be assumed that any reference to HTTP requests in this document can be used synonymously for HTTP and HTTPS.

The HTTP module allows the user to define the HTTP content type that will be used in HTTP POST operations. Please refer to the section titled "Callable Functions" below for more information.

The size of the buffers used in the Prompt module should be considered in relation to the size of the HTTP transactions expected. Note that the receive buffer does not necessarily need to be large enough to contain the entire HTTP response, however if it is smaller than the entire HTTP response then that response will be chunked (evented out in segments).

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_HttpClientSimcom_Instance GSM_HttpClientSimcom_Instance;
struct __GSM_HttpClientSimcom_Instance {
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_InternetConnection_Interface* InternetConnection;
    void (*EventHandler)(GSM_HttpClientSimcom_Instance* inst,
                         GSM_HttpClientSimcom_EventType event_type,
                         void* event_data);
};
```

Set the Power, Prompt, and Internet Connection module interface references (pointers).

Optionally set the "EventHandler" callback (more details in the Event Handler section below).

Call the initialisation routine.

```
void gsmHttpClientSimcomInit(GSM_HttpClientSimcom_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmHttpClientSimcomPoll(GSM_HttpClientSimcom_Instance* inst);
```

Callable Functions

The following functions may optionally be called by the host application:

gsmHttpClientSimcomGet

Perform an HTTP(S) GET request.

The function will return **true** if the HTTP module is not already busy with processing a GET or POST operation and will return **false** if the module is already busy processing a GET or POST operation.

```
bool gsmHttpClientSimcomGet(GSM_HttpClientSimcom_Instance* inst, char* url);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrHttpClientSimcomGet(char* url);
```

gsmHttpClientSimcomPost

Perform an HTTP(S) POST operation.

The function will return **true** if the HTTP module is not already busy with processing a GET or POST operation and will return **false** if the module is already busy processing a GET or POST operation.

Note that the Content Type can be set using the "gsmHttpClientSimcomPostSetContentType" function (detailed below).

```
bool gsmHttpClientSimcomPost(GSM_HttpClientSimcom_Instance* inst, char* url,
                             char* postData);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrHttpClientSimcomPost(char* url, char* postData);
```

gsmHttpClientSimcomPostSetContentType

Optionally set the HTTP content type to be used in the following HTTP(S) POST operations.

Please refer to HTTP headers standard for the possible content types. The HTTP module uses "application/x-www-form-urlencoded" by default.

```
void gsmHttpClientSimcomPostSetContentType(GSM_HttpClientSimcom_Instance*
                                           inst, char*
                                           contentTypeOrNullForDefault);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrHttpClientSimcomPostSetContentType(char*
                                              contentTypeOrNullForDefault);
```

gsmHttpClientSimcomHttpOperationPending

Get the operation pending status from the HTTP module (HTTP operation pending or not). "Pending" means that the operation is queued, but has not yet started.

```
bool gsmHttpClientSimcomHttpOperationPending(GSM_HttpClientSimcom_Instance*
                                             inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrHttpClientSimcomHttpOperationPending();
```

gsmHttpClientSimcomHttpOperationStarted

Get the operation started status from the HTTP module (HTTP operation started or not).

```
bool gsmHttpClientSimcomHttpOperationStarted(GSM_HttpClientSimcom_Instance*
                                             inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrHttpClientSimcomHttpOperationStarted();
```

gsmHttpClientSimcomCancel

Cancel the current pending HTTP operation.

The function will return **true** if there was an HTTP operation pending and return **false** if there was no HTTP operation pending.

```
bool gsmHttpClientSimcomCancel(GSM_HttpClientSimcom_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrHttpClientSimcomCancel();
```

gsmHttpClientSimcomHttpTermWhenDone

Optionally set whether the HTTP session should be terminated when the request is complete (false by default).

```
void gsmHttpClientSimcomHttpTermWhenDone(GSM_HttpClientSimcom_Instance* inst,
                                         bool termWhenDone);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrHttpClientSimcomHttpTermWhenDone(bool termWhenDone);
```

gsmHttpClientSimcomSetTimeouts

Optionally set the HTTP timeout values (in milliseconds) to be used by the HTTP module.

The httpActionTimeout_ms parameter (default 10s) is the timeout used when waiting for the response from an HTTP GET or POST operation. The httpReadTimeout_ms parameter (default 10s) is the timeout used while waiting for data to be read in response to the HTTP request sent.

```
void gsmHttpClientSimcomSetTimeouts(GSM_HttpClientSimcom_Instance* inst,
                                    uint32_t httpActionTimeout_ms, uint32_t
                                    httpReadTimeout_ms);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
void gsmmgrHttpClientSimcomSetTimeouts(uint32_t httpActionTimeout_ms, uint32_t
                                       httpReadTimeout_ms);
```

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the HTTP Client (SIMCom LTE) module:

```
typedef enum {
    GSM_HttpClientSimcom_EventType_HttpResult,
} GSM_HttpClientSimcom_EventType;
```

The event data ("event_date"), if set, differs per event (details below). In this case only one event can be triggered by the HTTP module.

GSM_HttpClientSimcom_EventType_HttpResult

This event will be triggered once a result for the HTTP operation is received by the modem:

```
typedef struct __GSM_HttpClientSimcom_Event_HttpResult
GSM_HttpClientSimcom_Event_HttpResult;
```

```
struct __GSM_HttpClientSimcom_Event_HttpResult {
    uint16_t ResultStatusCode;
    char*    DataResult;
    uint32_t DataLenTotal;
    uint32_t DataLenSegment;
    uint32_t DataOffset;
};
```

The struct fields are detailed below:

- ResultStatusCode – This is the HTTP status code result (e.g. 200 for "OK"). Note that the modem may also return custom codes which are not part of the HTTP standard – please see the modem documentation for more information.
- DataResult – This is the response data received from the HTTP server.
- DataLenTotal – This is the total length of the data received from the HTTP server. Note that depending on your buffer setup, this information may be broken up into multiple segments (chunked). Refer to the section titled "Additional Information" above for more information regarding the buffer setup.
- DataLenSegment – This is the length of the current segment of data. Each data segment when added together will add up to the total length of the response ("DataLenTotal").
- DataOffset – This is the offset from the start of the data received from the HTTP server to the start of the current segment of data.

MQTT Client (SIMCom LTE) Module

Summary

The MQTT Client (SIMCom LTE) module can be used to send (publish) and receive (subscribe to) MQTT messages, via an MQTT broker which it connects to.

Module Dependencies

The MQTT Client (SIMCom LTE) module dependencies are listed below:

- Power module.
- Prompt module.
- Basic Comms module.
- Network Registration module.
- Internet Connection (SIMCom LTE) module.

Additional Information

MQTT connections can be configured with a variety of different security configurations. This includes options for encrypted or unencrypted host servers (brokers) and whether authentication (username and password) is required or not. Please refer to the section titled "Callable Functions" below to ensure that the correct functions are used to configure your connection appropriately.

The MQTT Client (SIMCom LTE) module will automatically reconnect to a host server (broker) if the connection is lost. Note that if a "clean session" is used when connecting to the host server, then all prior subscriptions will be lost; the "ConnectionLost" and "ConnectSuccess" events can be monitored in order to handle this appropriately (see the "Event Handler" section below).

The size of the buffers used in the Prompt module should be considered in relation to the size of the MQTT transaction payloads expected. Note that the receive buffer does not necessarily need to be large enough to contain the entire MQTT payload, however if it is smaller than the entire MQTT payload then that payload will be chunked (evented out in segments).

Note that this module supports both encrypted and unencrypted connections, with the option for either authenticated (username and password) or unauthenticated access in each case. However, certificate-based authentication is not currently supported; if you require this feature, please let us know so we can prioritise its development.

Initialisation

(This code is auto-generated when using the LibGSM Code Generator)

Instance Struct (simplified):

```
typedef struct __GSM_MqttClientSimcom_Instance GSM_MqttClientSimcom_Instance;
struct __GSM_MqttClientSimcom_Instance {
```

```
    GSM_Power_Interface* Power;
    GSM_Prompt_Interface* Prompt;
    GSM_NetworkRegistration_Interface* NetworkRegistration;
    GSM_InternetConnection_Interface* InternetConnection;
    void  (*EventHandler)(GSM_MqttClientSimcom_Instance* inst,
                          GSM_MqttClientSimcom_EventType event_type,
                          void* event_data);
};
```

Set the Power, Prompt, Network Registration, and Internet Connection module interface references (pointers).

Optionally set the "EventHandler" callback (more details in the Event Handler section below)

Call the initialisation routine.

```
void gsmMqttClientSimcomInit(GSM_MqttClientSimcom_Instance* inst);
```

Polling

(This code is auto-generated when using the LibGSM Code Generator)

As with all library modules, the "poll" routine must be called regularly.

```
void gsmMqttClientSimcomPoll(GSM_MqttClientSimcom_Instance* inst);
```

Basic Workflow for MQTT Module

This section offers a quick overview of the essential steps needed to connect to an MQTT broker, publish messages, and subscribe to topics. It aims to help you get your MQTT application up and running as quickly as possible.

Detailed descriptions of the functions used can be found in the sections titled "Callable Functions" and "Event Handler".

Note that the example programs available can also provide a good reference to work from.

Starting an MQTT session

1. Connecting to a broker – This is done using the gsmMqttClientSimcomConnect() function, which accepts a GSM_MqttClientSimcom_Connect_Session struct parameter (containing the various settings for the connection).
    a. The gsmMqttClientSimcomInitConfigDefault() function can be used to set default connection parameters in the GSM_MqttClientSimcom_Connect_Session struct. Note that individual parameters can be updated, if necessary, after calling this function.

Note: You can disconnect from the broker using the gsmMqttClientSimcomDisconnect() function.

Publishing a message

Currently, only one message can be queued in the MQTT module at a time. Once the message has successfully been published then the next message can be queued to be published. You can easily "Daisy-Chain" messages to be published by monitoring the "PublishSuccess" event and queueing the next message as soon as the present one has been published.

If you would like an MQTT Publish Queue module to be developed then please submit your request to us so that development can be prioritised.

1. Queue a message – A message can be queued to be published using one of the following functions:
    a. The gsmMqttClientSimcomPublish() function. This function uses default values for the QoS value and the retained flag (QoS is set to 1 and the retained flag is set to true).
    b. The gsmMqttClientSimcomPublishEx() function. This function allows you to set the QoS and retained flags.
2. Message published successfully – The "PublishSuccess" event will be fired once the queued message has been published successfully.

Note: A queued message can be cancelled using the gsmMqttClientSimcomCancelPublish() function.

Subscribing to topics

Currently, only one subscribe request can be queued in the MQTT module at a time. Once the subscription has been successfully completed then the next subscription can be queued. You can easily "Daisy-Chain" subscriptions by monitoring the "SubscribeSuccess" event and queueing the next subscription as soon as the present one has been processed.

If you would like an MQTT Subscribe Queue module to be developed then please submit your request to us so that development can be prioritised.

1. Queue a subscription – A topic can be subscribed to using the following function:
    a. gsmMqttClientSimcomSubscribe().
2. Topic subscription successful – The "SubscribeSuccess" event will be fired once the queued subscription has been completed successfully.

Note: A subscription that has been queued can be cancelled using the gsmMqttClientSimcomCancelSubscribe() function, and you can unsubscribe from any topics using the gsmMqttClientSimcomUnsubscribe() function.

Incoming message routine

Once you have successfully subscribed to a topic, you will now start receiving any messages that are published to that topic. The following events (in this order) will be triggered when a message is received:

1. MsgRxTopic – This event will notify the user that a message has been received and will include the relevant topic information. If the topic is too long to be output in one segment, then this event will repeat until all the topic information has been output to the user.
2. MsgRxPayload – This event will include the relevant payload information. If the payload is too long to be output in one segment, then this event will repeat until all the payload information has been output to the user.
3. MsgRxEnd – This event will be triggered at the end of the received message.

The module will also output certain events if any errors are flagged during the incoming message routine:

1. MsgRxTopicHeaderError – This event will notify a user that the topic header could not be read. If the topic is long enough to be made up of multiple segments, then this segment will be missed.
2. MsgRxTopicError – This event will notify a user that the topic data, or topic segment data (if made up of multiple segments), could not be read.
3. MsgRxPayloadHeaderError – This event will notify a user that the payload header could not be read. If the payload is long enough to be made up of multiple segments, then this segment will be missed
4. MsgRxPayloadError – This event will notify a user that the payload data, or payload segment data (if made up of multiple segments), could not be read.

Callable Functions

The following functions may optionally be called by the host application:

gsmMqttClientSimcomConnect

This function is used to connect to an MQTT broker, which is necessary before any MQTT messages can be sent (published) or received (via subscription to a topic(s) on the broker).

The function will return **true** if the module is not already connected or attempting to connect to a host server (broker), and **false** if the module is already connected or attempting to connect to a host server (broker).

Note: Use the gsmMqttClientSimcomDisconnect() function to disconnect if you want to establish a new connection (with different parameters).

This function takes parameters in the form of a struct, the details of which are below. Note that the function "gsmMqttClientSimcomInitConfigDefault" (detailed further below) can be used to conveniently set default values on all of the struct fields, after which specific fields can then be updated if and as required (before calling gsmMqttClientSimcomConnect).

```
bool gsmMqttClientSimcomConnect(GSM_MqttClientSimcom_Instance* inst,
                                GSM_MqttClientSimcom_Connect_Session*
                                connect_session);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomConnect(GSM_MqttClientSimcom_Connect_Session*
                                   connect_session);
```

```
typedef struct __GSM_MqttClientSimcom_Connect_Session
GSM_MqttClientSimcom_Connect_Session;
struct __GSM_MqttClientSimcom_Connect_Session
{
    char*    MqttClientID;
    char*    HostUrl;
    uint16_t Port;
    GSM_MQTT_ServerType      ServerType;
    char*    Username;
    char*    Password;
    bool     IsCleanSession;
    uint16_t KeepAliveTime;
    GSM_MQTT_ProtocolVersion ProtocolVersion;
};
```

```
typedef enum
{
    GSM_MQTT_ServerType_Unencrypted_Unauthenticated, /* MQTT, no user/pwd */
    GSM_MQTT_ServerType_Unencrypted_Authenticated, /* MQTT, with user/pwd */
    GSM_MQTT_ServerType_Encrypted_Unauthenticated, /* MQTTS, no user/pwd */
    GSM_MQTT_ServerType_Encrypted_Authenticated, /* MQTTS, with user/pwd */
} GSM_MQTT_ServerType;
```

```
typedef enum
{
    GSM_MQTT_ProtocolVersion_V3_1,
    GSM_MQTT_ProtocolVersion_V3_1_1,
} GSM_MQTT_ProtocolVersion;
```

The GSM_MqttClientSimcom_Connect_Session struct fields are detailed below:

- MqttClientID – This is a string which must be a unique value on the broker, otherwise the broker will refuse the connection (if another client with the same Client ID is already connected). It is recommended to generate this from some unique value such as the MCU ID, or from some other unique ID set at production time, etc.
- HostUrl – This is the address of the server (broker), excluding the leading "mqtt://" or "mqtts://" as well as any trailing port (the port number is set in the Port field). E.g. "test.mosquitto.org".
- Port – TCP port on which to connect to the server (broker). This is usually 1883 for MQTT or 8883 for MQTTS.

- ServerType – This is an enum type ("GSM_MQTT_ServerType") which specifies whether the server (broker) is encrypted (MQTTS) or unencrypted (MQTT) as well as whether it requires authentication (username and password) or not.
- Username – If an authenticated server type is used (see the "ServerType" field), then the username should be set here.
- Password – If an authenticated server type is used (see the "ServerType" field), then the password should be set here.
- IsCleanSession – If the "Clean Session" flag is **not** set then the server (broker) will store topic subscriptions for this client as well as messages received, whilst the client is offline, and restore them when the client reconnects. If "Clean Session" **is** set then it will be necessary to resubscribe to all topics each time this client connects, and any messages received whilst this client is offline will not be stored on the server (broker) for potential later delivery (when the client does reconnect).
- KeepAliveTime – How often (in seconds) to ping the server (broker) in order to keep the connection alive. A suggested value is 30s. If you find that your server (broker) is disconnecting automatically ("ConnectionLost" event, detailed below) then try decreasing this value.
- ProtocolVersion – This is an enum type ("GSM_MQTT_ProtocolVersion", detailed above) which can be either v3.1 or v3.1.1.

The "ConnectSuccess" event will fire upon successful connection to the server (broker).

gsmMqttClientSimcomInitConfigDefault

This function can optionally be used to set default values in a "GSM_MqttClientSimcom_Connect_Session" struct (detailed above) which is used to connect to an MQTT broker. Individual fields can then be adjusted as required before calling gsmMqttClientSimcomConnect.

```
void
gsmMqttClientSimcomInitConfigDefault(GSM_MqttClientSimcom_Connect_Session*
connect_session);
```

The following convenience function is available when interfacing with the MQTT module:

```
void
gsmmgrMqttClientSimcomInitConfigDefault(GSM_MqttClientSimcom_Connect_Session*
connect_session);
```

Note: The struct and fields are detailed above under "gsmMqttClientSimcomConnect".

The default values set are:

- ProtocolVersion –v3.1.1.
- ServerType – Unencrypted, unauthenticated.
- KeepAliveTime – 30s.
- IsCleanSession – True.

- MqttClientID – "randomId12345". **You should update this ID after calling this function, to ensure that it is unique on the broker.**
- Username – 0 (null).
- Password – 0 (null).
- HostUrl – "test.mosquitto.org".

gsmMqttClientSimcomPublish

Publish a message to the broker, using default values for "QoS" (default value of 1) and "Is Retained" (default of "True").

Note that it is necessary to first connect to the broker ("gsmMqttClientSimcomConnect") before it will be possible to publish to it.

The function will return **true** if there isn't currently a publish in progress and **false** if there is already a publish in progress. If **false** is returned, wait for the publish to complete ("PublishSuccess" event, detailed below) and then try again.

See gsmMqttClientSimcomPublishEx() if you would like to define the QoS value and the retained flag status.

The "PublishSuccess" event will fire once the publish has been completed successfully.

```
bool gsmMqttClientSimcomPublish(GSM_MqttClientSimcom_Instance* inst, char*
                                publish_topic, char* publish_message);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomPublish(char* publish_topic, char*
                                   publish_message);
```

gsmMqttClientSimcomPublishEx

Publish a message to the broker.

Note that it is necessary to first connect to the broker ("gsmMqttClientSimcomConnect") before it will be possible to publish to it.

The function will return **true** if there isn't currently a publish in progress and **false** if there is already a publish in progress. If **false** is returned, then wait for the pending publish to complete ("PublishSuccess" event, detailed below) and then try again.

```
bool gsmMqttClientSimcomPublishEx(GSM_MqttClientSimcom_Instance* inst,
                                  GSM_MqttClientSimcom_Publish_Parameters*
                                  publish_parameters);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomPublishEx(GSM_MqttClientSimcom_Publish_Parameters*
                                     publish_parameters);
```

```
typedef struct __GSM_MqttClientSimcom_Publish_Parameters
GSM_MqttClientSimcom_Publish_Parameters;
struct __GSM_MqttClientSimcom_Publish_Parameters
{
    uint8_t PublishQos;
    bool    IsRetained;
    char*   PublishTopic;
    char*   PublishMessage;
};
```

The GSM_MqttClientSimcom_Publish_Parameters struct fields are detailed below:

- PublishQos – Publish Quality of Service (please see the MQTT specification for more information regarding QoS).
- IsRetained – Whether the message will be retained on the broker (please see the MQTT specification for more information).
- PublishTopic – The topic to publish to.
- PublishMessage – The message to publish.

The "PublishSuccess" event will fire once the publish has been completed successfully.

gsmMqttClientSimcomCancelPublish

Cancel a publish that is currently in progress.

The function will return **true** if the MQTT module is attempting to publish a message and **false** if the MQTT module is not attempting to publish a message (i.e. no message to cancel).

```
bool gsmMqttClientSimcomCancelPublish(GSM_MqttClientSimcom_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomCancelPublish();
```

gsmMqttClientSimcomSubscribe

Subscribe to a topic on the broker (including setting the QoS – Quality of Service).

Note that it is necessary to first connect to the broker ("gsmMqttClientSimcomConnect") before it will be possible to subscribe to any topics on it.

The function will return **true** if there isn't currently a subscription in progress and **false** if there is already a subscription in progress. If **false** is returned, wait for the pending subscription to complete ("SubscribeSuccess" event, detailed below) and then try again.

The "SubscribeSuccess" event will fire once the subscription has been completed successfully.

```
bool gsmMqttClientSimcomSubscribe(GSM_MqttClientSimcom_Instance* inst, char*
                                  subscribe_topic, uint8_t subscribe_QoS);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomSubscribe(char* subscribe_topic, uint8_t
                                     subscribe_QoS);
```

gsmMqttClientSimcomCancelSubscribe

Cancel subscription that is currently in progress.

The function will return **true** if the MQTT module is attempting to subscribe to a topic and **false** if the MQTT module is not attempting to subscribe to a topic (i.e. no subscription to cancel).

```
bool gsmMqttClientSimcomCancelSubscribe(GSM_MqttClientSimcom_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomCancelSubscribe();
```

gsmMqttClientSimcomUnsubscribe

Unsubscribe from a topic on the broker.

The function will return **true** if there isn't currently an unsubscribe in progress and **false** if there is already an unsubscribe in progress. If **false** is returned, wait for the unsubscribe to complete ("UnsubscribeSuccess" event, detailed below) and then try again.

```
bool gsmMqttClientSimcomUnsubscribe(GSM_MqttClientSimcom_Instance* inst, char*
                                    unsubscribe_topic);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomUnsubscribe(char* unsubscribe_topic);
```

The "UnsubscribeSuccess" event will fire once the topic has been unsubscribed from successfully.

gsmMqttClientSimcomCancelUnsubscribe

Cancel an unsubscribe attempt that is currently in progress.

The function will return **true** if the MQTT module is attempting to unsubscribe from a topic and **false** if the MQTT module is not attempting to unsubscribe from a topic (i.e. no unsubscribe attempt is in progress).

```
bool gsmMqttClientSimcomCancelUnsubscribe(GSM_MqttClientSimcom_Instance*
                                          inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomCancelUnsubscribe();
```

gsmMqttClientSimcomDisconnect

Disconnect from the server (broker).

The function will return **true** if there isn't currently a disconnect in progress and **false** if there is already a disconnect in progress.

```
gsmMqttClientSimcomDisconnect(GSM_MqttClientSimcom_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
bool gsmmgrMqttClientSimcomDisconnect();
```

gsmMqttClientServerConnectionState

This function returns the current state of connection to the server (broker).

Note: "Unknown" will be returned if the status has not yet been determined.

```
GSM_MQTT_ServerStatus
gsmMqttClientServerConnectionState(GSM_MqttClientSimcom_Instance* inst);
```

The following convenience function, which does not require the inst pointer, will be auto-generated when using the LibGSM Code Generator:

```
GSM_MQTT_ServerStatus gsmmgrMqttClientServerConnectionState();
```

```
typedef enum
{
    GSM_MQTT_ServerStatus_NotConnected,
    GSM_MQTT_ServerStatus_Disconnecting,
    GSM_MQTT_ServerStatus_Connecting,
    GSM_MQTT_ServerStatus_Connected,
    GSM_MQTT_ServerStatus_Unknown,
} GSM_MQTT_ServerStatus;
```

Event Handler

(This code is auto-generated when using the LibGSM Code Generator)

If an event handler callback is available ("EventHandler" field in the instance struct is set), then the following events may be raised by the MQTT Client (SIMCom LTE) module:

```
typedef enum
{
    GSM_MqttClientSimcom_EventType_ConnectSuccess,
    GSM_MqttClientSimcom_EventType_PublishSuccess,
    GSM_MqttClientSimcom_EventType_SubscribeSuccess,
    GSM_MqttClientSimcom_EventType_MsgRxTopic,
    GSM_MqttClientSimcom_EventType_MsgRxTopicHeaderError,
    GSM_MqttClientSimcom_EventType_MsgRxTopicError,
    GSM_MqttClientSimcom_EventType_MsgRxPayload,
    GSM_MqttClientSimcom_EventType_MsgRxPayloadHeaderError,
    GSM_MqttClientSimcom_EventType_MsgRxPayloadError,
    GSM_MqttClientSimcom_EventType_MsgRxEnd,
    GSM_MqttClientSimcom_EventType_UnsubscribeSuccess,
    GSM_MqttClientSimcom_EventType_DisconnectSuccess,
    GSM_MqttClientSimcom_EventType_ConnectionLost,
    GSM_MqttClientSimcom_EventType_NoNetwork,
    GSM_MqttClientSimcom_EventType_ErrorResponse,
} GSM_MqttClientSimcom_EventType;
```

The event data ("event_date"), if set, differs per event (details below).

GSM_MqttClientSimcom_EventType_ConnectSuccess

This event will be triggered once the modem has successfully connected to the broker (as a result of calling the gsmMqttClientSimcomConnect function).

GSM_MqttClientSimcom_EventType_PublishSuccess

This event will be triggered once the modem has successfully published a message (as a result of calling the gsmMqttClientSimcomPublish function).

GSM_MqttClientSimcom_EventType_SubscribeSuccess

This event will be triggered once the modem has successfully subscribed to a topic (as a result of calling the gsmMqttClientSimcomSubscribe function).

GSM_MqttClientSimcom_EventType_MsgRxTopic

This event will be triggered when the MQTT module receives the topic information for an incoming message from the modem. This is the first event in the incoming message routine.

Note that depending on the length of the topic, the MQTT module might need to break the topic into multiple segments. In this case, the event will repeat until all the data has been output.

The event data will be a pointer to the following struct:

```
typedef struct __GSM_MqttClientSimcom_Event_MsgRxTopic
GSM_MqttClientSimcom_Event_MsgRxTopic;
```

```
struct __GSM_MqttClientSimcom_Event_MsgRxTopic {
    uint16_t TopicLenTotal;
    uint16_t TopicLenSegment;
    uint16_t TopicSegmentOffset;
    char*    TopicSegment;
};
```

The struct fields are detailed below:

- TopicLenTotal – This is the total expected length of the topic for the message received.
- TopicLenSegment – This is the length of the topic segment that will be output (TopicSegment). Note that if the "TopicLenSegment" value is equal to the "TopicLenTotal" value then only one segment will be output (which will be the complete Topic Name). If "TopicSegmentOffset" + "TopicLenSegment" equals "TopicLenTotal" then this is the end of the Topic data.
- TopicSegmentOffset – This field will provide the offset from the start of the MQTT Topic to the start of the current segment (TopicSegment). This field can be used in combination with "TopicLenSegment" to determine if this is the last segment (TopicSegmentOffset + TopicLenSegment == TopicLenTotal).
- TopicSegment – This is the data segment of the topic. Note that if the "TopicLenSegment" value is equal to the "TopicLenTotal" value then this is the complete Topic Name; otherwise, this event will repeat until every topic segment is output, and you will need to combine each "TopicSegment" to obtain the complete Topic Name.

GSM_MqttClientSimcom_EventType_MsgRxTopicHeaderError

This event will only be triggered if the MQTT library module fails to read the topic header information (TopicLenSegment) that should be provided during the incoming message routine. This will result in a missed segment of the incoming topic data. See the MsgRxTopic event above.

GSM_MqttClientSimcom_EventType_MsgRxTopicError

This event will only be triggered if the MQTT module fails to read the topic segment (TopicSegment) data that is provided during the incoming message routine. See the MsgRxTopic event above.

GSM_MqttClientSimcom_EventType_MsgRxPayload

This event will be triggered when the MQTT module receives the payload information for an incoming message from the modem. This is part of the incoming message routine.

Note that depending on the length of the payload and the size of the buffers, it may be necessary to break the payload into multiple segments. In this case, this event will repeat until all the data has been output.

The event data will be a pointer to the following struct:

```
typedef struct __GSM_MqttClientSimcom_Event_MsgRxPayload
GSM_MqttClientSimcom_Event_MsgRxPayload;
struct __GSM_MqttClientSimcom_Event_MsgRxPayload
{
    uint32_t PayloadLenTotal;
    uint32_t PayloadLenSegment;
    uint16_t PayloadSegmentOffset;
    char*    PayloadSegment;
};
```

The struct fields are detailed below:

- PayloadLenTotal – This is the total expected length of the payload for the message received.
- PayloadLenSegment – This is the length of the payload segment that will be output (PayloadSegment). Note that if the "PayloadLenSegment" value is equal to the "PayloadLenTotal" value then only one segment will be output (which will be the complete Payload). If "PayloadSegmentOffset" + "PayloadLenSegment" equals "PayloadLenTotal" then this is the end of the Payload data.
- PayloadSegmentOffset – This field will provide the offset from the start of the MQTT Payload to the start of the current payload segment (PayloadSegment). This field can be used in combination with "PayloadLenSegment" to determine if this is the last segment (PayloadSegmentOffset + PayloadLenSegment == PayloadLenTotal).
- PayloadSegment – This is the data segment of the payload. Note that if the "PayloadLenSegment" value is equal to the "PayloadLenTotal" value then this is the complete Payload; otherwise, this event will repeat until every payload segment is output, and you will need to combine each "PayloadSegment" to obtain the complete Payload.

GSM_MqttClientSimcom_EventType_MsgRxPayloadHeaderError

This event will only be triggered if the MQTT module fails to read the payload header information (PayloadLenSegment) that is provided during the incoming message routine. This will result in a missed segment of the incoming payload data. See the MsgRxPayload event above.

GSM_MqttClientSimcom_EventType_MsgRxPayloadError

This event will only be triggered if the MQTT module fails to read the payload segment (PayloadSegment) data that is provided during the incoming message routine. See the MsgRxPayload event above.

GSM_MqttClientSimcom_EventType_MsgRxEnd

This event will be triggered when the MQTT module receives the "end of message" notification from the modem. This is the last event in the incoming message routine.

GSM_MqttClientSimcom_EventType_UnsubscribeSuccess

This event will be triggered once the modem has successfully unsubscribed from a topic (as a result of calling the gsmMqttClientSimcomUnsubscribe function).

GSM_MqttClientSimcom_EventType_DisconnectSuccess

This event will be triggered once the modem has successfully disconnected from the broker (as a result of calling the gsmMqttClientSimcomDisconnect function).

GSM_MqttClientSimcom_EventType_ConnectionLost

This event will only be triggered if the modem unexpectedly loses connection to the broker. The MQTT module will automatically attempt to reconnect. Note that if the connection to the host server was defined as a clean session, then all current topic subscriptions will be lost and you will need to resubscribe to them once the connection is reestablished.

If this event occurs frequently then try decreasing the value used for the "KeepAliveTime" field in the GSM_MqttClientSimcom_Connect_Session struct (which is used when connecting to the broker).

GSM_MqttClientSimcom_EventType_NoNetwork

This event will only be triggered if the modem loses network connection. The MQTT module will automatically attempt to reconnect. Note that if the connection to the host server was defined as a clean session, then all current topic subscriptions will be lost and you will need to resubscribe to them once the connection is reestablished.

GSM_MqttClientSimcom_EventType_ErrorResponse

This event will only be triggered if the MQTT module receives an error response from the modem. The event data will be a pointer to the following struct:

```
typedef struct __GSM_MqttClientSimcom_Event_ErrorResponse
GSM_MqttClientSimcom_Event_ErrorResponse;
struct __GSM_MqttClientSimcom_Event_ErrorResponse {
    GSM_MQTT_ErrorType ErrorType;
    uint8_t ErrorCode;
};
```

The GSM_MQTT_ErrorType enum is listed below:

```
typedef enum
{
    GSM_MQTT_ErrorType_MqttStartError,
    GSM_MQTT_ErrorType_AcquireClientIdError,
    GSM_MQTT_ErrorType_ConnectionError,
    GSM_MQTT_ErrorType_SubscribeError,
    GSM_MQTT_ErrorType_PublishError,
    GSM_MQTT_ErrorType_UnsubscribeError,
    GSM_MQTT_ErrorType_DisconnectError,
    GSM_MQTT_ErrorType_ReleaseClientIdError,
```

```
    GSM_MQTT_ErrorType_MqttStopError,
} GSM_MQTT_ErrorType;
```

The struct fields are detailed below:

- ErrorType – This is an Enum that provides more detail regarding during which part of the MQTT process the error occurred. For example, "GSM_MQTT_ErrorType_ConnectionError" would indicate that the error was received during the connection procedure. The ErrorType can be used in combination with the "ErrorCode" to determine why the error is being output.
- ErrorCode – This value can be used in combination with the modem datasheet (AT command manual) to determine what the error relates to. For example, error code 19 (SIMCom 7000 series modems) will be output when the "client is used"; if we continue the example from the previous point (ErrorType of GSM_MQTT_ErrorType_ConnectionError) then we can conclude that the connection to the broker is failing because the client ID you are using is already being used by another device connected to the broker.