UM2426
User manual

Getting started with the X-CUBE-CELLULAR cellular connectivity Expansion Package for STM32Cube

Introduction

This user manual describes the content and use of the X-CUBE-CELLULAR cellular connectivity Expansion Package for STM32Cube™.

The X-CUBE-CELLULAR Expansion Package enables connectivity over cellular networks. The network access technology depends on the cellular modem used: 2G, 3G, LTE Cat M1, or NB-IoT (also known as NB1). The cellular connectivity framework exposes standard APIs for easy integration of cloud connectors using the HTTP protocol.

The X-CUBE-CELLULAR Expansion Package for STM32Cube™ provides an application example that connects and subscribes to cloud services using the HTTP protocol in order to report data from the device to the server, as well as to receive commands from the remote server.

X-CUBE-CELLULAR is available for both P-L496G-CELL01 and P-L496G-CELL02 cellular-to-cloud packs. Each pack is composed of an STM32L496-based Discovery host board connected to an add-on cellular modem through the STMod+ connector. The add-on board of pack P-L496G-CELL01 is equipped with the UG96 modem (2G / 3G). The add-on board of pack P-L496G-CELL02 is equipped with the BG96 modem (LTE Cat M / NB-IoT / 2G fallback).

X-CUBE-CELLULAR is also available for the “Discovery IoT node cellular” set, which is a combination of the B-L475E-IOT01A IoT discovery board, X-NUCLEO-STMODA1 Arduino™ / STMod+ adapter, and MB1329 modem board with the BG96 modem.

The main features of the X-CUBE-CELLULAR Expansion Package are:

- Ready-to-run firmware examples using the 2G, 3G, LTE Cat M1, or NB-IoT protocols to support quick evaluation and development of IoT cloud applications
- Menu and command line through Virtual COM UART over USB ST-LINK to configure the connection to the Grovestreams cloud IoT platform (HTTP), and cellular connectivity (technology selection, bands, APN, and others)
- Cellular connection
- Reporting of such values as temperature, humidity, and pressure. The values are real if the MEMS add-on board (X-NUCLEO-IKS01A2) is connected, otherwise they are simulated. The sensors in the “Discovery IoT node cellular” set are always used
- Network radio level reporting

Contents

List of tables

List of figures

1 General information

This user manual describes the X-CUBE-CELLULAR Expansion Package and its use. It explains neither the cellular networks nor the cellular protocol stacks, the descriptions of which being available on the Internet.

This user manual primarily focuses on the P-L496G-CELL01 and P-L496G-CELL02 Discovery packs for descriptions, way of use, and examples. In this user manual, root path *Projects\STM32L496G-Discovery\* must be changed to *Projects\STM32L475E-Discovery\* when applying to the “Discovery IoT node cellular” set.

Refer to the *X-CUBE-CELLULAR cellular connectivity Expansion Package porting on other hardware* application note (AN5249) for adaptation to other hardware such as the “Discovery IoT node cellular” set.

1.1 Terms and definitions

Table 1 presents the definition of acronyms that are relevant for a better understanding of this document.

Table 1. List of acronyms

Term	Definition
API	Application programming interface
APN	Access point name
BSD	Berkeley software distribution
BSP	Board support package
C2C	Cellular to cloud
CID	Context ID (context identifier of a cellular connection)
COM	Cellular communication
DC	Data Cache
eUICC	Embedded UICC (UICC with remote profile feature)
eSIM	Embedded SIM
FEEPROM	Represents the embedded Flash memory of the STM32 MCU
HAL	Hardware abstraction layer
HTTP	Hypertext transfer protocol
ICMP	Internet message control protocol
IDE	Integrated development environment
IF	Interface
IoT	Internet of things (refer to *[4]*)
IPC	Inter-processor channel
ITM	Instruction trace module
LED	Light-emitting diode

Table 1. List of acronyms (continued)

Term	Definition
M2M	Machine to machine
NAT	Network address translation
NFMC	Network-friendly management configuration (refer to [4])
NIFMAN	Network IF manager
MNO	Mobile network operator
MVNO	Mobile virtual operator
PDN	Packet data network
PDU	Protocol data unit
PLMN	Public land mobile network
PPP	Point-to-point protocol
PPPoSIF	PPP over serial IF
PS	Packet switching
RAM	Random-access memory
ROM	Read-only memory
RSSI	Received-signal strength indication
RTC	Real-time clock
SMS	Short-message service
TCP	Transmission control protocol
UDP	User datagram protocol
UICC	Universal integrated circuit card (also referred to as SIM card)
URC	Unsolicited result code

The X-CUBE-CELLULAR Expansion Package runs on STM32 32-bit microcontrollers based on the Arm®[a] Cortex®-M processor.

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.2 References

1. *Development guidelines for STM32Cube Expansion Packages* user manual (UM2285)
2. *Development checklist for STM32Cube Expansion Packages* user manual (UM2312)
3. *Getting started with STM32CubeL4 for STM32L4 Series and STM32L4+ Series* user manual (UM1860)
4. *IoT Device Connection Efficiency Guidelines* (TSG.34/TS.34) from the GSM Association
5. *X-CUBE-CELLULAR cellular connectivity Expansion Package porting on other hardware* application note (AN5249)

2 Important note regarding the security

Caution: Application developers must take care of security aspects, and put mechanisms in place to protect the tokens and secrets used for the connections.

The application example provided in the X-CUBE-CELLULAR Expansion Package does not implement such protection mechanisms. It only presents a basic implementation for an easy understanding of the stack interface.

Warning: Use the HW only with the antenna connected. With no antenna connected, there is a risk of damage to the modem because of the power reflected from the antenna connector to the modem RF output.

---

3 Service connectivity description

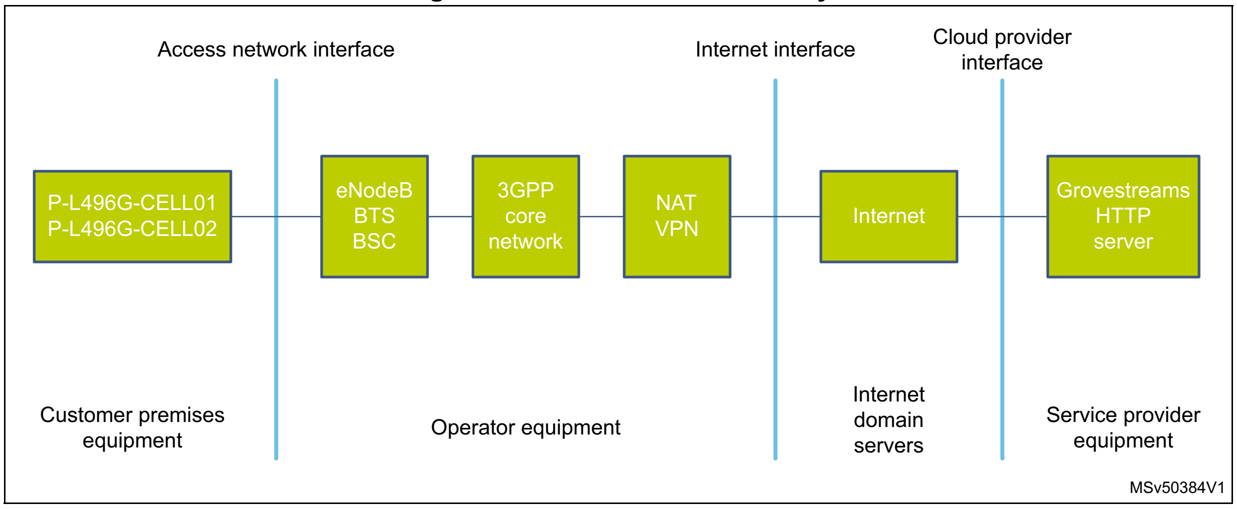
The X-CUBE-CELLULAR Expansion Package offers out-of-the-box connectivity for communication to the Internet through the HTTP protocol. It implements a complete middleware- and application-level stack in C language, which allows the connection of the C2C kit to a web site.

The first connectivity example provided connects to the Grovestreams web site. In this example, the board reports notifications to the Grovestreams web browser.

The second example provided implements the ping network-testing feature.

Figure 1 presents the cellular IoT connectivity handled by the X-CUBE-CELLULAR Expansion Package.

Figure 1. Cellular IoT connectivity



The cellular-to-cloud kit comprises an STM32-based main board, a cellular add-on modem board, and a prepaid SIM, which enables the registration to a cellular PLMN. The global roaming provided by the SIM provider allows device attachment from any country. The SIM only offers IP connectivity (meaning that SMS is not supported). The volume of data available in the prepaid offer depends on where it is used.

A private IP address is allocated to the device by the MNO or MVNO. Any client application running on device using TCP transaction request/response can reach a server located in the Internet by means of IP address translation (NAT) on the MNO/MVNO router.

4 Package description

This chapter details the content and use of the X-CUBE-CELLULAR Expansion Package.

4.1 General description

The X-CUBE-CELLULAR Expansion Package only provides software components running on the host STM32 MCU. Cellular modem firmware is not in the scope of this document.

The following integrated development environments are supported:

- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit (MDK-ARM)
- System Workbench for STM32 (referred to as SW4STM32)

Note: Refer to the release note available in the root folder of the delivery package for information about the IDE versions supported.

IAR™ binaries are provided in the package.

4.2 Modem socket versus LwIP

Either modem socket or LwIP can be used for the IP stack:

- Modem socket: the IP stack runs in modem FW
- LwIP: the LwIP stack runs on the STM32 side

This option is selected through a flag that is used during the compilation process. The generated FW is either for modem socket or for LwIP use. It is not possible to further change this setting through the boot menu.

Note: If modem socket is used, the software described in this user manual limits data plane support to TCP IPv4 Client application only. TCP server mode and UDP (both server and client) are not yet supported.

If LwIP is used, TCP and UDP (both server and client) are fully supported.

If LwIP is selected, the communication between host and modem is done through the PPP layer. There is a PPP client on the host side, and a PPP server on the modem side. PPPoSIF adapts the LwIP stack to a serial IF, while LwIP usually uses Ethernet interfacing.

Note: The LwIP mode is not supported for the “Discovery IoT node cellular” set.

4.3 Architecture

X-CUBE-CELLULAR runs on STM32 boards and allows sending or receiving IP packets to or from the Internet via an add-on cellular module.

Note: Some parts of X-CUBE-CELLULAR can be used in a bare OS environment. The complete stack only runs with FreeRTOS™.

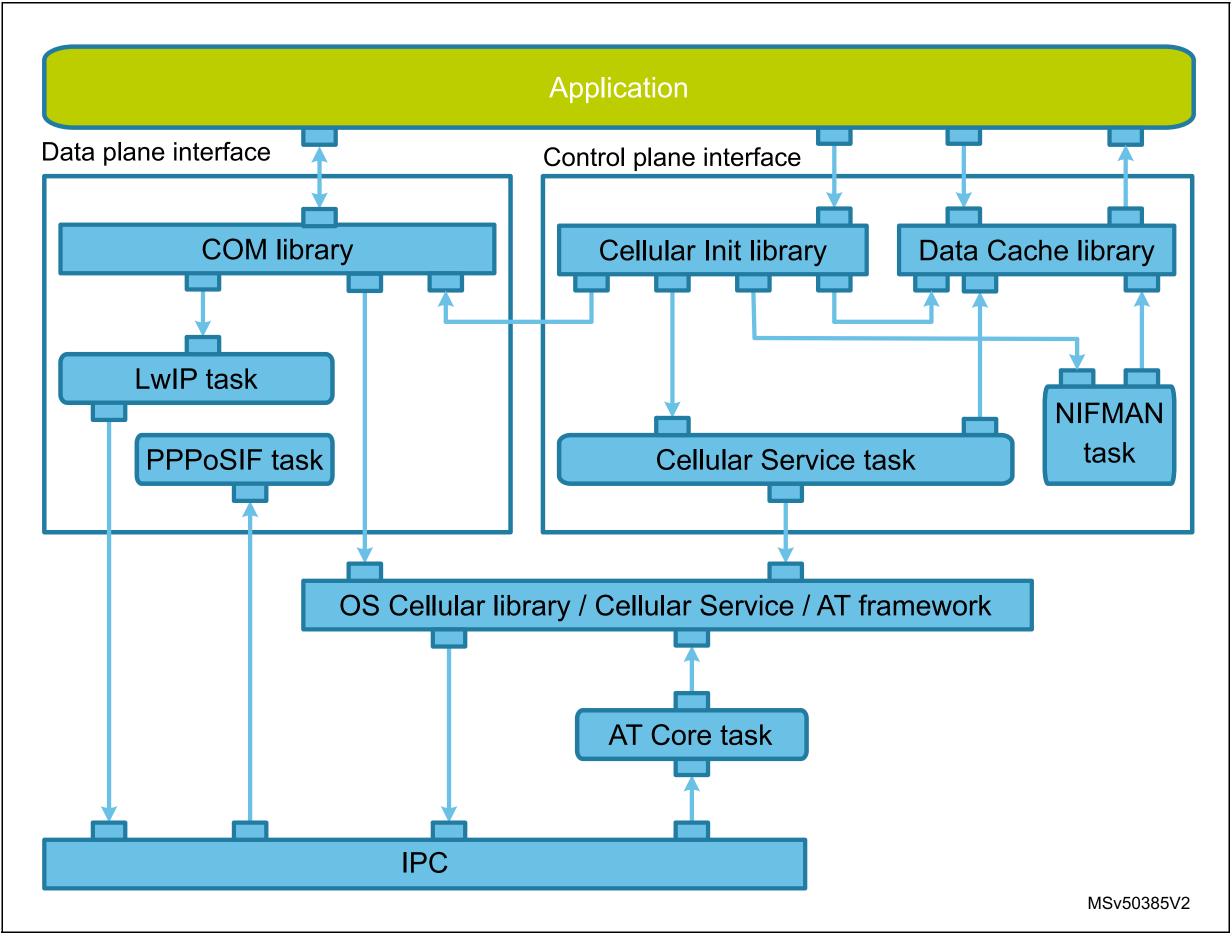
The package is split into the following components:

- STM32L4 Series HAL
- CMSIS/FreeRTOS™
- LwIP
- AT Service
- Cellular Service
- Data Cache
- IPC
- NIFMAN
- COM
- Cellular Init
- Utilities

4.3.1 Architecture concept

This section provides a high-level view of the software architecture supporting cellular connectivity, which is illustrated in *Figure 2*.

Figure 2. Architecture concept



The cellular connectivity stack exposes two main interfaces to the application:

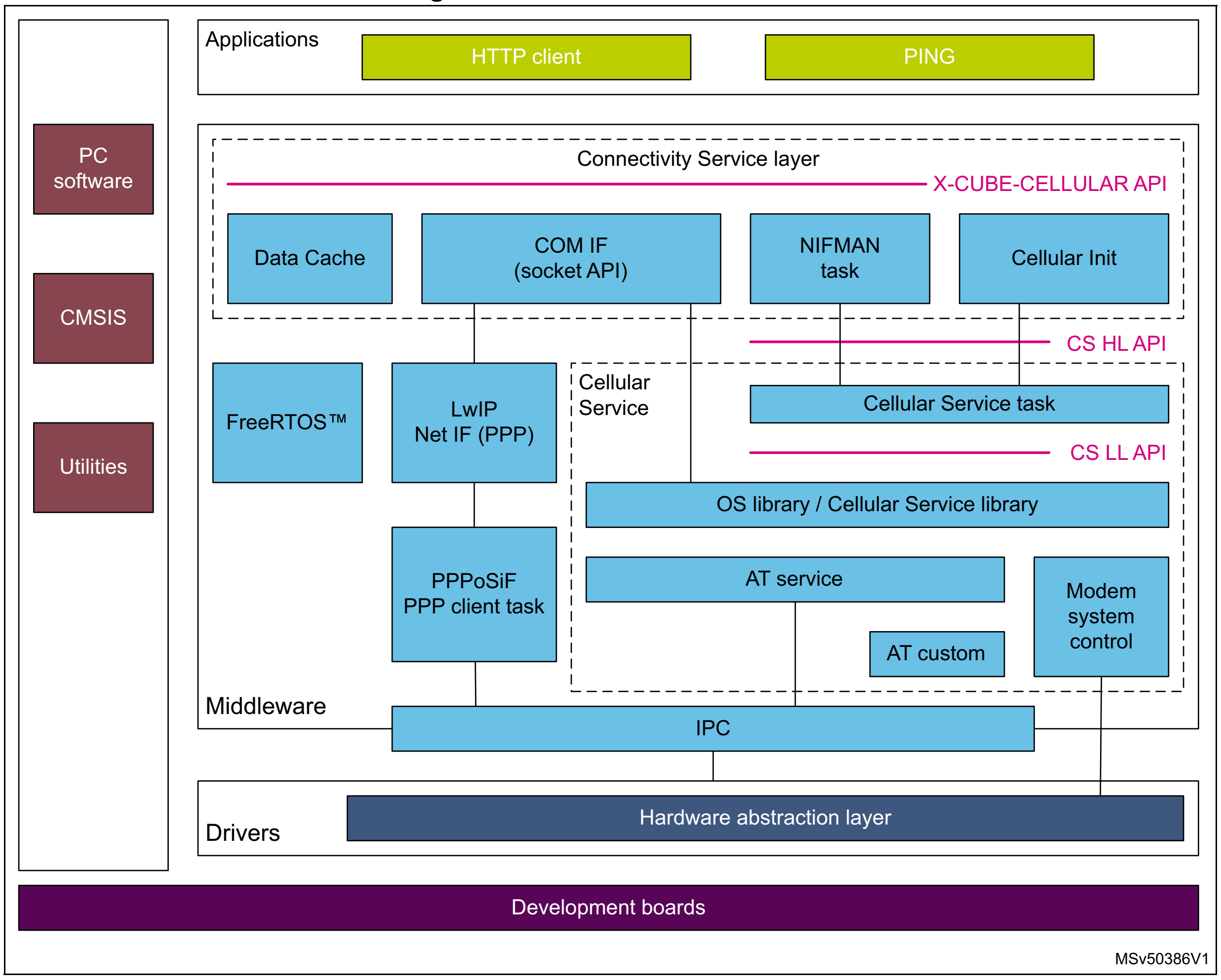
- The control plane interface: there are two interfaces for control. The cellular init library provides an API to initialize SW components and starts the Cellular Service. The Data Cache interface is used to read information related to cellular network like Signal Strength. (RSSI), and to get event notification like network registration state changes and network interface readiness.
- The data plane interface: also referred to as the COM interface, it is used to send and receive TCP or UDP segments to and from a remote client or server. The interface is based on standard BSD socket API in order to ease the integration of the application.

The IPC layer abstracts the actual HW bus interface used with the modem. The IPC supports two logical channels, each composed of one Tx (to the IPC) and one Rx (from the IPC). One is used for exchanging AT commands with the modem while the other one is used to carry PPP frames when LwIP is used. The selection of the active channel is controlled by Cellular Service.

4.3.2 Static architecture view

X-CUBE-CELLULAR static architecture is presented in *Figure 3*.

Figure 3. Static architecture view



- **HTTP client**: implements an HTTP client, which sends requests to the www.grovestreams.com cloud in the application. The HTTP client uses the Data Cache to monitor the network interface state changes (from NIFMAN), and the COM socket interface to send or receive HTTP packets over TCP. The HTTP client also implements recovery as defined by GSMA TS 34 when a remote HTTP server is not reachable.
- **Connectivity Service layer**:
  – **Data Cache**: framework that decouples the management of producer and consumer data (resource). Any resource state updated by a producer is pushed to the Data Cache (in RAM), which in turn informs the final consumer(s) to process the updated resource state via the callback provided by the consumer application. Data Cache is used by Cellular Service tasks to publish the *Cellular network information like RSSI*. It is also used by NIFMAN to publish the network interface readiness.
  – **COM IF**: a library that provides a collection of BSD-like socket functions to open, configure, and send or receive application PDU to remote TCP or UDP applications. A high-level ping service is also provided.
  – **NIFMAN**: the network interface manager task controls network interface activation. When LwIP is used, NIFMAN monitors the PPP server status (on the modem side), and starts or stops the PPP client accordingly. The application can then monitor the network interface status before opening a socket for data transfer.
  – **Cellular init library**: exposes a basic function to initialize and start Cellular Service components.
- **PPP client task / PPPoSIF**: optional component. It is only present when LwIP is used. It is in charge of establishing the PPP link with the modem.
- **LwIP / Net IF**: the LwIP component and its adaptation to PPP.
- **Cellular Service**:
  – **Cellular Service task** controls modem power-on and initialization, instructs the modem to perform network registration, activates the PDN (PDP context), and enters data transfer mode. It informs NIFMAN to setup the network interface (PPP link). It uses AT service to send AT commands to the modem.

    It implements a generic finite state machine to maintain consistent service state based on modem internal state change events (such as FOTA or reset), network registration state change events, and events related to the and PDP context status. It implements the network friendly features (NFMC) as defined in *[4]*. For example, when PDP activation fails because of a wrong APN, the Cellular Service task performs a new attempt after expiration of a back-off timer.

    The Cellular Service task stores the cellular configuration and network access parameters into the Flash memory, and configure the modem as per need. The configuration for example encompasses APN and CID settings, enabling and disabling NFMC, and setting the back-off timers.

    For system robustness, the Cellular Service task ensures that the modem is always operational by regularly polling the modem RSSI.
  – **Cellular Service OS**: is a library that offers a collection of functions to low-level Cellular Service. The library serializes the access to the single AT channel interface that is used to communicate with the modem. The functions are called by the COM service and the Cellular Service task.
  – **Cellular Service library**: offers a collection of blocking function calls to interact with a modem. Cellular Service is in charge of translating the request from the

Cellular Service task or COM service to a sequence of AT commands that must be sent to the modem. It finally calls a callback function (from the Cellular Service task or COM service) when an asynchronous event (URC) is received from the modem.

  - **AT Service**: provides a framework to send or receive AT commands to or from the modem over IPC. The AT Core task is in charge of processing the Cellular Service requests and translate them into AT commands. It is also in charge of processing AT commands response and URCs from the modem and forward them to Cellular Service. AT is split into two parts:

    1. a generic part, "core" (AT framework and manage standard AT commands)

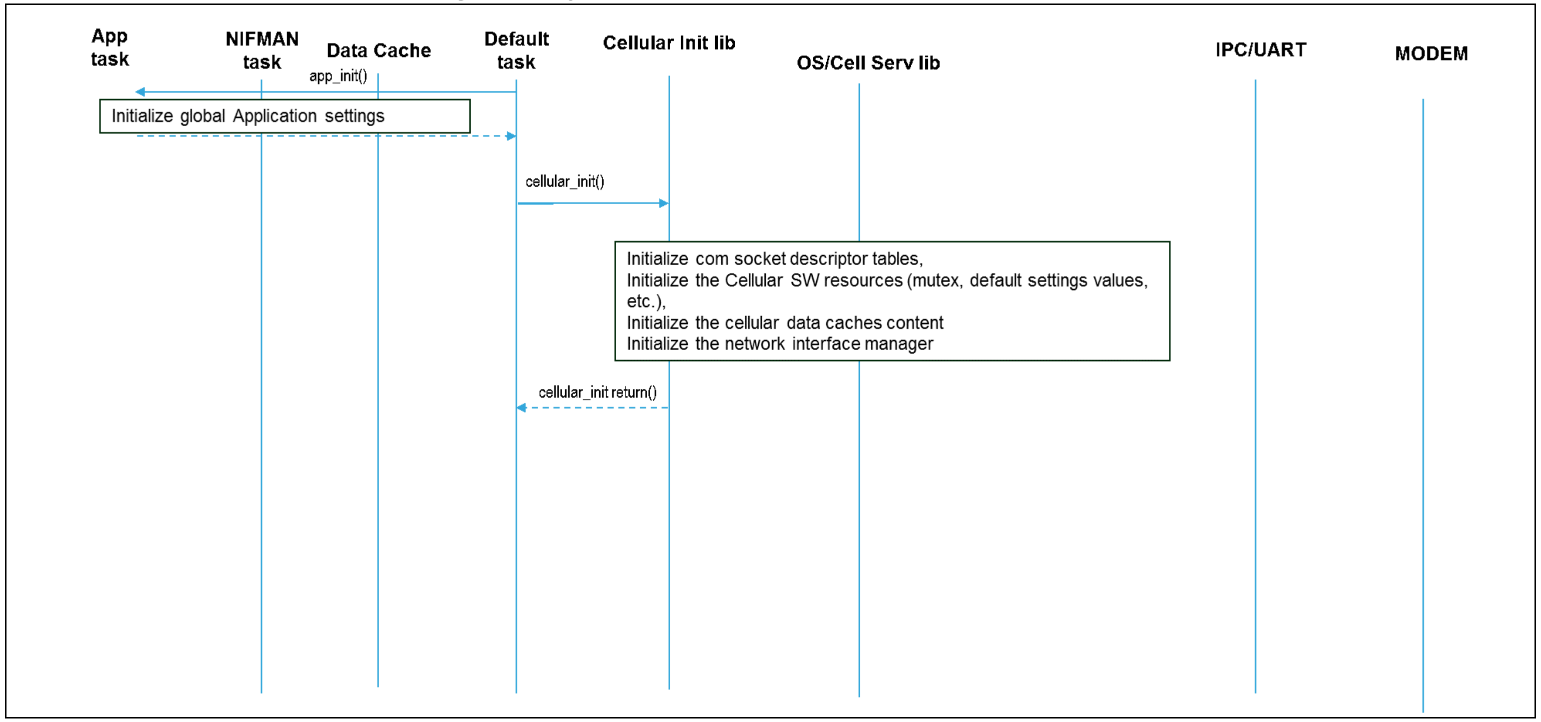
    2. a specific part, "custom" (implements specific modem behavior and AT commands)
  - **Modem system control**: support modem HW system control signaling (power on/off, reset). It is split into a generic and a specific part. The generic part exposes the generic API to the application (Cellular Service) while the specific part controls the GPIO dedicated to the modem.
- **IPC**: abstracts the actual physical interface (UART) to the upper layer. Supports the logical channel handler (FIFO) that is mapped to a physical channel. Supports two channels: character mode and stream mode:
  - Stream mode is used for data transfer (PPP).
  - Character mode is used to send AT commands.
- **Utilities**: provides tools such as debug and trace. Also provides the setup menu (over any terminal through a serial interface) to change the default configuration, which is hard coded during compilation and image creation.
- **FreeRTOS™ (and CMSIS)**: provides RTOS services to create the resources and scheduler needed by the software to run, such as threads and tasks, dynamic memory allocation, mutexes, and semaphores. A default task (*freertos.c*) is in charge of system initialization, creation of all application tasks. It finally initializes and starts the Cellular Service components by calling `cellular_init()` and `cellular_start()`.

4.3.3 Dynamic architecture view

X-CUBE-CELLULAR dynamic architecture is further presented as diagram sequences that illustrate the interactions between components for selected use cases:

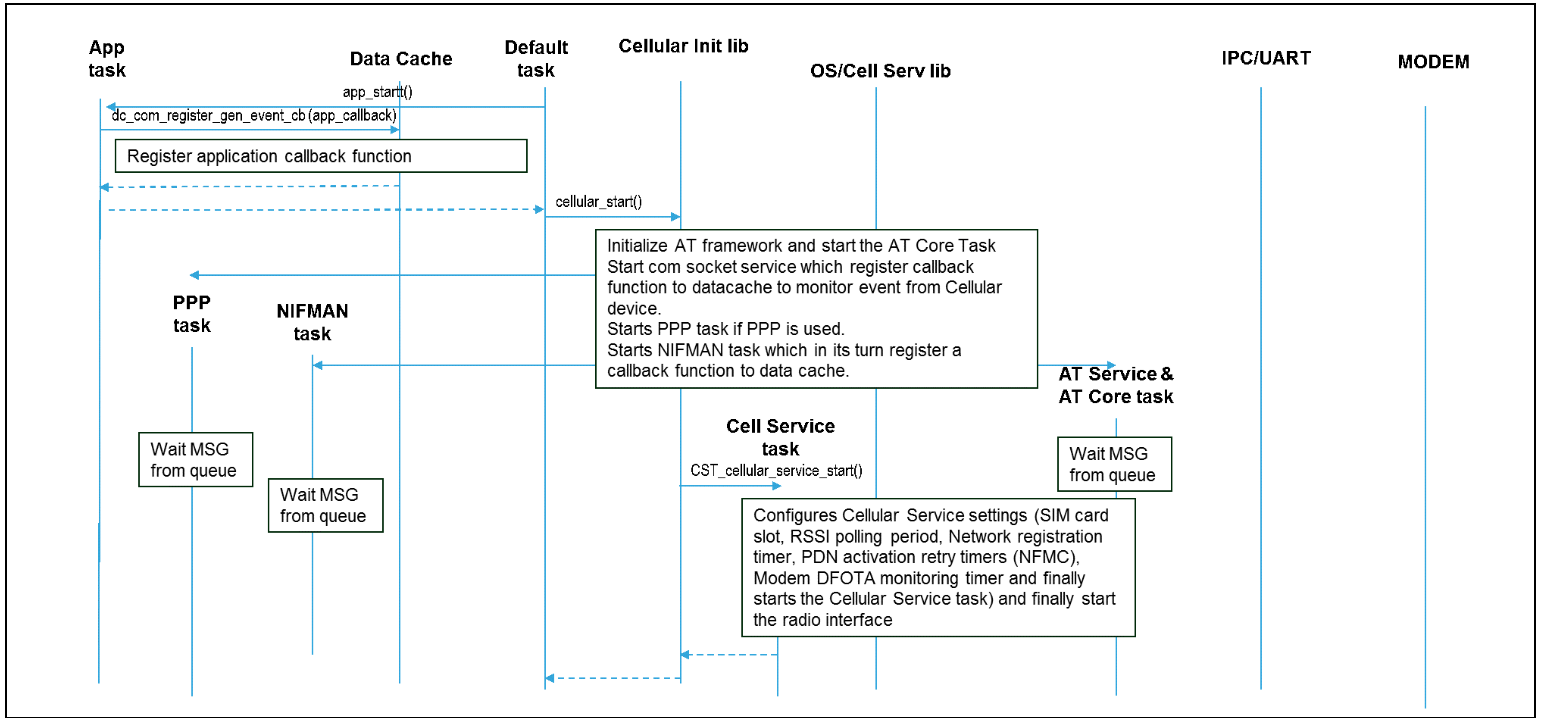
- *Figure 4: Dynamic architecture - Platform initialization*
- *Figure 5: Dynamic architecture - Platform service startup*
- *Figure 6: Dynamic architecture - Cellular Service start*
- *Figure 7: Dynamic architecture - Modem initialization*
- *Figure 8: Dynamic architecture - PLMN search and registration to a cellular network*
- *Figure 9: Dynamic architecture - Registration to a packet service domain*
- *Figure 10: Dynamic architecture - PDN activation (modem socket option)*
- *Figure 11: Dynamic architecture - PDN activation (LwIP socket option with UG96)*
- *Figure 12: Dynamic architecture - PDN activation (LwIP socket option with BG96)*
- *Figure 13: Dynamic architecture - Socket creation (modem socket option)*
- *Figure 14: Dynamic architecture - Socket creation (LwIP socket option)*
- *Figure 15: Dynamic architecture - Data transfer - Send data to remote application (modem socket option)*
- *Figure 16: Dynamic architecture - Data transfer - Send data to remote application (LwIP socket option)*
- *Figure 17: Dynamic architecture - Data transfer - Receive data from remote application (modem socket option)*
- *Figure 18: Dynamic architecture - Data transfer - Receive data from remote application (LwIP socket option)*

Figure 4. Dynamic architecture - Platform initialization


App task
NIFMAN task
Data Cache
Default task
Cellular Init lib
OS/Cell Serv lib
IPC/UART
MODEM
app_init()
Initialize global Application settings
cellular_init()
Initialize com socket descriptor tables,
Initialize the Cellular SW resources (mutex, default settings values, etc.),
Initialize the cellular data caches content
Initialize the network interface manager
cellular_init return()

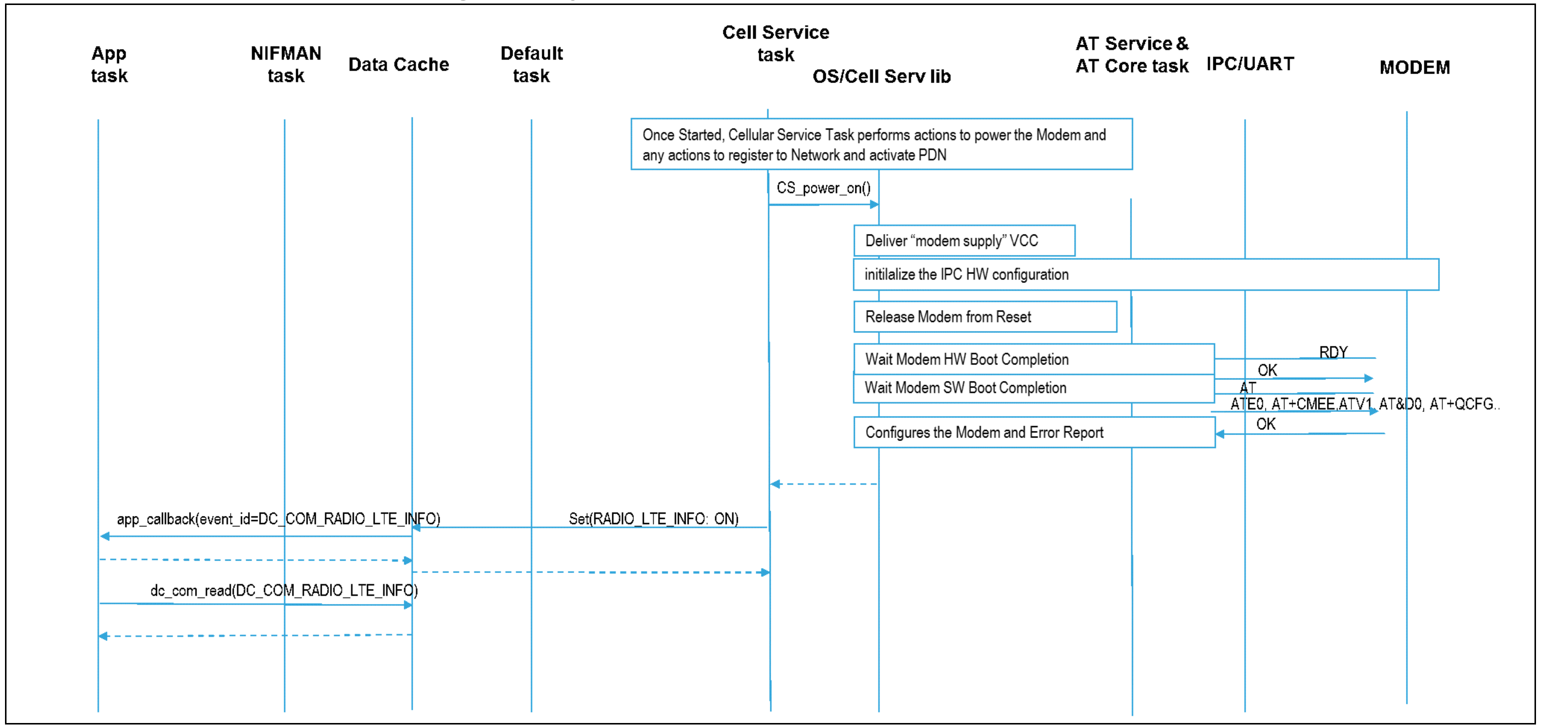
The default task calls `cellular_init` to initialize all software components and needed static resources.

Figure 5. Dynamic architecture - Platform service startup


App task
Data Cache
Default task
Cellular Init lib
OS/Cell Serv lib
IPC/UART
MODEM
app_start()
dc_com_register_gen_event_cb (app_callback)
Register application callback function
cellular_start()
Initialize AT framework and start the AT Core Task
Start com socket service which register callback function to datacache to monitor event from Cellular device.
Starts PPP task if PPP is used.
Starts NIFMAN task which in its turn register a callback function to data cache.
PPP task
NIFMAN task
AT Service & AT Core task
Wait MSG from queue
Wait MSG from queue
Wait MSG from queue
Cell Service task
CST_cellular_service_start()
Configures Cellular Service settings (SIM card slot, RSSI polling period, Network registration timer, PDN activation retry timers (NFMC), Modem DFOTA monitoring timer and finally starts the Cellular Service task) and finally start the radio interface

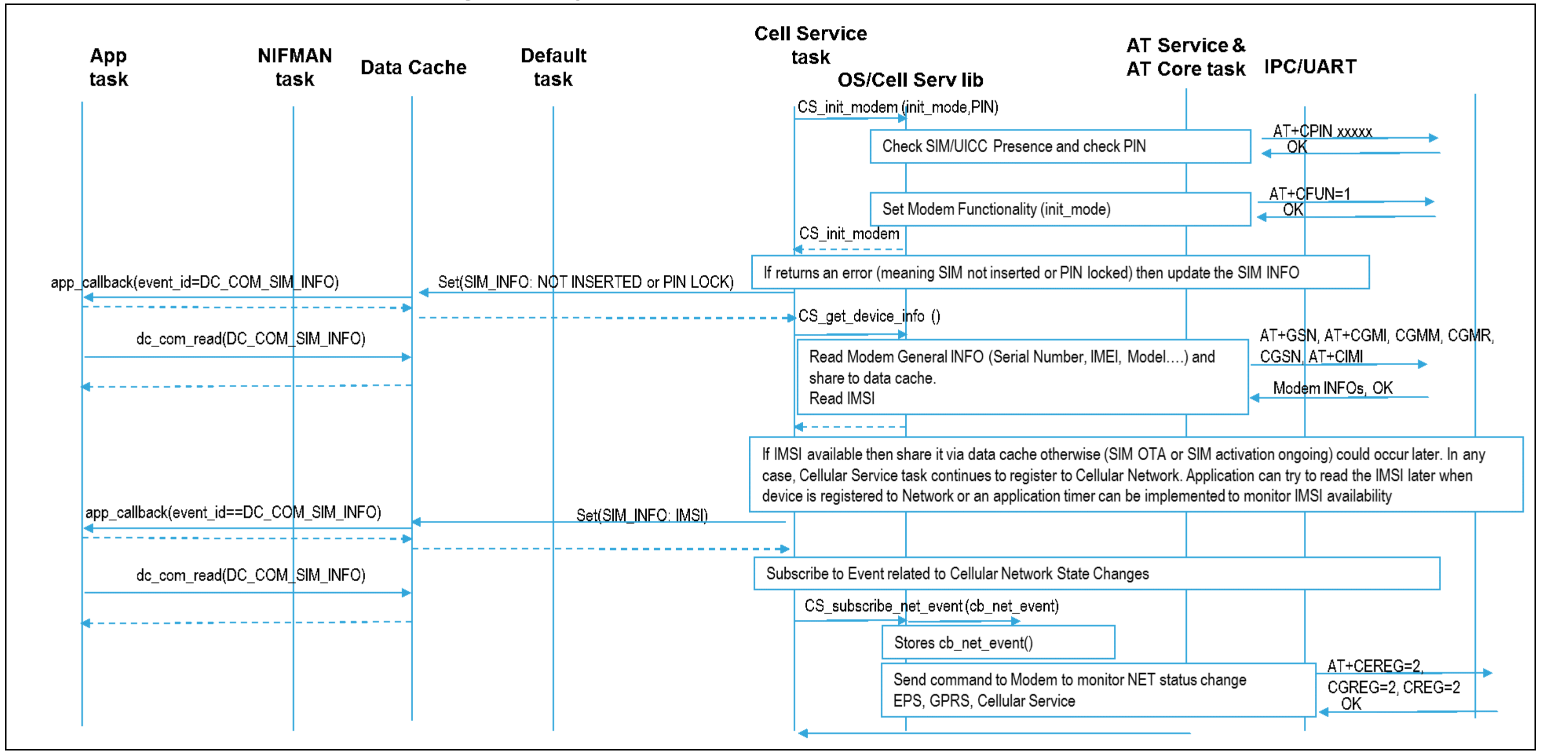
The default task calls the `cellular_start()` function to start software components tasks and to power on the modem device.

Figure 6. Dynamic architecture - Cellular Service start


App task
NIFMAN task
Data Cache
Default task
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
Once Started, Cellular Service Task performs actions to power the Modem and any actions to register to Network and activate PDN
CS_power_on()
Deliver "modem supply" VCC
initilalize the IPC HW configuration
Release Modem from Reset
Wait Modem HW Boot Completion
Wait Modem SW Boot Completion
RDY
OK
AT
ATE0, AT+CMEE,ATV1, AT&D0, AT+QCFG..
OK
Configures the Modem and Error Report
app_callback(event_id=DC_COM_RADIO_LTE_INFO)
Set(RADIO_LTE_INFO: ON)
dc_com_read(DC_COM_RADIO_LTE_INFO)

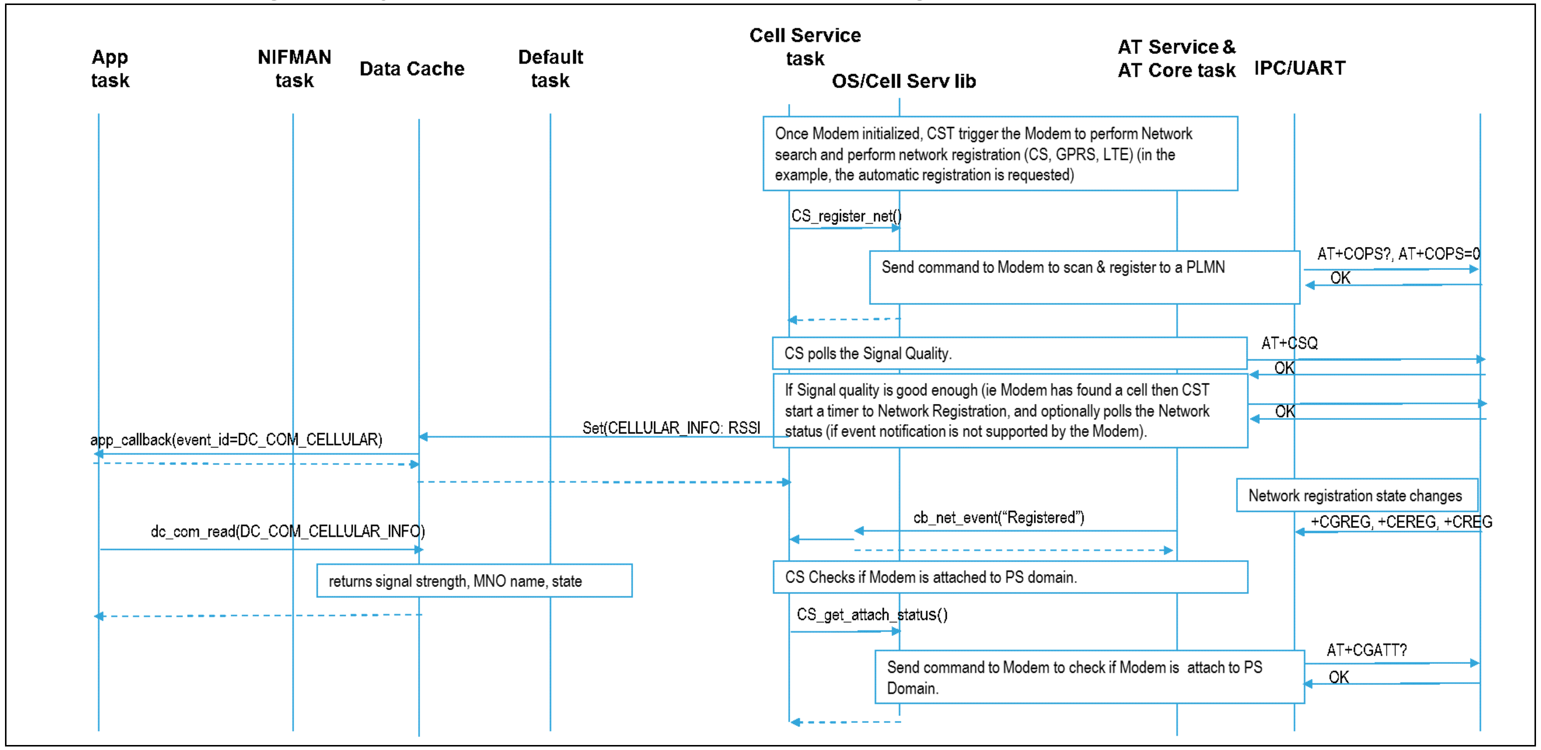
Once the Cellular Service task is started, it automatically performs all needed actions to power on the modem device.

Figure 7. Dynamic architecture - Modem initialization


App task
NIFMAN task
Data Cache
Default task
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
CS_init_modem (init_mode,PIN)
Check SIM/UICC Presence and check PIN
AT+CPIN xxxxx
OK
Set Modem Functionality (init_mode)
AT+CFUN=1
OK
CS_init_modem
If returns an error (meaning SIM not inserted or PIN locked) then update the SIM INFO
app_callback(event_id=DC_COM_SIM_INFO)
Set(SIM_INFO: NOT INSERTED or PIN LOCK)
CS_get_device_info ()
dc_com_read(DC_COM_SIM_INFO)
Read Modem General INFO (Serial Number, IMEI, Model....) and share to data cache.
Read IMSI
AT+GSN, AT+CGMI, CGMM, CGMR, CGSN, AT+CIMI
Modem INFOs, OK
If IMSI available then share it via data cache otherwise (SIM OTA or SIM activation ongoing) could occur later. In any case, Cellular Service task continues to register to Cellular Network. Application can try to read the IMSI later when device is registered to Network or an application timer can be implemented to monitor IMSI availability
app_callback(event_id==DC_COM_SIM_INFO)
Set(SIM_INFO: IMSI)
Subscribe to Event related to Cellular Network State Changes
dc_com_read(DC_COM_SIM_INFO)
CS_subscribe_net_event (cb_net_event)
Stores cb_net_event()
Send command to Modem to monitor NET status change EPS, GPRS, Cellular Service
AT+CEREG=2, CGREG=2, CREG=2
OK

Once the modem is powered up, the Cellular Service task initializes modem functionalities (full features, boot only).

Figure 8. Dynamic architecture - PLMN search and registration to a cellular network

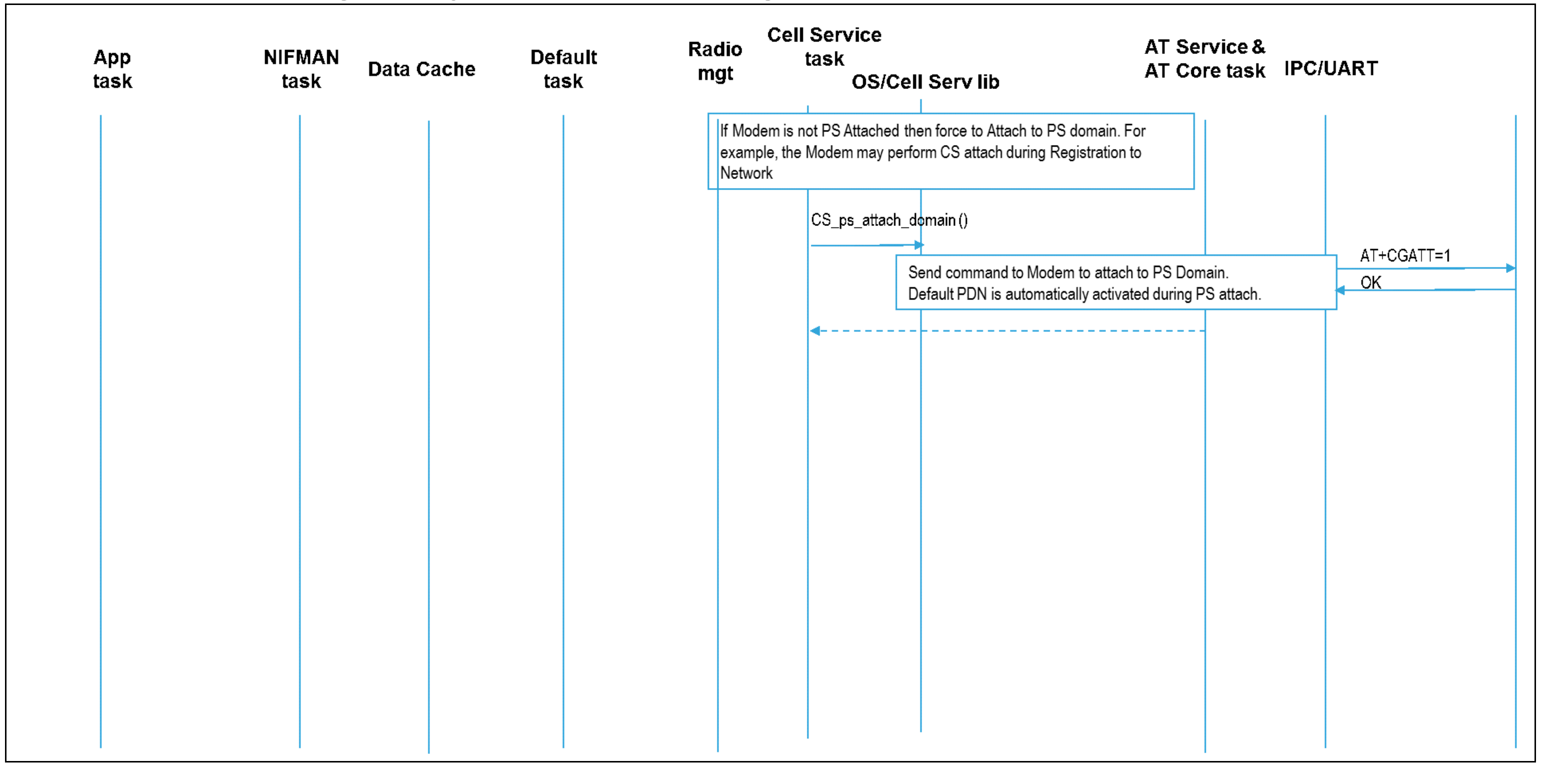

App task
NIFMAN task
Data Cache
Default task
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
Once Modem initialized, CST trigger the Modem to perform Network search and perform network registration (CS, GPRS, LTE) (in the example, the automatic registration is requested)
CS_register_net()
Send command to Modem to scan & register to a PLMN
AT+COPS?, AT+COPS=0
OK
CS polls the Signal Quality.
AT+CSQ
OK
If Signal quality is good enough (ie Modem has found a cell then CST start a timer to Network Registration, and optionally polls the Network status (if event notification is not supported by the Modem).
OK
Set(CELLULAR_INFO: RSSI
app_callback(event_id=DC_COM_CELLULAR)
Network registration state changes
cb_net_event("Registered")
+CGREG, +CEREG, +CREG
dc_com_read(DC_COM_CELLULAR_INFO)
returns signal strength, MNO name, state
CS Checks if Modem is attached to PS domain.
CS_get_attach_status()
AT+CGATT?
Send command to Modem to check if Modem is attach to PS Domain.
OK

Once the modem context is initialized, the Cellular Service task initiates the modem to perform network search (PLMN search) and register to the selected network. In the existing implementation, only automatic network search is supported.

Cellular Service regularly polls the signal quality to detect if a cell has been selected by the modem. Once a cell is selected, the Cellular Service task starts a timer to monitor network registration since there is a possibility that registration is rejected by the core network. When the network registration timer expires, the Cellular Service task instructs the modem to try network registration later (after back-off timer expiry) as defined in *[4]*.

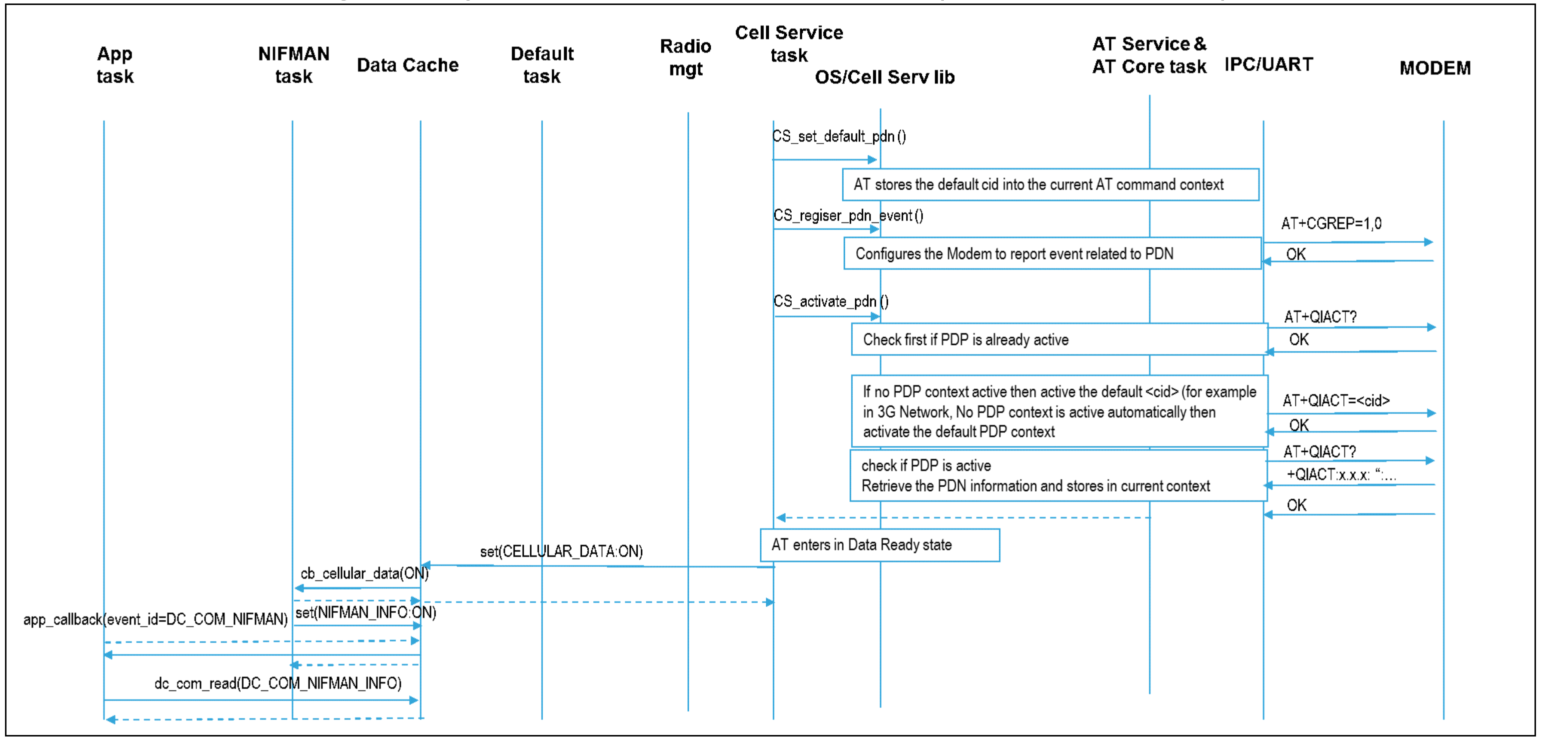
Once network registration is complete, the Cellular Service task verifies if the modem is registered to the packet domain. In 2G or 3G, the modem may only register to Circuit Service, whereas in LTE the modem is always attached to the packet network when registered.

Figure 9. Dynamic architecture - Registration to a packet service domain


App task
NIFMAN task
Data Cache
Default task
Radio mgt
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
If Modem is not PS Attached then force to Attach to PS domain. For example, the Modem may perform CS attach during Registration to Network
CS_ps_attach_domain ()
Send command to Modem to attach to PS Domain. Default PDN is automatically activated during PS attach.
AT+CGATT=1
OK

Since the target is to run IP connectivity, Cellular Service forces the modem to perform packet domain registration if it is not yet done automatically by the modem.

Figure 10. Dynamic architecture - PDN activation (modem socket option)


App task
NIFMAN task
Data Cache
Default task
Radio mgt
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
CS_set_default_pdn ()
AT stores the default cid into the current AT command context
CS_regiser_pdn_event ()
AT+CGREP=1,0
OK
Configures the Modem to report event related to PDN
CS_activate_pdn ()
AT+QIACT?
OK
Check first if PDP is already active
If no PDP context active then active the default <cid> (for example in 3G Network, No PDP context is active automatically then activate the default PDP context
AT+QIACT=<cid>
OK
AT+QIACT?
+QIACT:x.x.x: ":...
OK
check if PDP is active
Retrieve the PDN information and stores in current context
AT enters in Data Ready state
set(CELLULAR_DATA:ON)
cb_cellular_data(ON)
set(NIFMAN_INFO:ON)
app_callback(event_id=DC_COM_NIFMAN)
dc_com_read(DC_COM_NIFMAN_INFO)

Once the modem is attached to the PS domain, the Cellular Service task registers a callback to get the notifications related to the PDN (PDP context) from the modem. It also configures and selects the PDP context to be used for data transfer.

The application can register a callback (for example `app_client_notif_cb()`) to get notified about the readiness of the network interface (ON).

Figure 11. Dynamic architecture - PDN activation (LwIP socket option with UG96)

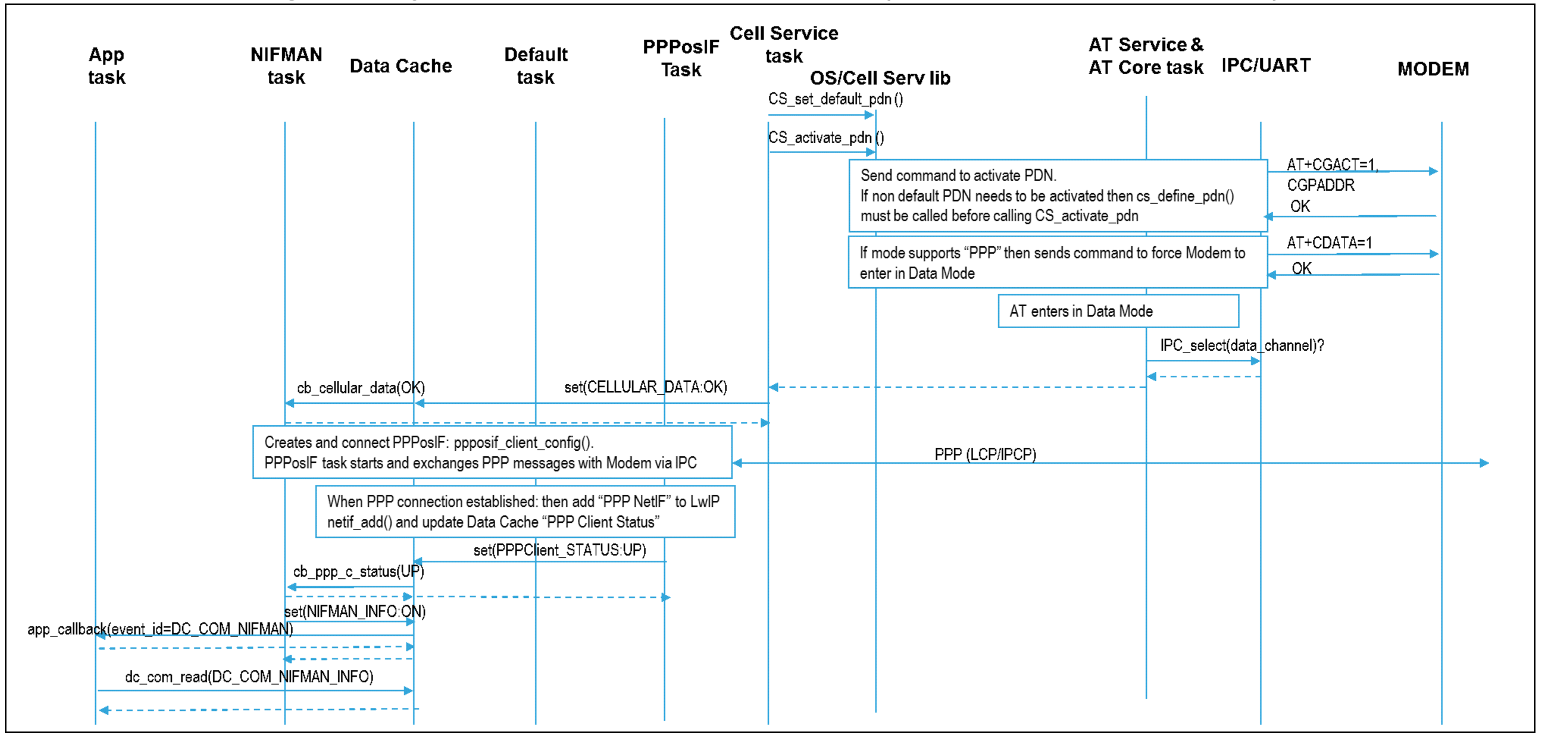
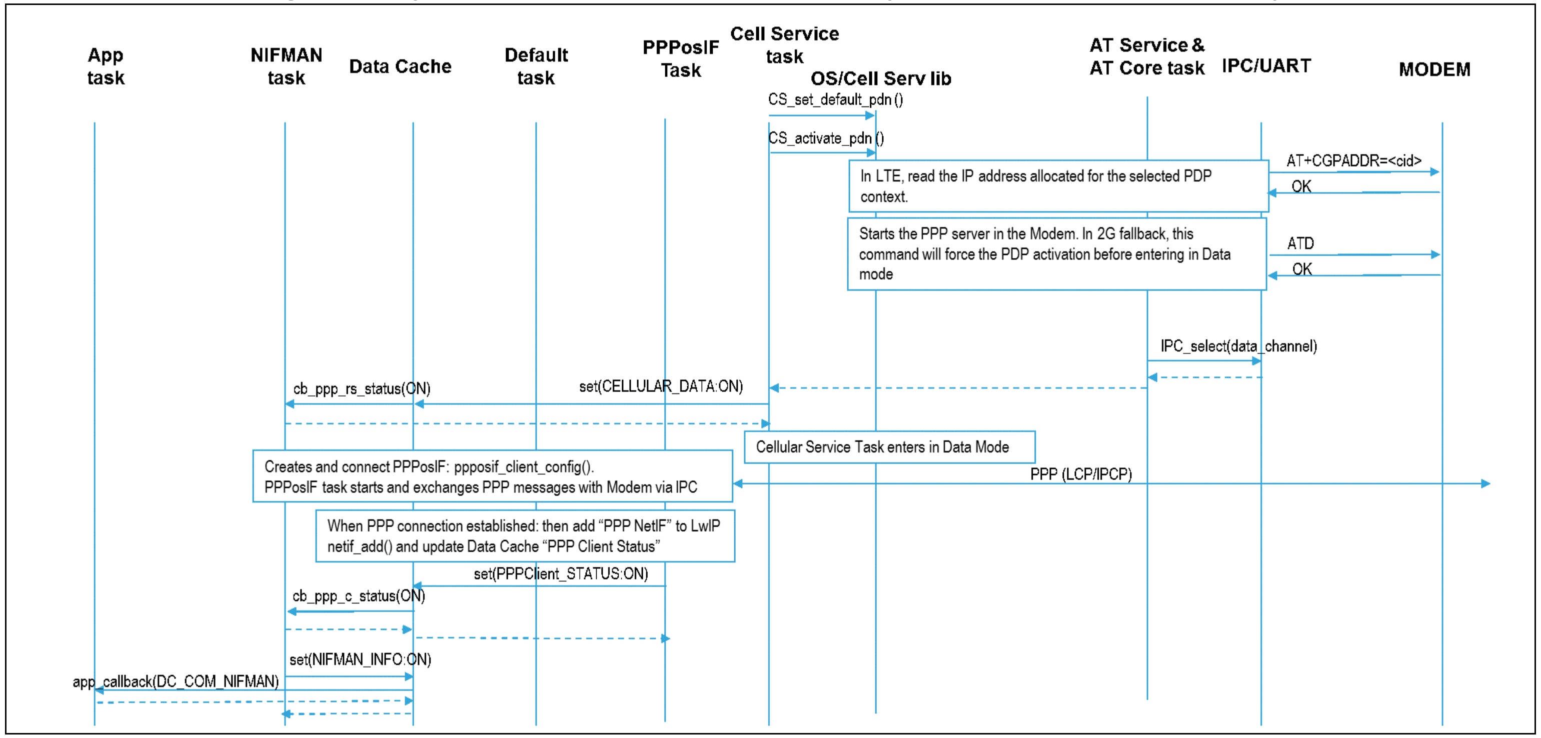

App task
NIFMAN task
Data Cache
Default task
PPPosIF Task
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
CS_set_default_pdn ()
CS_activate_pdn ()
Send command to activate PDN.
If non default PDN needs to be activated then cs_define_pdn() must be called before calling CS_activate_pdn
AT+CGACT=1, CGPADDR
OK
If mode supports "PPP" then sends command to force Modem to enter in Data Mode
AT+CDATA=1
OK
AT enters in Data Mode
IPC_select(data_channel)?
cb_cellular_data(OK)
set(CELLULAR_DATA:OK)
Creates and connect PPPosIF: pppposif_client_config().
PPPosIF task starts and exchanges PPP messages with Modem via IPC
PPP (LCP/IPCP)
When PPP connection established: then add "PPP NetIF" to LwIP netif_add() and update Data Cache "PPP Client Status"
set(PPPClient_STATUS:UP)
cb_ppp_c_status(UP)
set(NIFMAN_INFO:ON)
app_callback(event_id=DC_COM_NIFMAN)
dc_com_read(DC_COM_NIFMAN_INFO)

Figure 12. Dynamic architecture - PDN activation (LwIP socket option with BG96)

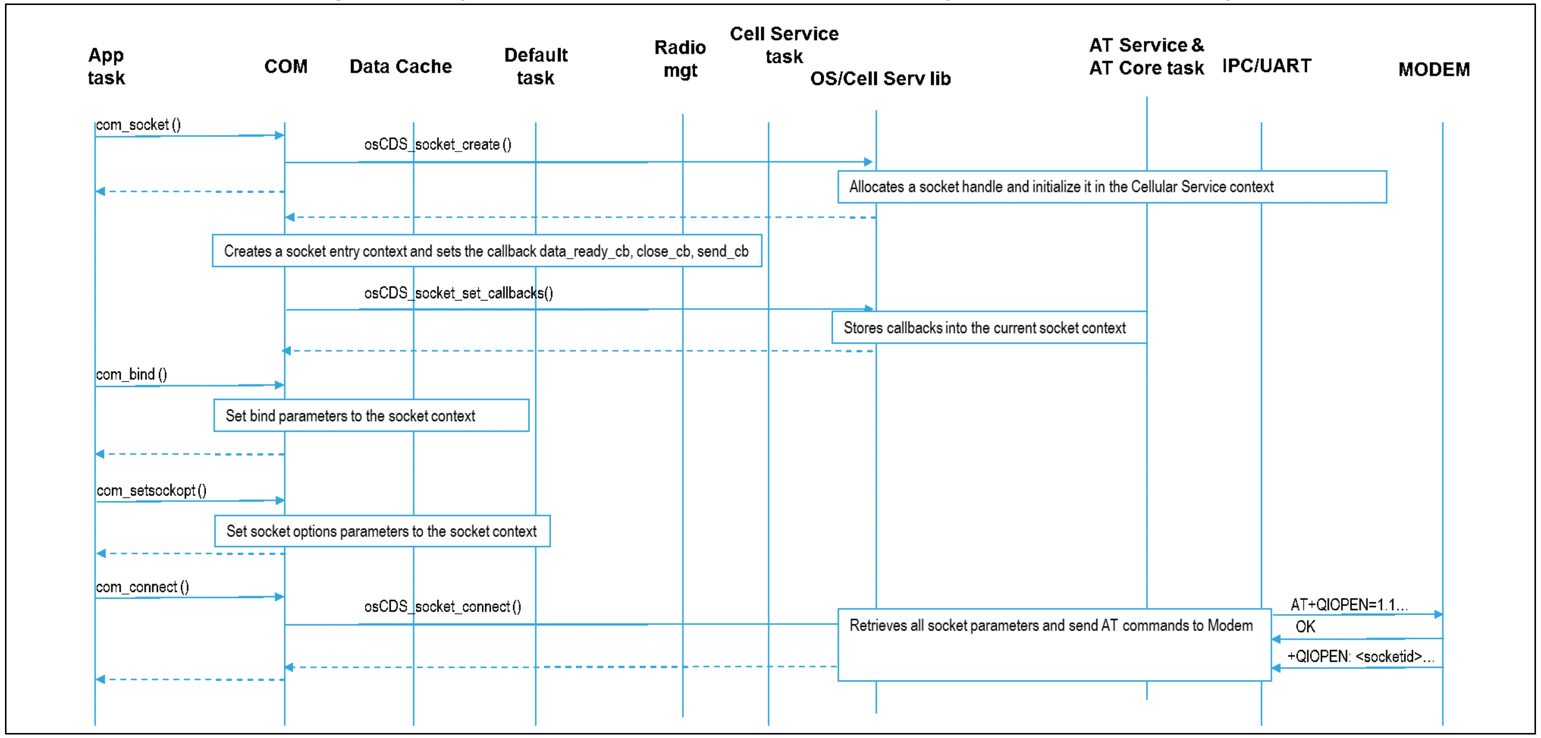

App task
NIFMAN task
Data Cache
Default task
PPPosIF Task
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
CS_set_default_pdn ()
CS_activate_pdn ()
AT+CGPADDR=<cid>
OK
In LTE, read the IP address allocated for the selected PDP context.
ATD
OK
Starts the PPP server in the Modem. In 2G fallback, this command will force the PDP activation before entering in Data mode
IPC_select(data_channel)
cb_ppp_rs_status(ON)
set(CELLULAR_DATA:ON)
Cellular Service Task enters in Data Mode
PPP (LCP/IPCP)
Creates and connect PPPosIF: ppposif_client_config().
PPPosIF task starts and exchanges PPP messages with Modem via IPC
When PPP connection established: then add "PPP NetIF" to LwIP netif_add() and update Data Cache "PPP Client Status"
set(PPPClient_STATUS:ON)
cb_ppp_c_status(ON)
set(NIFMAN_INFO:ON)
app_callback(DC_COM_NIFMAN)

The Cellular Service task calls the `CS_activate_pdn()` function in order to send AT commands to activate PDP context and start the PPP server running on the modem. If PDP context and PPP server are successfully activated, the Cellular Service task informs NIFMAN via the Data Cache.

Once, NIFMAN detects that the data network interface is ready, it instructs the PPPoSIF task to initiate the PPP connection to the PPP server running on the modem side.

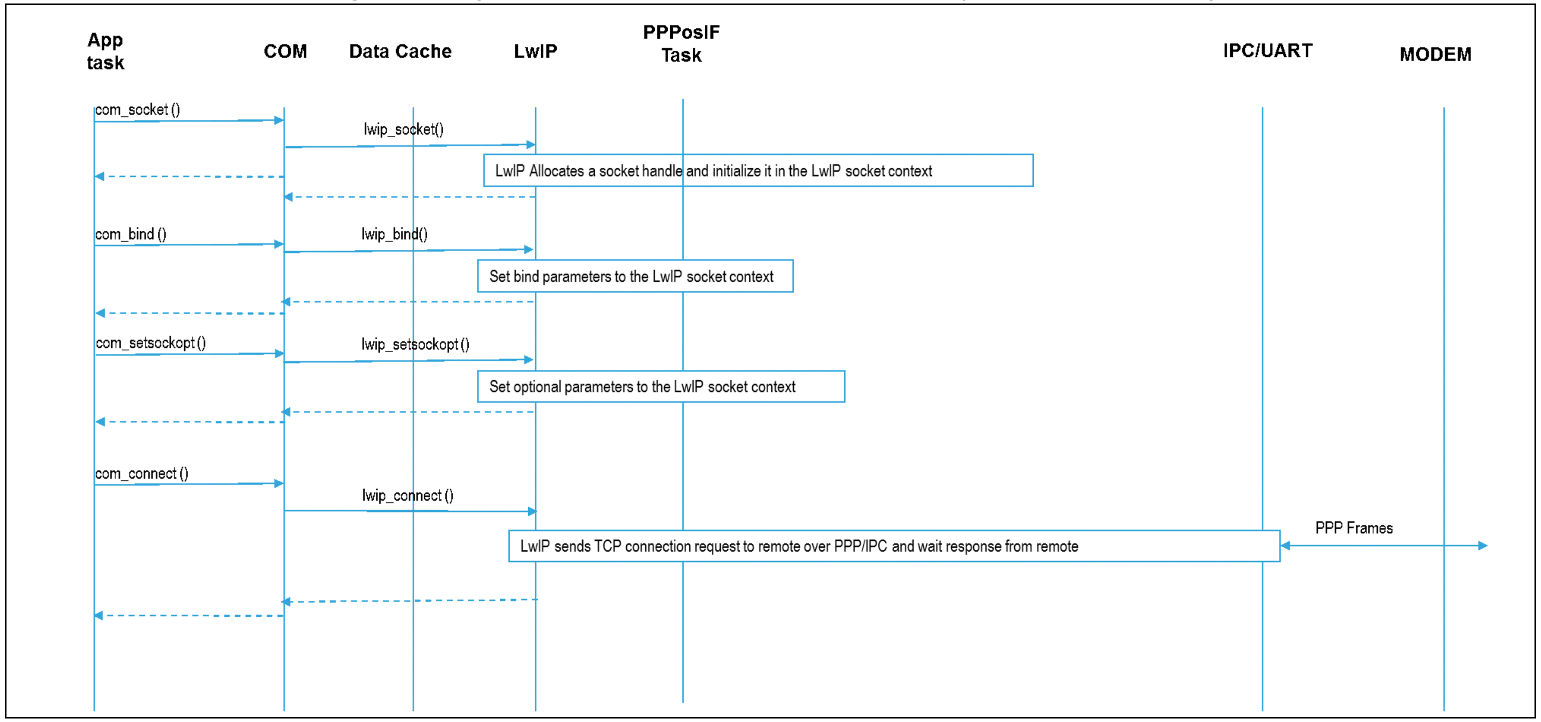
Once the PPP connection is established, NIFMAN is informed by the PPPoSIF task via the Data Cache and subsequently informs the application.

Figure 13. Dynamic architecture - Socket creation (modem socket option)


App task
COM
Data Cache
Default task
Radio mgt
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
com_socket ()
osCDS_socket_create ()
Allocates a socket handle and initialize it in the Cellular Service context
Creates a socket entry context and sets the callback data_ready_cb, close_cb, send_cb
osCDS_socket_set_callbacks()
Stores callbacks into the current socket context
com_bind ()
Set bind parameters to the socket context
com_setsockopt ()
Set socket options parameters to the socket context
com_connect ()
osCDS_socket_connect ()
Retrieves all socket parameters and send AT commands to Modem
AT+QIOPEN=1,1...
OK
+QIOPEN: <socketid>...

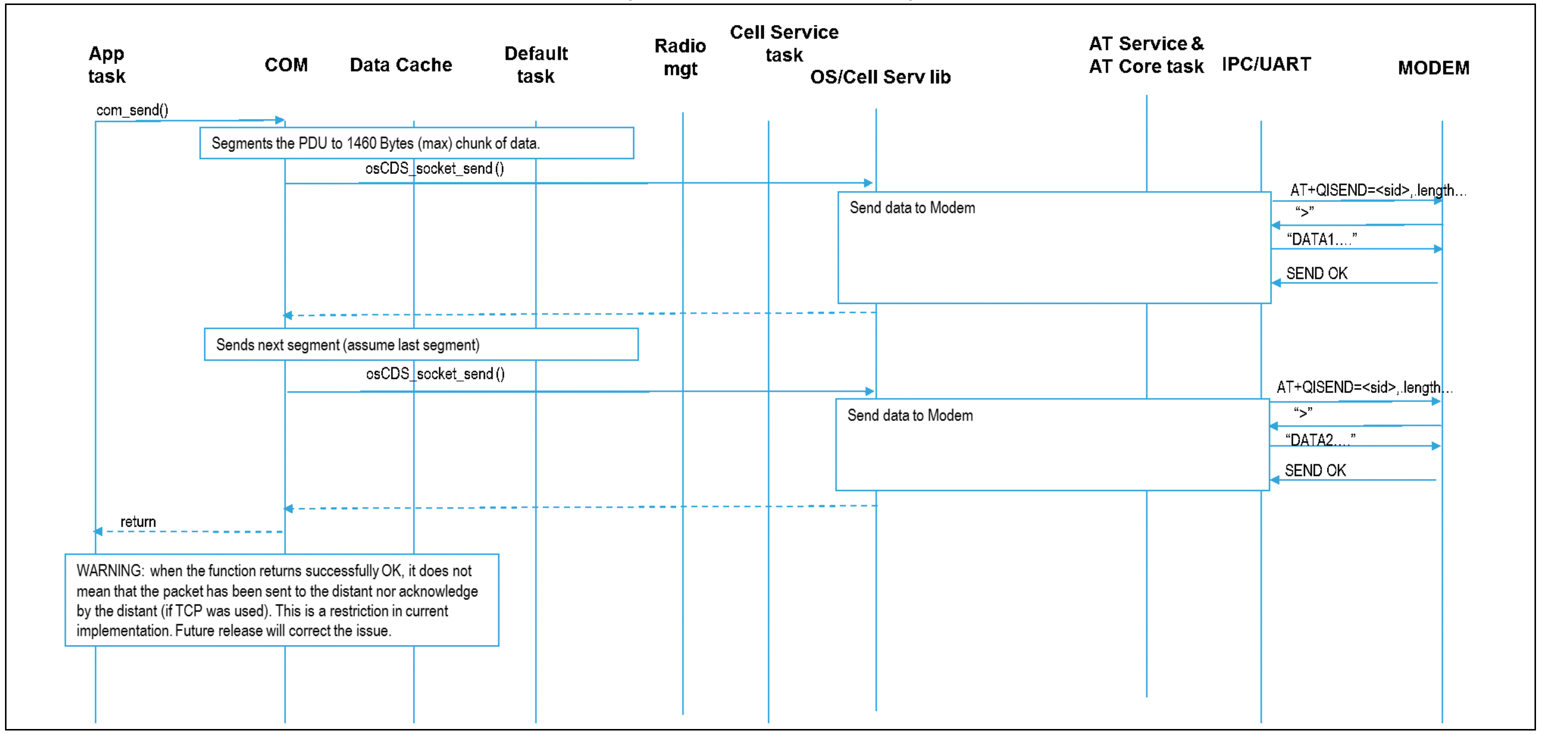
When the network interface is ready, the application can create a socket in order to exchange IP packets with the remote application. The application calls the `com_socket()` COM function, which in turn calls the Cellular Service library in order to send AT commands to the modem.

Figure 14. Dynamic architecture - Socket creation (LwIP socket option)


App task
COM
Data Cache
LwIP
PPPosIF Task
IPC/UART
MODEM
com_socket ()
lwip_socket()
LwIP Allocates a socket handle and initialize it in the LwIP socket context
com_bind ()
lwip_bind()
Set bind parameters to the LwIP socket context
com_setsockopt ()
lwip_setsockopt ()
Set optional parameters to the LwIP socket context
com_connect ()
lwip_connect ()
LwIP sends TCP connection request to remote over PPP/IPC and wait response from remote
PPP Frames

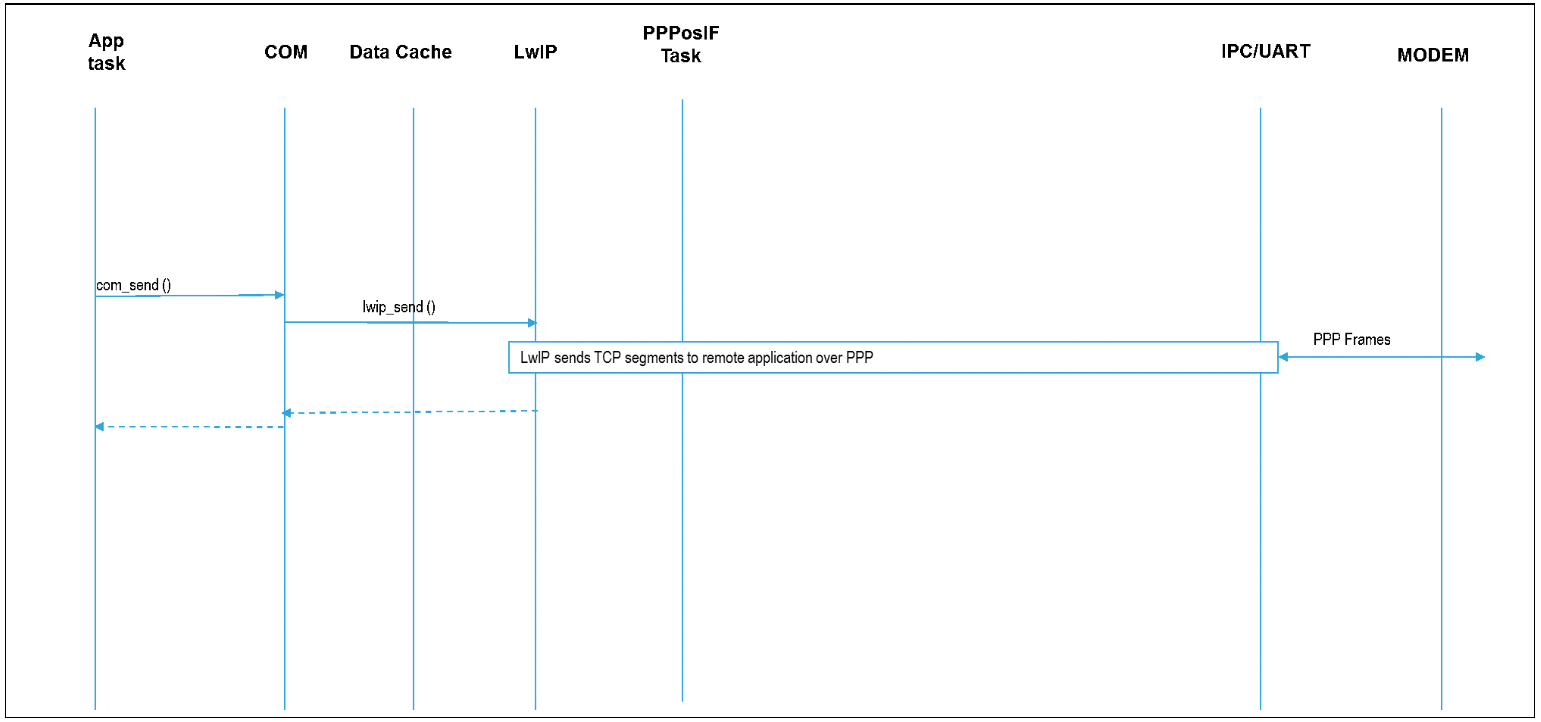
The COM interface directly maps each socket function to LwIP. This latter encapsulates the IP packets into PPP frames, which are sent to the Modem over IPC layer. PPP frames from the modem are received by the PPPoSIF task, which forwards them to the LwIP stack.

Figure 15. Dynamic architecture - Data transfer - Send data to remote application (modem socket option)


App task
COM
Data Cache
Default task
Radio mgt
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
com_send()
Segments the PDU to 1460 Bytes (max) chunk of data.
osCDS_socket_send ()
Send data to Modem
AT+QISEND=<sid>,.length…
">"
"DATA1…."
SEND OK
Sends next segment (assume last segment)
osCDS_socket_send ()
Send data to Modem
AT+QISEND=<sid>,.length…
">"
"DATA2…."
SEND OK
return
WARNING: when the function returns successfully OK, it does not mean that the packet has been sent to the distant nor acknowledge by the distant (if TCP was used). This is a restriction in current implementation. Future release will correct the issue.

The COM interface segments the PDU from the application into smaller chunks of data that it sends to the modem by means of the Cellular Service library function.

Figure 16. Dynamic architecture - Data transfer - Send data to remote application (LwIP socket option)



The `com_send()` function calls the `lwip_send()` function, which is implemented by the LwIP stack to exchange TCP segments remotely over the PPP layer.

Figure 17. Dynamic architecture - Data transfer - Receive data from remote application (modem socket option)

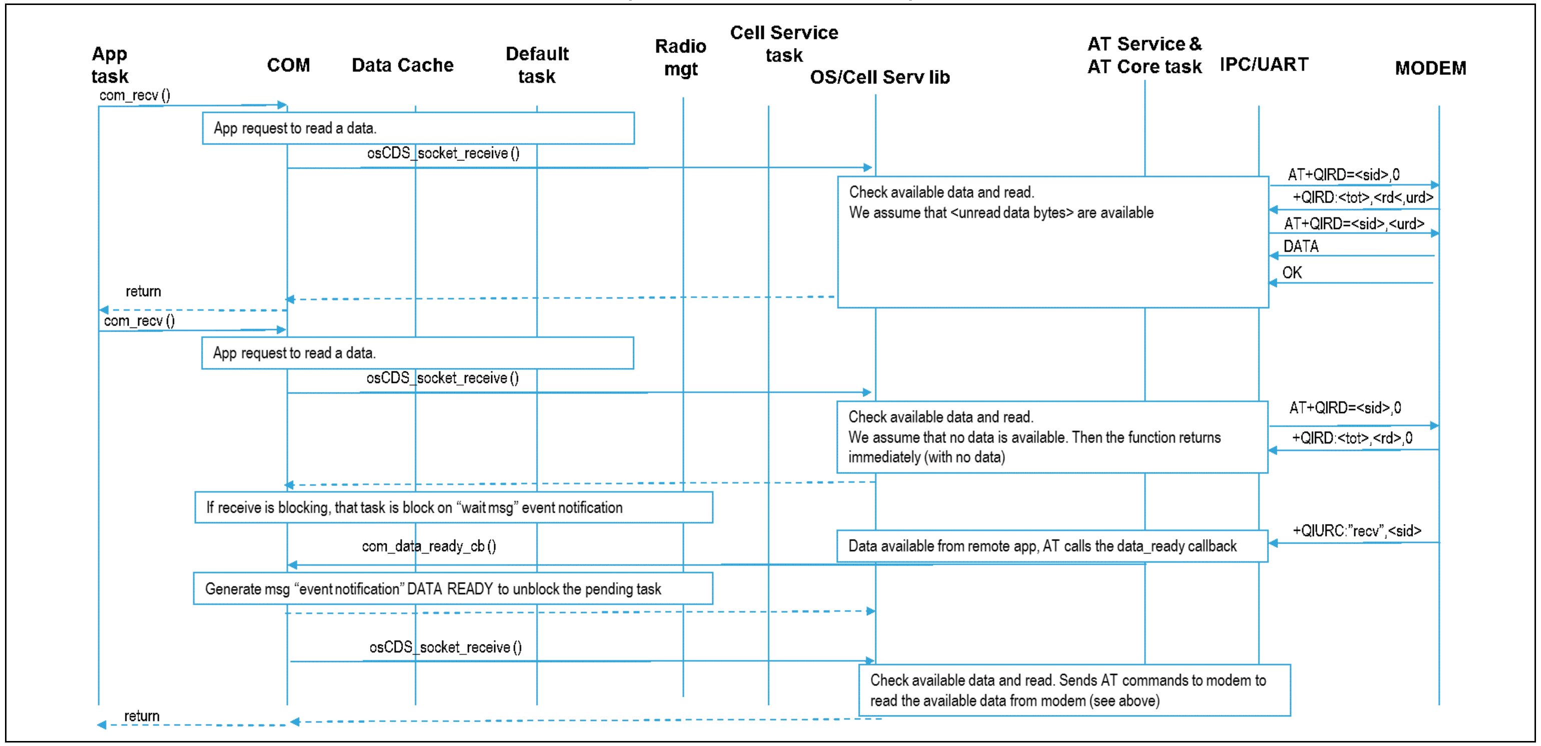
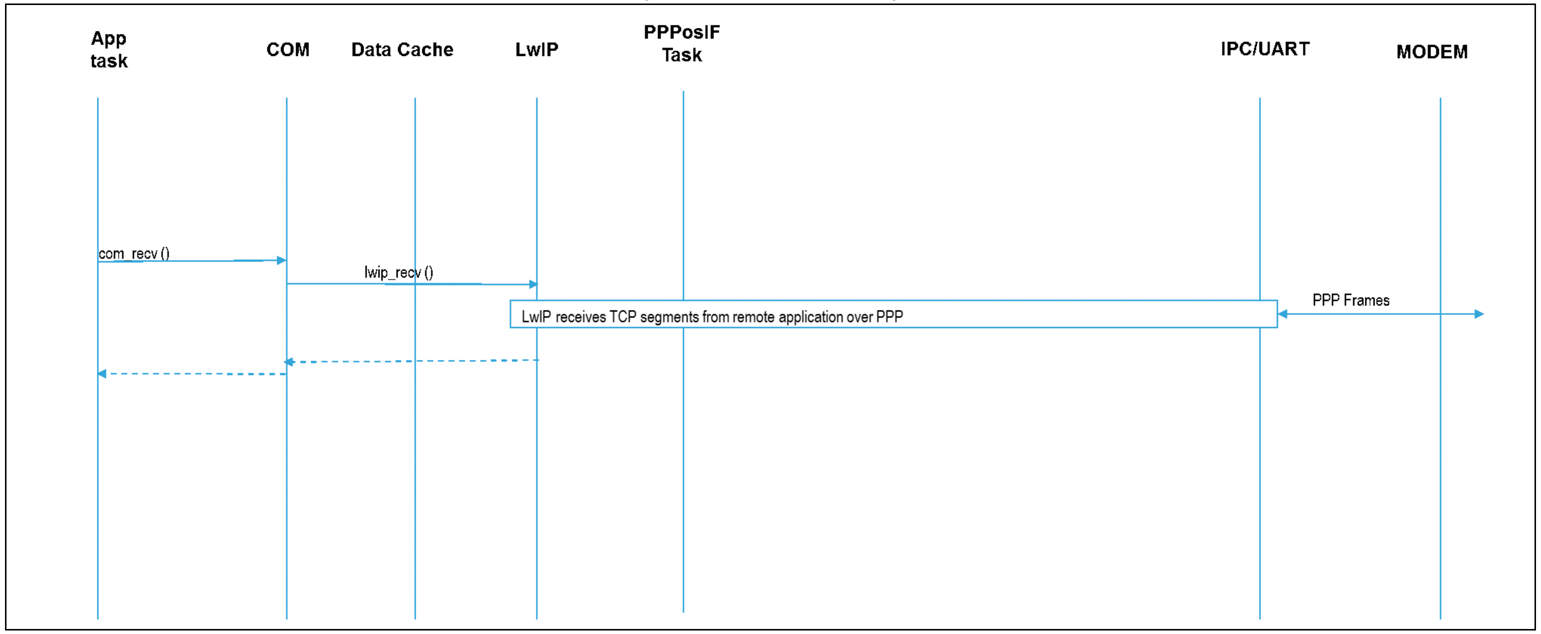

App task
COM
Data Cache
Default task
Radio mgt
Cell Service task
OS/Cell Serv lib
AT Service & AT Core task
IPC/UART
MODEM
com_recv ()
App request to read a data.
osCDS_socket_receive ()
Check available data and read.
We assume that <unread data bytes> are available
AT+QIRD=<sid>,0
+QIRD:<tot>,<rd<,urd>
AT+QIRD=<sid>,<urd>
DATA
OK
return
com_recv ()
App request to read a data.
osCDS_socket_receive ()
Check available data and read.
We assume that no data is available. Then the function returns immediately (with no data)
AT+QIRD=<sid>,0
+QIRD:<tot>,<rd>,0
If receive is blocking, that task is block on "wait msg" event notification
+QIURC:"recv",<sid>
com_data_ready_cb ()
Data available from remote app, AT calls the data_ready callback
Generate msg "event notification" DATA READY to unblock the pending task
osCDS_socket_receive ()
Check available data and read. Sends AT commands to modem to read the available data from modem (see above)
return

Figure 18. Dynamic architecture - Data transfer - Receive data from remote application (LwIP socket option)



4.4 X-CUBE-CELLULAR Expansion Package description

This section describes the software components of the X-CUBE-CELLULAR package.

X-CUBE-CELLULAR is an Expansion Package for STM32Cube™. Its main features are:

- Fully compliant with STM32Cube™ architecture
- Expands STM32Cube™ in order to enable the development of applications accessing and using various cloud platforms
- Based on the STM32CubeHAL, which is the hardware abstraction layer for STM32 microcontrollers

The software components used by the application are:

- **STM32Cube HAL**
  The HAL driver layer provides a generic multi-instance simple set of APIs (application programming interfaces) to interact with the upper layers (application, libraries and stacks).

  It is composed of generic and extension APIs. It is directly built around a generic architecture and allows the layers that are built upon, such as the middleware layer, implementing their functionalities without dependencies on the specific hardware configuration for a given microcontroller unit (MCU).

  This structure improves the library code reusability and guarantees an easy portability onto other devices.
- **Board support package (BSP)**
  The software package needs to support the peripherals on the STM32 boards apart from the MCU. This software is included in the board support package (BSP). This is a limited set of APIs which provides a programming interface for certain board specific peripherals such as the LED and the User button.
- **Application**
  HTTP and PING clients.
- **Middleware**
  Optionally LwIP.
- **FreeRTOS™**
  FreeRTOS™ is mandatory to run the tasks for X-CUBE-CELLULAR components.
- **Configuration files**
  component and platform configuration file are provided at project repository
  - *plf_features.h* defines the feature list to include in firmware.
  - *plf_hw_config.h* defines the mapping of the GPIO and HW interface specific to logical names to ease SW porting to another board. It also provides the HW bus interface configuration (such as the UART) used for communicating with the modem.
  - *plf_sw_config.h* provides platform SW configuration such as task priorities, trace, stack size monitoring and others. It also provides application behaviors, which can differ from one platform to another, such as modem polling timer value, button configuration and polling and others.
  - *plf_stack_size.h* defines the thread stack size.

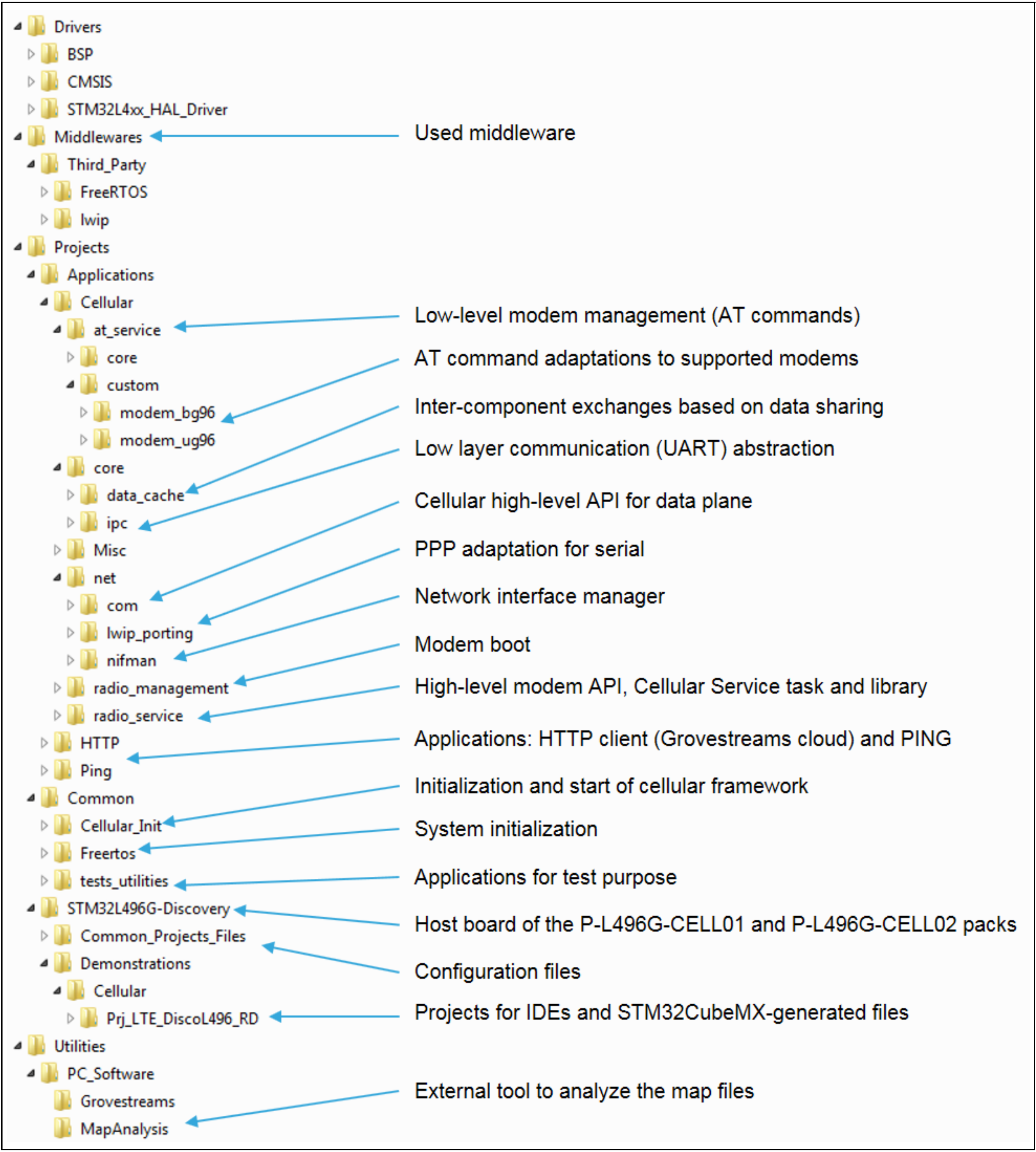
Other parameters can be customized. Additional details are provided in *Chapter 8: How to customize the software? on page 62*.

Some parameters can also be defined dynamically at runtime. They are stored in the Flash memory and re-used at the next platform boot if needed.

4.5 Folder structure

Figure 19 presents the folder structure of the X-CUBE-CELLULAR package.

Figure 19. Project file structure


Drivers
BSP
CMSIS
STM32L4xx_HAL_Driver
Middlewares
Third_Party
FreeRTOS
lwip
Projects
Applications
Cellular
at_service
core
custom
modem_bg96
modem_ug96
core
data_cache
ipc
Misc
net
com
lwip_porting
nifman
radio_management
radio_service
HTTP
Ping
Common
Cellular_Init
Freertos
tests_utilities
STM32L496G-Discovery
Common_Projects_Files
Demonstrations
Cellular
Prj_LTE_DiscoL496_RD
Utilities
PC_Software
Grovestreams
MapAnalysis
Used middleware
Low-level modem management (AT commands)
AT command adaptations to supported modems
Inter-component exchanges based on data sharing
Low layer communication (UART) abstraction
Cellular high-level API for data plane
PPP adaptation for serial
Network interface manager
Modem boot
High-level modem API, Cellular Service task and library
Applications: HTTP client (Grovestreams cloud) and PING
Initialization and start of cellular framework
System initialization
Applications for test purpose
Host board of the P-L496G-CELL01 and P-L496G-CELL02 packs
Configuration files
Projects for IDEs and STM32CubeMX-generated files
External tool to analyze the map files

Note: *The additional B-L475E-IOI01 directory with the same structure as the STM32L496G-Discovery directory is created in the Projects directory for the “Discovery IoT node cellular” set.*

4.6 Reset push-button

The reset push-button (black) is used to reset the board at any time. This action forces the reboot of the platform.

4.7 Application LED

The application LED is turned ON or OFF by the HTTP client application. Refer to *Figure 23: Hardware setup (P-L496G-CELL02 example)* in *Chapter 6: Hardware and software environment setup on page 39*.

4.8 Real-time clock

System date and time are managed by the RTC. At boot time, it is updated by the HTTP client application, which gets time and date from the Grovestreams server.

The system date can also be manually updated at boot setup.

The `HAL_RTC_GetTime()` function provides the time to the application.

Note: The hour value depends on the Grovestreams server time zone.

5 Cellular connectivity examples

This chapter describes the cellular connectivity available examples. Several examples that can run in parallel are provided, such as PING and HTTP client.

5.1 Real network or simulator

The worldwide coverage of 2G and 3G networks makes it possible to systematically run the examples for these technologies on real networks. The LTE cat M1 and NB1 technologies do not yet offer a similar global coverage. If such networks are not available, the user must use a Cat M1 / NB1 compliant network simulator.

In this document, assumption is made that a real network is available whatever the network technology used.

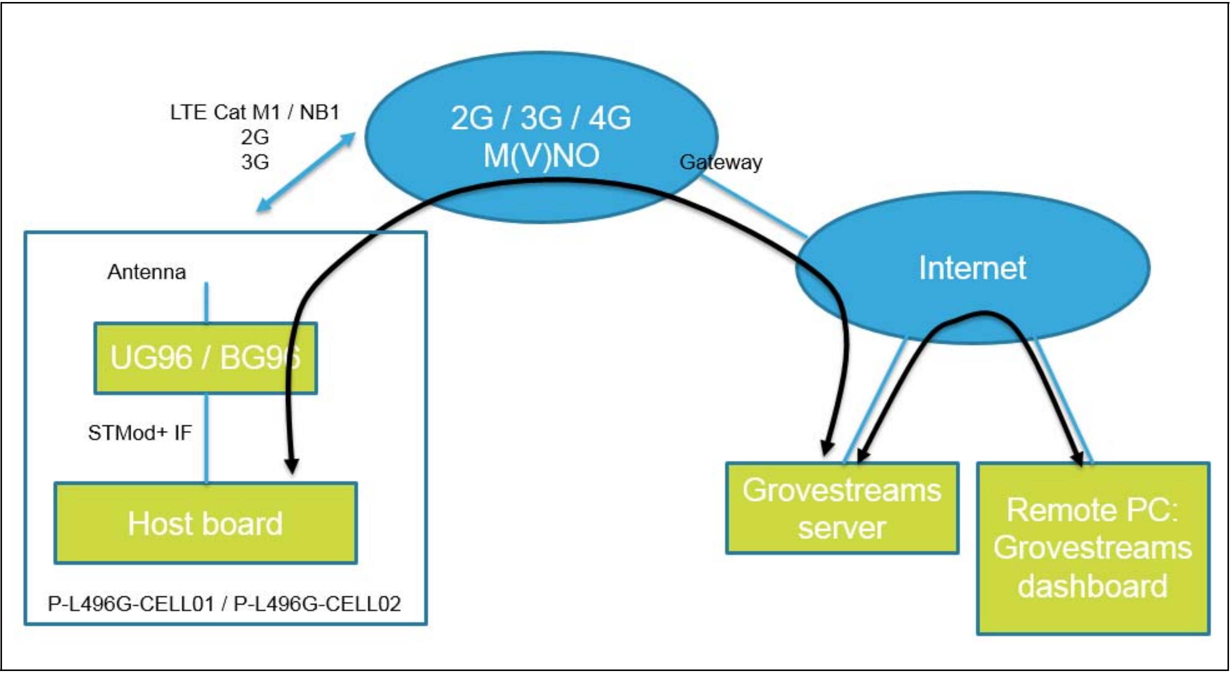
The add-on modem boards (in P-L496G-CELL01 and P-L496G-CELL02) embed an eSIM, provisioned by the EMnify MVNO profile:

- For the UG96 modem, 2G or 3G real networks are used
- For the BG96 modem, with the EMnify eSIM, only 2G fallback is possible. To use the BG96 modem in Cat M1 or NB1 mode, the user must insert a plastic SIM card compliant with this network technology from an MNO that has already deployed it.

5.2 Connection overview

The Grovestreams connection overview presented in *Figure 20*.

Figure 20. Grovestreams connection overview



5.3 PING example

The console command allows the generation of ping requests (refer to *Section 7.3: Console command on page 54*). The simple `ping` command generates 10 ping requests using default IP address 8.8.8.8. To specify another IP address, use the `ping <ddd.ddd.ddd.ddd>` command.

Ping command example:

```
ping 8.8.4.4
```

Ping result example:

```
<<< HTTP CLIENT ACTIVE 8.8.4.4>>>

>Ping: 32 bytes from 8.8.4.4: seq=01 time= 73ms
Ping: 32 bytes from 8.8.4.4: seq=02 time= 77ms
Ping: 32 bytes from 8.8.4.4: seq=03 time= 48ms
Ping: 32 bytes from 8.8.4.4: seq=04 time= 59ms
Ping: 32 bytes from 8.8.4.4: seq=05 time= 78ms
Ping: 32 bytes from 8.8.4.4: seq=06 time= 74ms
Ping: 32 bytes from 8.8.4.4: seq=07 time= 56ms
Ping: 32 bytes from 8.8.4.4: seq=08 time= 70ms
Ping: 32 bytes from 8.8.4.4: seq=09 time= 66ms
Ping: 32 bytes from 8.8.4.4: seq=10 time= 66ms
--- 8.8.4.4 Ping Statistics ---
Ping: min/avg/max = 48/66/78 ms ok = 10/10
<<< Ping Completed >>>
```

5.4 Grovestreams (HTTP) access example

The cellular connectivity demonstration for Grovestreams consists in two use cases:

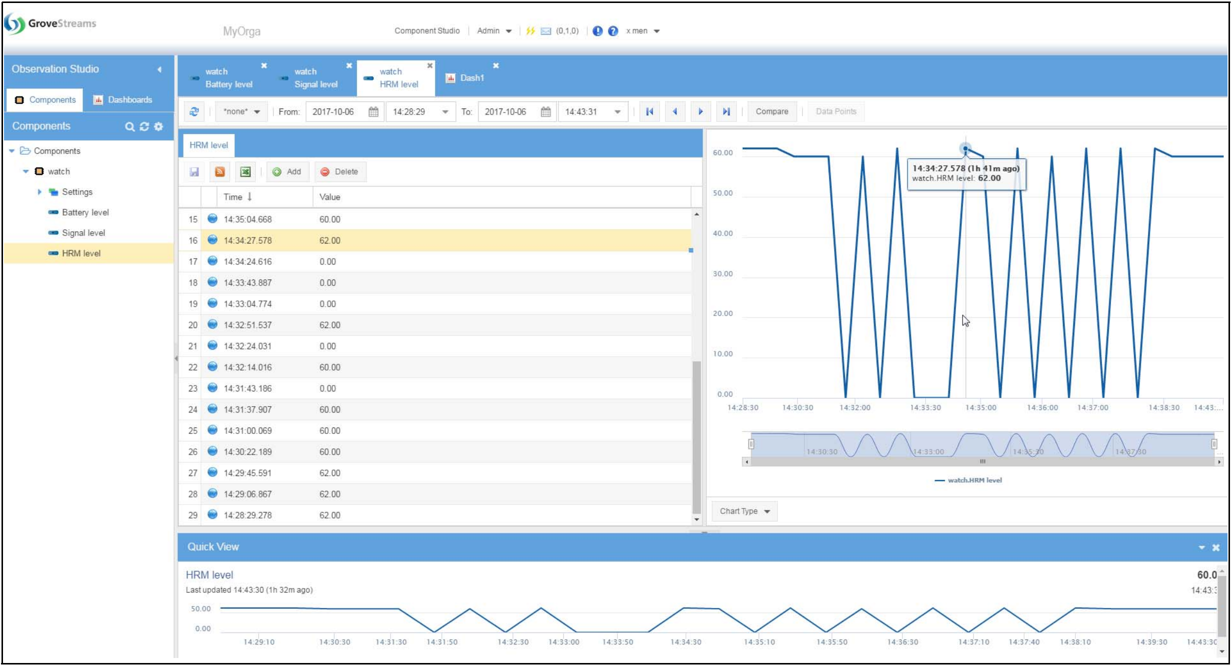
- The device periodically reports sensor data to the Grovestreams cloud IoT platform. The end-user connects to the Grovestreams web server dashboard.
- The end-user controls the LED application state from the dashboard.

The HTTP client on the MCU sets up a connection to the Grovestreams server through a dedicated account. It pushes data (temperature, humidity, pressure and cellular radio signal strength) to the Grovestreams server through the HTTP PUT command. An end-user connected to the Grovestreams platform with a standard web browser has access to the Grovestreams dashboard web page. The device polls the Grovestreams web server every two seconds by sending HTTP GET commands to retrieve any request from the end-user. By this mechanism, the end-user can switch the application LED on the host board ON and OFF.

For temperature, humidity, and pressure, if the X-NUCLEO-IKS01A2 board with sensors is plugged, real values are sent to the cloud server. Otherwise, simulated data are reported. With the “Discovery IoT node cellular” set, real values are always sent.

Figure 21 and *Figure 22* present the related Grovestreams interfaces.

Figure 21. Grovestreams web interface, component view


From: 2017-10-06 14:28:29 To: 2017-10-06 14:43:31

HRM level

	Time	Value
15	14:35:04.668	60.00
16	14:34:27.578	62.00
17	14:34:24.616	0.00
18	14:33:43.887	0.00
19	14:33:04.774	0.00
20	14:32:51.537	62.00
21	14:32:24.031	0.00
22	14:32:14.016	60.00
23	14:31:43.186	0.00
24	14:31:37.907	60.00
25	14:31:00.069	60.00
26	14:30:22.189	60.00
27	14:29:45.591	62.00
28	14:29:06.867	62.00
29	14:28:29.278	62.00

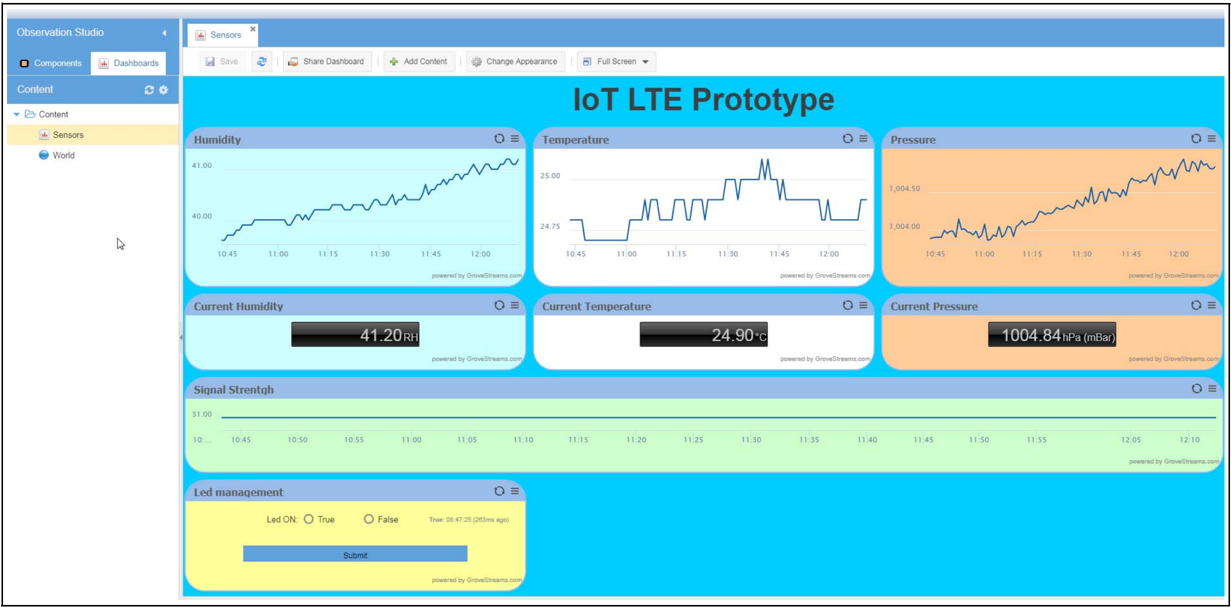
14:34:27.578 (1h 41m ago) watch.HRM level: 62.00

Quick View

HRM level

Last updated 14:43:30 (1h 32m ago)

Figure 22. Grovestreams web interface, dashboard view


IoT LTE Prototype

Current Humidity: 41.20 RH

Current Temperature: 24.90 °C

Current Pressure: 1004.84 hPa (mBar)

Led management: Led ON: True False

6 Hardware and software environment setup

The STM32 MCU FW must be programmed, whatever the test at stake. Modem FW can be upgraded too if needed.

Note: Before programming with a new firmware, connect the system to a PC with Teraterm and note the voucher number; this number is needed to activate the eSIM on the modem board. If the board is flashed before getting the voucher, re-flash the original image and get the voucher.
The embedded SIM in the "Discovery IoT node cellular" set is not provisoned with a M(V)NO profile: No eSIM activation is needed; A plastic SIM card must be used instead.

For the P-L496G-CELL01 and P-L496G-CELL02 Discovery packs, connect the host board to the modem board by means of the STMod+ connector (refer to *Figure 24*). For the "Discovery IoT node cellular" set, first connect the MB1329 modem board to the X-NUCLEO-STMODA1 adapter board, then connect the X-NUCLEO-STMODA1 to the B-L475E-IOT01A host board (refer to *Figure 25*).

When the UICC chip soldered on the add-on board is not used, insert a UICC compliant with the real network used into the UICC socket. The UICC socket is located at the back of the add-on board, behind the USB connector.

If the soldered UICC chip is used, it must be activated beforehand and selected at boot by means of the boot menu.

Power on the board by plugging its USB connector to a PC, USB power supply, or USB power bank. If traces must be displayed, the USB connector must be connected to a PC with an open console application.

Warning: Use the HW only with the antenna connected. With no antenna connected, there is a risk of damage to the modem because of the power reflected from the antenna connector to the modem RF output.

Figure 23 depicts the hardware setup.

Figure 23. Hardware setup (P-L496G-CELL02 example)

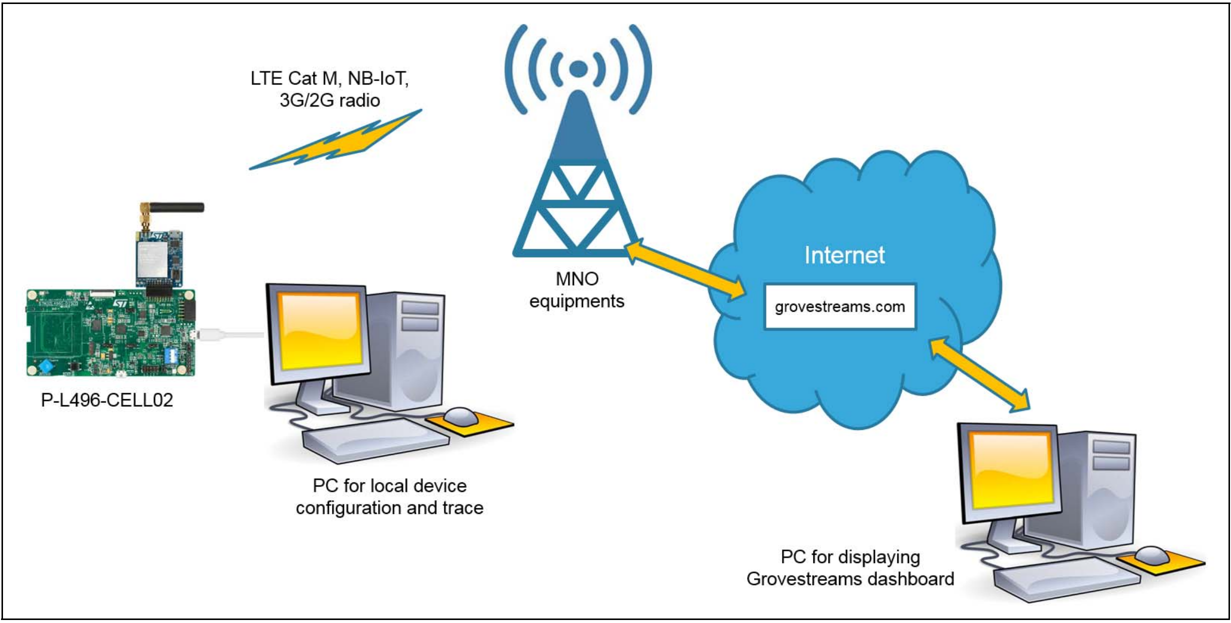


Figure 24 shows the P-L496G-CELL02 with the USB cable in place for power supply and optional trace / boot menu.

Figure 24. Hardware view (P-L496G-CELL02 example)

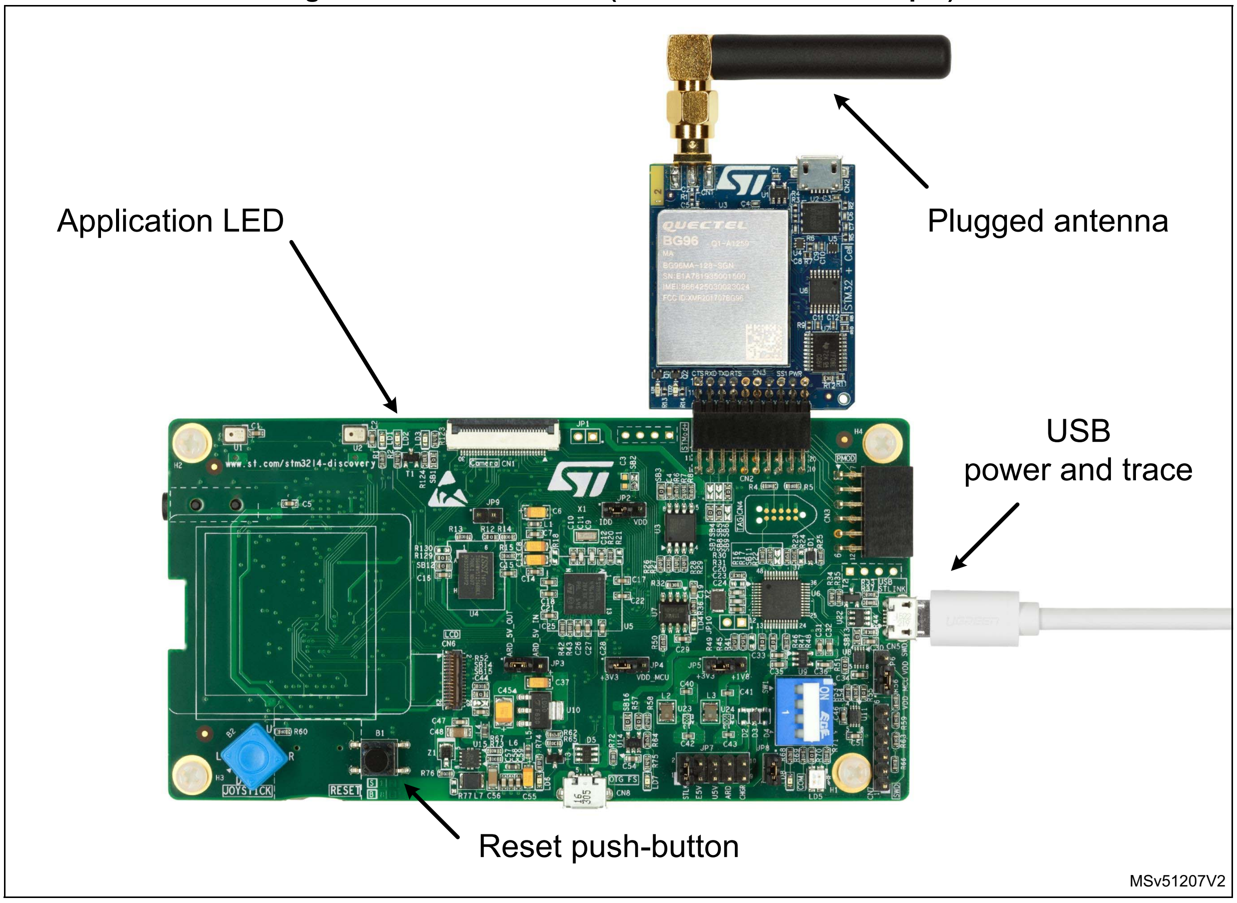
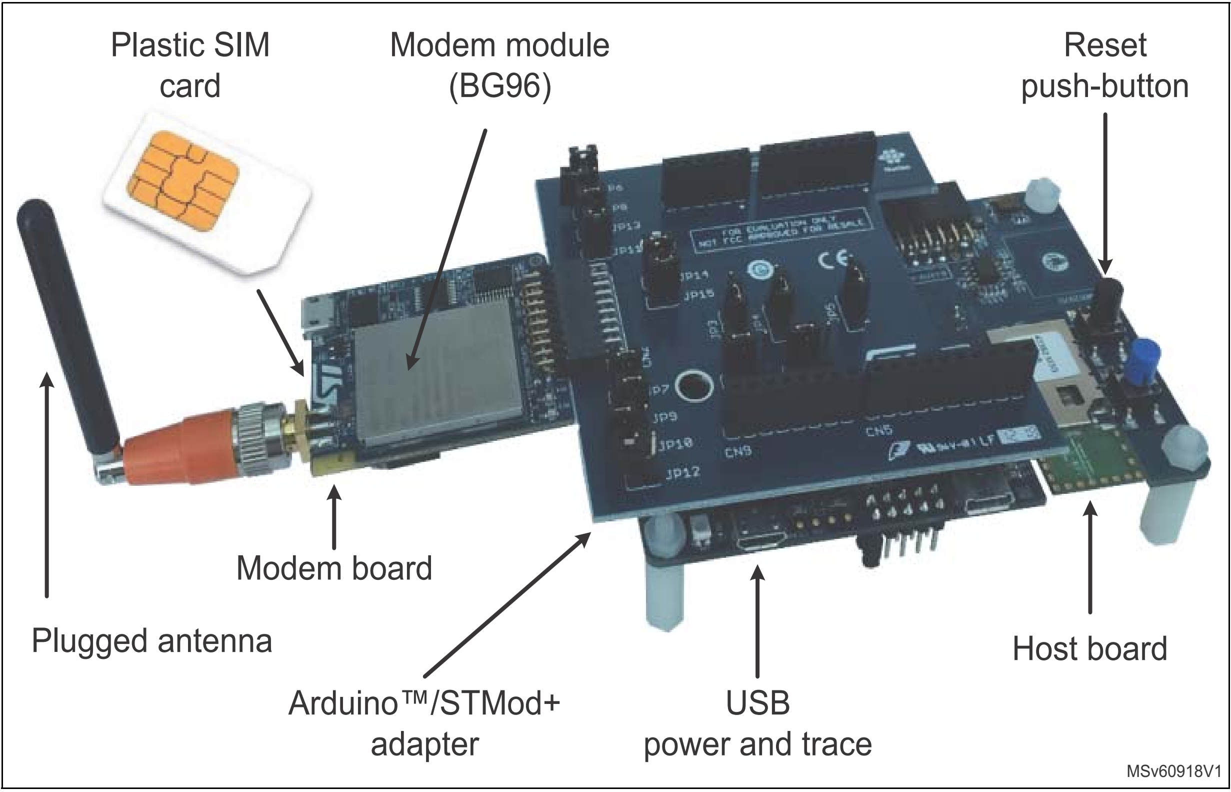


Figure 25 shows the “Discovery IoT node cellular” set with the B-L475E-IOT01A host, X-NUCLEO-STMODA1 adapter and MB1329 modem boards connected together.

Figure 25. Hardware view (“Discovery IoT node cellular” set example)



Summary of environment setup steps:

- Create the Grovestreams account (refer to *A.1: How to configure a Grovestreams account? on page 77*)
- Create a Grovestreams organization in the Grovestreams account (refer to *A.1*)
- Get the 2 needed API keys in the organization (refer to *A.1*)
- Update the provided *.txt* file with the user API keys previously read
- Connect the modem add-on board with its antenna to the host STM32 board (refer to *Figure 24*)
- Connect the STM32 board to a PC via a USB cable (refer to *Figure 24*)
- Program the STM32 board by dragging and dropping the STM32 FW image onto the newly appeared drive
- Start the terminal and set the parameters (refer to *Chapter 7: Interacting with the host board*)
- Reboot the board (press the black button)
- Select item "2" in the boot menu (refer to *Chapter 7: Interacting with the host board* for a complete description), select the SIM card to be used and the APN if needed, set the Grovestreams parameters, and save in Flash. The APN name is provided by the M(V)NO.

- Reboot the board (the trace is displayed in the terminal)
- Connect to the Grovestreams account, select the dashboard, double click on *sensor*, observe the data displayed and the toggling of the LED induced by the Grovestreams dashboard.

There are 2 important txt files for Grovestreams:

- *GS_ Blueprint.txt*: used during organization creation (use as such, no modification is needed)
- *GS_setup.txt*: used to configure the generic FW, to access the Grovestreams account. This file must be updated with the 2 API keys replaced by the API keys of the organization (refer to *A.1: How to configure a Grovestreams account? on page 77*)

Note: File "GS_setup.txt" is not mandatory. It is only an user-friendly method to enter personal Grovestreams parameters into the FW, avoiding to enter each parameter individually by using "File" > "Send file …" and selecting "GS_setup.txt".

Refer to Chapter 7: Interacting with the host board for entering Grovestreams parameters with file "GS_setup.txt" by using the menu at boot (Setup Menu > Grovestreams).

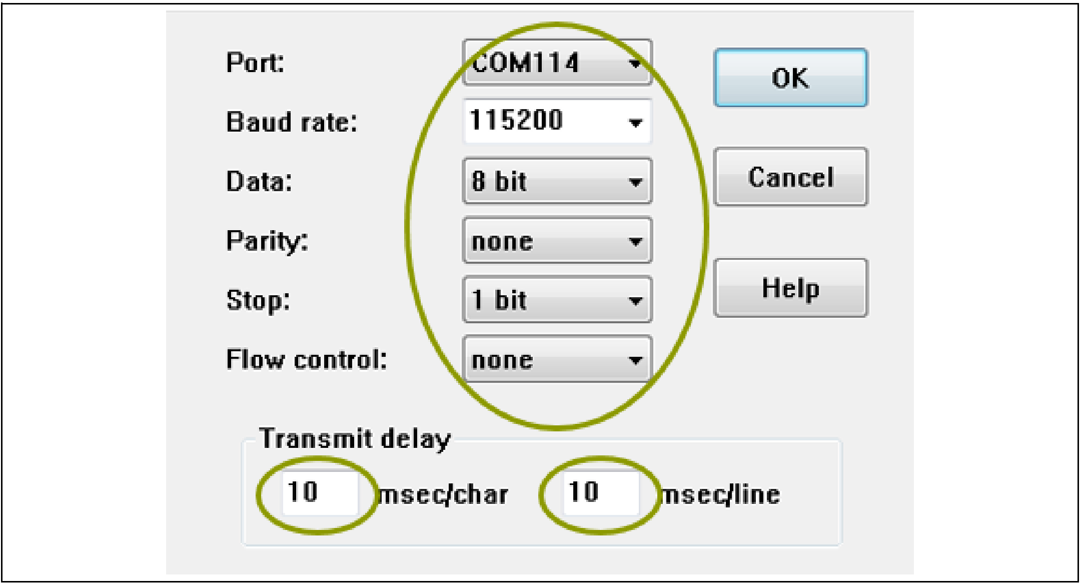
Refer to Chapter 7: Interacting with the host board for selecting the UICC or SIM socket, and the APN, by using the menu at boot with (Setup Menu > Cellular Service).

7 Interacting with the host board

To interact with the Host board a serial console is used (Virtual COM port over USB). With the Windows® operating system, the use of the Teraterm software is recommended.

Serial port settings for communicating with the host board are illustrated in *Figure 26*. The menu is reached through *Setup* > *Serial port*. Set *Baud rate* to 115200 to get Teraterm up and running. For the boot setup configuration, a *Transmit delay* of 10 ms must be applied.

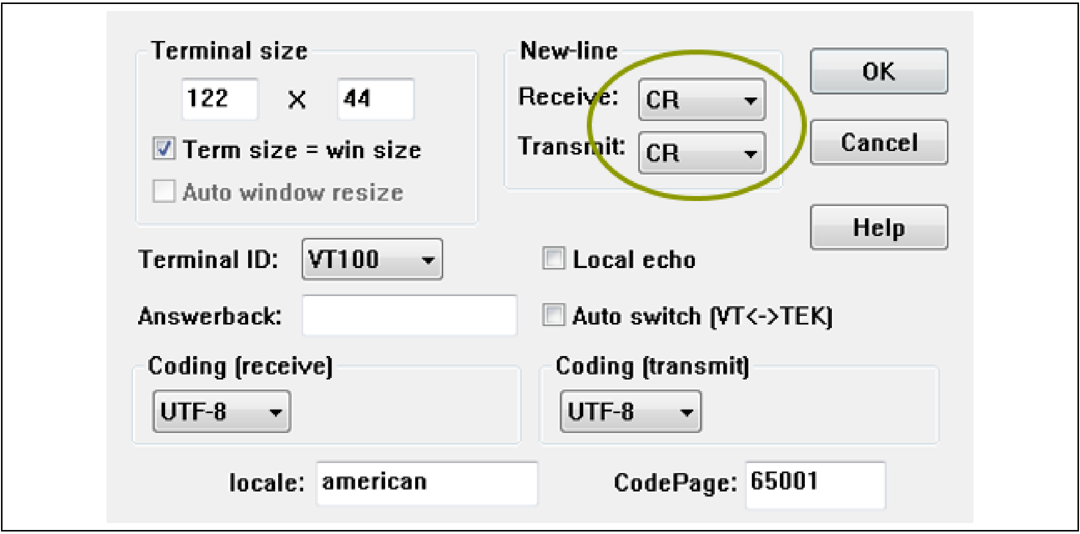
Figure 26. Serial port settings to interact with the host board



Port: COM114
Baud rate: 115200
Data: 8 bit
Parity: none
Stop: 1 bit
Flow control: none
Transmit delay
10 msec/char
10 msec/line

Figure 27 illustrates the menu for setting the terminal parameters. It is reached through *Setup* > *Terminal*. Set both *Receive* and *Transmit New-line* parameters to CR.

Figure 27. Serial port settings to interact with the host board (new-line)



Terminal size
122 X 44
☑ Term size = win size
☐ Auto window resize
New-line
Receive: CR
Transmit: CR
Terminal ID: VT100
Answerback:
☐ Local echo
☐ Auto switch (VT<->TEK)
Coding (receive) UTF-8
Coding (transmit) UTF-8
locale: american
CodePage: 65001

Once all terminal and serial port parameters are properly set, press the board reset button (black).

The HTTP service is active as soon as STM32 SW is running, sending data to the Grovestreams cloud. The Grovestreams dashboard displays the data values and switches the device LED ON / OFF.

Parameter setting is possible at boot time through the console. Refer to *Section 7.2: Boot menu* for details.

7.1 Debug

The debug trace is viewed through the Virtual COM port of the PC that is connected to the board and powers it.

Debug levels are customized in file *plf_sw_config.h*. Details are provided in *Section 8.2.3: Different kinds of available traces*, *Section 8.2.4: How to configure traces?* and *Section 8.5: Thread stack consumption monitoring*.

7.2 Boot menu

The boot menu is enabled by setting the `USE_DEFAULT_SETUP` variable to 0, which is the default value, in configuration file *Projects\STM32L496G-Discovery\Common_Projects_Files\inc\plf_sw_config.h*.

At boot stage, a menu is displayed via the serial console to select the firmware use:

0: *Start*	firmware starts immediately without waiting for 3 seconds
1: *Setup Menu*	configuration setup
2: *Modem power on*	modem boot only without starting firmware applications

If no character is entered after 3 seconds, the firmware starts normally.

Table 3 illustrates the boot menu.

Table 2. Boot menu

```
=============================
      STM32 CELLULAR
      Version: xxx
=============================
Select the application to run:

0: Start
1: Setup Menu
2: Modem power on
```

7.2.1 *“Start”* option

This option starts the Cellular Service normally without waiting for the countdown timer.

7.2.2 *“Modem power on”* option

Modem is powered on without starting Cellular Service nor an application. This is useful for flashing modem FW. Once powered on, if a USB cable is connected to the modem board, modem FW can be updated with the appropriated PC SW.

7.2.3 *“Setup Menu”* option

Selecting the *Setup Menu* option displays the setup menu via the serial console as shown in *Table 3*.

Table 3. Setup menu at boot

```
--------------------------------
   Setup Menu
--------------------------------
Select the component to config:
0: Quit
1: Date/Time (RTC)
2: Cellular Service
3: Grovestreams
4: Ping
8: Get list of config sources
9: Erase all feeprom(1) config
```

1. feeprom stands for FEEPROM, which is the embedded Flash memory of the host MCU.

0: Quit

Quits the menu and starts FW.

1: Date/Time (RTC)

Used for setting the system date and time (GMT).

Enter the new date and time according to the `Day, dd Month yyyy hh:mm:ss` format.

Example: `Mon, 11 Dec 2017 17:22:05`

2: Cellular Service

Used to set the Cellular Service parameters:

- APN/CID association
- SIM slot selection
- NFMC feature as defined in *[4]*

Refer to *Section 7.2.4: Cellular Service and Grovestreams configuration sub-menu* and *Section 7.2.6: Cellular Service configuration parameters* for Cellular Service configuration parameter setting.

3: Grovestreams

Setting of the Grovestreams parameters used by the HTTP Client to connect to the Grovestreams Server:

• Grovestreams URL	
• API Keys	Identifiers of the device for the PUT and GET HTTP requests
• Sensor list	List of sensors to get value from

Refer to *Section 7.2.4: Cellular Service and Grovestreams configuration sub-menu* and *Section 7.2.7: Grovestreams configuration parameters* for Grovestreams configuration parameter setting.

4: Ping

Used to set the ping parameters:

- Remote Host IP 1 to ping
- Remote Host IP 2 to ping

A remote host IP is formatted as “xxx.xxx.xxx.xxx”.

Examples:

• Grovestreams IP:	173.236.12.163
• Google™ IP:	8.8.8.8

8: Get list of config sources

Used for getting the list of the available configurations and their sources (either stored in FEEPROM or default values).

Table 4 shows an example for option *8: Get list of config sources* with FEEPROM configuration for the Cellular Service component and default configuration for the Grovestreams component.

Table 4. List of config sources example

```
---------------------------------
   List of config sources
---------------------------------
Cellular Service Config from FEEPROM
Grovestreams Config from DEFAULT
```

9: Erase all feeprom config

Used for erasing all setup configurations stored in FEEPROM and restoring the default settings.

7.2.4 Cellular Service and Grovestreams configuration sub-menu

Table 5 shows the configuration sub-menu displayed for Cellular Service or Grovestreams configuration.

Table 5. Configuration sub-menu

```
---------------------------------
  Setup Menu - Cellular Service
---------------------------------
 c : set config by console
 e : erase config in flash (restore default)
 l : list config
 q : quit
```

c : set config by console

Used for modifying the configuration and to storing it in FEEPROM.

e : erase config in flash

Erases the configuration stored in FEEPROM and restores the default settings.

l : list config

Lists the current configuration.

q : quit menu

Returns to the previous menu.

7.2.5 Set config by console option

When the *set config by console* option is selected though option *c* in the configuration sub-menu, the parameter values must be entered on the console.

There are three ways to enter parameter values:

- Enter the configuration manually from the keyboard
- Copy and paste the configuration from a configuration file
- Send a configuration file to the console by using the Teraterm *send* menu

For each parameter except *Version*, the current value of the parameter is displayed. If no value is entered (return key pressed), the current value is kept.

Version is the first configuration field. It allows checking the configuration version when a configuration file is used. If the version does not match, the configuration is aborted.

- If the configuration is entered manually, the version must be typed as displayed on the first configuration line
- If the configuration is entered from a text file, check that the configuration of the file (first line) match with version of the configuration displayed by the console

Example:

- Version (2): 2

Caution: The format of the configurations can change from one firmware version to the other. In such a case, the configuration stored in FEEPROM is erased at first boot and the default configuration is restored.

7.2.6 Cellular Service configuration parameters

Version (*n*)

Version of the configuration format (refer to *Section 7.2.5: Set config by console option*).

pdn config mode

Used for selecting the APN and CID definition modes.

- 0: Only the CID value is used and transmitted to the modem.
  This option can be used only if an APN/CID association has previously been stored in the modem.
- 1: The APN and CID defined (refer to *APN* and *CID*) are associated and sent to the modem.

The default value of *pdn config mode* is 1.

Caution: If the modem cannot store the configuration permanently, the *pdn config mode* value must be kept set to 1. This is the case for the UG96 modem.

If the modem has the capability to store the configuration permanently, the modem can be configured once using the *pdn config mode* value set to 1. This parameter can later be set to 0.

APN

APN to associated with the CID.

This parameter is taken into account only if parameter *pdn config mode* is set to 1 (refer to *pdn config mode*).

By default, the APN parameter is an empty string. If a non empty string is entered, the only way to set an empty string again is to erase the Cellular Service Flash configuration (refer to *e : erase config in flash*").

The default value of *APN* is an empty string.

CID

CID value to use.

- If parameter *pdn config mode* is set to 1, the CID is associated with the APN
- If parameter *pdn config mode* is set to 0, the APN is not used and the associated APN must be previously stored in the modem

Refer to *pdn config mode* for details about the CID/APN association.

The default value of *CID* is 1.

Sim slot

Sim slot list: specifies the list and order of SIM slots to use at boot time.

- 0: socket slot
- 1: embedded SIM slot
- 2: host SIM slot (not implemented)

The default value of *Sim slot* is 0.

Example: If the list of SIM slots to use is first "embedded SIM slot", then "socket slot", the value to set is 10 ("1" for "embedded SIM slot" and "0" for "socket slot").

NFMC activation

Used for enabling of disabling the NFMC feature.

- 0: NFMC feature disabled
- 1: NFMC feature enabled. Base-temporization parameters are used.

The default value of *NFMC activation* is 0.

The default values of the base-temporization parameters are:

- NFMC value 1: 60000 ms
- NFMC value 2: 120000 ms
- NFMC value 3: 240000 ms
- NFMC value 4: 480000 ms
- NFMC value 5: 960000 ms
- NFMC value 6: 192000 ms
- NFMC value 7: 3840000 ms

Refer to file *Utilities\PC_Software\Tools\Cellular_Service\emnify_cellular_config.txt* for an EMnify cellular configuration example.

7.2.7 Grovestreams configuration parameters

Version (*n*)

Version of the configuration format (refer to *Section 7.2.5: Set config by console option*).

Host to contact

URL, or IP address of host server and port.

Examples:

- Enter the IP of the host to contact (xxx.xxx.xxx.xxx:xxxxx)
- Enter the URL of the host to contact (www.url.com port)
  For instance: www.grovestreams.com 80

PUT API Key

API key for the HTTP PUT request. Used to send sensor values to the Grovestreams site. The API key is obtained from the Grovestreams site (refer to *A.1: How to configure a Grovestreams account? on page 77*).

GET API Key

API key for the HTTP GET request. Used to get values set on the Grovestreams site (LED state). The API key is obtained from the Grovestreams site (refer to *A.1: How to configure a Grovestreams account? on page 77*).

Component ID

Component ID of the device used in Grovestreams configuration.

The default value of *Component ID* is `Sensors_DEF`.

“PUT request” period

Period of the PUT request expressed in seconds. If no PUT request is needed, set this parameter to 0.

The default value of *“PUT request” period* is 25 while the minimum value is 10.

List of sensor value to send

This is a list of associations between the sensor types and the associated identifiers configured in Grovestreams.

Example: on the device the battery level type is 1. In the Grovestreams configuration, if the battery level identifier is set to `batlevel,` the association is:

- Sensor type: 1
- Channel ID: batlevel

Enter 0 as *sensor type* to exit the list.

List of sensor types:

- 0 : end of PUT request configuration
- 1 : Battery level
- 2 : Modem signal level
- 3 : HRM heart rate (not implemented)
- 4 : Pedometer (not implemented)
- 5 : Humidity (available if the sensor shield is plugged)
- 6 : Humidity (available if the sensor shield is plugged)
- 7 : Temperature (available if the sensor shield is plugged)
- 8 : Pressure (available if the sensor shield is plugged)

The default association values are `1/batlevel` and `2/siglevel`.

“GET request” period

Period of the GET request expressed in seconds. If no GET request is needed, set this parameter to 0.

The default value of *“GET request” period* is 2.

List of values to get

This is a list of associations between the type of values to get from Grovestreams and the associated identifiers configured in Grovestreams.

Example: on the device, the LED state type is 1. In the Grovestreams configuration, if the battery level identifier is set to `ledlight,` the association is:

- Sensor type: 1
- Channel ID: ledlight

Enter 0 as *sensor type* to exit the list.

List of sensor types:

- 0 : end of GET request configuration
- 1 : LED state

The default association value is `1/ledlight`.

7.2.8 Boot menu and configuration complete example

```
=============================
      STM32 CELLULAR
      Version: xxx
=============================
Select the application to run:

0: Start
1: Setup Menu
2: Modem power on

--------------------------------
Date: Mon 01/01/2000 - 02:00:42
--------------------------------
   Setup Menu
--------------------------------
Select the component to config:

0: Quit
1: Date/Time (RTC)
2: Cellular Service
3: Grovestreams
4: Ping
8: Get list of config sources
9: Erase all feeprom config

---------------------------------
   Setup Menu - Grovestreams
---------------------------------
c : set config by console
e : erase config in flash (restore default)
l : list config
q : quit
```

```
--------------------------------
 Grovestreams Config from UART
--------------------------------
Enter Grovestreams GET Key (e02848d7-0e7d-3fc4-9bd6-22f62d3c4xxx):
cc706cd8-ebab-3029-8ada-722d879a2xxx
Enter Grovestreams PUT Key (b92a7d9b-1bb3-38cf-8ea2-c1e0ff0dfxxx):
e02848d7-0e7d-3fc4-9bd6-22f62d3c4xxx
Enter Component ID (comp1): comp1

PUT request
Enter PUT request period  (25): 25

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (1): 1
Enter channel ID of sensorEnter Channel ID  (batlevel): batlevel

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (2): 2
Enter channel ID of sensorEnter Channel ID  (siglevel): siglevel

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (5): 5
Enter channel ID of sensorEnter Channel ID  (hum): humidity

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (6): 6
Enter channel ID of sensorEnter Channel ID  (temp): temperature
```

```
Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (7): 7
Enter channel ID of sensorEnter Channel ID  (press): pressure

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (0): 0
Enter GET request period  (2): 2

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (1): 1
Enter channel ID of sensorEnter Channel ID  (ledlight): ledlight

Config a sensor
Select sensor type
  0 : quit
Enter Selection (0 to quit)  (0): 0
save config in feeprom ? (y/n) :y
```

This example is complete with all possible items selected (5 PUT and 1 GET). It can be copied as an template text file and customized into the Teraterm installation folder (by default *C:\Program Files (x86)\teraterm\*), which is proposed by default when the *Send file* option of Teraterm is used.

7.3 Console command

After target boot, the commands presented in this section are available to get and set software component status.

7.3.1 Console command activation

- To activate the console command feature, the `USE_CMD_CONSOLE` compilation variable must set to 1 in file *Projects\STM32L496G-Discovery\Common_Projects_Files\inc\ plf_features.h*.

  ```
  #define USE_CMD_CONSOLE       (1)
  ```
- The board must be connected to a serial console as described in the introduction of *Chapter 7: Interacting with the host board on page 44*.

Note: The commands are case sensitive.

7.3.2 Trace commands

Trace commands are used to enable and disable the display of the trace.

trace off

Suspends trace display.

trace on

Resumes trace display.

7.3.3 Atcmd commands

Atcmd commands are used to send AT commands to the modem.

atcmd <at command>

Sends an AT command to the modem. The command result is displayed as trace.

Example:

```
>atcmd AT+CSQ
AT+CSQ<CR>
    <CR><LF>
    +CSQ: 28,99<CR><LF>
    <CR><LF>
    OK<CR><LF>
>
```

atcmd timeout [<timeout(ms)>]

Gets or sets the modem response timeout. The default value is 5 000 ms.

7.3.4 Cellullar service task (cst) commands

Cellullar service task commands are used to display information about the cellular context.

cst config

Displays the cellular configuration used.

Example:

```
>cst config
pdn mode     : Pdn set
APN          : "EM"
CID          : 1
Sim Slot 0 : MODEM SOCKET
Sim Slot 1 : MODEM EMBEDDED SIM
```

cst info

Displays modem information.

Example:

```
>cst info
Cellular Service Infos
Operator name : "Amarisoft_2 USIM"
IMEI          : 866425030127783
Nanuf name    : Quectel
Model         : BG96
Revision      : BG96MAR02A07M1G
Serial Number : 866425030127783
ICCID         :  89860000502000180722
```

cst state

Displays the cellular state.

Example:

```
>cst state
Cellular Service Task Status
Current State  : MODEM_DATA_READY_STATE
Signal Quality : 27
Sim Selected   : MODEM SOCKET
Sim MODEM SOCKET         : OK
Sim MODEM EMBEDDED SIM         : SIM NOT USED
Sim STM32 EMBEDDED SIM         : SIM NOT USED
Reset Count    : 0
```

7.3.5 HTTP client commands

HTTP client commands are used to enable or disable `httpclient` periodic processing.

httpclient on

Enables `httpclient` periodic processing.

httpclient off

Disables `httpclient` periodic processing.

7.3.6 Ping commands

The ping command is used to generate client ping requests.

ping [<IP ADDR:ddd.ddd.ddd.ddd>]

The IP ADDR address is optional. The default value is "8.8.8.8".

Example:

```
ping 8.8.4.4
<<< HTTP CLIENT ACTIVE 8.8.4.4>>>
>Ping: 32 bytes from 8.8.4.4: seq=01 time= 60ms
Ping: 32 bytes from 8.8.4.4: seq=02 time= 72ms
Ping: 32 bytes from 8.8.4.4: seq=03 time= 57ms
Ping: 32 bytes from 8.8.4.4: seq=04 time= 85ms
Ping: 32 bytes from 8.8.4.4: seq=05 time= 68ms
Ping: 32 bytes from 8.8.4.4: seq=06 time= 102ms
Ping: 32 bytes from 8.8.4.4: seq=07 time= 66ms
Ping: 32 bytes from 8.8.4.4: seq=08 time= 77ms
Ping: 32 bytes from 8.8.4.4: seq=09 time= 76ms
Ping: 32 bytes from 8.8.4.4: seq=10 time= 81ms
--- 8.8.4.4 Ping Statistics ---
Ping: min/avg/max = 57/74/102 ms ok = 10/10
<<< Ping Completed >>>
```

7.3.7 Modem configuration (modem config) commands

Modem configuration commands are used to modify the modem band configuration. Setting a new configuration is performed in two steps:

- enter the configuration parameters
  - `modem config iotopmode`: sets iotop mode
  - `modem config nwscanmode`: sets scan mode
  - `modem config gsmband`: sets the list of GSM bands to use
  - `modem config m1band`: sets the list of M1 bands to use
  - `modem config nb1band`: sets the list of M1 bands to use
  - `modem config scanseq`: sets the sequence order to scan
- send the new configuration to the modem
  - `modem config send`

  Other command available:
  - `modem config`: lists the configuration ready to send

Note: *To use these commands, it is advised to start firmware in "Modem power on" mode (option "2" of the boot menu).*
The new modem configuration is taken into account only after target reboot.

modem config

Displays the current configuration to send.

Example:

```
>modem config
scanmode   : AUTO
iotopmode : M1
GSM bands : f
900
1800
850
1900
any
M1 bands   : 1008
B4
B13
NB1 bands : a0e189f
B1
B2
B3
B4
B5
B8
B12
B13
B18
B19
B20
B26
B28
any
Scan seq : 020301
M1_NB1_GSM
```

modem config nwscanmode [GSM|LTE|AUTO]

Sets or gets the scan mode.

`AUTO`: the modem selects automatically the right mode.

Example:

```
> modem config nwscanmode AUTO
```

modem config scanseq [GSM_NB1_M1|GSM_M1_NB1|M1_GSM_NB1|M1_NB1_GSM|NB1_GSM_M1|NB1_M1_GSM]

Sets or gets the scan sequence (order of the bands scanned)

Example:

```
> modem config scanseq GSM_NB1_M1
```

First scans GSM, then NB1, then M1.

modem config iotopmode [M1|NB1|ALL]

Sets or gets iotop mode.

`ALL`: M1 and NB1 are both available.

Example:

```
> modem config iotopmode ALL
```

Modem config gsmband [900] [1800] [850] [1900] [nochange] [any]

Sets or gets gsm bands.

`any`: set all bands.

`nochange`: no change in GSM band configuration.

Example:

```
> modem config gsmband 900 1900
```

modem config m1band [B1] [B2] [B3] [B4] [B5] [B8] [B12] [B13] [B18] [B19] [B20] [B26] [B28] [B39] [nochange] [any]

Sets or gets M1 bands.

`any`: set all bands.

`nochange`: no change in M1 band configuration.

Example:

```
> modem config m1bands B1 B13
```

modem config nb1band [B1] [B2] [B3] [B4] [B5] [B8] [B12] [B13] [B18] [B19] [B20] [B26] [B28] [nochange] [any]

Sets or gets NB1 bands.

`any`: set all bands.

`nochange`: no change in NB1 band configuration.

Example:

```
> modem config nb1bands B1 B13
```

modem config send

Sends the new configuration to the modem.

Note: The new modem configuration is taken into account only after a target reboot.

Example:

```
> modem config send
```

Full example

For the following configuration:

```
BG96:>>>>> BG96 mode and bands configuration <<<<<
BG96:LTE Cat.M1 band active (scan rank = 1)
BG96:Cat.M1 BANDS config = 0x1008
BG96:CatM1_B4
BG96:CatM1_B13
BG96:GSM band active (scan rank = 2)
BG96:GSM BANDS config = 0xf
BG96:GSM_900
BG96:GSM_1800
BG96:GSM_850
BG96:GSM_1900
```

The set of commands to send is:

```
modem config nwscanmode AUTO
modem config iotopmode M1
modem config gsmband any
modem config m1band B4 B13
modem config nb1band any
modem config scanseq M1_NB1_GSM
modem config send
```

The reboot of the target is then needed (command lines start with the “>” prompt).

```
>modem config nwscanmode AUTO
>modem config iotopmode M1
>modem config gsmband any
GSM bands set
900
1800
850
1900
any

>modem config m1band B4 B13
M1 bands set
B4
B13
```

```
>modem config nb1band any
NB1 bands set
B1
B2
B3
B4
B5
B8
B12
B13
B18
B19
B20
B26
B28
any

>modem config scanseq M1_NB1_GSM
Scan Seq set
M1_NB1_GSM

>modem config send
ATParser:*** SEND (size=25) ***
AT+QCFG="nwscanmode",0,1<CR>
        <CR><LF>
        OK<CR><LF>
        ATParser:*** SEND (size=24) ***
AT+QCFG="iotopmode",0,1<CR>
        <CR><LF>
        OK<CR><LF>
        ATParser:*** SEND (size=29) ***
AT+QCFG="nwscanseq",020301,1<CR>
        <CR><LF>
        OK<CR><LF>
        ATParser:*** SEND (size=32) ***
AT+QCFG="band",f,1008,a0e189f,1<CR>
        <CR><LF>
        OK<CR><LF>
```

8 How to customize the software?

There are three possible software customization levels applicable to X-CUBE-CELLULAR, which are presented in this chapter: user customization, advanced user customization, and developer customization. This chapter also presents how to monitor thread stack consumption.

8.1 First customization level: user customization

The first customization level is the setup configuration at boot time (refer to *Chapter 7: Interacting with the host board on page 44*).

At this level, firmware is not modified. No customization-induced compilation is needed.

8.2 Second customization level: advanced user customization

At this level, firmware configuration modification is possible. Specific features can be added or removed and firmware configuration parameters can be modified as presented in sections *8.2.1*, *8.2.2*, *8.2.3*, and *8.2.4*.

Customization-induced recompilation is needed.

8.2.1 Adding/removing an application in firmware

The *Projects\STM32L496G-Discovery\Common_Projects_Files\inc\plf_features.h* configuration file allows the selection of the applications to be included in firmware.

Table 6 presents the compilation variables that can be defined or undefined as a function of the applications needed.

Table 6. Compilation variables for applications in firmware

Compilation variable[(1)]	Description
`#define USE_HTTP_CLIENT`	Includes the HTTP client (Grovestreams) application.
`#define USE_PING_CLIENT`	Includes ping utilities. The ping application uses COM PING API functionalities. If `USE_PING_CLIENT` is defined, `USE_COM_PING` must be defined too (see the corresponding entry in this table).
`#define USE_DC_MEMS`	Includes sensor management *Projects\Common\tests_utilities\dc_mems.c*. Note: this option can be set only on 32L496GDISCOVERY.
`#define USE_SIMU_MEMS`	Includes the simulation of sensors (no physical sensor available) *Projects\Common\tests_utilities\dc_mems.c* This option can be used also with `USE_DC_MEMS` defined. It is useful if no sensor shield is plugged.

Table 6. Compilation variables for applications in firmware (continued)

Compilation variable[1]	Description
`#define USE_DEFAULT_SETUP`	Defines if the boot setup menu is used or not: – 0: boot setup menu used – 1: boot setup menu not used. In this case, default parameters are set. These parameters are defined in file *<component>config.h* for each component using the setup menu *Projects\Applications\Cellular\radio_service\cellular\inc\cellular_service_config.h* *Projects\Common\applications\HTTP\inc\httpclient_config.h*
`#define USE_DC_EMUL`	Includes sensor emulation for Data Cache test *Projects\Common\tests_utilities\dc_emul.c*
`#define USE_DC_TEST`	Includes Data Cache test *Projects\Common\tests_utilities\dc_test.c*
`#define USE_COM_PING`	Includes ping functionalities in module COM. Because few memory is used with Ping functionalities, this define provides possibility to not include Ping functions if Ping is not used on platform. If `USE_PING_CLIENT` is defined, then `USE_COM_PING` has to be defined.
`#define COM_SOCKETS_ERRNO_COMPAT`	If activated, then when `USE_SOCKETS_TYPE` is set to `USE_SOCKETS_MODEM`, `com_getsockopt` with `COM_SO_ERROR` parameter returns a value compatible with *errno.h* (refer to file *com_sockets_err_compat.c* for the conversion).
`#define USE_CMD_CONSOLE`	Includes console command help (refer to *Section 7.3.1: Console command activation on page 55* and *Section A.5.2: Preparation of the measurements on page 92*).
`#define USE_CELPERF`	Includes cellular throughput performance measurements (refer to *Section A.5.2: Preparation of the measurements on page 92*).

1. All defines are independent: several applications can be selected together.

8.2.2 IP stack on MCU side or on modem side

The IP stack runs either on the MCU side or on the modem side. The default configuration is: modem side.

The *Projects\STM32L496G-Discovery\Common_Projects_Files\inc\plf_feature.h* configuration file include allows the definition of the location of the IP stack used.

Table 7 presents the compilation variable that defines the IP stack used.

Table 7. Compilation variable for IP stack selection

Compilation variable	Description
`#define USE_SOCKETS_TYPE`	Defines the IP stack used: – `USE_SOCKETS_LWIP`: IP stack on the MCU side (LwIP stack) – `USE_SOCKETS_MODEM`: IP stack on the modem side(1)

1. If the IP stack used is on the modem side, only the TCP IPv4 Client is supported.

8.2.3 Different kinds of available traces

Traces are centralized in the `TraceInterface` module.

C macros are defined in each module of X-CUBE-CELLULAR to manage the traces.

The user can easily modify or enrich the implementation according to his needs.

Enhanced UART traces

The traces are sent to the UART connected to the ST-Link, which uses the *TraceInterface* module (for fine trace selection, pretty buffer display, and others).

This is the recommended trace option. It is activated by default.

ITM traces

The Instrumentation Trace Macrocell activates traces that are less intrusive than UART traces. The *TraceInterface* module offers a basic implementation of ITM traces, which the user can enrich according to his needs.

It is activated by default.

Note: Enhanced UART traces and ITM traces are activated by default and provide the same trace at the same time on different communication channels.

To visualize the traces, the STM32 ST-Link Utility software must be installed.

The ITM trace is visualized after the following series of operation is performed:

1. Connect the USB ST-Link connector of the board to the computer
2. Connect the ST-Link Utility to the board (Menu *Target>Connect*)
3. Open the serial viewer (Menu *STLINK>Printf* via the SWO viewer)
4. Set the system clock to the correct value (usually 80 000 000 Hz)
5. Activate all stimulus ports
6. Press start to display the traces

Note: ST-Link Utility must be connected to the board. It is therefore not possible to debug via the IDE and visualize the ITM trace at the same time.

Standard `printf` traces

It is possible to select standard `printf` traces instead of enhanced UART traces.

It uses also the UART connected to ST-Link but offer less options and buffers are not displayed.

8.2.4 How to configure traces?

The configuration of traces is done in file *plf_sw_config.h* by means of the following flags:

- `SW_DEBUG_VERSION`
  - Set this flag to 1 (default value) to enable debug traces
  - Set this flag to 0 to disable debug traces (only setup menu traces will be displayed)
- `TRACE_IF_TRACES_UART`
  - Set this flag to 1 to enable enhanced UART traces (default value)
  - Set this flag to 0 to disable enhanced UART traces
- `TRACE_IF_TRACES_ITM`
  - -Set this flag to 1 to enable ITM traces (default value)
  - Set this flag to 0 to disable ITM traces
- `USE_PRINTF`
  - Set this flag to 1 to enable standard `printf` UART traces
  - Set this flag to 0 to disable standard `printf` UART traces (default value)

Note: When `printf` traces are activated, enhanced UART traces and ITM traces are disabled.

Each software module of X-CUBE-CELLULAR can be enabled or disabled using the corresponding flag beginning by “`USE_TRACE_`”, such as `USE_TRACE_HTTP_CLIENT` or `USE_TRACE_PING_CLIENT` for instance.

8.3 Third customization level: developer customization

8.3.1 Boot

The boot is the first part of the device initialization. It is included in file *main.c* (main function). This part concerns the HW initialization done by HAL. The file is generated by STM32CubeMX with the *.ioc* file provided. It must be updated only if a new HW is used or if the user configures peripherals on the host board (such as GPIO, I^{2}C, or others).

8.3.2 Initialization of software components

The SW components (application and middleware) are initialized in file *freeRTOS.c*. Each component comprises a static initialization to initialize its data structure (`<Component>_Init`), and a real time initialization to start the component thread (`<Component>_Start`). Both are called from the `StartDefaultTask` function in the right order.

8.3.3 Software customization

Firmware configuration parameters are included in files:

• *FreeRTOSConfig.h*:	includes FreeRTOS™ parameters
• *lwipopts.h*:	includes LwIP parameters
• *plf_hw_config.h*:	includes HW parameters (such as UART configuration, GPIO used, and others). A change is usually needed to adapt the software to a new board.
• *plf_sw_config.h*:	includes SW parameters (such as task priorities, trace activations and others)
• *plf_stack_size.h*:	includes thread stack sizes. The stack sizes included in this file are used to calculate the FreeRTOS™ heap size (contained in file *FreeRTOSConfig.h*).
• *plf_features.h*:	includes the selected applications

8.3.4 Firmware adaptation to a new HW configuration

To adapt the firmware to a new board or new HW configuration, follow these steps:

1. Create an STM32CubeMX project based on the new board
2. Configure the HW IPs as configured for the 32L496GDISCOVERY board
3. Generate the software configuration files from STM32CubeMX
4. Update file *plf_hw_config.h* to match the GPIO and HW handler names generated in the configuration files

8.3.5 Adding a new component

To add a new component (application or middleware), follow these steps:

1. Create an initialization function `<Component>_Init` to initialize the data structure of the application
2. Create a starting function `<Component>_Start` to start a component thread
3. Add in the `StartDefaultTask` function (in file *freertos.c*) the call of `<Component>_Init` and `<Component>_Start`
4. Add the stack priority constant in file *plf_sw_config.h* and the stack size in file *plf_stack_size.h*. Other convenient configuration compilation variables can be added as well.
5. Implement the component core. Each component owns at least one thread.

   Interaction with the other components and whole system is done by using Data Cache. Data Cache contains all system data to share between components (sensor values, network state, …). When a component updates a data in Data Cache, all subscribed callback are called.

   Generally a component subscribes a call back to Data Cache. The core of the application thread waits for a Data Cache event and process it.

File *httpclient.c* can be used as example for the creation of a new application.

8.4 Data cache

8.4.1 Introduction

The Data Cache allows the sharing of data and events by software components.

A software component (producer) creates a data entry and writes data in it. Each data entry is associated to an identifier.

The other components (consumers) can read the data by means of the identifier.

A component can subscribe a callback be informed when Data Cache data entry has been updated.

The Data Cache structure includes the `rt_state` field. This field contains the state of service and the validity of entry data.

- DC_SERVICE_UNAVAIL: field values of structure not significant
- DC_SERVICE_ON: service started (field values of structure not significant)
- Other value are entry dependent

8.4.2 Data Cache API

Writing into the Data Cache: dc_com_write

```
dc_com_status_t dc_com_write
(void *dc, dc_com_res_id_t res_id, void *data, uint16_t len)
```

- Input:
  - `dc`: reference to the Data Cache used. Must be set to `&dc_com_db`
  - `res_id`: identifier of the Data Cache entry to write
  - `data`: address of the data structure to write
  - `len`: length of the data structure to write
- Output: none
- Returns an error code

Reading from the Data Cache: `dc_com_read`

```
dc_com_status_t dc_com_read
(void *dc, dc_com_res_id_t res_id, void *data, uint16_t len)
```

- Input:
  - `dc`: reference to the Data Cache used. Must be set to `&dc_com_db`
  - `len`: length of the Data Cache structure to read
  - `res_id`: identifier of the Data Cache entry to read
- Output:
  - `data`: address of the data structure to read
- Returns an error code

Register a callback: `dc_com_register_gen_event_cb`

```
dc_com_reg_id_t dc_com_register_gen_event_cb
(dc_com_db_t *dc_db, dc_com_gen_event_callback_t notif_cb,
void *private_gui_data)
```

- Input:
  - `dc_db`: reference to the Data Cache used. Must be set to `&dc_com_db`
  - `notif_cb`: address of callback. This callback is called when a Data Cache entry has been updated. The callback is executed in the writing task context
  - `private_gui_data`: address of user private context. This address is passed as a parameter of the callback
- Output: none
- Returns an error code

8.4.3 Main Data Cache entries

Modem configuration

The modem configuration contains the cellular parameters used to configure the modem.

If the application needs to set its own modem configuration, it must set the `DC_COM_CELLULAR` Data Cache entry at boot time between the calls to `cellular_init()` and `cellular_start()`.

```
#include  dc_cellular.h
```

Identifier: `DC_COM_CELLULAR_PARAM`

```
typedef struct
{
  dc_service_rt_header_t header;
  dc_service_rt_state_t rt_state;
  uint8_t             set_pdn_mode;
  uint8_t             apn[DC_MAX_SIZE_APN];
  CS_PDN_conf_id_t    cid;
  uint8_t             sim_slot_nb;
  CST_sim_slot_type_t sim_slot[CST_SIM_SLOT_NB];
  uint8_t             nfmc_active;
  uint32_t            nfmc_value[DC_NFMC_TEMPO_NB];
} dc_cellular_params_t;
```

- `rt_status` values:
  - `DC_SERVICE_UNAVAIL`: service not initialized. The field values of structure are not significant
  - `DC_SERVICE_ON`: service started (field values of structure not significant)
  - Other value not used
- `set_pdn_mode`: used for selecting the APN and CID definition modes.
  - 0: Only the CID value is used and transmitted to the modem.

    This option can be used only if an APN/CID association has previously been stored in the modem.
  - 1: The APN and CID defined (refer to APN and CID) are associated and sent to the modem.
- `apn`: APN Value (string)
- `cid`: CID value
- `sim_slot_nb`: number of SIM slots used (max 3)
- `sim_slot`: table of SIM slots used

  `CST_SIM_SLOT_MODEM_SOCKET` = 0

  `CST_SIM_SLOT_MODEM_EMBEDDED_SIM` = 1

  `CST_SIM_SLOT_STM32_EMBEDDED_SIM` = 2
- `nfmc_active`: this flag specifies if NMFC must be activated. If yes, the `nmfc_value` is used.
- `nfmc_value`: table of NMFC values allowing the calculation of NFMC tempos. This field is used only if `nfmc_active==1`.

<u>Example to set the modem configuration:</u>

```
void set_modem_configuration(void)
{
  dc_cellular_params_t cellular_params;

  cellular_params.set_pdn_mode = 1U;        /* PDN to set */
  memset(cellular_params.apn, 0, DC_MAX_SIZE_APN);
  strcpy(cellular_params.apn, "APN");       /* APN value */
  cellular_params.cid = 1;                  /* CID Value */
  cellular_params.sim_slot_nb = 1;         /* Number of sim slot */
  cellular_params.sim_slot[0] = CST_SIM_SLOT_MODEM_SOCKET;
                        /* SIM slot to use : Modem socket  */
  cellular_params.nfmc_active = 0U;         /* NFMC disabled */
  cellular_params.rt_state = DC_SERVICE_ON;/* Modem Config valid */
  /* Write modem config to Data Cache */
  dc_com_write(&dc_com_db, DC_COM_CELLULAR_PARAM,
  (void *)&cellular_params, sizeof(cellular_params));
}
/* StartDefaultTask in freertos.c  */

void StartDefaultTask(void const *argument)
{
  /* ... */
  cellular_init();
  /* ... */
  set_modem_configuration();
  /* ... */
  cellular_start();
  /* ... */
}
```

Cellular information

Contains the main cellular information received from the modem after the boot.

```
#include  dc_cellular.h
```

Identifier: `DC_COM_CELLULAR_INFO`

```
typedef struct
{
  dc_service_rt_header_t header;
  dc_service_rt_state_t rt_state;

  uint32_t    cs_signal_level;
  int32_t     cs_signal_level_db;

  int8_t imei[DC_MAX_SIZE_IMEI];
  int8_t mno_name[DC_MAX_SIZE_MNO_NAME];
  int8_t manufacturer_name[DC_MAX_SIZE_MANUFACT_NAME];
  int8_t model[DC_MAX_SIZE_MODEL];
  int8_t revision[DC_MAX_SIZE_REV];
  int8_t serial_number[DC_MAX_SIZE_SN];
  int8_t iccid[DC_MAX_SIZE_ICCID];

} dc_cellular_info_t;
```

- `rt_status` values
  - `DC_SERVICE_UNAVAIL`: service not initialized. The field values of structure are not significant
  - `DC_SERVICE_RUN`: modem powered on and initialized. The field values of structure are significant except MNO name
  - `DC_SERVICE_ON`: modem attached. All the field values of structure are significant.
  - Other value not used

Example:

```
    #include "dc_cellular.h"

    dc_cellular_info_t      cst_cellular_info;
    dc_com_read(&dc_com_db, DC_COM_CELLULAR_INFO,
    (void *)&cst_cellular_info, sizeof(dc_cellular_info_t));
```

Sim information

Contains information of the available SIM.

```
#include  dc_cellular.h
```

Identifier: `DC_COM_SIM_INFO`

```
typedef struct
{
  dc_service_rt_header_t header;
  dc_service_rt_state_t  rt_state;
  int8_t                 imsi[DC_MAX_SIZE_IMSI];
  uint16_t               index_slot;
  uint16_t               active_slot;
  dc_cs_sim_status_t     sim_status[CST_SIM_SLOT_NB];
} dc_sim_info_t;
```

- `rt_status` values
  - `DC_SERVICE_UNAVAIL`: service not initialized. The field values of structure are not significant
  - `DC_SERVICE_ON`: modem powered on and initialized. The other field values of structure are significant
  - Other value not used

Example

```
#include "dc_cellular.h"

dc_sim_info_t           cst_sim_info;
dc_com_read(&dc_com_db, DC_COM_SIM_INFO, (void *)&cst_sim_info,
sizeof(dc_sim_info_t));
```

Data Network information

Contains information about the network.

```
#include  dc_cellular.h
```

Identifier: `DC_COM_NIFMAN_INFO`

```
typedef struct
{
  dc_service_rt_header_t header;
  dc_service_rt_state_t  rt_state;
  dc_nifman_network_t    network;
  dc_network_addr_t      ip_addr;
} dc_nifman_info_t;
```

- `rt_status` values
  - `DC_SERVICE_UNAVAIL`: service not initialized. The field values of structure are not significant
  - `DC_SERVICE_ON`: network available. The other field values of structure are significant
  - `DC_SERVICE_OFF`: network not available
  - `DC_SERVICE_FAIL`: network stack returns error. Network is not available
  - `DC_SERVICE_SHUTTING_DOWN`: network shut down. Network is not available
  - Other value not used

Example:

```
#include "dc_cellular.h"
dc_nifman_info_t           nifman_info;
dc_com_read(&dc_com_db, DC_COM_NIFMAN_INFO, (void
*)&nifman_info, sizeof(dc_nifman_info_t));
```

8.4.4 Example of producer/consumer code

The following example shows how NIFMAN Data Cache entry in managed. It shows:

- the NIFMAN code: NIFMAN is the producer of NIFMAN Data Cache entry
- `httpclient` as an example of consumer: at the beginning `httpclient` waits for the network start.

Producer code (NIFMAN)

dc_cellular.h

```
/* Declaration of Data Cache structure */
typedef struct
{
  dc_service_rt_header_t header;
  dc_service_rt_state_t  rt_state;
  dc_nifman_network_t    network;
  dc_network_addr_t      ip_addr;
} dc_nifman_info_t;

/* Declaration of NIFMAN Data Cache identifier */
extern dc_com_res_id_t    DC_COM_NIFMAN        ;
```

nifman.c

```
/* Update of NIMAN Data Cache structure */
dc_nifman_info_t           nifman_info;
dc_com_status_t          dc_status ;
/* read the current Data Cache structure tu update */
dc_status   = dc_com_read(&dc_com_db, DC_COM_NIFMAN_INFO, (void
*)&nifman_info, (uint16_t)sizeof(nifman_info));
/* modify structure values */
nifman_info.network      =  DC_CELLULAR_SOCKET_MODEM;
nifman_info.rt_state     =  DC_SERVICE_ON;
```

```
dc_status  = dc_com_write(&dc_com_db, DC_COM_NIFMAN_INFO, (void
*)&nifman_info, (uint16_t)sizeof(nifman_info));
```

Consumer code (httpclient)

```
void http_client_start(void)
{
   /* … */
   /* Registration callback to Data Cache - for Network On/Off
detection */
   dc_com_register_gen_event_cb(&dc_com_db, http_client_notif_cb,
(void *) NULL);
   /* … */
}

/* http client callback code */
static void http_client_notif_cb(dc_com_event_id_t dc_event_id,
                                 void *private_gui_data)
{
  /* Test if it is the NIFMAN event */
  if (dc_event_id == DC_COM_NIFMAN_INFO)
  {
    dc_nifman_info_t  dc_nifman_info;
    /* read the NIFMAN Data Cache structure */
    dc_com_read(&dc_com_db, DC_COM_NIFMAN_INFO,
                (void *)&dc_nifman_info, sizeof(dc_nifman_info));

    if (dc_nifman_info.rt_state == DC_SERVICE_ON)
    {
      /* the Network is On */
      network_is_on = HTTPCLIENT_TRUE;
      http_client_ip_addr = dc_nifman_info.ip_addr.addr;
      /* wakeup  the http client task to complete the processing */
      osMessagePut(http_client_queue, (uint32_t)dc_event_id, 0U);
    }
    else
```

```
        {
          /* the Network is Off */
          network_is_on = HTTPCLIENT_FALSE;
        }
      }
    }
```

8.5 Thread stack consumption monitoring

The `Stack Analysis` module enables the monitoring of the thread stack consumption.

Each time a thread is created, its registration must be added to `Stack Analysis` to provide its stack size allocation (this is already done for all the threads declared in the project).

Table 8 shows, as for any new thread creation, the addition of a call to service `stackAnalysis_addThreadStackSizeByHandle()`.

Table 8. New thread registration example

```
osThreadDef(defaultTask, StartDefaultTask, CTRL_THREAD_PRIO, 0,
            CTRL_THREAD_STACK_SIZE);
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
#if (STACK_ANALYSIS_TRACE == 1)
     stackAnalysis_addThreadStackSizeByHandle(defaultTaskHandle,
CTRL_THREAD_STACK_SIZE);
#endif /* STACK_ANALYSIS_TRACE */
```

The number of project threads is calculated according to feature activation (refer to file *plf_stack_size.h)* as shown in *Table 9*.

Table 9. Number of project threads setting example

```
#define THREAD_NUMBER                             \
               USED_TCPIP_THREAD                  \
              +USED_DEFAULT_THREAD                \
              +USED_PPPOSIF_CLIENT_THREAD         \
              +USED_DC_CTRL_THREAD                \
              +USED_ATCORE_THREAD                 \
              +USED_NIFMAN_THREAD                 \
              +USED_DC_TEST_THREAD                \
              +USED_DC_MEMS_THREAD                \
              +USED_DC_EMUL_THREAD                \
              +USED_HTTPCLIENT_THREAD             \
              +USED_PINGCLIENT_THREAD             \
              +USED_FREERTOS_TIMER_THREAD         \
              +USED_FREERTOS_IDLE_THREAD          \
              +USED_CMD_THREAD                    \
              +USED_CELLULAR_SERVICE_THREAD
```

To monitor the consumption of the thread stacks during software execution, the lines shown in *Table 10* are added to `StartDefaultTask`, at the end of initialization.

Table 10. Code for thread stack consumption monitoring

```
    for(;;)
    {
#if (STACK_ANALYSIS_TRACE == 1)
        stackAnalysis_trace(true);
        /* print stack analysis status every 5 sec*/
        osDelay(5000);
#else
        osDelay(1000);
#endif
    }
```

The monitoring is not activated by default. To activate it (`printf` of all thread stack size evolutions), change in file *plf_sw_config.h*

```
#define STACK_ANALYSIS_TRACE   (0)
```

by

```
#define STACK_ANALYSIS_TRACE   (1)
```

The monitoring default configuration results in a print of all thread stack size maximum occupation every 5 seconds.

For more options regarding the configuration monitoring refer to file *stack_analysis.h.*

Appendix A Support material

A.1 How to configure a Grovestreams account?

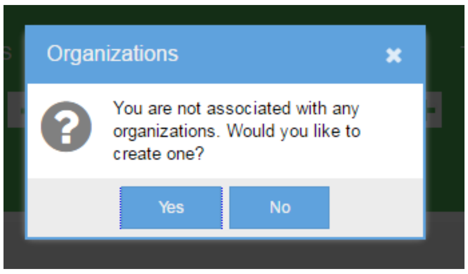
The different steps are:

1. Create an email account if needed (a valid email account is needed to create a Grovestreams account)
2. Create a Grovestreams account (free sign up)
3. Setup the Grovestreams account (create the organization with the blueprint)
4. Get the needed API keys (used in STM32 MCU FW)

The series of figures from *Figure 28* to *Figure 33* illustrate the creation and configuration of a Grovestreams account.

Figure 28 shows the login screen for the creation of a Grovestreams organization.

Figure 28. Grovestreams organization creation acceptance screen

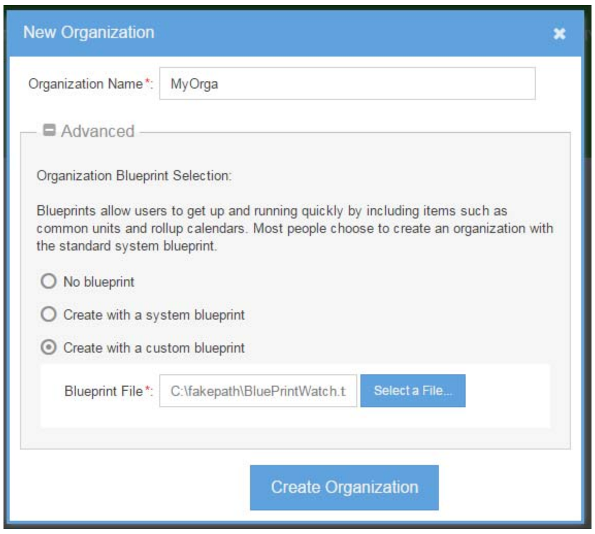

Organizations

You are not associated with any organizations. Would you like to create one?

The blueprint (template) is used for the creation of the organization as shown in *Figure 29*:

- Enter the organization name in field *Organization Name*
- Tick the *Create with a custom blueprint* option
- Select the *Blueprint File*
- Validate with *Create Organization*

Figure 29. Grovestreams organization creation screen


New Organization

Organization Name*: MyOrga

Advanced

Organization Blueprint Selection:

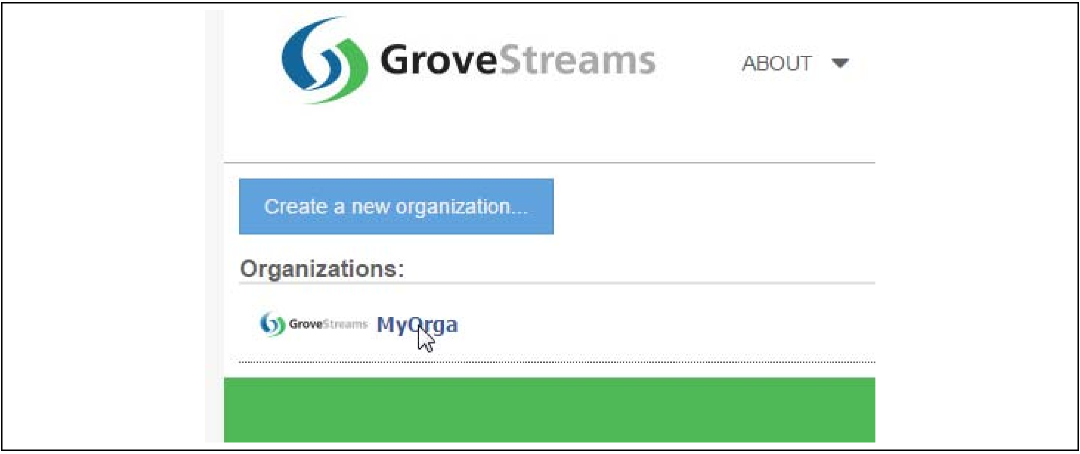
Blueprints allow users to get up and running quickly by including items such as common units and rollup calendars. Most people choose to create an organization with the standard system blueprint.

No blueprint
Create with a system blueprint
[✓ Create with a custom blueprint]

Blueprint File*: C:\fakepath\BluePrintWatch.t

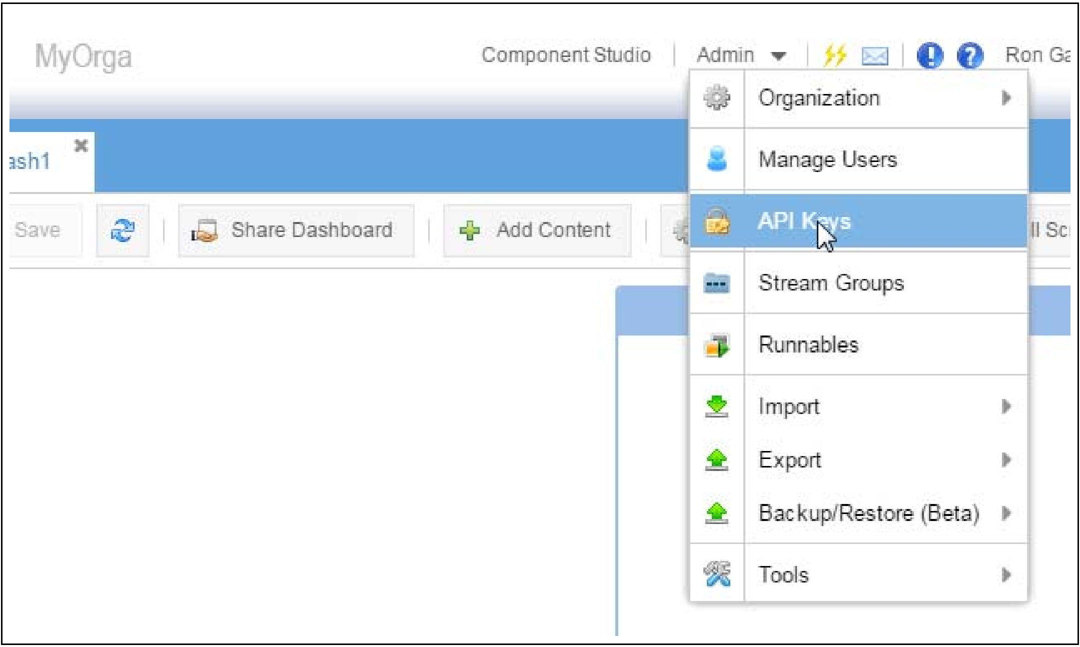
Get access to the organization as shown in *Figure 30*.

Figure 30. Grovestreams organization access screen



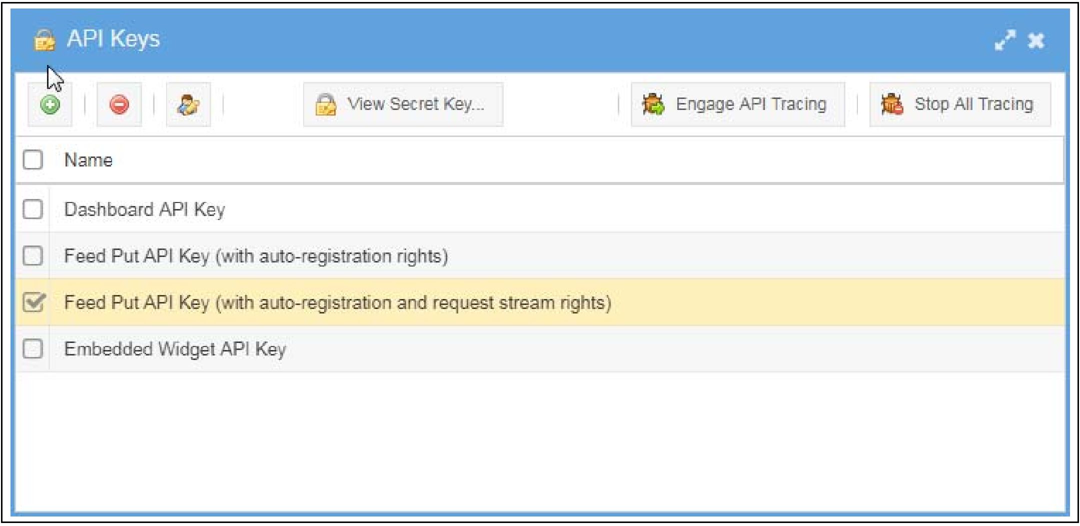
Prepare to copy the API keys by selecting *Admin* > *API Keys* as shown in *Figure 31*.

Figure 31. Grovestreams organization administration menu



Select *Feed Put API Key* as shown in *Figure 32*.

Figure 32. Grovestreams API key selection screen



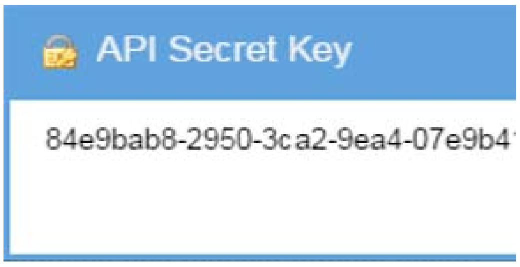
API Keys

View Secret Key... Engage API Tracing Stop All Tracing

	Name
☐	Dashboard API Key
☐	Feed Put API Key (with auto-registration rights)
☑	Feed Put API Key (with auto-registration and request stream rights)
☐	Embedded Widget API Key

Select *View Secret Key* to display the key as illustrated in *Figure 33*.

Figure 33. Grovestreams API key display screen



API Secret Key

84e9bab8-2950-3ca2-9ea4-07e9b4

The API key can be copied and pasted as per need. Proceed similarly for the dashboard API key, by un-selecting *Feed Put API Key* before selecting *Dashboard API Key.*

A.2 How to activate the soldered SIM card?

The C2C kits are provided with a UICC chip pre-provisioned by the EMnify MVNO, which allows worldwide 2G and 3G roaming. This is not an eUICC but an embedded SIM. It is a classical SIM profile provisioned in a chip during the factory process. It is not an eUICC that allows remote profile update.

Note: This section does not apply to the “Discovery IoT node cellular” set, which features no embedded SIM provisioned by an M(V)NO.

LTE Cat M1 and NB1 technologies are not supported by EMnify. In case of UICC activation on a BG96 board, only 2G fallback is possible.

To use the BG96 modem in Cat M1 or NB1 mode, the user must operate with the microSIM plastic card.

There is no restriction for UG96 add-on boards.

The steps to activate EMnify on the UICC chip are:

1. Boot the board with a terminal connected so that a text can be read
2. Get the voucher from the displayed text
3. Connect with a PC to https://stm32-c2c.com
4. Register the board using the voucher
5. Select the EMnify item in the displayed board
6. Follow the activation procedure
7. The EMnify profile is immediately activated (double click on the SIM to display the SIM status)

A.3 Frequently asked questions

Q: Where to find the blueprint file for creating a Grovestreams organization?

A: In */Utilities/PC_Software/Grovestreams*, as file *GS_Blueprint.txt.*

Q: Where to find the text file for updating Grovestreams parameters?

A: In */Utilities/PC_Software/Grovestreams*, as file *GS_Setup.txt.*

Q: How to know if the cellular application is up and running?

A: Either with the trace on Teraterm or by connecting to the Grovestreams account and checking that the data is updated.

Q: Nothing is displayed in Teraterm.

A: Ensure that Teraterm options (such as *Baud rate*) are correctly set.

Q: Wrong echo in Teraterm during setup.

A: Ensure that Teraterm options (such as *New-line* and *Transmit delay*) are correctly set.

Q: It boots but there is an issue with the network.

A: Check that the correct SIM is selected (Plastic SIM or soldered UICC; Plastic SIM by default).

Q: How to check that the EMnify profile on UICC is correctly activated?

A: Connect to the EMnify account and check the status.

Q: EMnify is correctly activated but it does not work.

A: Check in the EMnify account if the correct tariff is selected.

Q: The log indicates it is properly up and running but nothing changes in Grovestreams.

A: Check that the correct API keys are used (the ones associated with the organization used for the demonstration).

Q: The voucher cannot be read because the initial image was overwritten.

A: Browse to www.stm32-c2c.com and download *Restore factory firmware*.

Q: With the out-of-the-box FW, there is no information on the console.

A: Set the correct parameter in Teraterm (*Baud rate* as 9600 to obtain the voucher) and make sure that the modem is not connected upside-down to the host board through the STMod+ connector.

A.4 X-CUBE-CELLULAR API descriptions

This section presents the API used by upper layers (applications).

The API are exposed from 2 modules, COM, and Data Cache.

A.4.1 COM API

The pieces of software in this section are extracted from file *Application\net\com\inc\com_sockets.h.*

Socket management

Table 11. COM API management - Socket handle creation

```
/**
  * @brief  Socket handle creation
  * @note   Create a communication endpoint called socket
  * @param  family   - address family
  * @param  type     - connection type
  * @param  protocol - protocol type
  * @retval int32_t  - socket handle or error value
  */
int32_t com_socket(int32_t family, int32_t type, int32_t protocol);
```

Table 12. COM API management - Socket option set

```
/**
  * @brief  Socket option set
  * @note   Set option for the socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  level     - level at which the option is defined
  * @param  optname   - option name for which the value is to be set
  * @param  optval    - pointer to the buffer containing the option value
  * @param  optlen    - size of the buffer containing the option value
  * @retval int32_t   - ok or error value
  */
int32_t com_setsockopt(int32_t sock, int32_t level, int32_t optname,
                       const void *optval, int32_t optlen);
```

Table 13. COM API management - Socket option get

```
/**
  * @brief  Socket option get
  * @note   Get option for a socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  level     - level at which option is defined
  * @param  optname   - option name for which the value is requested
  * @param  optval    - pointer to the buffer that will contain the option value
  * @param  optlen    - size of the buffer that will contain the option value
  * @retval int32_t   - ok or error value
  */
int32_t com_getsockopt(int32_t sock, int32_t level, int32_t optname,
                       void *optval, int32_t *optlen);
```

Table 14. COM API management - Socket bind

```
/**
  * @brief  Socket bind
  * @note   Assign a local address and port to a socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  addr      - local IP address and port
  * @param  addrlen   - addr length
  * @retval int32_t   - ok or error value
  */
int32_t com_bind(int32_t sock,
                 const com_sockaddr_t *addr, int32_t addrlen);
```

Table 15. COM API management - Socket close

```
/**
  * @brief  Socket close
  * @note   Close a socket and release socket handle
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @retval int32_t   - ok or error value
  */
int32_t com_closesocket(int32_t sock);
```

Client functionalities

Table 16. COM API client - Socket connect

```
/**
  * @brief  Socket connect
  * @note   Connect socket to a remote host
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  addr      - remote IP address and port
  * @param  addrlen   - addr length
  * @retval int32_t   - ok or error value
  */
int32_t com_connect(int32_t sock,
                    const com_sockaddr_t *addr, int32_t addrlen);
```

Table 17. COM API client - Socket send data

```
/**
  * @brief  Socket send data
  * @note   Send data on already connected socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  buf       - pointer to application data buffer to send
  * @param  len       - size of the data to send (in bytes)
  * @param  flags     - options
  * @retval int32_t   - number of bytes sent or error value
  */
int32_t com_send(int32_t sock,
                 const com_char_t *buf, int32_t len, int32_t flags);
```

Table 18. COM API client - Socket receive data

```
/**
  * @brief  Socket receive data
  * @note   Receive data on already connected socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  buf       - pointer to application data buffer to store the data to
  * @param  len       - size of application data buffer (in bytes)
  * @param  flags     - options
  * @retval int32_t   - number of bytes received or error value
  */
int32_t com_recv(int32_t sock,
                 com_char_t *buf, int32_t len, int32_t flags);
```

Server functionalitie

Table 19. COM API server - Socket listen

```
/**
  * @brief  Socket listen
  * @note   Set socket in listening mode
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  backlog   - number of connection requests that can be queued
  * @retval int32_t   - ok or error value
  */
int32_t com_listen(int32_t sock, int32_t backlog);
```

Table 20. COM API server - Socket accept

```
/**
  * @brief  Socket accept
  * @note   Accept a connect request for a listening socket
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  addr      - IP address and port number of the accepted connection
  * @param  len       - addr length
  * @retval int32_t   - ok or error value
  */
int32_t com_accept(int32_t sock,com_sockaddr_t *addr, int32_t *addrlen);
```

Table 21. COM API server - Socket send to data

```
/**
  * @brief  Socket send to data
  * @note   Send data to a remote host
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  buf       - pointer to application data buffer to send
  * @param  len       - length of the data to send (in bytes)
  * @param  flags     - options
  * @param  addr      - remote IP address and port number
  * @param  len       - addr length
  * @retval int32_t   - number of bytes sent or error value
  */
int32_t com_sendto(int32_t sock,
                   const com_char_t *buf, int32_t len, int32_t flags,
                   const com_sockaddr_t *to, int32_t tolen);
```

Table 22. COM API server - Socket receive from data

```
/**
  * @brief  Socket receive from data
  * @note   Receive data from a remote host
  * @param  sock      - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  buf       - pointer to application data buffer to store the data to
  * @param  len       - size of application data buffer (in bytes)
  * @param  flags     - options
  * @param  addr      - remote IP address and port number
  * @param  len       - addr length
  * @retval int32_t   - number of bytes received or error value
  */
int32_t com_recvfrom(int32_t sock,
                     com_char_t *buf, int32_t len, int32_t flags,
                     com_sockaddr_t *from, int32_t *fromlen);
```

Other functionalities

Table 23. COM API other - Component initialization

```
/**
  * @brief  Component initialization
  * @note   must be called only one time and
  *         before using any other functions of com_*
  * @param  None
  * @retval bool      - true/false init ok/nok
  */
com_bool_t com_init(void);
```

Table 24. COM API other - Component start

```
/**
  * @brief  Component start
  * @note   must be called only one time but
  *         after com_init and dc_start
  *         and before using any other functions of com_*
  * @param  None
  * @retval None
  */
void com_start(void);
```

Table 25. COM API other - Get host IP from host name

```
/**
  * @brief  Get host IP from host name
  * @note   Retrieve host IP address from host name
  * @param  name       - host name
  * @param  addr       - host IP corresponding to host name
  * @retval int32_t    - ok or error value
  */
int32_t com_gethostbyname(const com_char_t *name,
                          com_sockaddr_t *addr);
```

Table 26. COM API other - Get peer name

```
/**
  * @brief  Get peer name
  * @note   Retrieve IP address and port number
  * @param  sock       - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  name       - IP address and port number of the peer
  * @param  namelen    - name length
  * @retval int32_t    - ok or error value
  */
int32_t com_getpeername(int32_t sock,
                        com_sockaddr_t *name, int32_t *namelen);
```

Table 27. COM API other - Get sock name

```
/**
  * @brief  Get sock name
  * @note   Retrieve local IP address and port number
  * @param  sock       - socket handle obtained with com_socket
  * @note   socket handle on which operation has to be done
  * @param  name       - IP address and port number
  * @param  namelen    - name length
  * @retval int32_t    - ok or error value
  */
int32_t com_getsockname(int32_t sock,
                        com_sockaddr_t *name, int32_t *namelen);
```

Table 28. COM API other - Ping APIs

```
#if (USE_COM_PING == 1)
/**
  * @brief  Ping handle creation
  * @note   Create a ping session
  * @param  None
  * @retval int32_t  - ping handle or error value
  */
int32_t com_ping(void);

/**
  * @brief  Ping process request
  * @note   Create a ping session
  * @param  ping     - ping handle obtained with com_ping
  * @note   ping handle on which operation has to be done
  * @param  addr     - remote IP address and port
  * @param  addrlen  - addr length
  * @param  timeout  - timeout for ping response (in sec)
  * @param  rsp      - ping response
  * @retval int32_t  - ok or error value
  */
int32_t com_ping_process(int32_t ping,
                         const com_sockaddr_t *addr, int32_t addrlen,
                         uint8_t timeout, com_ping_rsp_t *rsp);

/**
  * @brief  Ping close
  * @note   Close a ping session and release ping handle
  * @param  ping      - ping handle obtained with com_socket
  * @note   ping handle on which operation has to be done
  * @retval int32_t   - ok or error value
  */
int32_t com_closeping(int32_t ping);
#endif /* USE_COM_PING == 1 */
;
```

A.4.2 Data Cache API

The Data Cache API is split into several files and covers event ID, data ID and services.

Event ID

Table 29. Data Cache API in *dc_control.h* (event ID)

```
extern dc_com_res_id_t    DC_COM_BUTTON_UP    ;
extern dc_com_res_id_t    DC_COM_BUTTON_DN    ;
extern dc_com_res_id_t    DC_COM_BUTTON_RIGHT ;
extern dc_com_res_id_t    DC_COM_BUTTON_LEFT  ;
extern dc_com_res_id_t    DC_COM_BUTTON_SEL   ;
```

Cellular data ID

Table 30. Data Cache API in *dc_cellular.h* (cellular data ID)

```
extern dc_com_res_id_t    DC_COM_CELLULAR      ;
extern dc_com_res_id_t    DC_COM_PPP_CLIENT    ;
extern dc_com_res_id_t    DC_COM_CELLULAR_DATA ;
extern dc_com_res_id_t    DC_COM_RADIO_LTE     ;
extern dc_com_res_id_t    DC_COM_NIFMAN        ;
extern dc_com_res_id_t    DC_COM_NFMC_TEMPO    ;
extern dc_com_res_id_t    DC_COM_SIM_INFO      ;
```

Sensor data ID

Table 31. Data Cache API in *dc_mems.h* (sensor data ID)

```
extern dc_com_res_id_t  DC_COM_PRESSURE      ;
extern dc_com_res_id_t  DC_COM_HUMIDITY      ;
extern dc_com_res_id_t  DC_COM_TEMPERATURE   ;
extern dc_com_res_id_t  DC_COM_ACCELEROMETER ;
extern dc_com_res_id_t  DC_COM_GYROSCOPE     ;
extern dc_com_res_id_t  DC_COM_MAGNETOMETER  ;
```

Services

Table 32. Data Cache API in *dc_common.h* (services)

```
/**
  * @brief  to register a generic cb to a given dc_db
  * @param  dc_db                data base reference
  * @param  notif_cb             user callback
  * @param  private_gui_data     user context
  * @retval dc_com_reg_id_t      created entry identifier
  */
dc_com_reg_id_t dc_com_register_gen_event_cb(
    dc_com_db_t*dc_db,
    dc_com_gen_event_callback_t notif_cb, /* the user event callback */
    void *private_gui_data);              /* user private data */

/**
  * @brief  update a data info in the DC
  * @param  dc                   data base reference
  * @param  res_id               resource id
  * @param  data                 data to write
  * @param  len                  len of data to write
  * @retval dc_com_status_t      return status
  */
dc_com_status_t dc_com_write (void *dc, dc_com_res_id_t res_id, void *data, uint16_t
len);

/**
  * @brief  read current data info in the DC
  * @param  dc                   data base reference
  * @param  res_id               resource id
  * @param  data                 data to read
  * @param  len                  len of data to read
  * @retval dc_com_status_t      return status
  */
dc_com_status_t dc_com_read  (void *dc, dc_com_res_id_t res_id, void *data, uint16_t
len);

/**
  * @brief  send an event to DC
  * @param  dc                   data base reference
  * @param  event_id             event id
  * @retval dc_com_status_t      return status
  */
dc_com_status_t dc_com_write_event (void *dc, dc_com_event_id_t event_id);
```

Table 33. Data Cache API in *dc_time.h* (services)

```
/**
  * @brief  set system date and time
  * @param  dc_time_date_rt_info     (in) date to set
  * @param  dc_time_data_type_t      (in) time to set
  * @retval dc_srv_ret_t             return status
  */
dc_srv_ret_t dc_srv_get_time_date(dc_time_date_rt_info_t* dc_time_date_rt_info,
                                  dc_time_data_type_t time_date);

/**
  * @brief  get system date and time
  * @param  dc_time_date_rt_info     (out) date
  * @param  dc_time_data_type_t      (out) time
  * @retval dc_srv_ret_t             return status
  */
dc_srv_ret_t dc_srv_set_time_date(const dc_time_date_rt_info_t* time,
                                  dc_time_data_type_t time_date);
```

Table 34. Data Cache API in *cellular_init.h* (services)

```
/**
  * @brief  get cellular component initialization
  * @retval no return value
  */
void cellular_init(void);
/**
  * @brief  get cellular component start
  * @retval no return value
  */
void cellular_start(void);
```

A.5 How to measure cellular throughput?

A.5.1 Introduction

Throughput measurements are done by sending data to a server.

The measurements are done successively for buffer data of 16, 32, 64, 128,256, 512, 1024 and 1400 bytes.

For each data size, several iterations are done to increase accuracy.

A.5.2 Preparation of the measurements

- Server installation
  - Get the server IP ADDR using the `ipconfig` command

    ```
    ipconfig
    ```
  - Copy the *Utilities\PC_Software\Performance\perf_server\perf_tcp_rcv_server.c* file onto a Linux® PC connected to the network
  - Compile the *perf_tcp_rcv_server.c* file using

    ```
    cc -o perf_tcp_rcv_server perf_tcp_rcv_server.c
    ```
  - Start the server

    ```
    ./perf_tcp_rcv_server
    ```
- Target preparation
  - Enable performance feature :

    in file *Projects\STM32L496G-Discovery\Common_Projects_Files\inc\plf_features.h* update the following lines:

    ```
    #define USE_HTTP_CLIENT (0) // 0: not activated, 1: activated
    #define USE_CMD_CONSOLE (1) // 0: not activated, 1: activated
    #define USE_CELPERF (1) // 0: not activated, 1: activated
    ```
  - Generate and load and start new firmware
  - Connect a serial console to the target (refer to *Chapter 7: Interacting with the host board on page 44*)
  - Wait for the network start
  - Disable trace using command on the serial console

    ```
    trace off
    ```
  - Set right server IP address using command on the serial console

    ```
    perf addr <IP addr of server: ddd.ddd.ddd.ddd>
    ```
  - Start throughput measure. Note: the measurements takes several minutes.

    ```
    perf start
    ```

A.5.3 Example

```
>trace off
 <<< TRACE INACTIVE >>>
>perf start

socket connect OK
tcpip perf snd started...

 size    iter   data(B) time(ms)         throughput(Byte/s)
   16    5000     80000  103843            776
   32    3000     96000   66772           1454
   64    1000     64000   24837           2666
  128    1000    128000   30330           4266
  256     500    128000   20894           6400
  512     200    102400   12925           8533
 1024     100    102400   10986          10240
 1400     100    140000   14277          10000
Perf measure completed
```

A.6 How to select BG96 modem configuration bands?

The configuration of BG96 modem bands can be done using console commands (refer to *Chapter 7: Interacting with the host board on page 44*).

Revision history

Table 35. Document revision history

Date	Revision	Changes
28-Jun-2018	1	Initial release.
2-Nov-2018	2	X-CUBE-CELLULAR support extended to the "Discovery IoT node cellular" set: – Updated *Introduction*, *Chapter 1: General information*, *Section 4.2: Modem socket versus LwIP*, *Chapter 6: Hardware and software environment setup*, *Section 5.4: Grovestreams (HTTP) access example*, and *A.2 How to activate the soldered SIM card?* Augmented X-CUBE-CELLULAR feature description: – Added *Section 7.3: Console command*, *Section 8.4: Data cache*, *Section 8.2.3: Different kinds of available traces*, *Section 8.2.4: How to configure traces?*, and *A.6 How to select BG96 modem configuration bands?* Updated X-CUBE-CELLULAR feature description: – NIFMAN in *Section 4.3.2: Static architecture view* – PING in *Section 5.3: PING example*, *Section 7.2.3: "Setup Menu" option*, and *Section 7.2.8: Boot menu and configuration complete example* – Menu in *Section 7.2: Boot menu* – Compilation variables in *Table 6: Compilation variables for applications in firmware*