AN5715

Application note

G3-PLC Hybrid PLC & RF SW package

Introduction

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

The package provides the software ecosystem for ST's G3-PLC Hybrid PLC & RF connectivity technology evaluation, based on the ELVKST8500GH868, EVLKST8500GH915 kits that include all the functions required for plug-and-play communication networking.

The host controller application firmware example for STM32G070RB, based on FreeRTOS, allows testing the PLC and RF communication exploiting the IPv6 layer interface of the ST8500 modem. The G3-PLC Hybrid PLC & RF communication stack has full flexibility to be configured in any of the available bandplans for both PLC (CEN-A, CEN-B or FCC) and RF (according to the RF Sub-GHz module selection).

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 ST hybrid PLC & RF solution is based on open standards and enables seamless integration into existing G3-PLC networks and adoption in multiple applications and systems.

1 G3-PLC Hybrid PLC & RF software solution overview

1.1 What is the G3-PLC Hybrid PLC & RF?

The G3-PLC 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 and wireless media.

The 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.

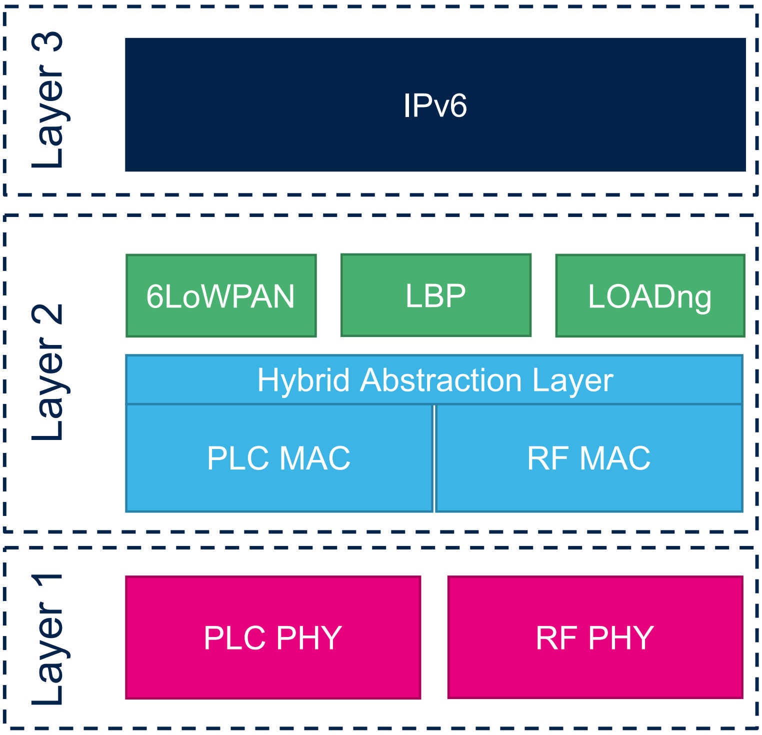
In many network conditions, none of the communication technologies can consistently achieve connectivity >99% on their own. The hybrid solution maximizes coverage and connectivity, avoiding the high cost that would be associated to deployment of a different solution to achieve the remaining 1% connectivity.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

The G3-PLC 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 Protocol basics

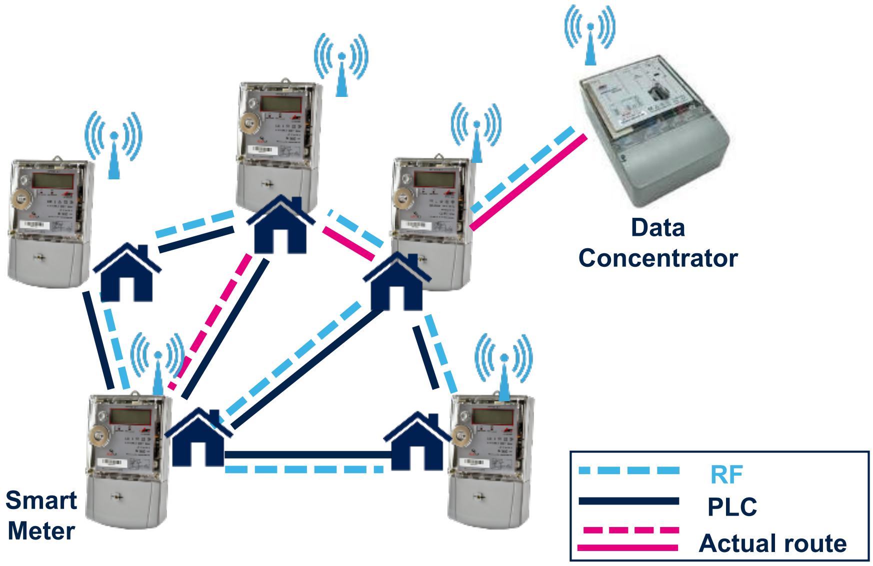
Figure 1. **G3 Hybrid PLC & RF protocol stack**



The G3-PLC Hybrid PLC & RF 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 G3-PLC 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.

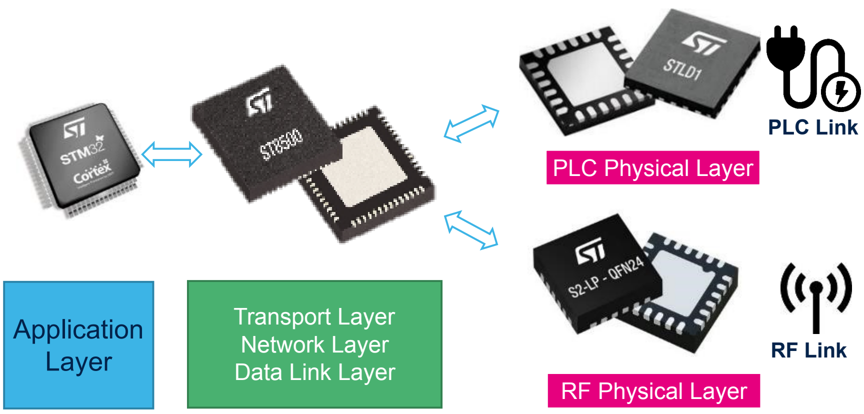
The principle of operation is quite simple: each node in a network having hybrid connectivity decides which medium to use to reach better performances and coverage, realizing a hop-by-hop automatic selection which is totally transparent to the end user. At the same time, the high flexibility of the implementation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

Figure 2. **Hybrid network example**



1.3 Supported Hardware and evaluation boards

Figure 3. **ST chipset implementing the G3-PLC Hybrid PLC & RF protocol stack**



The ST G3-PLC Hybrid PLC & RF is based on three main devices (ST8500, STLD1 and S2-LP) building together the G3-PLC Hybrid PLC & RF modem, 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-PLC Hybrid PLC & RF stack down to layer 2 (MAC layer) and the PHY layer of the PLC medium.
- 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.

The ST G3-PLC Hybrid PLC & RF implementation can run on several evaluation and development kits. Listed below are the ones where the STSW-ST8500GH STM32 firmware described in this document can be used without adaptation. Although the EVALKITST8500-1 can run the same protocol stack in PLC only mode, the STM32 code provided cannot be used without an adaptation.

Table 1. ST8500 G3-PLC Hybrid PLC & RF supported boards

Part number	Hybrid support	PLC connectivity	RF connectivity
EVLKST8500GH868	YES	0-500 kHz	863-870 MHz
EVLKST8500GH915	YES	0-500 kHz	860-940 MHz

2 The G3-PLC Hybrid PLC & RF software package

The STSW-ST8500GH contains a software package for the STM32G070RB microcontroller based on STM32CubeMx. The structure of the package is described hereafter (please refer to the specific release note to check if there are changes in your version).

- NUCLEO-G070RB
  - Documents
  - Drivers
  - EWARM
  - G3_Applications
  - Inc
  - Middlewares
  - Src
  - User_Applications
  - NUCLEO_G070RB.ioc

The file *NUCLEO_G070RB.ioc* is the STM32CubeMx project file. It can be used 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 *Documents* folder includes a quick start guide on the User application and the release note of the package; the user should check this document to see what changes have occurred since the last official release.

The *Drivers* folder contains CMSIS and STM32 HAL (Hardware Abstraction Layer) files for the selected platform.

The *EWARM* folder contains the IAR project; to know the version of the IAR toolchain used you can refer to the Release Note.

The *G3_Applications* folder contains all the protocol specific firmware, comprising the API and the G3-PLC applications.

The *Inc* and *Src* folders contain the source code generated by the STM32CubeMx project and framework code, while the *Middlewares* folder contains the FreeRTOS specific source files.

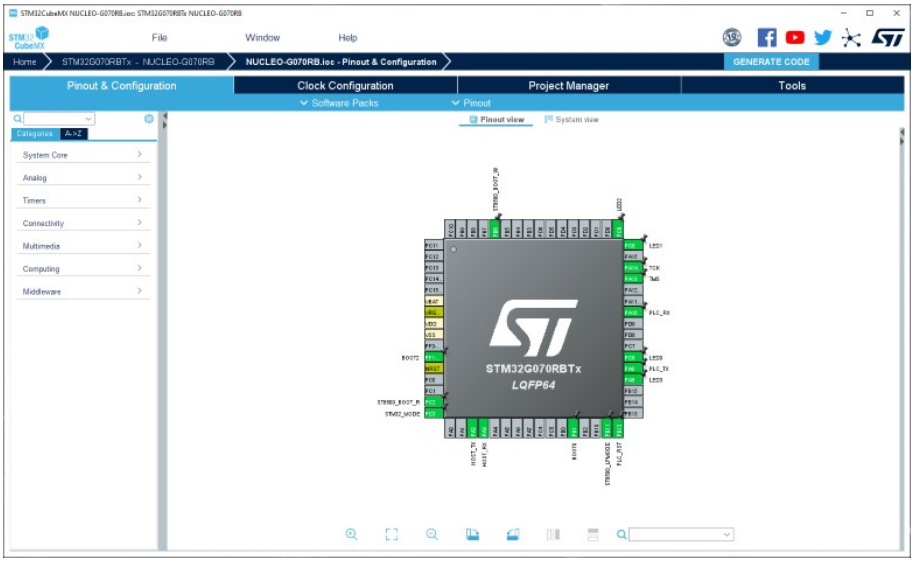
The *User_Applications* folder contains the files related to an example of user application: a UDP responder implementation with its FreeRTOS task and a serial terminal handling.

3 STM32CubeMX project

3.1 Project description

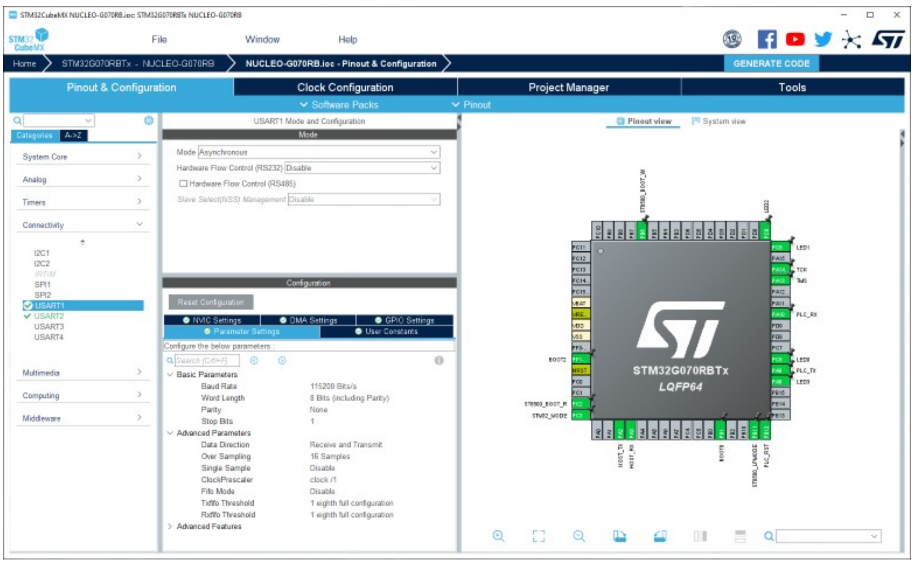
At first opening, the STM32CubeMx project shows, on the left side, the list of categories the settings are referring to, while on the right panel the default view is on the STM32 pinout. By moving on the active pins (the ones not grayed), it is possible to see how they are configured and, by clicking, the alternative settings are shown.

Figure 4. STM32CubeMx project initial view



To see/change the configuration of each peripheral, a side panel will be 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

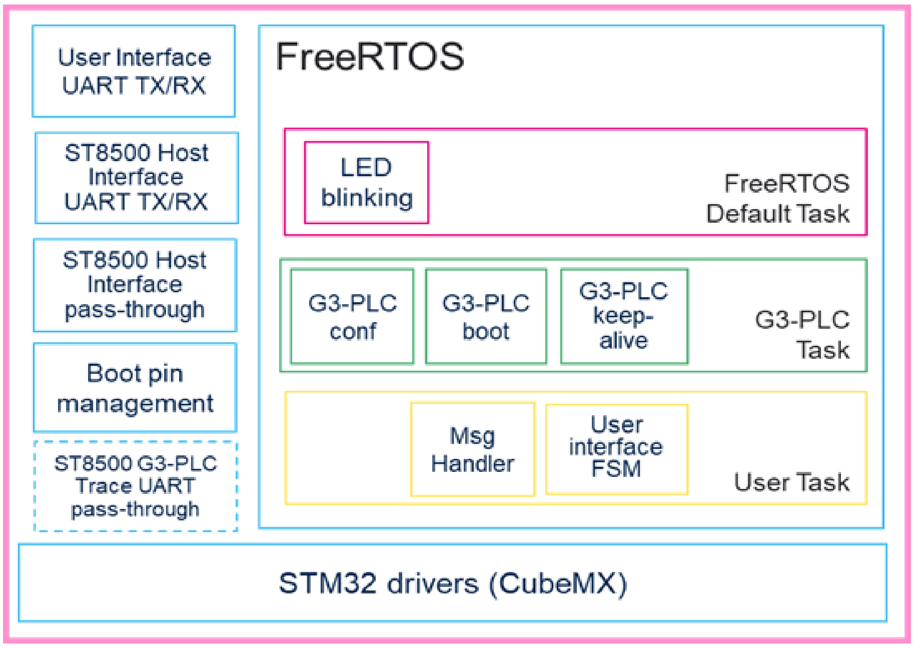


Any information about the STM32CubeMx ecosystem can be found on the st.com website www.st.com (https://www.st.com/content/st_com/en/stm32cube-ecosystem.html).

3.2 FreeRTOS subsystem

The FreeRTOS middleware is generated automatically by the STM32CubeMx IDE and the resulting FW structure is shown in Figure 6 .

Figure 6. G3-PLC Hybrid PLC & RF SW package architecture

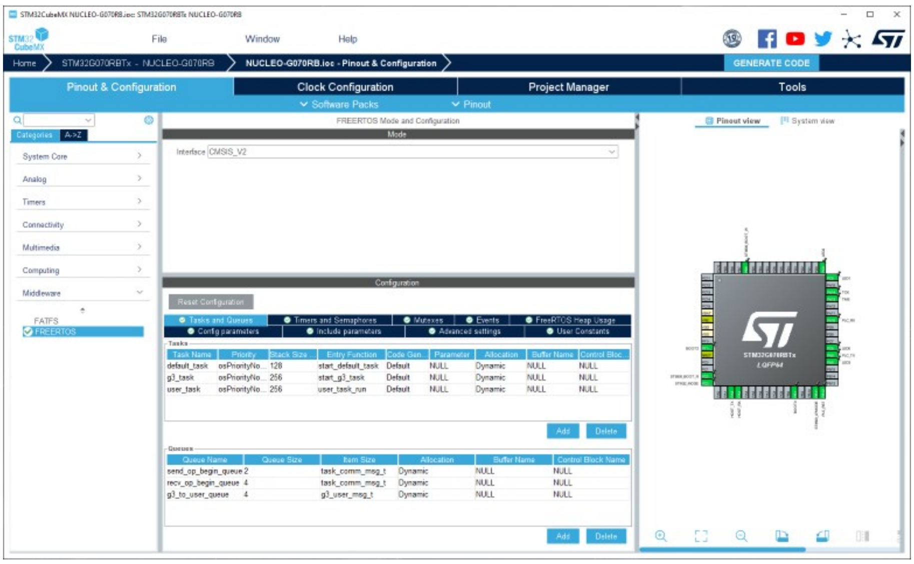


The STM32CubeMX generates the STM32 drivers and HAL and configures the middleware (in this case, the FreeRTOS).

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

For this project, the firmware package STM32Cube_FW_G0_V1.3.0 has been used as a starting point.

Figure 7. STM32CubeMx FreeRTOS configuration panel



The G3-PLC Hybrid PLC & RF SW project makes use of the CMSIS_V2 interface and generates three tasks:

- the FreeRTOS Default task that blinks an LED to signal it is working;

- the FreeRTOS G3-PLC task that handles the PLC communication. A complete description of the *g3_task* is given in Section 4 .
- the FreeRTOS User task that handles an example of UDP responder and a serial terminal. A complete description of the *user_task* is given in Section 5 .

3.3 UART interface modules

The G3-PLC Hybrid PLC & RF SW project includes several modules related to UART communication between the ST8500, the STM32 and the PC:

- ST8500 Host Interface UART TX/RX
- ST8500 Host Interface pass-through
- User Interface UART TX/RX.

The first two modules enable the control of the ST8500 Host Interface, that is the UART interface of the ST8500 implementing commands related to the G3-PLC hybrid protocol stack.

The ST8500 Host Interface UART TX/RX module basically takes care of the low level UART messaging between the STM32 and the ST8500; it is implemented in the file *g3_hif.c*.

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

- *g3_hif_uart_rx_handler*, called inside the UART Rx ISR, decodes a message coming from the ST8500 Host Interface then queues the message for the G3 Task when the UART reception has ended and the message has been verified;
- *hif_uart_send_message*, called from the G3 Task main loop, which initiates the transmission of a message to the ST8500 Host Interface
- *g3_hif_uart_tx_handler*, called inside the UART Tx ISR, feeding the UART with the content of the message to be transmitted to the ST8500 Host Interface.

The ST8500 Host Interface pass-through module gives direct access to the ST8500 Host Interface from the PC via the NUCLEO-G070RB USB connector. When activated through a specific switch on the board, this module takes complete control of the STM32, reconfiguring the pins of the two UARTs (the one connected to the ST8500 and the one connected to the USB) as GPIOs, to provide a transparent link. The serial communication is forwarded from the USB to the ST8500 by the STM32 with an EXTI (External Interrupt) mechanism that controls the output GPIOs (former UART TX pins) whenever the input GPIOs change (former UART RX pins).

The User Interface UART TX/RX module controls the UART dedicated to interaction between the PC and the STM32 User Task. The User Interface UART is connected to the PC through the ST-LINK USB on the NUCLEO-G070RB board.

3.4 Task communication mechanism

Task communication between the G3 Task and the ST8500 Host Interface is achieved by two queues named *send_op_begin_queue* and *recv_op_begin_queue.* These are monodirectional queues, and the usage is as follows:

- *send_op_begin_queue* is used by the G3 Task to queue commands to be sent to the ST8500, the access functions are:
  - *task_comm_put_send_buffer*, to queue a message
  - *task_comm_get_send_buffer*, to dequeue a message
- *recv_op_begin_queue* is used by the ST8500 Host Interface to send messages to the G3 Task; the access functions are:
  - *task_comm_put_recv_buffer*, to queue a message
  - *task_comm_get_recv_buffer*, to dequeue a message

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

- *osMessageQueueNew*
- *osMessageQueuePut*
- *osMessageQueueGet*

The message format is defined by the following structure:

```
typedef struct {
    g3_msg_ids_t msg_id;
    uint8_t cmd_id;
    void *buff;
    size_t size;
} task_comm_msg_t;
```

msg_id is a parameter which indicates if the message is related to the G3 Initialization (*G3_INIT*) or to the ST8500 Host Interface (*G3_PROT*); *cmd_id* is the command code valid for the ST8500 Host Interface, meaningful only for *G3_PROT* messages; *buff* is the pointer to the data associated with the command; *size* is the length of *buff* in bytes.

To avoid deep copy operations in the ISR, which could lead to data loss at higher baud rates, or with low speed STM32 devices, shared buffers are used to exchange data between the UART and the G3-PLC task; these buffers are dynamically allocated through a specific module (*mem_pool.c* and *mem_pool.h*). Four 2048-byte shared buffers are defined; to allocate and deallocate the buffers the following interface functions are defined (see file *mem_pool.h*):

- *Mem_Alloc*, to allocate a buffer
- *Mem_Free*, to deallocate the buffer
- *Mem_FreeAll*, to deallocate all the buffers and free the entire memory pool

Note: *mem_pool* has only been used for storing data of received messages (i.e. from the ST8500 Host Interface to the G3-PLC Task). Transmitted message data, needing more complex management, are stored using static memory allocation. Extending *mem_pool* management to transmitted message data could be a possible optimization.

4 G3-PLC Applications

4.1 Introduction

The G3-PLC task implements three modules:
- G3-PLC Config module
- G3-PLC Boot module
- G3-PLC Keep-alive module

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

4.2 The G3-PLC Config module

The G3-PLC Hybrid PLC & RF node can be configured on several parameters: the role of the node in the network (PAN Device or PAN 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 startup 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-PLC 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/HW reset event, the host waits until the ST8500 notifies the end of NVM reconfiguration by sending the message *HOSTIF-HWRESET.Confirm* on the Host Interface.

The G3-PLC Config module takes care of customizing the configuration of the node in three steps:
1. G3-PLC Attributes table initialization
2. G3-PLC Platform configuration
3. G3-PLC Attributes configuration

4.2.1 Attributes table initialization

The list of parameters to be configured is stored into a static array of type *G3_LIB_PIB_t*; the Attributes table is defined in the file *g3_app_attrib_tbl.c* as follows:

```
static G3_LIB_PIB_t g3_app_attrib_tbl[] = {
#ifdef DEVICE_TYPE_COORD
    {.Attribute = {.ID = MAC_PANID_ID, .Index = 0}, .Len = 2, .Value = {0x57U, 0x8CU}},
    {.Attribute = {.ID = MAC_SHORTADDRESS_ID, .Index = 0}, .Len = 2, .Value = {0x00U, 0x00U}}
#else
    {.Attribute = {.ID = ADP_EAPPSKKEY_ID, .Index = 0}, .Len = 16, .Value = DEFAULT_PSK}
#endif
};
```

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

The user can easily extend this configuration to adapt to their needs, by adding rows to the two tables. During the node startup, this table is prepared to be used by the G3-PLC Attribute configuration described below.

4.2.2 G3-PLC Platform configuration

The G3-PLC Hybrid PLC & RF stack must be properly initialized and configured before starting the operations. The G3-PLC config module performs this operation at the startup of the G3-PLC task, in the *g3_app_conf_start* function.

```
void g3_app_conf_start(void)
{
    static G3_LIB_SWResetRequest_t sw_reset_req;

    hi_g3lib_swresetreq_set(    &sw_reset_req,
                PROT_CENA_BANDPLAN,
                dev_type,
                HI_CFG_IPV6_BOOT_MODE);

    task_comm_put_send_buffer(    G3_PROT,
                    HIF_G3LIB_SWRESET_REQ,
                    &sw_reset_req,
                    sizeof(sw_reset_req),
                    NO_WAIT);
}
```

The *hi_g3lib_swresetreq_set* function accepts three parameters:

- The PLC operating band (Cenelec A, Cenelec B or FCC)
- The Device Type (PAN Coordinator or PAN Device)
- The Operating mode (*HI_PHY_MODE, HI_MAC_MODE, HI_ADP_MODE, HI_BOOT_MODE, HI_IPV6_ADP_MODE, HI_IPV6_BOOT_MODE*)

The G3-PLC task supports all the PLC operating bands and Device type, but it requires the modem to be set in *HI_IPV6_BOOT_MODE* to properly operate.

Upon reception of G3LIB-SWRESET.Confirm message on the ST8500 Host Interface, the configuration of the RF module is performed by the User Application task calling the *g3_conf_fsm_handle_swreset_cnf* function. It simply feeds the structure containing the RF config parameters using hard-coded values, depending on the selected RF module (X-NUCLEO-S2868A2 or EVALS2915A1/A2):

```
rfconfig_req.data = (s2lp_configdata_t) {
#ifdef RF_IS_868
    .RadioBaseFreq = 863100000U,
    .RadioModSelect = 0x00U,
    .RadioDataRate = 50000U,
    .RadioFreqDeviation = 12500U,
    .RadioBandwidth = 100000U,
    .RadioCsBlanking = 0x01U,
    .RadioXtalFreq = 50000000U,
    .RadioRssiGain = 14U,
    .PowerdBm = 35U,
    .PktCRCMode = 0xA0U,
    .PktEnFEC = 0x00U,
    .PktEnWhitening = 0x01U,
    .PktEnInterleaving = 0x01U,
    .IrqGpioPin = 0x00U,
    .MCUClkEnable = 0x00U,
    .MCUClkGpioPin = 0xFFU,
    .MCUClkXORatio = 0xFFU,
    .MCUClkRCORatio = 0xFFU,
    .MCUClkClockCycles = 0xFFU,
    .ExtSmpsEnable = 0x00U,
    .FEMEnabled = 0x00U,AN5715 - Rev 1 page 11/27
    .FEMGpioPinCSD = 0xFFU,
    .FEMGpioPinCPS = 0xFFU,
    .FEMGpioPinCTX = 0xFFU,
    .FEMTxBypassEn = 0x00U
#else /* RF_IS_915 */
    .RadioBaseFreq = 915000000U,
    .RadioModSelect = 0x00U,
    .RadioDataRate = 50000U,
    .RadioFreqDeviation = 12500U,
    .RadioBandwidth = 100000U,
    .RadioCsBlanking = 0x01U,
    .RadioXtalFreq = 50000000U,
    .RadioRssiGain = 14U,
    .PowerdBm = 45U, /* Modified for 915 */
    .PktCRCMode = 0xA0U,
    .PktEnFEC = 0x00U,
    .PktEnWhitening = 0x01U,
    .PktEnInterleaving = 0x01U,
    .IrqGpioPin = 0x03U, /* Modified for 915 */
    .MCUClkEnable = 0x00U,
    .MCUClkGpioPin = 0xFFU,
    .MCUClkXORatio = 0xFFU,
    .MCUClkRCORatio = 0xFFU,
    .MCUClkClockCycles = 0xFFU,
    .ExtSmpsEnable = 0x00U,
    .FEMEnabled = 0x01U, /* Modified for 915 */
    .FEMGpioPinCSD = 0x00U, /* Modified for 915 */
    .FEMGpioPinCPS = 0x01U, /* Modified for 915 */
    .FEMGpioPinCTX = 0x02U, /* Modified for 915 */
    .FEMTxBypassEn = 0x00U
#endif
};
```

Note that the frequency for the RF module is fixed by the selected configuration (both coordinator and device have 863.1 MHz option and 915 MHz option), so, to ensure that all nodes of the network can communicate via RF, it might be necessary to set a custom RF frequency for all nodes.

Please check the compatible frequency range for EVLKST8500GH868 and EVLKST8500GH915 in the respective documentation on st.com.

4.3 G3-PLC Attributes configuration

Once the G3-PLC task has started, and the operations in the NVM are concluded (after the reception of the *HOSTIF-HWRESET.Confirm* message from the ST8500 Host Interface), the G3-PLC Config module starts setting all the attributes found in the *g3_app_attrib_tbl[ ]* table, completing the configuration of the node.

4.4 The G3-PLC Boot module

The G3-PLC Boot module makes use of the G3-PLC 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 and (at least) one PAN Device.

4.4.1 PAN Device implementation

In G3-PLC Hybrid PLC & RF networks, at startup 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-Device-Pansort.Ind* message from the ST8500 Host Interface.

Since the PAN Device must decide which agent is used to join the network, it is up to the host application to sort the list of agents. The function *g3_app_pansort_handle_pansort_ind* sorts the list of Agents using the Route Cost to PAN Coordinator as metric and, at the end, sends back the ordered list of agents to the ST8500 Host Interface through the message *Boot-Device-Pansort-Req.*

```
static void g3_app_pansort_handle_pansort_ind(const void *msg)
{
    uint16_t len;
    static BOOT_DevicePANSortRequest_t pan_sort_req;
    const BOOT_DevicePANSortIndication_t *pansort_ind = msg;

    len = hi_boot_dev_pansort_req_set(    &pan_sort_req,
                    pansort_ind->PANCount,
                    pansort_ind->PANDescriptor);

    if (!pan_sort_req.PANCount)
        return;

    qsort(    pan_sort_req.PANDescriptor,
        pan_sort_req.PANCount,
        sizeof(pan_sort_req.PANDescriptor[0]),
        pan_descr_compare);

    task_comm_put_send_buffer(    G3_PROT,
                    HIF_BOOT_DEVICE_PANSORT_REQ,
                    &pan_sort_req,
                    len ,
                    NO_WAIT);
}
```

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

4.4.2 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.Ind* message on the ST8500 Host Interface to the G3-PLC Boot module. The G3-PLC 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 table *g3_app_boot_data_tbl*, which is hardcoded into the PAN Coordinator G3-PLC Boot module: when a PAN Device, whose Extended Address is listed in the table, tries to join the network, then the corresponding Short Address and PSK are assigned.

If the Extended Address of the PAN Device is not present in the table, then it is assigned a progressive Short Address and a default PSK.

In a real implementation, the dynamic allocation should be avoided, so that a malicious node is not allowed to access the network. Similarly, a hardcoded table is not the right choice to store this information and, for this purpose, the use of a secure element to store locally the table, or the use of a secure connection towards an authentication server, is recommended.

The access to the network is granted in the *g3_app_boot_handle_srv_getpsk_ind* function:

```
static void g3_app_boot_handle_srv_getpsk_ind(const void *msg)
{
    const uint8_t default_psk[] = DEFAULT_PSK;
    static BOOT_ServerSetPSKRequest_t setpsk_req;
    const g3_app_boot_data_t *boot_data;
    uint16_t len;
    const BOOT_ServerGetPSKIndication_t *getpsk_ind = msg;

    boot_data = g3_app_boot_tbl_find_elem(getpsk_ind->ExtendedAddress);

    if (!boot_data)
    {
        boot_data = g3_app_boot_temp_tbl_find_elem(getpsk_ind->ExtendedAddress);

        if (!boot_data)
            g3_app_boot_data_tbl_store_elem(    g3_app_boot_get_next_short_addr(),
                            getpsk_ind->ExtendedAddress,
                            default_psk);

        boot_data = g3_app_boot_temp_tbl_find_elem(getpsk_ind->ExtendedAddress);

        if (!boot_data)
            return;
    }

    len = hi_boot_srv_setpsk_req_set(    &setpsk_req,
                    boot_data->ext_addr,
                    boot_data->psk,
                    boot_data->short_addr);

    task_comm_put_send_buffer(    G3_PROT,
                    HIF_BOOT_SERVER_SETPSK_REQ,
                    &setpsk_req,
                    len,
                    NO_WAIT);
}
```

As a general hint, the user could:

- Modify the *g3_app_boot_tbl_find_elem* function, to retrieve the data from the secure source;
- Remove all the instructions inside the outermost *if* branch, leaving only a return statement, to avoid the automatic registration of unknown nodes;

4.5 The G3-PLC Keep-alive module

In a real network, the channel conditions can vary during uptime, and sometimes it may happen that a PAN Device stops being able to 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 (6 hours by default); 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-PLC 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.

4.5.1 PAN Device implementation

The Keep-alive module in the PAN Device keeps track of the last time a data communication with the PAN Coordinator has been successfully performed. If there is no communication for more than *KEEP_ALIVE_DEV_SIDE_TIMEOUT*, the device leaves the PAN and it retries the bootstrap procedure; this is managed by the *g3_app_ka_dev_check_timeout* function:

```
static void g3_app_ka_dev_check_timeout(void *unused)
{
    if (    ka_dev_info.ka_dev_timeout_active &&
        st_is_timeout_expired(ka_dev_info.ka_dev_timeout))
    {
        // we use a fake buffer address here to be able to check for errors (buff ==NULL)
        task_comm_put_send_buffer(    G3_PROT,
                        HIF_BOOT_DEVICE_LEAVE_REQ,
                        utils_get_fake_buff_addr(),
                        0,
                        NO_WAIT);

        ka_dev_info.ka_dev_timeout_active = false;
    }
}
```

4.5.2 PAN Coordinator implementation

The Keep-alive module in the PAN Coordinator makes use of the ICMP Echo feature to communicate periodically with each PAN Device, and the interval between two echo requests (*KEEP_ALIVE_CHECK_TIMEOUT_DELAY*) is configurable.

If there is no successful communication with a PAN Device for more than *KEEP_ALIVE_DEV_LAST_SEEN_TIMEOUT* (configurable), the PAN Coordinator forces an ICMP echo message to the PAN Device and, eventually, if the communication fails again, it stops pinging the device and sends a *HIF_BOOT_SERVER_KICK_REQ* to the ST8500 Host Interface, to kick the PAN Device out of the network. This is managed by the *g3_app_ka_panc_kick_dev* function:

```
static void g3_app_ka_panc_kick_dev(uint16_t short_addr)
{
    uint16_t len;
    ka_device_data_t *dev_data = g3_app_ka_device_find( &ka_panc_info.device_table,
                        short_addr);
    assert(dev_data != NULL);

    if (!dev_data)
        return;

    len = hi_boot_srv_kick_req_set(    &ka_panc_info.boot_kick_req,
                dev_data->short_addr,
                dev_data->extended_addr);

    task_comm_put_send_buffer(    G3_PROT,
                HIF_BOOT_SERVER_KICK_REQ,
                &ka_panc_info.boot_kick_req,
                len,
                NO_WAIT);

    if (     ka_panc_info.currently_checking_device &&
        (short_addr == ka_panc_info.ka_panc_ping_dst_short_addr) )
    {
        g3_app_ka_ping_stop();
    }

    g3_app_ka_device_remove(&ka_panc_info.device_table, dev_data);
}
```

5 User Applications

5.1 Introduction

A specific task, named User task, has been defined to handle the user application. The corresponding files are in the *User_Application* folder. The User task implements an example of UDP responder launched via a serial terminal.

This section describes the User task and explains how it is handled.

5.2 User task interface with the ST8500 Host Interface and the G3-PLC task

The User task can send messages to the ST8500 Host Interface. To do this, it uses the monodirectional queue *send_op_begin_queue*. This queue is then used by both the G3-PLC task and the User task.

When the G3-PLC task receives some messages from the ST8500, they are forwarded to the User task, if needed, via the *g3_to_user_queue*. The messages from the G3-PLC task to the User task have the following format:

```
typedef struct
{
    uint8_t cmd_id;
    void *buff;
    size_t size;
} g3_user_msg_t;
```

The *cmd_id*, *buff* and *size* are the same as the ones explained in Section 3.4 .

5.3 User task interface with the serial terminal

The User task interacts with the user via any PC serial terminal (like Teraterm) using the User Interface UART RX/TX port (i.e. STM32G070RB USART2 as defined in CubeMX). RX and TX messages are managed inside the *user_term_rxtx.c* file.

5.3.1 RX communication (from terminal to User task)

Bytes received from UART are first packetized to create a command message. Its content is defined by *User_Term_CmdBody_t*:

```
typedef struct
{
    uint32_t Type;
    uint32_t Length;
    uint8_t Payload[TERM_CMD_PAYLD_MAX_SIZE];
} User_Term_CmdBody_t;
```

TERM_CMD_PAYLD_MAX_SIZE gives the maximum length for one string to be sent from terminal (set to 100).

To be considered as valid, a string shall end with a CR or LF character.

Once created, received command messages are buffered so that they can be processed by the User task. The buffer is defined by *User_Term_RxFifo_t*:

```
typedef struct
{
    uint32_t ReadIdx;
    uint32_t WriteIdx;
    User_Term_CmdBody_t Data[TERM_RX_FIFO_SIZE];
} User_Term_RxFifo_t;
```

Buffer size is defined to *TERM_RX_FIFO_SIZE* (= 8).

Command message packetization and RX buffer feeding are handled under UART ISR.

5.3.2 TX communication (from User task to terminal)

The User task can send data to terminal by either using *User_Term_Printf()* function (sending character strings to be displayed on terminal) or *User_Term_Printb()* function (sending bytes to be displayed on terminal).

These functions feed data bytes in a buffer defined by *User_Term_TxFifo_t*:

```
typedef struct
{
    uint32_t ReadIdx;
    uint32_t WriteIdx;
    uint8_t Data[TERM_TX_FIFO_SIZE];
} User_Term_TxFifo_t;
```

Buffer size is defined to *TERM_TX_FIFO_SIZE* (= 512).

Bytes from TX buffer are then sent over UART through UART ISR.

5.4 User task internals

5.4.1 User interface FSM

The User interface FSM is implemented in *user_if_fsm.c*. Its main goal is to manage the serial terminal display and various actions in the system (e.g. interactions with the G3-PLC layer), depending on terminal user inputs on one side and events coming from the G3-PLC task and the ST8500 Host Interface on the other side.

The actual implementation ensures:

- User terminal menu navigation in *UserIf_Fsm_FsmBody()* function and its state sub-functions
- Various test executions (i.e. UDP responder) via its *UserIf_Fsm_XxxExec()* functions
- The display of User task events (e.g. from G3-PLC task) by calling *UserIf_Fsm_ParseUserEvents()*. The management of such asynchronous events is independent from FSM current state and execution.

The implementation FSM 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).

5.4.2 G3-PLC task and ST8500 Host Interface management

The User task interface management with the ST8500 Host Interface and the G3-PLC task is implemented in *user_g3_common.c*. This interface management ensures:

- The parsing of incoming messages from G3-PLC task using the *UserG3_MsgHandler()* function;
- The further processing of the incoming messages from G3-PLC task using the *UserG3_HandleXxx()* sub-functions;
- The implementation of some specific commands to be sent to the ST8500 Host Interface (e.g. UDP data transfer request commands);
- The implementation of an API offering generic G3-PLC services:
  - A set of functions enabling specific executions, such as *UserG3_StartUdpDataTransfer()*, that can be used for example by the User interface FSM;
  - A structure named *UserG3_CommonEventData_t* gathering data to be made available to the User interface FSM (User/G3-PLC events, upcoming UDP transfer content).

5.5 UDP responder example

One goal of the User task is to give an example of a UDP responder handled via a serial terminal.

To establish a UDP transfer, the following data are needed:

- Local port: it is fixed in this UDP responder example;

- Remote port: it is fixed in this UDP responder example;
  - The recipient IPv6 address:
    - The Coordinator gets the IPv6 address of the Device from the short address in the *G3BOOT-SRV-JOIN.Indication* message received from the G3-PLC task and forwarded to the User task.
    - The Device gets the Coordinator IPv6 address from *G3ICMP-ECHO.Request* -> *G3ICMP-ECHOREQ.Indication* message received from the G3-PLC task and forwarded to the User task.

To be able to make a UDP transfer, some messages should be sent to the ST8500 Host Interface in a specific order. After guiding the user through terminal menus, the *UserIf_Fsm_FsmBody()* function sequences the orders using its *Exec* sub-functions, such as *UserIf_Fsm_UdpTestSingleDataReqOriginatorExec()*, so that it prepares the UDP connection and launches it. The FSM function displays in parallel the required information on the serial terminal (e.g. *G3UDP-DATA.Ind* event and its data content).

Several types of tests can be handled: simple UDP responder, UDP data loopback example, etc.

In response to the requests, the ST8500 Host Interface sends back some confirm/indication messages (e.g. *G3UDP-CONN-SET.Cnf*, *G3ICMP-ECHO.Ind*, *G3UDP-DATA.Ind*). These are forwarded to the User task and treated in the *UserG3_MsgHandler()* function. Those messages are displayed as User/G3-PLC events in the serial terminal.

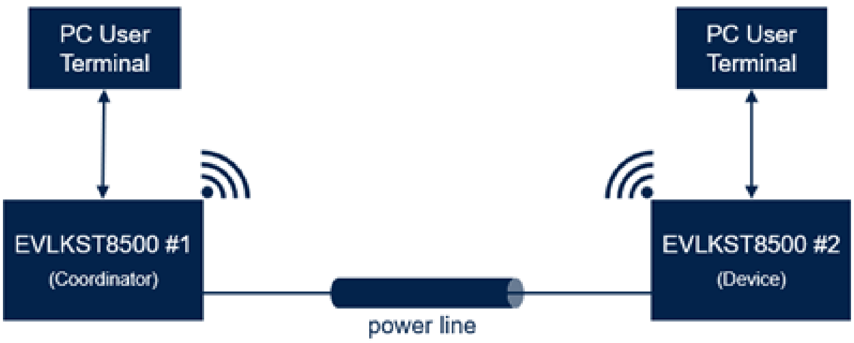
UDP data transfer sequencing operates as follows: once the bootstrap operation is done and the needed data are captured (in particular, IPv6 addresses), a connection is established via a *G3UDP-CONN-SET.Request* on both sender (Local) and receiver (Remote) side. After that, a UDP transfer is launched via a *G3UDP-DATA.Request* on sender side.

5.6 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 launchable tests/demo. It also displays upcoming specific events such as G3-PLC network bootstrap messages. A serial terminal software must be available on the PC side and a USB cable is connected between the EVLKST8500GH 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



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

6 Extending the G3-PLC 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 keep the compatibility with the STM32CubeMx environment, and allow automatic regeneration of the source code, the user should add to or 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 source files contained inside *Src* that are not generated by CubeMX (like *crc*, *mem_pool*, *soft_timer*, *st8500_handler*, *task_comm*, *timed_events* and *utils*) and any source file added by the user themselves.

6.1 Extending G3-PLC Task with additional features

To extend the G3-PLC Task with an additional module, the user should take care of defining:

- An initialization function (if needed) to be called from the *g3_task_init* function
- A function implementing the module state machine, to be called inside the endless loop of the function *g3_app_task*

Should the new module request the usage of ST8500 Host Interface commands not to be already managed, the hints provided in Section 6.2 should be applied. Otherwise, the helper functions defined in the *hi_msg_impl.c* file and the ST8500 Host Interface message parser functions already defined, should be amended to support the new module.

6.2 Managing ST8500 G3-PLC Host Interface messages

The addition of new features and new tasks could require that the user implement additional helper functions to ease the preparation of commands to be sent to, and to decode the messages coming from, the ST8500 G3-PLC Host Interface. To do that, the user should modify *hi_msgs_impl.c* with the required functions.

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 (as described in AN5603 - ST G3-PLC Hybrid solution) 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_echo_req_set*, which is reported here below. This function accepts as parameters:

- *msg_*, the pointer to a buffer which contains the command sequence to be sent to the ST8500 Host Interface
- *dst_addr*, the IPv6 address of the destination node
- *handle*, a user-defined ID which should be used to correlate the request with the answer
- *data_len*, the length of the payload associated with the echo message

- *data*, the buffer containing the payload associated with the echo message

```
uint16_t hi_ipv6_echo_req_set(    void *msg_, const ip6_addr_t *dst_addr, uint8_t
                handle, uint16_t data_len, const uint8_t *data)
{
    IP_G3IcmpDataRequest_t *msg = (IP_G3IcmpDataRequest_t *)msg_;
    msg->DestAddress = *dst_addr;
    msg->Handle = handle;
    msg->DataLen = data_len;

    if (data_len <= sizeof((IP_G3IcmpDataRequest_t *)0)->Data)
        memcpy(msg->Data, data, data_len);
    else
        UTILS_HANDLE_ERROR_AND_RETURN_VAL(NULL,
                    "hi_ipv6_echo_request: data_len value too high", 0);

    return G3_ICMP_DATA_REQ_CURR_LEN(msg);
}
```

An example of how this helper function should be used is given in the function *g3_app_ka_fsm_send_echo_req*.

```
static ka_panc_state_t g3_app_ka_fsm_send_echo_req(ka_panc_event_t event)
{
    ip6_addr_t ip_dst_addr;
    uint16_t req_len;
    static IP_G3IcmpDataRequest_t icmp_data_req;

    ka_panc_info.ka_panc_event = KA_EV_PANC_NO_EVENT;

    hi_ipv6_set_ipaddr(    &ip_dst_addr, ka_info.pan_id,
                ka_panc_info.ka_panc_ping_dst_short_addr);

    req_len = hi_ipv6_echo_req_set(    &icmp_data_req, &ip_dst_addr,
                    ka_panc_info.ping_handle++,
                    sizeof(g3_app_ka_data), g3_app_ka_data);

    task_comm_put_send_buffer(    G3_PROT,
                    HIF_ICMP_ECHO_REQ,
                    &icmp_data_req,
                    req_len,
                    NO_WAIT);

    st_set_timer(SOFT_TIMER_ID_G3APP_KA_PAN, KA_PAN_PING_TIMEOUT);

    return KA_ST_PANC_WAIT_FOR_ECHO_CNF;
}
```

The helper function *hi_ipv6_echo_req_set* is called to prepare the command, then the *task_comm_put_send_buffer* function is used to push the command to the ST8500 Host Interface.

To manage incoming messages from the ST8500 Host Interface, such as indications or confirm messages, the user should extend the function *g3_app_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 plus the one for the PAN Sorting) are already defined.

```
static void g3_app_msg_handler(const msg_t *msg)
{
    const void *msg_payload = hif_rx_get_msg_payload(msg->buff, NULL);

    if (g3_app_conf_msg_needed(msg)) {
        g3_app_conf_msg_handler(msg->cmd_id, msg_payload);
        g3_send_msg_to_user_task(msg->cmd_id,(void *)msg_payload,msg->size);
    }

    if (g3_app_pansort_msg_needed(msg)) {AN5715 - Rev 1 page 20/27
        g3_app_pansort_msg_handler(msg->cmd_id, msg_payload);
        g3_send_msg_to_user_task(msg->cmd_id,(void *)msg_payload,msg->size);
    }

    if (g3_app_boot_msg_needed(msg)) {
        g3_app_boot_msg_handler(msg->cmd_id, msg_payload);
        g3_send_msg_to_user_task(msg->cmd_id,(void *)msg_payload,msg->size);
    }

    if (g3_app_ka_msg_needed(msg)) {
        g3_app_ka_msg_handler(msg->cmd_id, msg_payload);
        g3_send_msg_to_user_task(msg->cmd_id,(void *)msg_payload,msg->size);
    }

    task_comm_put_recv_used_buffer(msg, NO_WAIT);
}
```

6.3 Adding a user-defined task

The user can easily add one or more user-defined tasks to this example and the way the G3-PLC 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-PLC Hybrid PLC & RF protocol, the user should:

- Extend the G3-PLC Task with an additional module, to manage communication between the User Task and the ST8500 G3-PLC Host Interface (see section 6.1)
  – Extend the support of ST8500 G3-PLC Host Interface commands (see section 6.2)
- Implement the User Task and provide a messaging service between the User Task and the G3-PLC Task, possibly using the *task_comm* interface to handle communication between tasks (the format and messages to be exchanged are out of the scope of this document)
- In case the communication between the tasks involves the exchange of a large amount of data, implement a memory pool infrastructure (or use the CMSIS_V2 *mem_pool* feature)

Revision history

Table 3. **Document revision history**

Date	Version	Changes
19-Oct-2021	1	Initial release.

Contents

List of tables

List of figures