G3-Hybrid PLC & RF software package

Introduction

This document describes the software architecture and implementation of the host controller application firmware example included in the STSW-ST8500GH-2 software package.

The package provides the software ecosystem for ST's G3-Hybrid technology evaluation, combining Power Line Communication (PLC) & Radio Frequency (RF) sub-GHz connectivity. The package is based on the ELVKST8500GH-2, that includes all the functions required for plug-and-play communication networking.

The host controller application firmware example, based on FreeRTOS and available for STM32G070RB, STM32G474RE or STM32L476RG, allows testing the PLC and RF communication, evaluating the functionalities of the G3-Hybrid protocol and making use of the IPv6 layer interface of the ST8500 modem. The G3-Hybrid communication stack has full flexibility to be configured in any of the available bandplans for both PLC (CENELEC-A, CENELEC-B or FCC) and RF, based on local regulations.

Messages between two nodes in the PLC & RF hybrid network are sent over the best available medium: PLC or RF. The media selection for each link in the network is done automatically and adjusted dynamically, enabling highly efficient hybrid mesh networking.

The G3-Hybrid solution by ST is based on open standards, is fully compliant with G3-Alliance specifications and enables seamless integration into existing G3-PLC networks and adoption in multiple applications and systems.

1 G3-Hybrid software solution overview

1.1 What is the G3-Hybrid?

The G3-Hybrid is the first industry hybrid communication standard offering extended capabilities for Smart Grid and IoT applications in one seamlessly managed network over both wired (PLC) and wireless (RF) media.

The G3-Hybrid protocol stack is built using open standard IEEE 802.15.4-2015 in addition to the existing G3-PLC protocol. Each device in the mesh network can use PLC as well as RF for communication. Depending on the actual conditions in the field, messages between two devices are sent over the most reliable channel available. The channel selection for each link in the network is done automatically and adjusted dynamically.

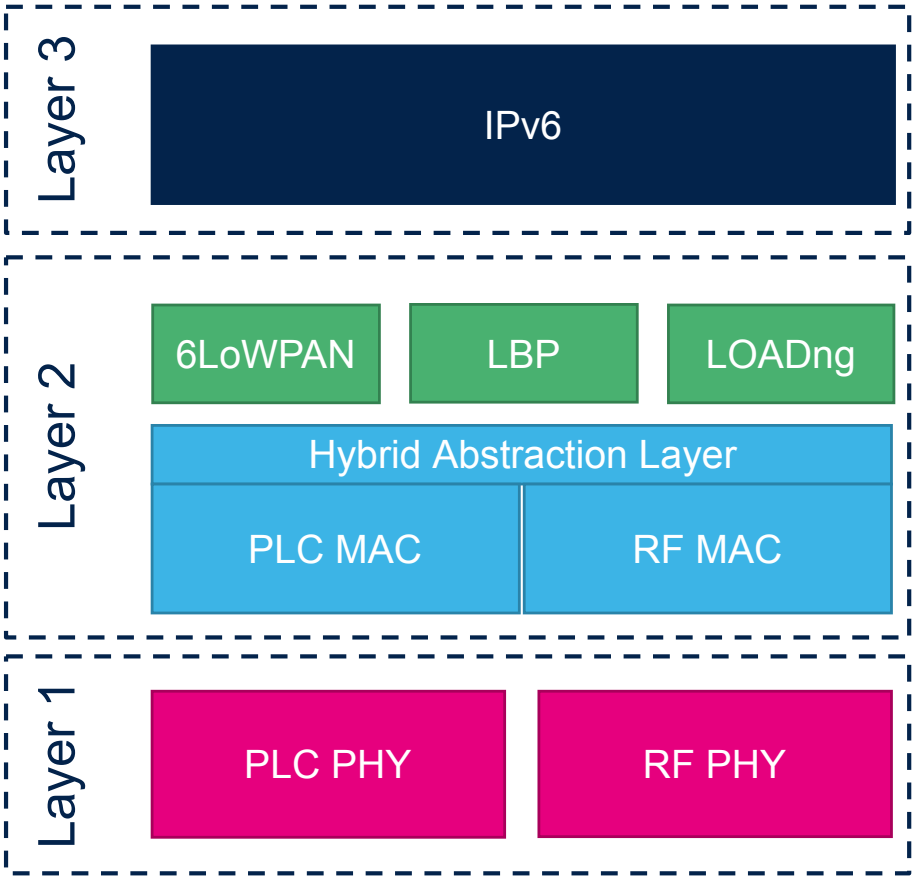
In many network conditions, none of the communication technologies can consistently achieve connectivity >99% on their own. The hybrid solution maximizes coverage and connectivity, avoiding the high cost that would be associated to deployment of a different solution to achieve the remaining 1% connectivity. The G3-Hybrid profile can provide a more efficient and cost-effective solution for smart grids, smart cities and industrial applications.

The G3-Hybrid profile is:

- fully compatible and interoperable with existing G3-PLC implementations, so it is possible to mix hybrid and non-hybrid nodes in the same network;
- available for all PLC bandplans and supporting a wide range of non-licensed Sub-GHz bands worldwide.

1.2 G3-Hybrid protocol basics

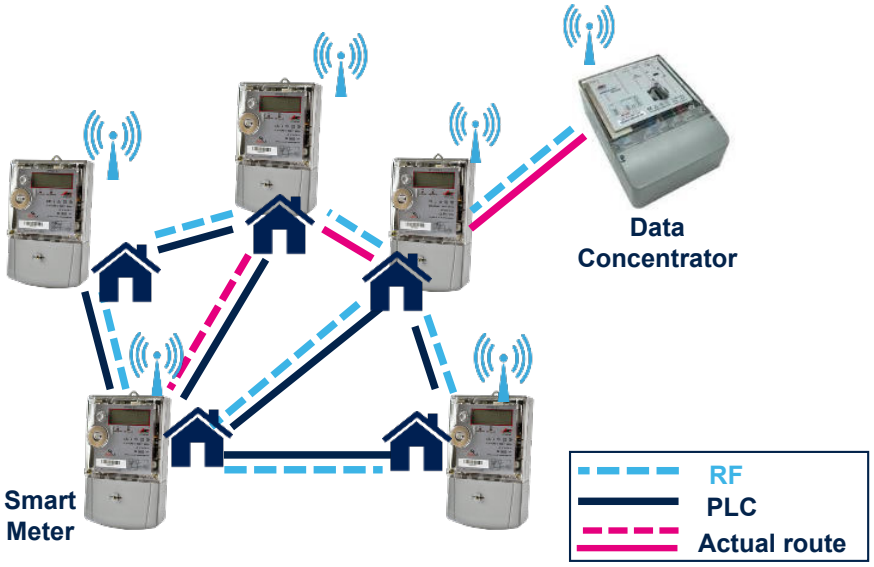
Figure 1. G3-Hybrid protocol stack



The G3-Hybrid protocol stack (see Figure 1) is based on the well-known G3-PLC protocol standard. To get the best from the two communication technologies involved, Narrow-Band PLC and Sub-GHz RF, the existing MAC layer has been replicated (and adapted) on top of the RF PHY, and a Hybrid Abstraction Layer has been developed on top of the two MAC layers, to guarantee seamless integration of the two media in one single managed network.

The principle of operation is quite simple: each node in a network having hybrid connectivity decides which medium to use to reach better performances and coverage, realizing a hop-by-hop automatic selection which is totally transparent to the end user. At the same time, the high flexibility of the implementation allows to mix hybrid nodes with PLC-only and/or RF-only nodes in a single network.

Figure 2. Hybrid PLC/RF network example



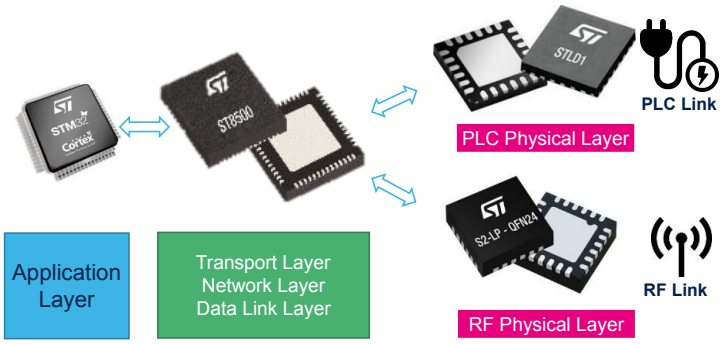
Since the software package described in this document is based on G3-Hybrid and works above MAC layer, given that the PLC/RF MAC layers are masked by the Hybrid Abstraction Layer, the G3-Hybrid protocol can be simply referred to as "G3" for conciseness.

1.3 Supported hardware and evaluation boards

The ST's G3 solution is based on three main devices (ST8500, STLD1 and S2-LP) building together the G3 modem chipset, with an STM32 microcontroller for the customer application firmware.

- The ST8500 programmable power line communication modem system-on-chip is at the heart of the system, implementing the full G3 stack, except the RF PHY layer.
- The S2-LP ultra-low power, high performance, Sub-GHz transceiver implements the PHY layer of the RF medium and the Sub-GHz RF radio.
- The STLD1 is the Line Driver for the PLC medium.

Figure 3. **ST chipset implementing the G3 protocol stack**



The ST's G3 implementation can run on several evaluation and development kits. Listed below are the ones where the STSW-ST8500GH-2 STM32 firmware described in this document can be used without adaptation. **Although the EVALKITST8500-1 and the EVLKST8500GH868/EVLKST8500GH915 kits can run the same protocol stack in PLC only or hybrid mode, the STM32 code provided in this package cannot be used without hardware-specific porting.**

Table 1. **ST8500 evaluation kits supported by STSW-ST8500GH-2**

Part number	Hybrid support	PLC connectivity	RF connectivity
EVLKST8500GH-2	YES	0-500 kHz	860-940 MHz

2 The G3 software package

The STSW-ST8500GH-2 contains a software package for the STM32G070RB, STM32G474RE, and STM32L476RG microcontrollers based on the STM32CubeMX V6.9.2 and the STM32CubeIDE V1.13.2.

Each microcontroller has a dedicated project related to its evaluation board (NUCLEO-G070RB, NUCLEO-G474RE or NUCLEO-L476RG). Since most of the source code is shared between the three different projects (being platform-independent) a folder named *\NUCLEO-Common* is linked to each project.

The structure of the package is described hereafter (refer to the specific release note to check if there are changes in your version).

- *\Documents*
- *\NUCLEO-G070RB*
  - *\.settings*
  - *\Core*
  - *\Drivers*
  - *\FATFS*
  - *\Middlewares*
  - *.cproject*
  - *.mxproject*
  - *.project*
  - *NUCLEO-G070RB.ioc*
  - *STM32G070RBTX_FLASH.ld*
- *\NUCLEO-G474RE* (organized in the same way as NUCLEO-G070RB)
- *\NUCLEO-L476RG* (organized in the same way as NUCLEO-G070RB)
- *\NUCLEO-Common*
  - *\Crypto*
  - *\G3_Applications*
  - *\Inc*
  - *\Modules*
  - *\Src*
  - *\Third_Party*
  - *\User_Applications*
- *versioning.py*
- *set_version.py*

Each microcontroller project includes the following directories:

- The*\Core* folder contains the source code related to the main program for the given microcontroller generated by STM32CubeMX, including *main.c*, and the *\Startup* folder (which has the assembly files responsible for the microcontroller preliminary startup operations such as stack pointer setup, filling the BSS section with zeros, system initialization, and calling the main function).
- The *\Drivers* folder contains CMSIS and STM32 HAL (Hardware Abstraction Layer) files for the selected microcontroller platform.
- The *\FATFS* folder contains the source code related to the FAT application and the User Disk IO driver.
- The *\Middlewares* folder contains the FreeRTOS specific source files, the FAT file system, the Modbus library, and the Disk IO library for the SD card.
- The file *NUCLEO_XYYYZZ.ioc* is the STM32CubeMX project file. You can use it to modify and/or regenerate the project (to modify the toolchain, the pin assignment, the target STM32, etc.) within the constraints of the STM32CubeMX environment.
- The file *STM32XYYYZZWW_FLASH.ld* is the linker script file. It lists and describes all memory sections allocated inside the microcontroller (FLASH and RAM).

The shared NUCLEO-Common directory includes the following sub-directories:

- The *Crypto* folder contains the cryptographic library used to compute the AES-128 functions required for the EAP-PSK protocol.
- The *G3_Applications* folder contains all the protocol specific firmware, comprising of the API and the G3 applications.
- The *Inc* and *Src* folders contain source code files related to settings, software version, and FreeRTOS configuration.
- The *Modules* folder contains the source code for all the modules used in the application.
  - The *Debug_Print* module is used for printing log messages for debug purposes.
  - The *Host_Uart* module is used for handling the Host UART.
  - The *Image_Download* module is used to handle the PE/RTE image downloads at startup in "Boot From UART" mode.
  - The *Image_Management* module is used to manage the ST8500 images stored inside the SPI Flash memory.
  - The *Pin_Management* module is used to read and write GPIO pins.
  - The *SFlash_Driver* module is used to handle the SPI Flash memory.
  - The *User_Uart* module is used for handling the User UART.
  - The *Utility* module is used for internal functionalities present in most parts of the applications.
- The *Third_Party* folder contains the Modbus library and the user Disk IO SPI driver implementation for the SD card.
- The *User_Applications* folder contains the files related to an example of User Datagram Protocol (UDP) user application: a UDP sender/receiver implementation with its FreeRTOS task and a serial terminal with several features.

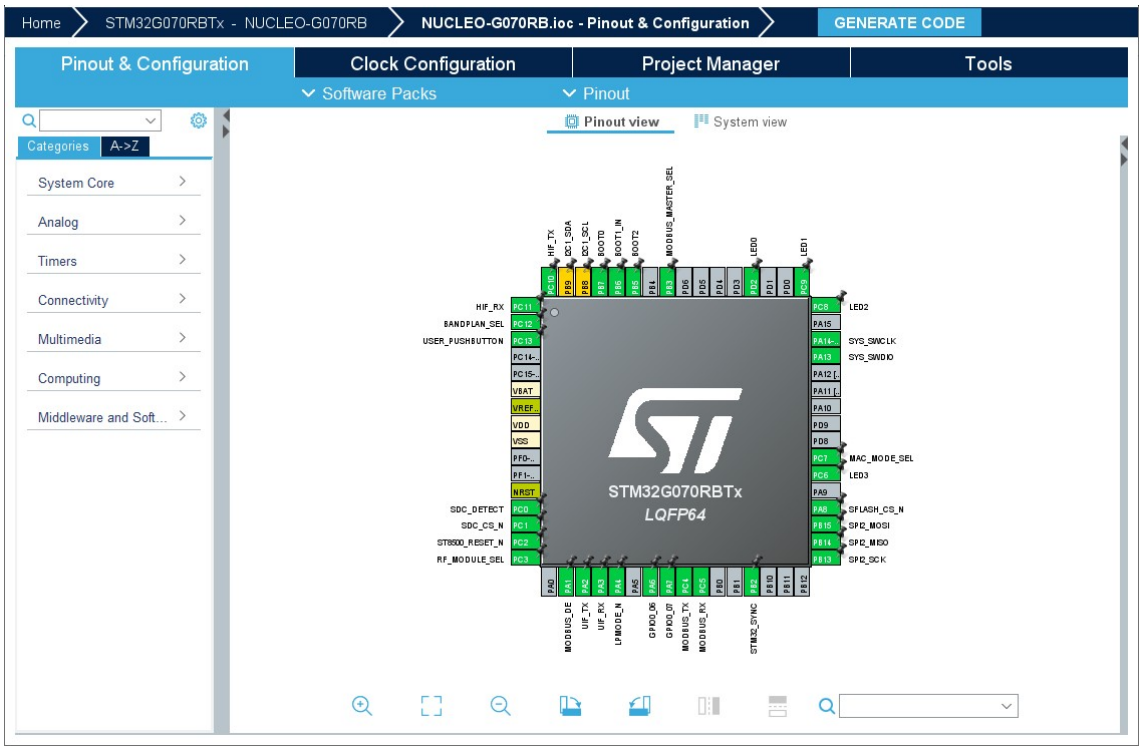
On the same level of the three microcontroller projects and the *NUCLEO-Common* folder, the *Documents* folder includes a quick start guide on the user application and the release note of the package. Check the release note to see the changes that have occurred since the last official release.

3 STM32CubeMX project

3.1 Project description

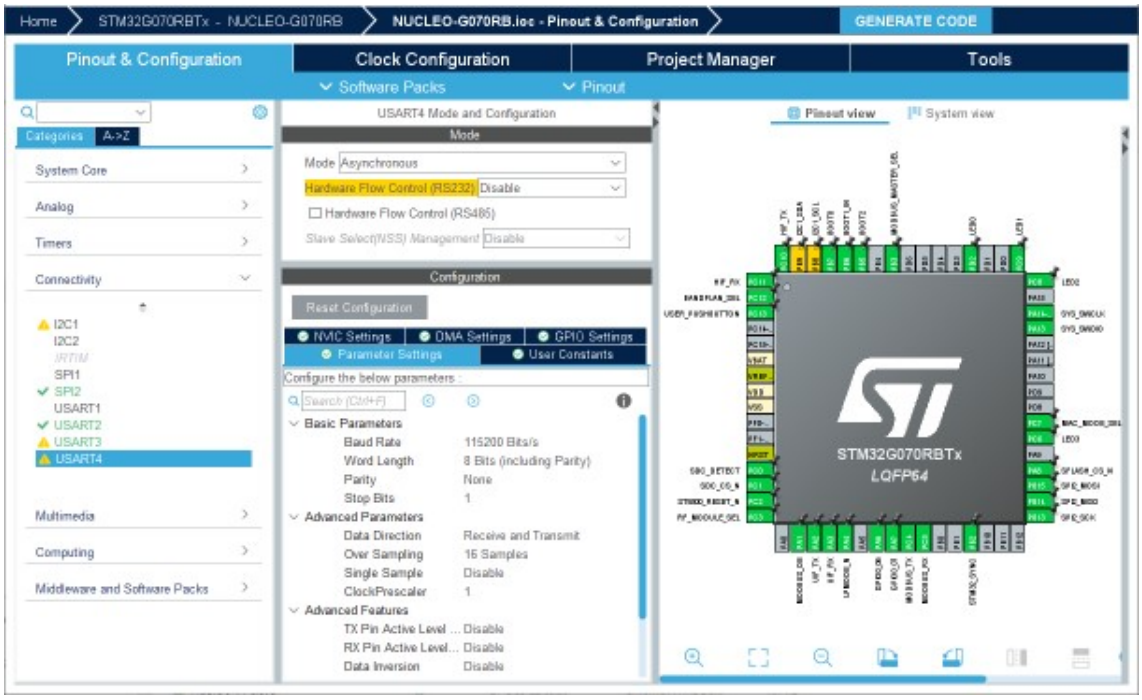
Each of the three STM32 projects has its own related STM32CubeMX project. When first opened, the STM32CubeMX project shows, on the left side, the list of categories the settings refer to, while on the right panel the default view is on the STM32 pinout. By moving on the active pins (the ones not grayed), it is possible to see how they are configured and, when clicking, the alternative settings are shown.

Figure 4. STM32CubeMX project initial view (NUCLEO-G070RB.ioc)



To see/change the configuration of each peripheral, a side panel is displayed by clicking on a specific peripheral from the left side list, with related operating mode and configuration parameters.

Figure 5. **STM32CubeMX project peripheral configuration view**

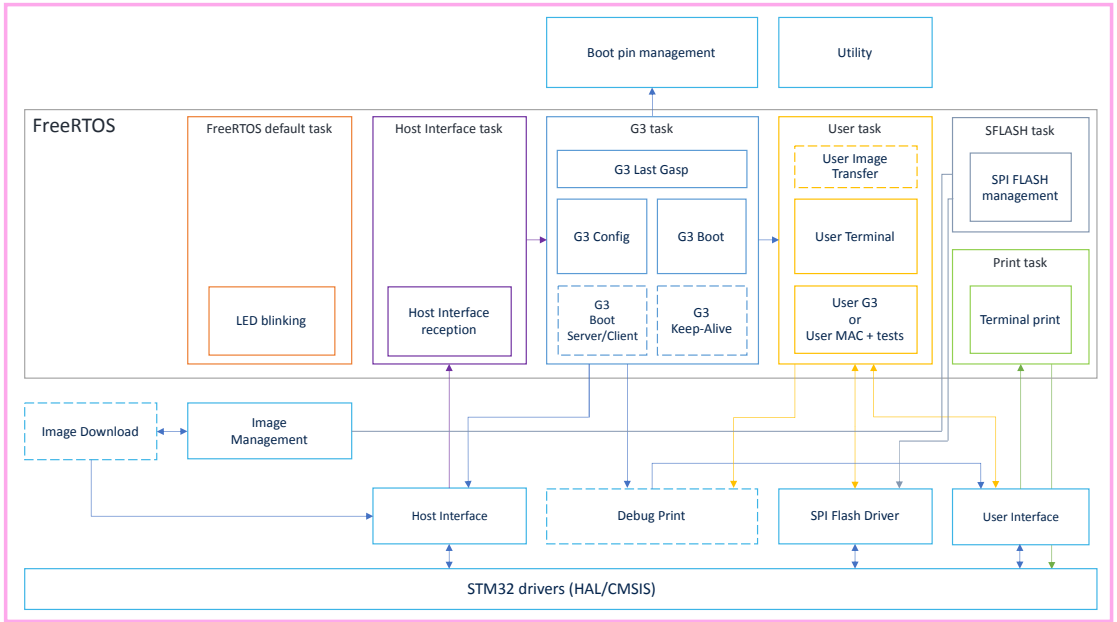


Any information about the STM32CubeMX ecosystem can be found on the STMicroelectonics website.

3.2 FreeRTOS subsystem

The FreeRTOS middleware is generated automatically by STM32CubeMX. The overall software architecture is shown in Figure 6 .

Figure 6. **STSW-ST8500GH-2 project software architecture**



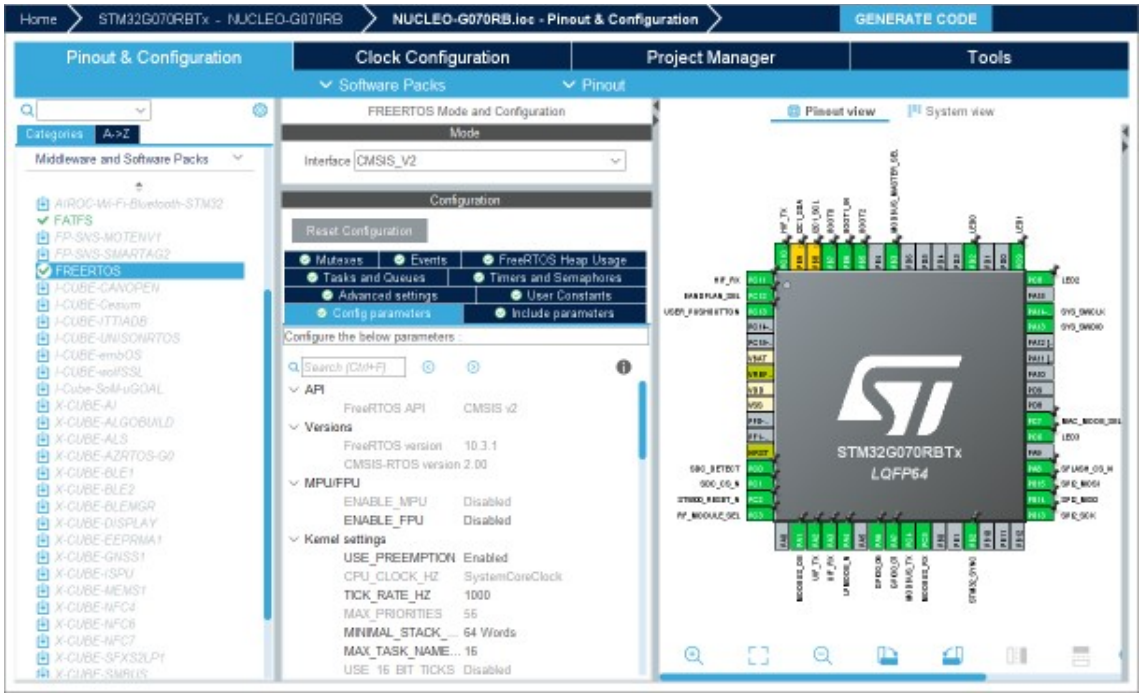
The User Application can change its functions depending on the platform running mode. In case BOOT or ADP mod is running, the User task executes UDP application with several features. In case MAC mode is running, the User task executes a testing application that verifies the HW functionalities and lets the user "ping" other devices at MAC level.

The STM32CubeMX generates the STM32 HAL/CMSIS drivers and configures the middleware (in this case, FreeRTOS and FAT File System).

The FreeRTOS configuration parameters can be modified in the FreeRTOS tab under the Middleware section inside STM32CubeMX (see Figure 7).

For these projects, STM32Cube_FW_G0_V1.6.1, STM32Cube FW_G4 V1.5.1 and STM32Cube FW_L4 V1.18.0 have been used as FW packages.

Figure 7. STM32CubeMX FreeRTOS configuration panel



The STSW-ST8500GH-2 software project makes use of the CMSIS_V2 interface and includes several OS resources. Inside *rtos_config.c* all main OS resources are allocated and the *FreeRTOS_DefaultInit* initializes each one of them. The FreeRTOS configuration is done in such a way, instead of using the configuration utility of STM32CubeMX, in order to simplify modifications and to avoid the replication of the FreeRTOS configuration for each STM32CubeMX project (".ioc" file), having a single shared configuration instead. All allocations and initializations use simplified custom macros (defined in the upper part of *rtos_config.c*). The usage of these macros is explained in details in the comments inside the source code.

The tasks generated inside *rtos_config.c* are the followings:

- The FreeRTOS default task that blinks a LED to signal it is working (stops in case of fault).
- The Host Interface task that handles the reception and validation of messages from the Host Interface. A complete description of the *host_if_task* is given in Section 4.4.1 .
- The Print task that handles the low-level printing operation to serial terminal. A complete description of the *print_task* is given in Section 4.4.2 .
- The SFlash task that handles the operations on the SPI Flash memory connected to the STM32. A complete description of the *sflash_task* is given in Section 4.5 .
- The G3 task that handles the G3 protocol. A complete description of the *g3_task* is given in Section 5 .
- The User task that handles an UDP example application or a MAC testing application. A complete description of the *user_task* is given in Section 6 .

There are four specific message queues, *host_if_queue*, *g3_queue*, *user_queue*, *sflash_queue*, respectively for the message reception of the Host Interface task, the G3 task, the User task and the SFlash task.

In addition, the following resources are used by the OS:

- *eventSync*: event flag used to synchronize all application tasks after their initialization. It is useful to execute all initialization functions before the main task routines.
- *hostStreamBuffer*: stream buffer used to transfer data to print to the Print task.
- *mutexPrint*: mutex used to make sure that only one task sends data to the Print task at a time.
- *semHostIfTxComplete*: semaphore used to keep the G3 task blocked while the last G3 request message is being transmitted to the Host Interface through DMA.
- *semUserIfTxComplete*: semaphore used to keep the Print task blocked while the ASCII data is being transmitted to the serial terminal through DMA.
- *semStartPrint*: semaphore used to unblock the Print task whenever new ASCII data is available to be printed.
- *semConfirmation*: semaphore used to make sure that no new request is sent to the ST8500 before a confirmation to the previous one is received.
- *semSPI*: semaphore used to block the User task during the SPI transaction with the SPI Flash memory.
- *bootTimer*: timer used by the boot application and/or the boot server application to trigger timed events, such as temporized actions, timeouts and re-connections.
- *kaTimer*: timer used by the Keep-Alive application to trigger timed events, such as pings and timeouts.
- *serverTimer*: timer used by the boot server application to add delays between the Re-keying procedure (the delays can be deactivated by setting *ENABLE_REKEYING_DELAYS* to 0 inside *settings.h*).
- *commTimer*: timer used by the User G3 or the User MAC to trigger a timeout when a confirm message or an indication message is not received within the expected amount of time.
- *userTimeoutTimer*: timer used by the User Terminal to trigger user-defined timeouts.
- *transferTimer*: timer used by the User Image Transfer to trigger timeout events during the image transfer.

4 Application modules

4.1 Introduction

The application modules are software components that are used by the main program and/or the OS tasks to implement specific functions. They include device drivers and serial interfaces but also utility functions that assure the inter-task communication and the memory allocation.

The most relevant application modules described by this chapter are:

- The Memory pools;
- The Task communication mechanism;
- The UART interface modules, divided in:
  - Host Interface;
  - User Interface;
- The SPI Flash driver;
- The image downloader.

4.2 Memory pools

The project uses custom memory pools when it is necessary to allocate a considerable amount of data. These custom memory pools are static shared buffers that can be reused and they are implemented inside *mem_pool.c*.

Three types of memory pools with different sizes are defined to occupy the least amount of memory for each message. If necessary, it is possible to adjust the size and the number of memory pools of each type by editing the relative values inside *mem_pool.h*. The initialization of the memory pools, performed by the *mem_pool_init*, is called inside the *main* function.

The following interface functions are used to allocate and deallocate the memory pools:

- *MEMPOOL_MALLOC*, to allocate a memory pool of a given size (the smallest compatible memory pool is chosen).
- *MEMPOOL_FREE*, to deallocate the memory pool.

These functions rely respectively on the *mem_pool_alloc* function and on the *mem_pool_free* function. In addition, a function named *mem_pool_check* is available to check if a pointer points to a memory pool or not.

To avoid memory leakages, it is of utmost importance to always call *MEMPOOL_FREE* after each *MEMPOOL_MALLOC* call.

In case memory pools need to be debugged due to problems involving them, the *MEMPOOL_DEBUG* macro inside *settings.h* can be adjusted to one of the available debug levels (explained in the comments above the macro). For high values of *MEMPOOL_DEBUG*, information about the number of allocations/deallocations, the number of currently used pools, and the record of allocated pools of each type can be examined inside the *mem_pool* structure with the help of the debugger.

4.3 Task communication mechanism

Task communication between the Host Interface task, the G3 task, the User task and the SFlash task is achieved by four queues named *host_if_queue*, *g3_queue* , *user_queue* and *sflash_queue*.

Each message sent to one of these queues has a internal message ID that determinates its usage:

- *HIF_TX_MSG*: used only to send G3 messages from the G3 task through the Host Interface.
- *HIF_RX_MSG*: used only to transfer messages from the Host Interface reception handler to the Host Interface task, in order to parse and verify them inside the task.
- *G3_RX_MSG*: used for G3 messages to be handled by the G3 task or the User task, if forwarded. Their reception unblocks the G3 task/User task.
- *BOOT_SRV_MSG*: used to unblock the G3 task and execute the Boot Server request handler and the Boot Server FSM (used only in case the Boot Server module is enabled).
- *BOOT_REKEY_MSG*: used to unblock the G3 task and trigger the Re-keying FSM of the Boot Server (used only in case the Boot Server module is enabled).
- *BOOT_CLT_MSG*: used to unblock the G3 task and execute the Boot Client request handler and the Boot Client FSM (used only in case the Boot Client module is enabled).
- *KA_MSG*: used to unblock the G3 task and execute the Keep-Alive FSM when a Keep-Alive related event is triggered.
- *LAST_GASP_MSG*: used to unblock the G3 task and activate the Last Gasp mode.
- *USER_MSG*: used only to unblock the User task and execute all of its FSMs.
- *SFLASH_MSG* used to unblock the SFlash task and execute an operation on the SPI Flash memory.

The function used to exchange messages between tasks are called through four macros:

- *RTOS_PUT_MSG*: Calls *taskCommPut*, passing message ID and data payload as parameters.
- *RTOS_GET_MSG*: Calls *taskCommGet* with infinite timeout, blocking a task indefinitely until a message is received through the queue.
- *RTOS_GET_MSG_TIMEOUT*: Calls *taskCommGet* with finite timeout, blocking a task until a message is received through the queue or the timeout is reached.
- *RTOS_MSG_AVAILABLE*: Returns true if at least one message is available to extract in a given queue.

These functions rely on the CMSIS_V2 queuing mechanism through the functions:

- *osMessageQueueGet*
- *osMessageQueuePut*
- *osMessageQueueGetCount*

The queue message format is defined by the task message (*task_msg_t*) structure:

- *message_type* is the message ID that describes the purpose of the message;
- *data* is the pointer to the data associated with the message.

Task messages are used to unblock tasks and share data between them. The main kind of shared data are G3 messages of the G3 protocol. A G3 message has a slightly different format, defined by the *g3_msg_t* structure:

- *command_id* is the command ID hat describes the nature of the G3 message;
- *size* is the size of the payload data (in bytes) associated with the G3 message.
- *payload* is the pointer to the payload data associated with the G3 message.

With the *g3_copy_and_send_message* function (inside *g3_comm.c*) it is possible to create and encapsulate a G3 message (*g3_msg_t*) as the payload of a task message and send the task message to the *g3_queue*. This function uses shared buffers that are statically allocated through memory pools (see Section 4.2 Memory pools).

In particular, the *g3_copy_and_send_message* function uses one memory pool for the message context (*g3_msg_t* structure) and one for the payload (*payload* field of the *g3_msg_t* structure). These two pools are freed once the message has been completely processed by the *g3_discard_message* function.

Since the *g3_copy_and_send_message* function allocates a pool for the payload and fully copies it, a good practise is to avoid calling *g3_copy_and_send_message* and call *g3_send_message* instead when the message payload is already allocated inside a memory pool and there is no need to copy it. The *g3_send_message* function, indeed, allocates only one memory pool for the message context, without deep-copying the payload.

For instance, the Host Interface task, after receiving a task message containing the pointer to a reception structure (*host_if_msg_rx_t*) relative to an incoming message, checks its integrity and then calls *g3_send_message* to forward the message exempted of the header. The Host Interface reception handler already allocates a memory pool for the payload and the CRC field during the reception of the message, and thus it is not necessary to allocate another one. In *g3_send_message*, the address of the already allocated payload memory pool is re-assigned to the buffer variable inside the G3 message.

The G3 messages sent to the *g3_queue* with *g3_copy_and_send_message* and *g3_send_message* are processed by the G3 task and eventually forwarded to the User task.

When forwarded, the User task must call *g3_discard_message* for the exhausted G3 message after the completion of its use, in order to free the memory pools allocated for the G3 message.

When not forwarded, the G3 task must call itself the *g3_discard_message* when the G3 message is no longer needed.

4.4 UART interface modules

The STSW-ST8500GH-2 software project includes two modules related to UART communication between the ST8500, the STM32 and the PC:

- Host Interface
- User Interface

This chapter describes the implementation of these modules in detail.

Note: The Host UART pass-through module present in the old STSW-ST8500GH software package has been removed in this release because the motherboard of the EVLKST8500GH-2 evaluation kit features an hardware pass-through via a specific USB connector, eliminating the need of a software pass-through.

4.4.1 Host Interface

The Host Interface module enables the control of the Host UART, the UART interface of the ST8500, implementing commands related to the G3-Hybrid protocol stack.

The Host Interface module basically takes care of the low-level UART messaging between the STM32 and the ST8500; it is implemented in the file *host_if.c*.

Besides the initialization functions, the core of the operations is managed by three functions:

- *host_if_rx_handler*, called inside the Host UART ISR when the current reception is complete (*HAL_UART_RxCpltCallback* inside *callbacks.c*), receives the messages coming from the Host UART and then queues it for the Host Interface task (*hif_task*) when their reception is complete. Each message is received in three steps, handled by a local FSM:
  1. Reception of the first SYNC byte;
  2. Reception of the remaining header (second SYNC byte, command ID, message length...), plus the Error Code (EC);
  3. Reception of the payload and the CRC16-XMODEM.

  To avoid deep copy operations in the ISR, which could lead to a loss of data at higher baud rates, or with low speed STM32 devices, this function passes only the pointer to the reception structure of the message, allocated inside a memory pool, without parsing or copying anything.
- *host_if_tx_handler*, called inside the Host UART ISR when the transmission is complete (*HAL_UART_TxCpltCallback* inside *callbacks.c*), frees the memory pool allocated as transmission buffer to allow the transmission of a new message through the Host UART.
- *host_if_send_message*, called from the G3 task main loop, prepares and initiates the dispatch of a message through the Host UART.

The reception of the Host Interface messages is completed inside the *host_if_task_exec*, implemented inside *host_if_task.c*. This task receives the messages coming from the *host_if_rx_handler*, parses them by calling *host_if_parse_message*, verifying the SYNC field and the CRC16-XMODEM, and then queues them for the G3 task, releasing the *semConfirmation* semaphore when a confirm type message is recognized.

4.4.2 User Interface

The User Interface module handles the User UART, the serial port dedicated to the interaction between the device connected through a USB cable (a PC is assumed, for instance) and the User task. The User UART is connected to the PC through the ST-LINK USB on the Nucleo board. This module, implemented in the file *user_if.c,* receives the serial input from the PC directly when the User UART reception ISR is called, while the output is sent to a dedicated FreeRTOS low priority task, that is the Print task.

Besides the initialization and service functions, the core of the operations is managed by five functions:

- *user_if_get_input*, called inside the User task, extracts the input data from the dedicated buffer to make it available for the operations of the task;

- *user_if_printf*, formats data into a UTF-8 string like the well-known "printf" function. It is possible to include the current timestamp (*ENABLE_TIMESTAMP* must be set to 1), change the text color (*ENABLE_COLORS* must be set to 1) and add a label related to the string. After the string is encoded inside a buffer, the *user_if_low_level_print* function is called to transmit the string to the stream buffer (*hostStreamBuffer*), when FreeRTOS is running. The *mutexPrint* mutex is used to make sure that only one task at a time sends data to the stream buffer. If FreeRTOS is not running, it transmits data directly to the User UART, in polling;
- *user_if_print_raw*, encodes an unformatted UTF-8 string inside a buffer and calls the *user_if_low_level_print* function;
- *user_if_rx_handler*, called inside the User UART ISR when the reception is complete (*HAL_UART_RxCpltCallback* inside *callbacks.c*), requests data from the User UART Interface to put it in a dedicated buffer (*user_if_fifo_rx*). In order to unblock the user task and let it process the incoming input, this function also queues a message to the user task;
- *user_if_tx_handler*, called inside the User UART ISR when the transmission is complete (*HAL_UART_TxCpltCallback* inside *callbacks.c*), releases a semaphore to allow the transmission of new data through the User UART.

A specific task, named Print task, has been defined to handle the printing operation to the serial terminal (through the User UART). When unlocked, the Print task extracts the data from the stream buffer (*hostStreamBuffer*), transmits it through the User UART with the DMA and then remains blocked until all data is transmitted. The Print task repeats these instructions until there is no more data to print.

The Print task is regulated by two FreeRTOS semaphores:

- *semStartPrint*, used to block the Print task whenever no data to print is present in the stream buffer. It is released at the end of the *user_if_low_level_print* function (when FreeRTOS is running) when data to print is available;
- *semUserIfTxComplete*, used to keep the task blocked while the DMA is transmitting the data.

The Print task has a lower priority (*osPriorityLow*), in order not to execute when more critical tasks are running.

4.5 SPI Flash Driver

The SPI Flash Driver is the software component responsible for the SPI communication with the SPI Flash connected to the STM32. It is implemented inside *sflash_driver.c* and it includes the following functions:

- *SFLASH_GetDeviceId*: reads the device ID information from the SPI Flash memory chip;
- *SFLASH_Read*: reads the SPI Flash memory, storing the read data inside a buffer;
- *SFLASH_Write*: writes the SPI Flash memory, taking the data to write from a buffer;
- *SFLASH_Erase*: erases one or more sectors of the SPI Flash memory; and
- *SFLASH_BulkErase*: erases all sectors of the SPI Flash memory.

All of the driver functions are wrapped by the *sflash_command* function, implemented inside *sflash.c*, which is the only function that needs to be called whenever it is necessary to perform an operation on the SPI Flash memory. This wrapper function behaves differently depending on FreeRTOS kernel status:

- When FreeRTOS is not running, the *sflash_command* calls directly the SPI Flash driver functions, with a blocking effect.
- When FreeRTOS is running, the *sflash_command* sends a *SFLASH_MSG* task message to the SFlash task, with a blocking effect only in case of a "read" or "get ID" operation.

The SFlash task, implemented inside *sflash_task.c*, upon receiving a *SFLASH_MSG* task message, performs the requested operation on the SPI Flash memory. In this way, any task that requests the operation avoids being blocked in case of an erase or write operation, since the SFlash task performs them in the background.

Since a mechanism involving a semaphore is used to unblock the requesting task when a SPI Flash read operation is completed, it is mandatory to set the *SFLASH_SEM_NUM* macro (inside *sflash_task.c*) to the total number of tasks that may request read operations on the SPI Flash memory.

4.6 Image downloader

When activated through a specific switch on the board, this module downloads both the RTE and the PE images from the STM32 SPI Flash memory to the RAM of the ST8500. This is archived at start-up by interacting with the first level bootloader integrated in the ST8500, before initializing the operative system (FreeRTOS). This is handled entirely by the *downloadImageToST8500* function, that, in order:

1. Saves the current baud rate of the Host UART.
2. Validates the ST8500 images in the STM32 SPI Flash, if they still need to be validated.
3. Changes the baud rate of the Host UART to the default baud rate value (9600 bps) of the ST8500's bootloader.
4. Searches for the RTE and PE images, looking for primary ones.
5. If at least one of the two images is missing, searches for the missing images, looking for secondary ones.
6. Sets the communication baud rate (and the Host UART baud rate) to the maximum value (921600 bps).
7. Downloads the RTE image to the ST8500.
8. Downloads the PE image to the ST8500.
9. Restores the original baud rate of the Host UART.

To download the images from the STM32 SPI Flash, it is necessary to store them on the SPI Flash in the first place. This can be done with STM32CubeProgrammer, following the procedure described in Section 8 .

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_DOWNLOAD macro to 1 or 0 inside settings.h.

5 G3 applications

5.1 Introduction

The G3 task implements the following modules:

- G3 Configuration module
- G3 Boot module
- G3 Boot Server module (optional, only on coordinator implementation)
- G3 Boot Client module (optional, only on device implementation)
- G3 Keep-Alive module (optional)
- G3 Last Gasp module (optional)

This section describes the G3 task and gives some hints on how it could be adapted to fit specific cases.

5.2 The G3 Configuration module

The G3 Hybrid PLC & RF node can be configured on several parameters: the running mode of the mode (*IPV6_BOOT*, *IPV6_ADP* or *MAC*), the role of the node in the network (Device or Coordinator), the PLC band (CENELEC A, CENELEC B, FCC), the RF frequency and transmission mode, and so on. Such parameters must be configured at the start-up of the node to ensure that it properly behaves in the network.

In case an SPI Flash memory is connected to the ST8500, all the configurations are written in this Non-Volatile Memory (NVM) at specific sectors; the NVM content is retrieved and the ST8500 G3 configuration restored at each power-on/HW reset. The NVM management is almost transparent to the user but, to avoid conflicts between the ST8500 retrieving parameters from the NVM and the Host trying to configure the modem, after each power-on/hardware reset event, the host application waits until the ST8500 notifies the end of NVM reconfiguration by sending a hardware reset confirm through the Host Interface.

The G3 Configuration module takes care of customizing the configuration of the node in two steps:

1. G3 attributes table initialization
2. G3 platform configuration

5.2.1 Attributes table initialization

The list of parameters to be configured is stored into a static type *G3_LIB_PIB_t* array, the attribute table; the attributes are inserted in it with the function *g3_attrib_tbl_add*, defined in the file *g3_app_attrib_tbl.c*, that is called multiple times inside *g3_app_attrib_tbl_init*, the attribute table initialization function. It is possible to add attributes only for a specific mode, only for a specific role (only for PAN Coordinator or only for PAN Device) or for both PAN Coordinator and PAN Device.

In case the node is working in MAC mode, only the *MAC_PAN_ID* attribute is mandatory.

In IPV6-BOOT/IPV6-ADP mode, in case the node is configured as a PAN Coordinator, the only two mandatory parameters to be configured are the *MAC_PAN_ID* and the *MAC_SHORTADDRESS_ID*.

Instead, if the node is configured as PAN Device, the only mandatory attribute to be set is *ADP_EAPPSKKEY_ID* (as the Short Address is configured during the Bootstrap procedure).

Other attributes that are not mandatory but still suggested to set are:

- *ADP_COORDSHORTADDRESS_ID* set to 0 (only for PAN Device);
- *MAC_KEYTABLE_ID* set to the default GMK value at index 0 (only for PAN Coordinator);
- *ADP_ACTIVEKEYINDEX_ID* set to 0 (only for PAN Coordinator).
- *G3_LIB_PEEVENTINDICATION_ID* set to its default value.

The user can easily extend this configuration, if necessary, by adding rows to the two tables with the *g3_attrib_tbl_add* function (the *MAX_ATTRIBUTE_NUMBER* macro must be adjusted to the maximum number of attributes to set). During the node start-up, this table is prepared to be used by the G3 platform configuration described below.

5.2.2 G3 platform configuration

The G3-Hybrid PLC & RF stack must be properly initialized and configured before starting the operations. The G3 Configuration module performs this task with a proper Configuration FSM (executed by calling the *g3_app_conf* function) started before the *for* loop cycle of the G3 task, with the *g3_app_conf_start* function, by sending a *G3LIB-SWRESET.Request* to the ST8500.

The software reset request accepts three parameters:

- The PLC operating band (CENELEC-A, CENELEC-B or FCC)
- The Device type (PAN Coordinator or PAN Device)
- The Operating mode (PHY, MAC, ADP, BOOT, IPV6-ADP or IPV6-BOOT)

The selection of the band is done by reading the status of a GPIO (*BANDPLAN_SEL*) at start-up, making it possible to work in CENELEC-A or FCC. The G3 task supports all the PLC operating bandplans (it is possible to work even in CENELEC-B by modifying the source code) and device types, but it requires the modem to be set in IPV6-BOOT mode to properly operate. The modem is set in IPV6-ADP mode only in case of PAN Coordinator with Boot Server at application level (see Section 5.3.1.1), since the Boot Server module replaces the Boot layer inside the ST8500, interfacing itself with the ADP layer. It is possible to work in MAC mode by setting a specific level on another configuration GPIO (*MAC_MODE_SEL*). In that case, the configuration module and the User task work in different ways.

Upon reception of the *G3LIB-SWRESET.Confirm* from the Host Interface, the first attribute in the attribute table is extracted and set on the platform with a *G3LIB-SET.Request*. Afterwards, each successive attribute in the table is set as soon as the *G3LIB-SET.Confirm* is received for the previous one. When all attributes in the table have been set, a *HOSTIF-DBGTOOL.Request* is sent to detect the S2LP module presence. In case the S2LP module is detected (*RfConf* field equal to “1”), the configuration of the RF module is performed by sending a *HOSTIF-RFCONFIGSET.Request*. This simply feeds the structure containing the RF configuration parameters using hard-coded values, depending on the selected RF module (X-NUCLEO-S2**868**A2 or EVALS2**915**A1/A2). The selection of the RF module is done by reading the status of a GPIO (*RF_MODULE_SEL*) at start-up. In case the S2LP is absent, this step is skipped.

Note that the frequency range for the RF link is chosen depending on the USE_STANDARD_ETSI_RF and the USE_STANDARD_FCC_RF values. These macros are mutually exclusive, so if one is set to 1, the other one must be set to 0.

- If USE_STANDARD_ETSI_RF is set to 1, the base frequency is set to 863,1 MHz and the power gain is set to 17 (in order to obtain an output power of +14 dBm)
- If USE_STANDARD_FCC_RF is set to 1, the base frequency is set to 915 MHz and the power gain is set to 30 (in order to obtain an output power of +27 dBm)

After the configuration of the S2LP module, the platform configuration for the PAN Device (and also for the MAC mode) is complete. The PAN Coordinator implementation, instead, includes two additional steps to stop the Boot Server (the default start uses a fixed PAN ID) and then start it again with a custom PAN ID value (the short address has to be 0).

The reception of the *G3BOOT-SRV-START.Confirm*, completes the platform configuration for the PAN Coordinator, rendering the Configuration FSM inactive.

5.3 The G3 Boot module

The G3 Boot module makes use of the G3-Hybrid PLC & RF bootstrap implementation to manage the registration of each PAN Device in the network to the PAN Coordinator. It is important to remember here that it is required to run a network having (only) one PAN Coordinator with 0 as short address and (at least) one PAN Device.

On the PAN Coordinator side, the Boot module can be extended by moving the Boot Server in the application. More details about this are given in Section 5.3.1.1 .

On the PAN Device side, the Boot module can be extended by moving the Boot Client in the application. More details about this are given in Section 5.3.2.1 .

5.3.1 Boot module - PAN Coordinator implementation

Each time a PAN Device asks to enter the network, independently from the fact that the selected agent is the PAN Coordinator or another PAN Device, the request is eventually forwarded to the ST8500 acting as PAN Coordinator, which generates a *G3BOOT-SRV-GETPSK.Indication* message on the ST8500 Host Interface to the G3 Boot module. The G3 Boot module gets from it the Pre-Shared Key (PSK) of the specific PAN Device and the short address the PAN Coordinator wants to assign to it.

The PAN Coordinator assigns the short address to the requesting PAN Devices either with static or dynamic allocation, using the associated PSK.

The static allocation is supported by the *g3_boot_access_table* (inside *g3_app_boot.c*), which is hard-coded in the PAN Coordinator G3 Boot module. There are two possible access mode:

- Black list mode (default): when a PAN Device whose Extended Address is listed in the access table tries to join the network, the connection is refused. Connection is accepted for PAN Devices whose Extended Address is not listed in the access table. The default PSK is assigned for all bootstrap procedures and short addresses are assigned in ascending order.
- White list mode: when a PAN Device whose Extended Address is listed in the access table tries to join the network, the connection is accepted and the corresponding Short Address and PSK are assigned. Connection is refused for PAN Devices whose Extended Address is not listed in the access table.

The access mode can be selected by setting the *SELECTED_LIST_MODE* macro inside *settings.h* either to *BLACK_LIST_MODE or WHITE_LIST_MODE*.

In a real implementation, since the list contains sensitive data such as the PSK of each node, the use of a secure element to store locally the table and/or the use of a secure connection towards an authentication server are recommended.

Once the bootstrap process is complete, a *G3BOOT-SRV-JOIN.Indication* is received on PAN Coordinator side and a new entry is added in the *connected_devices* array of *boot_server*, a structure shared with the Boot Server module (present even when the Boot Server module is disabled). The entry will remain until that PAN Device leaves the PAN or it is kicked out. The entry is a *boot_device_t* structure containing also information relative to the Keep-Alive module (see the Section 5.4.2 of the Keep-Alive module).

5.3.1.1 *The G3 Boot Server module*

On the PAN Coordinator side, it is possible to use the Boot layer embedded in the G3 library of the platform, running the ST8500 in *IPV6_BOOT* mode, or add a custom implementation of the Boot layer at application layer, running the ST8500 in *IPV6_ADP* mode. This can be chosen by setting *ENABLE_BOOT_SERVER_ON_HOST* inside *settings.h* to 1, for Boot layer on application, or to 0, for Boot layer on platform. The Boot layer of the PAN Coordinator is constituted by a Boot Server responsible for handling the bootstrap process of the PAN Devices, when they try to join the PAN, and other functions, such as updating the GMK when requested. If the Boot layer is at application level, the Boot Server module is enabled and all G3 Boot messages are sent to the G3 task instead of the Host Interface. This chapter describes the main functions of the Boot layer when implemented at application level (*ENABLE_BOOT_SERVER_ON_HOST* set to 1). The application Boot layer implementation is similar to the one of the platform Boot layer, so that from the outside both implementations behave almost in the same way (the application Boot layer is managed with the same messages used with the platform Boot layer) and there is no need to change the rest of the application when a Boot layer implementation is chosen instead of the other.

The implementation of the Boot Server is distributed between *g3_app_boot_srv.c* (Boot layer management), *g3_boot_srv_eap.c* (EAP-PSK protocol) and *g3_boot_srv_join_entry_tbl.c* (joining entry table management).

In addition to the initialization function, the Boot Server module main functions are *g3_app_boot_srv* and *g3_app_boot_srv_msg_handler*, that manage the ADP message exchange with the ST8500, and *g3_app_boot_srv_req_handler* that handles the Boot layer requests coming from the G3 task or the User task.

The Boot Server main role is to serve Boot requests coming from the applications and execute the bootstrap procedure when a join request is received from a PAN Device. The *boot_server* structure contains all data needed for the Boot Server activities, such as the connected device list, the current state/sub-state of the Boot Server FSM, the PAN ID and the short address, the active GMK and its index, and other variables.

The Boot Server starts receiving a *BOOT-SRV-START.Request* from the application. After that, the following sequence is followed:

1. The *bootTimer* is started to wait *BOOT_SERVER_START_WAIT_TIME* seconds.
2. When the wait is finished, the Boot Server sends an *ADP-DISCOVERY.Request*, with *BOOT_SERVER_DISCOVERY_TIME* as duration, to the Host Interface.
3. Upon receiving the *ADP-DISCOVERY.Confirm*, if no other PAN was detected, the Boot Server sends an *ADP-NETWORK-START.Request* with the chosen PAN ID to the Host Interface. If at least another PAN was detected, the Boot Server restarts by sending itself another *BOOT-SRV-START.Request*.
4. When the *ADP-NETWORK-START.Confirm* is received, the Boot Server goes in the active state and a *BOOT-SRV-START.Confirm*Confirm is sent back to the G3 task.

When the Boot Server is active, it is ready to handle bootstrap procedure, kick requests and re-keying requests at any time. The total necessary time to start the Boot Server is almost equal to *BOOT_SERVER_START_WAIT_TIME* plus *BOOT_SERVER_DISCOVERY_TIME* (see *g3_app_boot_constants.h*), in seconds.

The *BOOT_SERVER_DISCOVERY_TIME* macro is the time (in seconds) granted to make sure no beacons are received in response to the discovery request. This macro should be set to a value high enough to make sure no other PAN Coordinator is nearby. By default, in order to minimize the boot time, it is set to 1, assuming no other PAN Coordinator is present.

The bootstrap procedure consists in the following sequence:

1. The Boot Server receives an *ADP-LBP.Indication* carrying an EAP join message from a new PAN Device.
2. A new entry is added to the joining entry table.
3. The Boot Server sends an EAP-PSK #1 message back to the PAN Device.
4. The Boot Server receives an *ADP-LBP.Indication* carrying an EAP-PSK #2 message from the PAN Device.
5. The Boot Server sends a *BOOT-SRV-GETPSK.Indication* to the Boot module of the application.
6. The Boot Server receives a *BOOT-SRV-SETPSK.Request* from the Boot module of the application.
7. The Boot Server sends an EAP-PSK #3 message back to the PAN Device.
8. The Boot Server receives an *ADP-LBP.Indication* carrying an EAP-PSK #4 message from the PAN Device.
9. The Boot Server sends an EAP accept message back to the PAN Device and a *BOOT-SRV-JOIN.Indication* to the G3 application. A new entry is added to the connected device list.
10. The entry relative to the device just connected is removed from the joining entry table.

In case of timeout at any point (*bootTimer* is used to measure the time at each step), the procedure is aborted and the relative joining entry is removed.

The bootstrap parameters carried by the EAP-PSK #3 message may vary if the Re-keying procedure is in progress, in order to synchronize the new device with the new GMK.

The bootstrap procedure functions are also used by the Re-keying procedure, explained in detail in Section 5.3.1.2 .

5.3.1.2 *GMK update (Re-keying)*

The GMK update, also called Re-keying, is implemented in *g3_boot_srv_eap.c*. The FSM function that manages the whole process is *g3_boot_srv_eap_rekeying_fsm*. This function is called:

- when the Boot Server receives the *BOOT-SRV-REKEYING.Request* from the application;
- at the reception of the EAP-PSK message #2 and #4 from the Host Interface;
- at the reception of the EAP accept message from the Host Interface;
- at the reception of the configuration parameter result message from the Host Interface;
- when a *BOOT_REKEY_MSG* is sent to the G3 task (and *g3_app_boot_srv_rekeying* is called);
- when the *bootTimer* timeout is reached;
- at the reception of the *G3LIB-SET.Confirm* from the Host Interface.

The context information regarding the Re-keying process are stored inside the *boot_server* structure, that contains two GMK and the index of the active GMK. It is mandatory that these variables are always aligned with the attributes set on the ST8500 platform. That is possible by setting the *ADP_ACTIVEKEYINDEX* and the *MAC_KEYTABLE* attributes to the same values as the *boot_server* structure at start-up (see Section 5.2.1).

The GMK update procedure consists in the following steps:

1. Once the *BOOT-SRV-REKEYING.Request* is received by its Boot Server, the coordinator writes the new GMK (*MAC_KEYTABLE* attribute) at the unused index (if the current GMK index is 0, it is written at index 1) by sending a *G3LIB-SET.Request* to the ST8500 platform.
2. Then, after the reception of a successful *G3LIB-SET.Confirm*, the coordinator executes a partial bootstrap procedure, from the transmission of the EAP-PSK message #1 to the reception of EAP-PSK message #4, with all connected PAN Devices, passing them only the new GMK.
3. Once all PAN Devices have completed the partial bootstrap successfully, a configuration parameter message with the *GMK-Activation* parameter is sent to all PAN Devices, in order to switch the active GMK index to the new value.
4. Once all devices have answered with the configuration parameter result message, the coordinator switches the current active GMK index (setting it from 0 to 1 or from 1 to 0) by sending a *G3LIB-SET.Request* with the *ADP_ACTIVEKEYINDEX* attribute set to the new value.
5. After the reception of a successful *G3LIB-SET.Confirm*, the Boot Server updates the variable of the active GMK index and sends back a positive *BOOT-SRV-REKEYING.Confirm* to the application.

For future implementations, additional steps (after step 4) to remove the previous GMK on the PAN Devices and the PAN Coordinator could be added.

In case of error, timeout or abort (requested by *BOOT-SRV-ABORT-RK.Request*), a roll-back procedure is started, reverting the actions performed until the failed step. The steps of the roll-back procedure are the following (the initial step depends on the failed step of the re-keying procedure):

1. The Boot Server of the coordinator changes the index of the active GMK to its original value, by setting the *ADP_ACTIVEKEYINDEX* attribute.
2. The Boot Server sends a configuration parameter message with the *GMK-Activation* parameter to each connected device, to switch their index of the active GMK to its original value.
3. The Boot Server sends back a negative *BOOT-SRV-REKEYING.Confirm* to the application.

For future implementations, additional steps (after step 2) to remove the new GMK from the PAN Devices and the PAN Coordinator could be added.

5.3.2 Boot module - PAN Device implementation

In G3-Hybrid PLC & RF networks, at start-up each PAN Device tries to locate a reachable PAN through the Discovery procedure. The result of this procedure is that a certain number of agents are identified (the PAN Coordinator and/or some PAN Devices already connected to the network). The list of agents is then passed to the host application through the *BOOT-DEV-PANSORT.Indication* message from the Host Interface (or from the Boot Client module, if enabled).

Since the PAN Device must decide which agent shall be used to join the network, it is up to the host application to sort the list of agents. As sorting criteria, the route cost to PAN Coordinator is chosen by default. The criteria can be changed by changing the value of *PANSORT_CRITERION_PRIMARY*, *PANSORT_ORDER_PRIMARY* and/or *PANSORT_CRITERION_SECONDARY*, *PANSORT_ORDER_SECONDARY* inside *settings.h*, or by modifying the *g3_pansort_compare* function.

The function *g3_boot_handle_pansort_ind* (inside *g3_app_boot.c*) sorts the list of agents functions using the chosen criteria and, at the end, sends back the ordered list of agents to the Host Interface (or to the Boot Client module, if enabled) through the *BOOT-DEV-PANSORT.Request* message.

At this point, the G3 Hybrid PLC & RF implementation starts trying to join the network with the bootstrap procedure through the first agent and, in case of failure, scans the entire table until the node is registered.

The PAN Device is able to leave the joined PAN by sending a *BOOT-DEV-LEAVE.Request* to the Host Interface (or to the Boot Client module, if enabled).

To re-reconnect the PAN Device, a *BOOT-DEV-START.Request* can be sent to the Host Interface (or to the Boot Client module, if enabled). This will re-trigger the same steps performed at start-up (Discovery, PAN sort, bootstrap) until the PAN Device joins a PAN (the old one or a new one).

5.3.2.1 The G3 Boot Client module

On the PAN Device side, it is possible to either use the Boot layer embedded in the G3 library of the platform, running the ST8500 in *IPV6_BOOT* mode, or add a custom implementation of the Boot layer at the application layer, running the ST8500 in *IPV6_ADP* mode. This can be chosen by setting *ENABLE_BOOT_CLIENT_ON_HOST* inside *settings.h* to 1, for Boot layer on application, or to 0, for Boot layer on platform. The Boot layer of the PAN Device is constituted by a Boot Client responsible for handling the its bootstrap process, required to connect to a PAN. If the Boot layer is at application level, the Boot Client module is enabled and all G3 Boot messages are sent to the G3 task instead of the Host Interface. This chapter describes the main functions of the Boot layer when implemented at application level (*ENABLE_BOOT_CLIENT_ON_HOST* set to 1). The application Boot layer implementation is similar to the one of the platform Boot layer, so that from the outside both implementations behave in the same way (the application Boot layer is managed with the same messages used with the platform Boot layer) and there is no need to change the rest of the application when a Boot layer implementation is chosen instead of the other.

Compared to the implementation of the Boot Server of the PAN Coordinator implementation, the Boot Client does not include the management of the EAP-PSK protocol2, which even in *IPV6_ADP* mode is handled by the platform.

The Boot Client is implemented inside the *g3_app_boot_clt.c* (Boot layer management).

In addition to the initialization function, the Boot Client module's main functions are *g3_app_boot_clt* and *g3_app_boot_clt_msg_handler*, that manage the ADP message exchange with the ST8500, and *g3_app_boot_clt_req_handler* that handles the Boot layer's requests coming from the G3 task or the User task.

The Boot Client's main role is to serve Boot requests coming from the applications and execute the bootstrap procedure at startup. The *boot_client* structure contains all data needed for the Boot Client's activities, such as: the connected device list, the current state/sub-state of the Boot Client FSM, the PAN ID, the short address, and other variables.

The Boot Client starts by sending a *HOSTIF-NVM.Request* to the platform in order to check if the fast restore is enabled. Depending on the *CONFIG* value read from the NVM, either a fast restore or a normal start is selected. In the case of a normal start, the following sequence is performed:

1. The *bootTimer* is started to wait *BOOT_CLIENT_START_WAIT_TIME* seconds.
2. When the wait is finished, the Boot Client sends an *ADP-DISCOVERY.Request*, with *BOOT_CLIENT_DISCOVERY_TIME* as the duration, to the Host Interface.
3. Upon receiving the *ADP-DISCOVERY.Confirm*, if at least another PAN is detected, the Boot Client sends an *BOOT-PANSORT.Indication* with the list of discovered PAN descriptors. If no PAN is detected, the Boot Client restarts the procedure from step 1.
4. When the *BOOT-PANSORT.Request* is received or when the timeout for its reception is reached, the Boot Client sends an *ADP-NETJOIN.Request* to the Host Interface to connect to the first entry of the PAN descriptor list from the *BOOT-PANSORT.Request* message (or from the previous *ADP-DISCOVERY.Confirm* if the timeout is reached).
5. Upon receiving a positive *ADP-NETJOIN.Confirm*, a *ADP-ROUTEDISCOVERY.Request* message is sent to the Host Interface. In case a negative *ADP-NETJOIN.Confirm* is received, another *ADP-NETJOIN.Request* associated to the same PAN descriptor is sent to the Host Interface. The Boot Client attempts to connect to the PAN using a PAN descriptor a maximum of *BOOT_CLIENT_ASSOCIATION_MAX_RETRIES* times before proceeding to the next entry of the PAN descriptor list.
6. Upon receiving the *ADP-ROUTEDISCOVERY.Confirm*, a positive *BOOT-DEV-START.Confirm* message is sent back to the G3 task, which gives information on the PAN ID and the assigned short address.

In the case of a fast restore start, the following sequence is performed:

- A *HOSTIF-NVM.Request* is sent to the Host Interface to read the necessary data to perform the fast restore.
- Upon receiving the *HOSTIF-NVM.Confirm* message, the attributes needed to restore the connection saved in NVM are set one by one with the *G3LIB-SET.Request*. When the last attribute is set, a *ADP-ROUTEDISCOVERY.Request* message is sent to the Host Interface.
- Upon receiving the *ADP-ROUTEDISCOVERY.Confirm*, a positive *BOOT-DEV-START.Confirm* message is sent back to the G3 task, which gives information on the PAN ID and the assigned short address.

When the Boot Client is connected to a PAN, it is ready to handle leave requests at any time, in case the device needs to disconnect from the PAN.

5.4 The G3 Keep-Alive module

In a real network, the channel conditions can vary during uptime, and sometimes it may happen that a PAN Device cannot exchange data with the PAN Coordinator (or vice versa), meaning that its route to the PAN Coordinator is not valid anymore. Even if the channel conditions remain acceptable, the entries in the routing table of the nodes have a limited validity time; if there is no communication for a longer time, the entry is invalidated.

Usually this condition is asserted as soon as the PAN Device tries to send data to the PAN Coordinator (or vice versa); this triggers a Route Repair mechanism which takes time to be completed, significantly increasing the round-trip-delay of the triggering data message and the overall traffic in the network. In many cases, this is not a problem, but there are cases where this delay may become unacceptable, either because it is larger than the validity time of the data transferred, or because it may lead to a temporary network instability if many nodes try to recover their route to the PAN Coordinator in a short period of time.

To lower the occurrence of this event, especially on networks where PAN Devices do not communicate regularly with the PAN Coordinator, it is good practice to implement a Keep-Alive mechanism, which should be able to:

- periodically monitor the connection;
- try to repair the route if the connection is not stable (this is done automatically by the G3 Hybrid PLC & RF implementation);
- disconnect from the network and restart the bootstrap procedure in case the failure cannot be recovered.

The Keep-Alive module takes care of this process.

Note: This feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_ICMP_KEEP_ALIVE macro to 1 or 0 inside settings.h.

5.4.1 Keep-Alive module - PAN device implementation

The Keep-Alive module on the PAN Device side keeps track of the last time a ICMP echo with the PAN Coordinator has been performed successfully. If there is no communication for more than *KA_DEVICE_LEAVE_TIME* milliseconds, the *kaTimer* triggers the *g3_ka_device_timeout_expired* function and the PAN Device sends a *G3BOOT-DEV-LEAVE.Request* to disconnect from the PAN.

To reconnect, a *G3BOOT-DEV-START.Request* can be sent to the ST8500 platform.

5.4.2 Keep-Alive module - PAN Coordinator implementation

The Keep-Alive module in the PAN Coordinator makes use of the ICMP echo feature (also called "ping") to communicate periodically with each PAN Device. The PAN Devices considered by the Keep-Alive module are the same inside the connected device list of the *boot_server* structure of the Boot/Boot Server module.

The Keep-Alive mechanism uses a “lives” system, that consists of subtracting one “life” each time a ping fails and resetting the lives back to the maximum amount each time a ping is successful. Each device starts with *KEEP_ALIVE_LIVES_N* lives (2 by default) when it joins the PAN.

The Keep-Alive is constituted by a FSM that works in combination with the *kaTimer*, used for timed Keep-Alive events, such as the "ping event".

When the Keep-Alive is started by calling *g3_app_ka_start* on the Coordinator, the *kaTimer* is started. On its next timeout the ping event is triggered.

When the ping event is triggered, the first PAN Device of the connected device list is pinged by sending a *ICMP-ECHO.Request* with its IPv6 address as destination to the Host Interface. From there, every *KEEP_ALIVE_CHECK_NEXT_DELAY* milliseconds the subsequent PAN Device is pinged. When all PAN Devices in the list have been pinged, the sequence of pings is restarted from the first PAN Device after *KEEP_ALIVE_CHECK_PERIOD* milliseconds (120 seconds by default). When a PAN Device is pinged, the *kaTimer* is started.

If the *ICMP-ECHOREP.Indication* from that PAN Device is not received before the *kaTimer* timeouts, the "echo timeout" event is triggered, and the number of lives of that PAN Device is decreased by 1. Receiving a *negative ICMP-ECHO.Confirm* also decrements the number of lives, with the exception of *G3_BUSY* error code as confirm status. In that case, the *ICMP-ECHO.Request* is sent again after *KEEP_ALIVE_CHECK_RETRY_DELAY* ms.

The number of lives of a PAN Device gets restored to the maximum value when its *ICMP-ECHOREP.Indication* is received correctly after its ping. If the number of lives of a PAN Device goes to 0, that PAN Device is immediately kicked out of the PAN with a *BOOT-SRV-KICK.Request*.

5.5 The Last Gasp module

The Last Gasp feature is a specific transmission mode designed to alert a nearby node of a power outage.

It is implemented inside *g3_app_last_gasp.c*, mostly on a PAN Device side, and it consists of sending a broadcast UDP packet with the purpose of informing the PAN Coordinator that a power outage was detected.

Since the PLC medium is not reliable in case of an outage, the Last Gasp mode only uses RF packets.

In a real metering application, the Last Gasp mode should be triggered by the detection of a power outage. However, in this example software, the feature is demonstrated by pressing the blue push-button (GPIO PC13) of the Nucleo board of the evaluation kit, for simplicity.

In the PAN Device implementation, the *g3_last_gasp_fsm_manager* function is responsible for handling all steps involved in the activation and the execution of the Last Gasp FSM.

These steps consist in:

- Acquiring the bootstrap information upon receiving the *G3BOOT-DEV-START.Confirm*;
- Waiting for the Last Gasp activation event;
- Upon detecting the activation event, broadcasting the Last Gasp message as a UDP packet using the *last_gasp_msg_t* structure as payload, with the short address of the device as *gasped_short_addr*; and
- Leaving the PAN.

After all steps have been concluded, the Last Gasp cannot be activated again and the node can only transmit RF broadcast packets. In order to restore the modem to its normal status, an HW-RESET is necessary.

Since it is possible that many hops are necessary to reach the coordinator, the Last Gasp module also provides a forwarding mechanism for incoming Last Gasp UDP packets.

The Last Gasp module uses a dedicated UDP connection (with the local and remote port set to 50 by default). All UDP packets received from this connection are evaluated by:

- Checking that the *id* value of the Last Gasp message is equal to its expected value; and
- Checking that the *visited_device* array does not contain the short address of the forwarding PAN Device.

If these initial checks are passed, the short address of the forwarding PAN Device is added to the *visited_device* array and then the Last Gasp activation is evaluated:

- If the Last Gasp mode is not active on the forwarding device (normal mode), the Last Gasp message is forwarded in unicast directly to the PAN Coordinator, without further need of being forwarded.
- Instead, if the Last Gasp mode is active on the forwarding device (Last Gasp mode), if the *hop_count* value is less than *LAST_GASP_MAXHOPS*, the Last Gasp message is forwarded in broadcast to the surrounding neighbors with the *hop_count* pre-incremented.

The idea behind this forwarding mechanism is to avoid having a PAN Device forwarding multiple times the same Last Gasp message. The maximum number of short addresses that can be listed inside the *visited_device* array is exactly equal to *LAST_GASP_MAXHOPS*.

5.6 The G3 task internal functions

The G3 task is composed of an initialization function, *g3_task_init*, that is executed once after FreeRTOS has started, and a main routine function, *g3_task_exec*, that is repeatedly executed afterwards.

The *g3_task_init* is responsible the following initialization:

- G3 Configuration module;
- G3 Boot module;
- G3 Boot Server/Client module (only for with Boot Server/Client at application level);
- G3 Keep-Alive module;
- G3 Last Gasp module (only for PAN Device);
- Host Interface;

The main routine functions, *g3_task_exec*, instead, is composed of two parts:

- a start-up block that is executed only once, responsible for receiving the initial *HOSTIF-HWRESET.Confirm* message, handling an eventual fast restore, changing the baud rate for the Host Interface and starting the G3 Configuration module;
- an endless *for* loop that takes care of receiving, forwarding and/or transmitting G3 messages and running all modules.

The implemented reception/transmission mechanism is made that only two requests can be sent through the Host Interface at a time. Before sending another request, the confirmation to at least one previous request must be received (this is implemented using the *semConfirmation* semaphore). The only exceptions to this rule are the *G3BOOT-DEV-START.Request* message, that consents the transmission of more G3 requests before its confirm message is received, and the *G3ICMP.ECHO.Request* and *G3UDP-DATA.Request*, that are transmitted to the platform only when confirmations have been received for all previous requests.

To avoid critical blocks, in case the confirmation is missed by error, a timeout (*TIMEOUT_CNF*) for the confirmation reception is present. The G3 task always stays in the *BLOCKED* state, and it executes its routine only when a task message is sent to the *g3_queue*.

6 User applications

6.1 Introduction

A specific task, named User task, has been defined to handle the user application. The corresponding files are in the *\User_Applications* folder. The User task implements an example of a UDP application with a serial terminal interface and multiple features.

This section describes the User task explaining how it is handled.

6.2 User task interface with the G3 task

When the G3 task receives a G3 message, it can forward it to the User task by calling the *g3_msg_forward* function. This function sends the G3 message to the *user_queue* queue if the G3 message is considered necessary for the User task.

Inside the User task, the G3 messages are parsed by the *user_msg_handler* function and later discarded, at the end on the cycle, to free the allocated memory pools.

6.3 User task interface with the serial terminal

The User task can interact with the user via any PC serial terminal (like Teraterm) through the User Interface. The incoming/outgoing terminal data is handled inside the *user_if.c* file.

6.3.1 Reception (from terminal to User task)

Bytes received from the User Interface are first assembled in a packet to create a command message. Its content is defined by the *user_input_t* structure. The *USERIF_INPUT_MAX_SIZE* macro sets the maximum length, in bytes, for one string to be sent from the terminal (set to 128 by default). To be considered as valid, a string must end with a CR character (by pressing the Enter key).

Once created inside *user_if_rx_handler*, the received command message is buffered in a FIFO buffer and an empty message is sent to the *user_queue*, so that the User task can extract the command by calling *user_if_get_input* and process it. The whole reception control is defined by the *user_if_fifo_rx* local structure.

6.3.2 Transmission (from User task to terminal)

The User task can print data on the terminal by invoking the *user_if_printf*, function. This function can be called with multiple printing macros (*PRINT*, *PRINT_NOTS*, *PRINT_RAW*...) defined in *user_if.h*, depending on the desired print format. The *user_if_printf* function, after formatting the string to print on the terminal, calls the *user_if_low_level_print* function, that sends strings of characters to the Print task through the *hostStreamBuffer* stream buffer.

The stream buffer size is set by the *USERIF_TX_FIFO_SIZE* macro.

The bytes put in the host stream buffer are extracted and transmitted through the User Interface by the Print task, during its execution time. See Section 4.4.2 for further details regarding the Print task implementation.

6.4 User task internal functions

The User task is composed by an initialization function, *user_app_init*, and a looped routine function, *user_app_exec*. The *user_app_exec* function handles the messages forwarded from the G3 task, by calling *user_msg_handler*, and executes different routines depending on the working PLC mode, that is selected at start-up with a specific GPIO (PC3 by default).

In *IPV6_BOOT* and *IPV6_ADP* mode, the *user_app_exec* function manages the UDP communication with the G3 task (see Section 6.4.1), the IPv6 version of the User Terminal (see Section 6.4.2) and the User Image Transfer (see Section 6.4.3).

In *MAC* mode, the *user_app_exec* function manages the MAC test mode (see Section 6.4.4), the MAC version of the User Terminal (see Section 6.4.2).

The User task always stays in the *BLOCKED* state, and it executes its routine only when a message is sent to the *user_queue*.

6.4.1 G3 communication module

The G3 communication module acts as interface between the User task and the G3 task and it is implemented in *user_g3_common.c*. This module ensures:

- The parsing of incoming messages from G3 task, using the *UserG3_MsgHandler* function that is called by the *user_msg_handler* for the G3 communication related messages. The parsed messages generate user events and call specific sub-functions, depending on the received message. At his discretion, the user can modify or add more handlers inside the *UserG3_MsgHandler* function to customize the reception handling of confirms and indications coming form the ST8500 platform.
- The implementation of UserG3 FSM, the *UserG3_FsmManager* function, that is used to handle the G3 communication at IPv6 level, including the connection setup, the transmission of UDP packets (*G3UDP-DATA.Request*), the reception of UDP packets (*G3UDP-DATA.Indication*) and other G3 requests. The UserG3 FSM, once started by calling *UserG3_StartUdpConnSetup*, sets all the connections located in *connection_table*, that are pointed inside the *connection_list* array (more connections can be easily added). Any connection can be modified afterwards by calling *UserG3_ModifyUdpConnection*. A UDP packet can be sent through a specific connection by calling the *UserG3_SendUdpData* or the *UserG3_SendUdpDataToShortAddress* function, specifying:
  - the ID of the connection to use;
  - the destination IPv6 or short address (depending on the function);
  - the pointer to the buffer that contains the data to send;
  - the number of bytes to send.

The *UserG3_SendUdpDataToShortAddress* is a wrapper function that calls *UserG3_SendUdpData* after converting the given short address to the corresponding IPv6 address.

The *UserG3_SendUdpData* is the main function used to transmit UDP packets and it uses a memory pool to transfer the payload. If an already existing memory pool is passed to this function, that memory pool will be used instead, without allocating a new one. Either way, the memory pool gets deallocated inside *userg3_fsm_send_data*, after the *G3UDP-DATA.Request* message is prepared for the transmission through the Host Interface.

The *userg3_set_next_connection* and *userg3_fsm_send_data* functions also starts the *commTimer* to handle an eventual timeout of the *G3UDP-CONN-SET.Confirm* or the *G3UDP-DATA.Confirm* messages. This timer is stopped when these messages are received, in order to prevent an invalid timeout event.

In case a UDP packet is received, it is automatically stored inside a memory pool by the *userg3_handle_udp_data_ind* function. The user can access the payload data received from a given connection by calling *UserG3_GetUdpData()* with its ID as parameter and then free the memory pool by calling *UserG3_DiscardUdpData()* with the same connection ID after the data is no longer necessary.

The implementation of a structure named *userg3_common* gathers data to make it available to the rest of the user application, including the User Terminal (G3 user events, last UDP packet content received, online status, platform information...).

6.4.2 User Terminal module

The User Terminal is implemented in *user_terminal.c*, with the *UserTerminal_Init* and the *UserTerminal_FsmManager_IPv6*/*UserTerminal_FsmManager_MAC* functions. Its main goal is to manage the serial terminal and various actions in the system (e.g. interactions with the IPv6 or layer), depending on terminal user inputs (from the User Interface) on one side and events coming from the G3 task on the other side. The *UserTerminal_FsmManager_MAC*, in particular, is called only in *MAC* mode and its purpose is to start the MAC test with multiple parameters received from the User Interface.

Each time *UserTerminal_FsmManager_IPv6* is called:

1. User events are parsed by calling *user_term_parse_user_events*.
2. It is checked if the Escape key was typed on the terminal by the user to reset the FSM.
3. The current state function is executed.

Parsed events are used to inform the User Terminal that an event occurred and it is possible to proceed to the next action. The *USER_EVENT_RISEN* macro is used to assert that an event just occurred, while the *USER_EVENT_OK* is used to verify if the event was positive (successful) or negative (failed).

The User Terminal has two sub-states for each state, one for printing information and instructions on the terminal, and one to acquire input from the user or wait a specific user event (such as the reception of a confirm or indication).

Some states or sub-states (like the *user_term_state_test_exec* function) are organized in multiple steps. By interacting with the terminal, it is possible to navigate through the various states, sub-states and steps, with the possibility to return to the main menu by pressing the Escape key. A user input command/string is acquired from the User Interface reception FIFO buffer by calling the *user_if_get_input* function.

Inside several functions, when it is necessary to re-execute the User Terminal afterward, an empty task message is sent to the *user_queue* to re-trigger the same FSM.

The actual implementation of the User Terminal ensures:

- exchange of UDP messages for testing purposes (with different kinds of tests);
- display the current UDP connections;
- display the list of connected devices (PAN Coordinator) or the device information (PAN Device);
- UDP transfer of a PE/RTE image inside the STM32 SPI Flash memory from the PAN Coordinator to a PAN Device*;
- erasure of the STM32 SPI Flash memory (single sector or mass erase)*;
- possibility to kick a PAN Device (PAN Coordinator) or to leave the network (PAN Device)*;
- activate/deactivate the fast restore (only for PAN Device)*;
- updating the active GMK of all connected devices in the PAN (only for PAN Coordinator with Boot Server at application level)*;
- reset the system (both the STM32 and the ST8500 platform).

See Section 6.6 for further details about the implementation of these features.

To handle events such as reception timeouts, the *userTimeoutTimer* is started with the *user_term_set_timeout*. When the *UserTerminal_TimeoutCallback* is called, a warning message is printed on the terminal and the User task is unblocked. The *user_term_timeout_reached* function is used to evaluate if a timeout occurred. Lastly, the *user_term_remove_timeout* stops the *userTimeoutTimer*, preventing the timeout to occur.

The implementation of the User Terminal is fully upgradable so that it can be easily adapted to further user needs (e.g. other user-defined tests, specific routines for demo purposes).

Note: The features marked with "" can be enabled or disabled inside settings.h.*

6.4.3 User Image Transfer module

In the EVLKST8500GH-2 kit, the STM32 microcontroller is connected via SPI to an external SPI Flash memory (W25Q16JV). This is useful to store ST8500 images (RTE and PE type) that can be fed to the ST8500 at startup, if the “Boot from UART” mode is selected with the switches. The UDP application is able to transfer ST8500 images stored in the SPI Flash from the PAN Coordinator to other connected PAN Devices. The User Image Transfer is responsible for this feature. For UDP communication, a specific reserved UDP connection is used for the transfer (with a local and remote UDP port equal to 2000 and null remote IPv6).

Inside the *user_image_transfer.c* file, an FSM function named *UserImgTransfer_FsmManager*, depending on the device type, handles the UDP transfer procedure in transmission (PAN Coordinator) or reception (PAN Device). In the PAN Coordinator implementation, the transfer is started by calling *UserImgTransfer_StartSend*, while on the PAN Device implementation, the transfer is started by calling *UserImgTransfer_StartReceive*.

It is possible to abort the transfer, by calling *UserImgTransfer_Stop*, or get its current status and error code, by calling respectively *UserImgTransfer_IsComplete* and *UserImgTransfer_GetError.*

The User Image Transfer FSM on the sender side executes the following procedure:

1. Once the transfer starts, a UDP packet with information regarding the image characteristics (*image_info_t*) is sent to the image receiver.
2. When the acknowledge UDP packet (*image_ack_t*) is received from the image receiver, if there is still data to send, the SPI Flash memory is read by calling *getDataBlock* to get the next block of image data. Otherwise, the procedure ends.
3. The block of image data read from the SPI Flash memory is sent as a UDP packet (*image_data_t*) to the image receiver.
4. The procedure is repeated from step 2.

The User Image Transfer FSM on the receiver side executes the following procedure:

1. Once the transfer starts, the SPI Flash memory slot selected for the incoming image is erased.

2. After the erasure is complete, the UDP packet with information regarding the incoming image characteristics (*image_info_t*) is waited.
3. When the UDP packet with the image information is received, the procedure continues from step 4.
4. At this point, an UDP packet is sent to the image sender as an acknowledge (*image_ack_t*).
5. When the confirmation of the acknowledge UDP packet transmission is received, if there is still data to receive, the UDP packet containing the next block of image data (*image_data_t*) from the image sender is waited. Otherwise, the procedure ends.
6. When the UDP packet with the image data (*image_data_t*) is received, it is written inside the SPI Flash memory by calling *setDataBlock*.
7. The procedure is repeated from step 4.

Both the sender and the receiver use the *user_img_transfer_fsm* structure to handle the various procedure steps and keep track of the transfer progress.

Note that the SPI Flash memory is driven inside the SFlash task, in order not to block the User task while the SPI Flash memory is read or written.

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_IMAGE_TRANSFER macro to 1 or 0 inside settings.h.

6.4.4 MAC test mode

The "MAC test mode" is activated at startup by pulling to GND a specific GPIO (PC7 by default) and it is used for testing the entire HW connectivity (SD card, RS485, and SPI Flash memory) of the EVLKST8500GH-2 evaluation kit. This testing mode requires an SD card to be inserted and another evaluation kit both connected via RS485 and PLC in the same mode and a band plan. For the RS485 test, one of the two kits must be configured in controller mode by pulling to GND a specific GPIO (PB3 by default).

The MAC mode test can be directly executed by pressing the blue push-button (EXTI on PC13) on the Nucleo board configured as a controller after a fresh restart of the two boards in MAC mode. During the test, first all connectivity is tested, then an exchange of PLC and RF MAC frames is performed with the other kit. At the end of the test, the result is displayed through the LEDs (using the values from *mac_test_state_t*). On the PAN coordinator implementation, only the MAC communication test is performed, excluding the SD card, RS485, and SPI Flash memory tests to save program memory space.

When the test is started by pressing the blue push-button, the first MAC frame is transmitted in broadcast because the tester device is able to acquire the extended address of the DUT only after its first response. After that, the following MAC frames are transmitted in unicast to the extended address extracted from the first MAC response.

When testing with more than two nodes connected, the test cannot be performed using the mechanism described previously since more than one kit would answer to the broadcast frame. Instead, it is possible to initiate the test from the User Terminal (through the User Interface), by sending a command string with the following format (including the dots):

EXTENDED_ADDRESS.TEST_TYPE.NUMBER_OF_FRAMES

Where:

- *EXTENDED_ADDRESS* is the extended address of the device to exchange the frames with.
- *TEST_TYPE* can be either:
  - *PLC* to exchange only PLC MAC frames;
  - *RF* to exchange only RF MAC frames; or
  - *PLCRF* to exchange PLC and RF MAC frames.
- *NUMBER_OF_FRAMES* is the number of MAC frames exchanged for each type of physical medium used (when *TEST_TYPE* is equal to *PLCRF*, the total number of exchanged MAC frames is two times the *NUMBER_OF_FRAMES*).

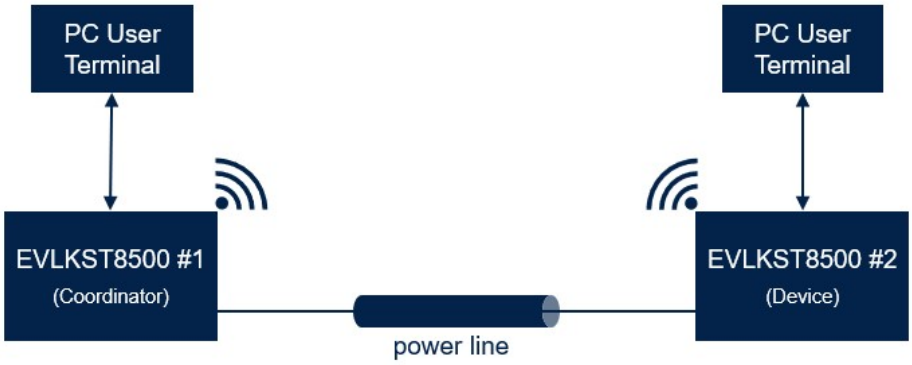
The test, if initiated from the User Terminal, starts immediately with a MAC frame transmitted in unicast to the specified extended address, allowing for the coexistence of many kits connected in the same area.

6.5 Use of serial terminal

Using a PC serial terminal allows the user to start the evaluation of some features like UDP data transfers via some tests/demonstrations. It also displays upcoming specific events such as G3 network bootstrap messages. A serial terminal software must be available on the PC side and a USB cable is connected between the EVLKST8500GH-2 board (USB connector of the NUCLEO board) and the PC.

A typical evaluation setup with two evaluation kits is represented in the following image:

Figure 8. Typical test setup using the PC serial terminal



Further evaluation kits might be added, either as PAN Devices, or as G3 sniffers using the G3 graphical user interface (GUI) included in the STSW-ST8500GH-2 package.

More information about User terminal configuration, handling and test execution is available in "*ST8500 Hybrid PLC&RF connectivity development kit - User Terminal guidelines*".

6.6 UDP application example features

One goal of the User task is to give an example of a UDP application handled via a serial terminal. The implemented UDP application offers the following features:

- multiple ways to test the exchange of UDP messages (UDP tests, explained in Section 6.6.1);
- display UDP connections;
- display connected devices (PAN Coordinator) or the device information (PAN Device);
- ST8500 image transfer from/to the STM32 SPI Flash memory (UDP Image Transfer, explained in Section 6.6.2);
- STM32 SPI Flash memory management (SPI Flash Management, explained in Section 6.6.3);
- manage connected device or PAN connection (PAN/Device management, explained in Section 6.6.4);
- enabling/disabling Fast Restore (Fast Restore, explained in Section 6.6.5);
- execute the update of the Group Master Key (GMK) used in the PAN (GMK update/Re-keying, explained in Section 6.6.6);
- reset the system entirely (STM32, ST8500 and S2-LP).

All these features are available in the main menu, that is displayed once the device has connected to the PAN. Each feature can be selected through a specific menu option. Each of the following sub-sections describes one of the features in details.

Note: Some of these options can be disabled inside settings.h to save embedded Flash memory space.

6.6.1 UDP tests

The UDP tests consist of exchanging UDP messages between two connected devices (coordinator-device or device-device) and are implemented inside the *user_term_state_test_exec* function called inside *UserTerminal_FsmManager* when the relative menu entry is selected. There are different possible types of UDP tests:

- Basic: a UDP string packet is sent from a device to another;
- Multiple Basic: many UDP string packets are sent from a device to another;

- Loop-back: a UDP string packet is sent from a device to another, then it is sent back to the first device, as it is.
- Multicast: a UDP string packet is sent in broadcast and it is therefore received by all devices connected to the PAN.

The *user_term_state_test_exec* function is divided into multiple guided steps, that depend on the chosen test type and role for the device. A device can indeed act as an originator, or as a recipient/looper, depending on the type of test. For each test, two devices are always required: one must act as an originator, and the other must act as a recipient/looper. In addition, other settings requirements must be met:

- the Remote UDP port of the recipient/looper must match the Local UDP port of the originator, or it must be zero;
- the Local UDP port of the recipient/looper must match the Remote UDP port of the originator, and it must not be zero;
- the Remote IPv6 address of the recipient/looper must match the IPv6 address of the originator, or it must be null;

On the originator board side, the destination short address is prompted to the user in one of the guided steps, while the destination PAN is always the PAN the device is connected to.

In the loop-back UDP test, the destination IPv6 address and UDP port are taken from the received UDP packet header.

Each test function in transmission sends UDP packets by calling *UserG3_SendUdpData* with *TEST_CONN_ID* as connection ID, and each test function in reception checks for the UDP data indication event to detect the reception of incoming UDP packets.

6.6.2 UDP image transfer

It is possible to start and stop the UDP transfer of a PE/RTE image from the User Terminal. Inside the *user_term_state_transfer* function, depending on the device type, either *user_term_udp_transfer_tx* (PAN Coordinator) or *user_term_udp_transfer_rx* (PAN Device) is called. In the PAN Coordinator case, before calling *user_term_udp_transfer_tx*, the user is prompted for the destination short address of the recipient PAN Device. These functions have several steps necessary to handle the image transfer.

The image sender device proceeds with the following steps (*user_term_udp_transfer_tx*):

1. Displays the SPI Flash memory content on the terminal by calling *checkMemoryContent*.
2. Prompts the user for the selection of the memory slot with the image to send.
3. Once the slot is selected, checks the image validity, size and calculates the CRC16 of the image. If the image is valid, the image transfer is started by calling *UserImgTransfer_StartSend*.
4. The end of the image transfer is polled (when the User task is unblocked) by calling *UserImgTransfer_IsComplete*. When the transfer ends, its result is evaluated with the *UserImgTransfer_GetError* function.

The image receiver device proceeds with the following steps (*user_term_udp_transfer_rx*):

1. Displays the SPI Flash content on the terminal by calling *checkMemoryContent*.
2. Prompts the user for the selection of the slot for the image to receive.
3. Once the slot is selected, warns if the selected slot is already occupied by another image and starts the transfer by calling *UserImgTransfer_StartReceive*.
4. The end of the image transfer is polled (when the User task is unblocked) by calling *UserImgTransfer_IsComplete*. When the transfer ends, its result is evaluated with the *UserImgTransfer_GetError* function.

Both sender and receiver use the *user_term_transfer* structure to handle the various procedure steps of the transfer. The internal procedure of the image transfer and the exchange of UDP packets is handled inside the User Image Transfer module (see Section 6.4.3).

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_IMAGE_TRANSFER macro to 1 or 0 inside settings.h.

6.6.3 SPI Flash memory management

It is possible to examine and, eventually, erase the content of the SPI Flash memory connected to the STM32 microcontroller via SPI. This is entirely handled inside the *user_term_state_erase_sflash* state function that shows the memory content with the *checkMemoryContent* function, prompts the user to select a specific sector or the whole Flash memory and then proceeds to perform the erasure by either calling *eraseMemorySector*, for a single sector erase, or *eraseMemory*, for a bulk erase.

All functions that interact with the SPI Flash memory (implemented inside *image_management.c*) make use of the *sflash_command* wrapper function. See Section 4.5 for more details.

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_SFLASH_MANAGEMENT macro to 1 or 0 inside settings.h.

6.6.4 Device management

The devices connected to the PAN can be managed by the user terminal of the PAN Coordinator. In particular, the PAN Coordinator can kick out a specific connected PAN Device by selecting the relative option and inserting the short address of the PAN Device to kick out. The confirm of the kick request is then waited. The user terminal goes back to the main menu after its reception.

On PAN Device side, instead, each PAN Device can leave the PAN with the relative option from the user terminal. In that case, the user is prompted for confirmation. If the user confirms, the leave request is sent and then the user terminal waits for the reception of the leave confirm. The user terminal goes back to the main menu after its reception.

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_DEVICE_MANAGEMENT macro to 1 or 0 inside settings.h.

6.6.5 Fast Restore

On the PAN Device implementation, if the user wishes to enable or disable the Fast Restore, it is possible to do so through the User Terminal. The *user_term_state_fast_restore* manages this feature, prompting the user for confirmation after its selection. If the user confirms its choice, a *HOSTIF-NVM.Request* is sent to the Host Interface to write the bit of the ST8500 NVM responsible for the activation of the Fast Restore at startup.

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_FAST_RESTORE macro to 1 or 0 inside settings.h.

6.6.6 GMK update request

On PAN Coordinator side, the user can request the update of the Group Master Key (GMK) used by all devices of the PAN (the PAN Coordinator itself and all connected PAN Devices) with a specific command from the user terminal. This feature requires the Boot Server to be implemented at application level (*ENABLE_BOOT_SERVER_ON_HOST* inside *settings.h* set to 1). Once the relative option is selected, the *user_term_state_rekeying* is called and the user is prompted for the new GMK. If the inserted GMK is valid, the *UserG3_SendRekeyingRequest* is called and the *BOOT-SRV-REKEYING.Request* is sent to the application Boot Server. The User Interface FSM then waits for the *BOOT-SRV-REKEYING.Confirm* that signals the completion of the update. The procedure can be aborted by the user through the user terminal: in that case, a roll-back procedure to restore the original GMK is started from the step where the update is interrupted. After a successful GMK update procedure, all devices in the PAN communicate with messages encrypted with the new GMK. For more details about the GMK update procedure, see Section 5.3.1.2 .

Note: this feature can be disabled to save embedded Flash memory space. To enable or disable it, set the ENABLE_REKEYING macro to 1 or 0 inside settings.h.

7 Extending the G3 Hybrid PLC & RF software example

In this section, a few hints on how the user can customize the code are presented, to add new features or modify/remove the existing ones. As a general remark, to maintain compatibility with the STM32CubeMX environment, and allow automatic regeneration of the source code, the user should modify system generated files (the ones included in the folders *Drivers*, *Inc*, *Src* and *Middlewares*) only inside a specific section indicated by a couple of markers like the ones reported below, which are present in all the modifiable system generated functions:

```
/* USER CODE BEGIN <func_name> */
```

...

```
/* USER CODE END <func_name> */
```

All the user code typed inside such user code space is kept during regeneration, while the code typed outside is removed and completely lost. Of course, this is not required for the files contained in the *G3_Applications* folder, the *Modules* folder, the *User_Applications* folder and any source file added by the user.

7.1 Extending G3 task with additional features

To extend the G3 task with an additional module, the user should make sure to define:

- an initialization function (if needed) for the new module to be called from the *g3_task_init* function;
- a message handler function called when needed inside *g3_msg_handler*;
- a function implementing the new module state machine, to be called inside the endless loop of the function *g3_task_exec*, after the message handler has been executed (if it must be triggered upon receiving specific G3 messages) or when a non-G3 task message is received from the G3 task (the Keep-Alive implementation can be studied as reference).

Should the new module request the usage of Host Interface commands not to be already managed, the hints provided in Section 7.2 should be applied. Otherwise, the helper functions defined in the *hi_msgs_impl.c* file and the Host Interface message parser functions already defined, should be amended to support the new module.

Note: the G3 task, in order to execute, always needs to be unblocked by sending a message to the g3_queue.

7.2 Managing Host Interface messages

The addition of new features and new tasks could require that the user implements additional helper functions to ease the preparation of commands to be sent to, and to decode the messages coming from, the Host Interface. To do that, the user should add the required functions to the *hi_msgs_impl.c* source file.

As an example of how the user should proceed, the implementation of the command *G3ICMP_ECHO.Request* is described. This command executes an ICMP echo request to a specific node. The format of the Host Interface command implementing this feature is shown in Table 2.

Table 2. G3ICMP_ECHO.Request command format

#Bytes	Label	Values	Description
16	DESTADDR	-	The 16 bytes IPv6 remote destination address
1	HANDLE	0x0-0xFF	Handle returned in the G3ICMP-ECHO.Confirm
2	DATALEN	0-1500	The ECHO request payload length
0-1500	DATA	-	The ECHO request payload

The helper function which compiles the command is the *hi_ipv6_echoreq_fill*, which is reported here below. This function accepts as parameters:

- *msg_*, pointer to the buffer which contains the command sequence to be sent to the Host Interface;
- *dst_addr*, IPv6 address of the destination node;

- *handle*, user-defined ID which should be used to correlate the request with the answer;
- *data_len*, length of the payload associated with the echo message;
- *data*, pointer to the buffer containing the payload associated with the echo message.

An example of how this helper function should be used is given in the function *hi_ipv6_echoreq_fill*, that prepares the message inside the *IP_G3IcmpDataRequest_t* structure and returns the length of the prepared message.

Afterwards, the *g3_send_message* function is called using the proper command ID macro (*HIF_ICMP_ECHO_REQ*), the pointer to the *IP_G3IcmpDataRequest_t* structure and the calculated length as parameters. The message type is set to *HIF_TX_MSG*, therefore it is sent to the Host Interface.

To manage incoming messages from the Host Interface, such as indications or confirm messages, the user should extend the function *g3_msg_handler*, adding a new handler, or extending one of the existing handlers if the purpose is to extend an existing functionality. Four handlers (one for each implemented module) are already defined. Each handler is called only if the received message is needed by that functionality (for instance, *g3_app_conf_msg_needed* returns true if *g3_app_conf_msg_handler* needs to process the message in question). If the message is to be processed also by the User task, the *UserG3_MsgNeeded* function, called inside *g3_msg_forward*, must return true for the command ID of the received message. Then, the *RTOS_PUT_MSG* macro forwards the message to the User task.

7.3 Adding a user-defined task

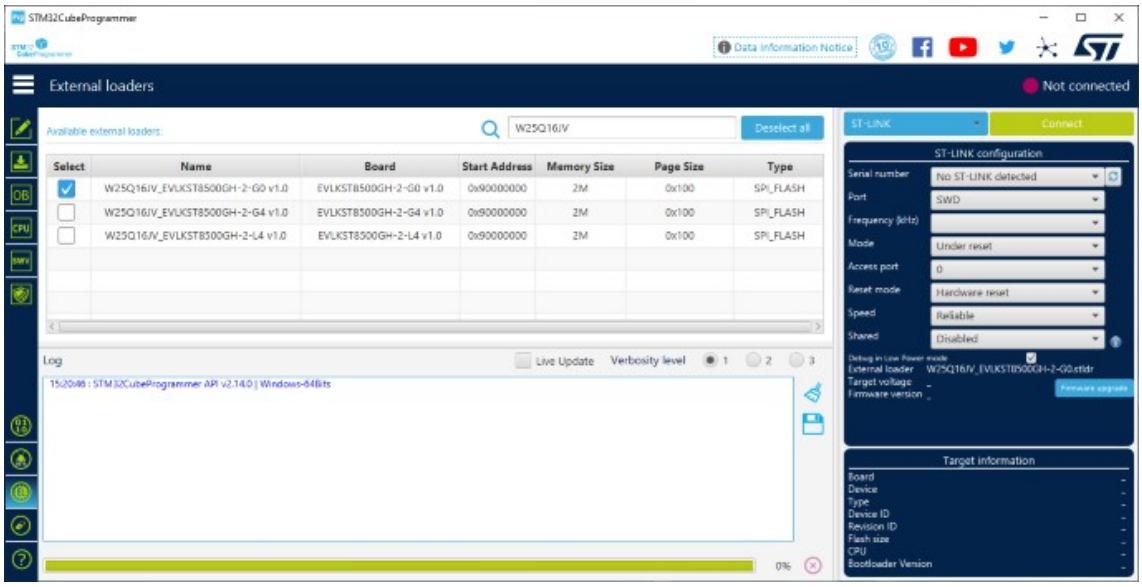
The user can easily add one or more user-defined tasks to this example and the way the G3 task has been implemented should be an example on how the customer should proceed. In general, in case the user-defined task needs to make use of some of the features of the G3-Hybrid PLC & RF protocol, the user should:

- Extend the support of Host Interface commands (see Section 7.2).
- In case the communication between the tasks involves the exchange of a large amount of data, use the existing memory pool infrastructure, customizing it if needed (see Section 4.2).
- Use the user event macros and the functions from *user_g3_common.h* to interact with the G3 task, to set connections and send/receive UPD data or send other requests to the G3 stack (the user terminal example inside *user_terminal.c* should be followed).

8 STM32 SPI Flash programming

To download the RTE and PE images to the ST8500 when it is in the Boot From UART mode, first, it is necessary to store such images on the SPI Flash connected to the STM32. The images can be acquired by transferring them with the UDP Image Transfer (see Section 6.6.2) or by using a specific External Loader inside STM32Cube Programmer named *W25Q16JV_EVLKST8500GH-2-XX*, where "*XX*" depends on the Nucleo board involved ("G0" for NUCLEO-G070RB, "G4" for NUCLEO-G474RE, "L4" for NUCLEO-L476RG). This External loader sees the STM32 SPI Flash memory space mapped to the virtual address range 0x90000000-0x901FFFFF.

Figure 9. STM32Cube Programmer External loaders menu

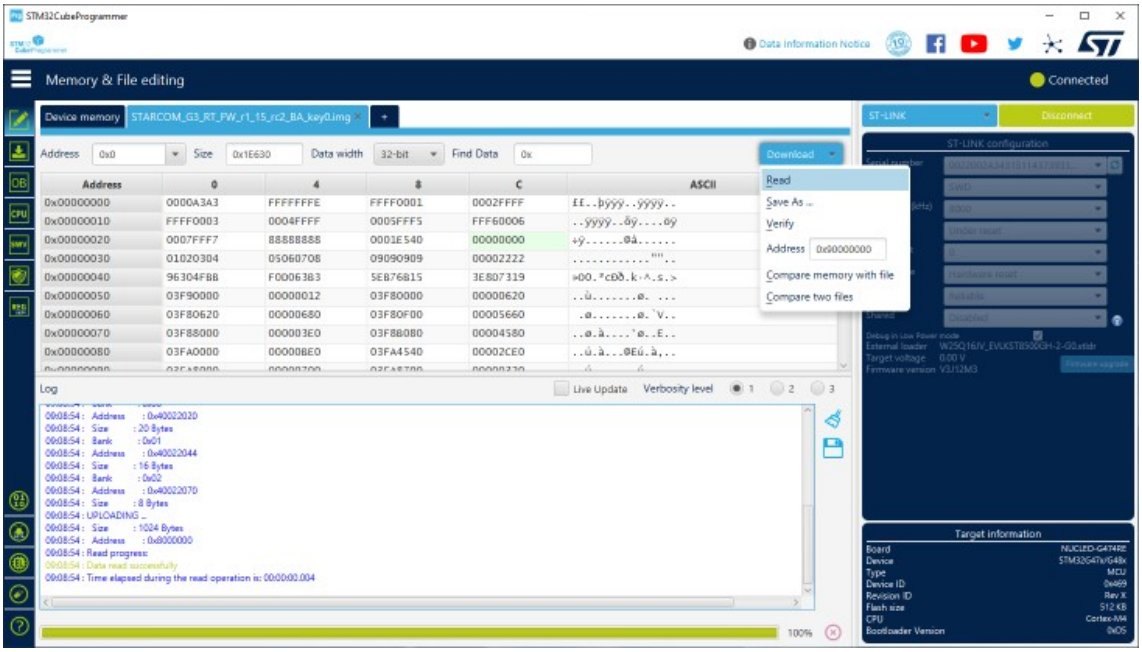


To add the *W25Q16JV_EVLKST8500GH-2-XX* external loader to the list inside STM32CubeIDE, copy and paste the *W25Q16JV_EVLKST8500GH-2-XX.stldr* file from the *ExtenalLoader* folder of the package to the *\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\ExternalLoader* folder. Once the file has been copied, follow these steps to program the STM32 SPI Flash memory:

1. Open STM32CubeProgrammer.
2. In the External loaders menu, select the *W25Q16JV_EVLKST8500GH-2-XX* external loader.
3. Connect to the STM32 via ST-LINK by clicking on "Connect".

4. In the Memory & File edition menu, open the RTE image file *<filename>.img*, change its address to 0x90000000, and then click “Download”.

Figure 10. STM32Cube Programmer Memory & File edition menu



5. In the Memory & File edition menu, open the PE image file *<filename>.img*, change its address to 0x90040000, and then click “Download”.
6. Disconnect from the STM32.

Once this procedure has been completed, the STM32 SPI Flash memory shall contain the RTE image at address 0x0 and the PE image at address 0x40000.

Revision history

Table 3. **Document revision history**

Date	Version	Changes
19-Oct-2021	1	Initial release.
15-Mar-2022	2	Revisited document for the V1.5.0 software package.
25-Jan-2023	3	Revisited document for the V1.7.5 software package.
07-Sep-2023	4	Revisited document for the V2.0.6 software package (STSW-ST8500GH-2) and the EVLKST8500GH-2 kit.
07-Nov-2023	5	Revisited document for the V2.3.0 software package.

Contents

List of tables

List of figures