EuroCPS - SmartSSL

Milestone #3 Technical Documentation

Table of contents

1. Purpose and scope of this document

This document includes the technical documentation of deliverable items up to D3.1 as defined in EuroCPS Deliverable D3.2.

1.1. List of abbreviations

API	Application Programming Interface
CCU	Central Computing Unit
DALI	Digital Addressable Lighting Interface
DPS	DALI Power Supply
DU	Dispatcher Unit
if-CCU-LDCU	Interface between the Central Computing Unit and the LED DAQ and Control Unit
if-CCU-MSM	Interface between the Central Computing Unit and The Maintenance Subnetwork Modem
if-CCU-SU	Interface between the Central Computing Unti and the Sensor Unit
if-DU-IN	Interface between the Dispatcher Unit and the Internet
if-DU-MS	Interface between the Dispatcher Unit and the Maintenance Subnetwork
if-LDCU-DPS	Interface between the LED DAQ and Control Unit and the DALI Power Supply
if-PU-MS	Interface between the Pylon Unit and the Maintenance Subnetwork
if-PU-SS	Interface between the Pylon Unit and the Smart city Subnetwork
IN	Internet
LED	Light Emitting Diode
LDCU	LED DAQ and Control Unit
MS	Maintenance Subnetwork
MSM	Maintenance Subnetwork Modem
NC	Network Coordinator
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PLC	Power Line Communication
POR	Power On Reset
PPP	Point-to-Point Protocol
PU	Pylon Unit
RF	Radio Frequency
RS	Remote Station
SU	Sensor Unit
SSL	Solid State Lighting

2. Introduction

The problem of today's high degree of "LEDification" of lighting installations is that it still happens in a "retrofit" manner. Though luminaires designed for conventional light sources (incandescent bulbs, CFLs, gas discharge lamps) are replaced by LED luminaires, their physical operating environments (i.e. the mechanical supports, the power supply, and the applied control system) remain untouched, therefore one cannot take full advantage of applying LEDs. Even energy saving can be further increased with smart control of LEDs considering the actual temperature and LED characteristics and maintenance costs could be further reduced if more information about the LED luminaires and their physical environment were available. The objectives are as follows:

- Renew LED based street-lighting solutions with improved/extended communications (detection of human presence, improved adaptation to ambient conditions) especially dedicated to small rural communities, where improved safety and significantly reduced operating costs are key factors to justify investment into new street-lighting installations. Due to the limited resources of smaller communities the option of gradual introduction of new features in new lighting installations is also an important aspect.
- Introduce the self-diagnostics and intelligent LED control functions implemented both for street-lighting and for indoor solutions such as smart office lighting.
- Be a pilot implementation of self-identification in order to allow complete life cycle traceability of every individual module – this allows charging recycling costs of the product manufacturer. This is important to reduce the flood of cheap, low-quality Asian products on the European market where the recycling costs are not included in the price – providing an unfair market advantage to the manufacturers of such products.
- Develop an intelligent LED driver which allows
    - advanced self-diagnostics functions for the entire LED luminaires through an appropriate communications interface.
    - adjustment of the LEDs' electrical operating point based on the actual environmental temperature in order to provide extra energy saving (reducing the current when efficiency increases due to the lower temperature) and increased safety/reliability of operation (limiting the forward current at high temperatures to protect the LEDs).
- Develop a central control unit for the luminaires with a sensor interface and communications interface with an architecture inspired by the principles of the OSI model, allowing high degree of independence of the physical medium available for data transmission and providing flexibility in the actual communications protocol being used; allowing street-lighting operators to tailor the smart LED luminaires to their existing infrastructure or re-configuring (updating) the luminaires when the infrastructure is upgraded.

3. System architecture

3.1. Network topology

As an answer to the arising needs the network topology depicted in Fig. 1 has been proposed.

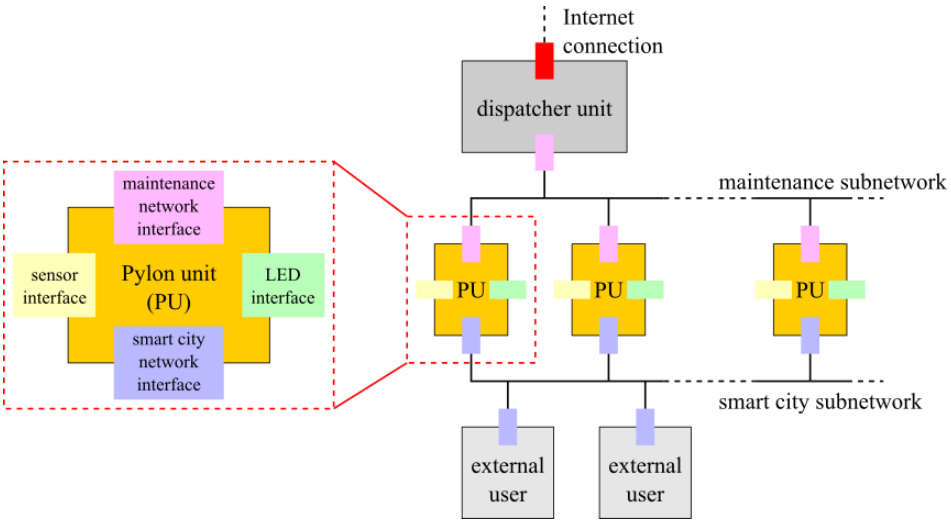


Fig. 1. The overall network topology with the nodes' internal and external interfaces.

In this dual network topology, two physically independent networks are constructed. The specialty of this topology is that the subnetworks share their nodes (Pylon Units, PUs) and these nodes eventually establish a logical connection between the subnetworks in a controllable manner. The roles of the subnetworks are the following:

- **Maintenance subnetwork**: It ensures remote access to the luminaires for a lighting dispatcher. The first objective is to provide the lighting dispatcher with self-diagnostics and identification information and make it possible to control the operation of the luminaires. It is economically more favorable to exploit the already installed infrastructure than introduce new ones, therefore, there are two possible ways for the maintenance subnetwork's physical layer implementation; wireless communication or power line communication (PLC [1]).
- **Smart city subnetwork**: By implementing widely-used communication interfaces, the PUs may be used to form subnetworks supporting smart-city applications.

As it may be seen in Fig. 1, the shared nodes implement two abstract interfaces implementing additional private functionalities. The sensor interface is used to connect sensors to the PUs. These sensors are used for data acquisition regarding the ambient conditions (e.g. temperature). The LED interface implements the lower levels of control services with regard to intelligent LED driving. All these interfaces follow the principles of the OSI model. The uniqueness and flexibility of this system architecture lies in the fact that the above mentioned interfaces are implemented by the same computation unit inside the PUs. That means that the different interface protocol stacks may share their application layer (see Fig. 2), which, as a result of the well-abstracted protocol structure, has a comprehensive access to the system resources, no matter, what they may be, including low-level LED driver protocols, sensors, and networks. The first demonstrational realization of the above

described system utilizes well-known technologies regarding physical layers. However, the arising new technologies may inspire new services with special demands that today's communication standards cannot cope with, justifying and necessitating the layered protocol structure.

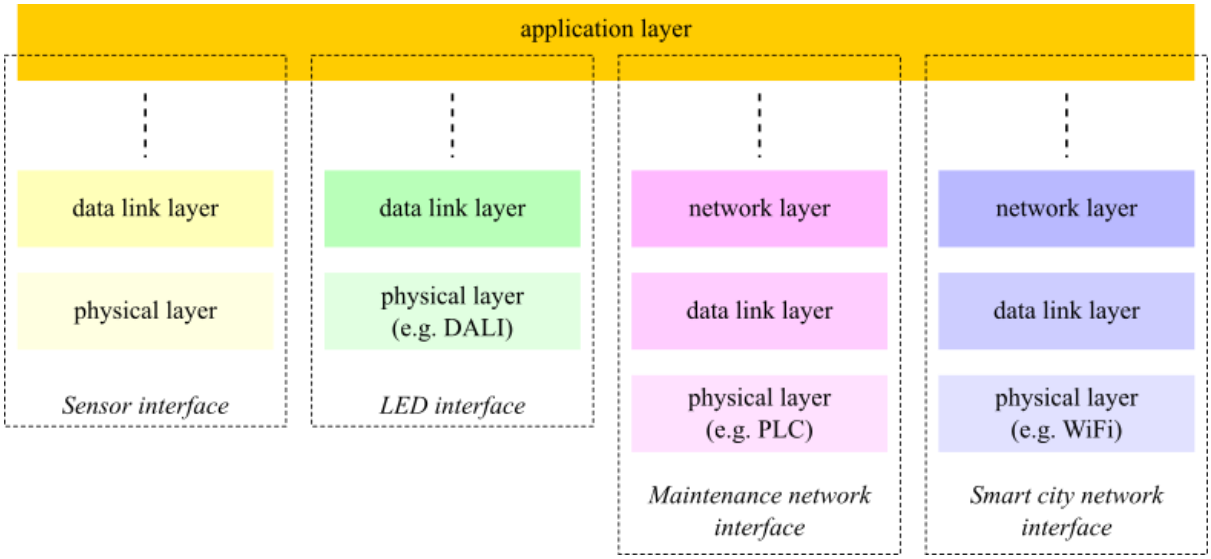


Fig. 2. Protocol stacks with shared application layer implemented in the PUs.

The smart city subnetwork is primarily intended to support future services. Besides, the future-proof nature of the maintenance subnetwork is also ensured by the component referred to as *dispatcher unit* in Fig. 1. This entity is responsible for managing the maintenance subnetwork and concentrating its data traffic from the viewpoint of a higher level application playing the role of the lighting dispatcher host. The main objective of this host application is to facilitate a link to the Internet. This connection widens the range of possible services even further in the direction of web-based and cloud-based applications and services.

By connecting the smart lighting system to the Internet, our proposed solution may become part of the Internet of Things. Allowing arbitrary sensor data measured in the pylon units to accumulate in a cloud-based database can open new horizons in smart city applications. Accessing these measurement data and controlling the smart lighting network using a web-based application allows:

- Supporting weather forecasting services. Accumulating temperature and relative humidity measurements coming from sensors built-in the luminaires may provide fine grained information regarding local weather conditions. The acquired local weather data may be used to supplement weather forecasting algorithms.
- The wireless communication ability of each pylon unit enables providing support for traffic management. Smart vehicles may connect to the smart city subnetwork of the proposed solution to acquire location information of other connected vehicles. Autonomously driven vehicles may share location information in order to avoid collisions. The smart city subnetwork may also be used to dismiss current social navigation applications such as Waze [2].
- The smart city subnetwork of the pylon units may be used to support navigation applications in finding free parking slots. The pylon units may contain sensors that track parking place occupation and these data may be accessed from navigation applications using the Internet.
- Proximity sensors built-in to the luminaires may sense pedestrian presence and may be used for autonomous dimming and dynamic power management of the smart lighting system. According to the proximity sensor data, power consumption may be

reduced dramatically by powering off luminaires where no pedestrian activity can be detected.

- The smart city subnetwork of the lighting fixtures may be used as global positioning beacons supporting and extending current GPS navigation applications. UAVs such as drones may use this network as navigation beacons to supplement orientation in densely populated urban or covered areas where GPS positioning is not available or not accurate.

Although the maintenance subnetwork and the smart system subnetwork are referred to as two distinct interconnection media with their unique service sets and protocol stacks, the demonstrational implementation defines the same services for both of them. That means that the “smart city services” demonstrated by this realization are practically identical to those available on the maintenance subnetwork. However, their physical layers significantly differ from each other. The maintenance subnetwork applies PLC, while the smart system subnetwork utilizes WiFi. In an envisioned future realization, the two subnetworks significantly differ in the services they provide. In this case, the security and authorization issues arising because of the combined private/public nature of the services are concentrated into the application layer implemented by the pylon units.
Another difference between the envisioned system architecture and the demonstrational one is that in the latter one, the smart city subnetwork is a virtual network, established by point-to-point links between the external users and the pylon units. The access to other pylon units is indirectly provided by the maintenance subnetwork (see Fig. 3). In a real-life future application, the smart city subnetwork presumably has to cope with more frequently and progressively changing requirements (e.g. bandwidth, data volume, etc.), therefore, the physical-level separation seems to be justified, even if this solution demands more complicated software and hardware resources and solutions.

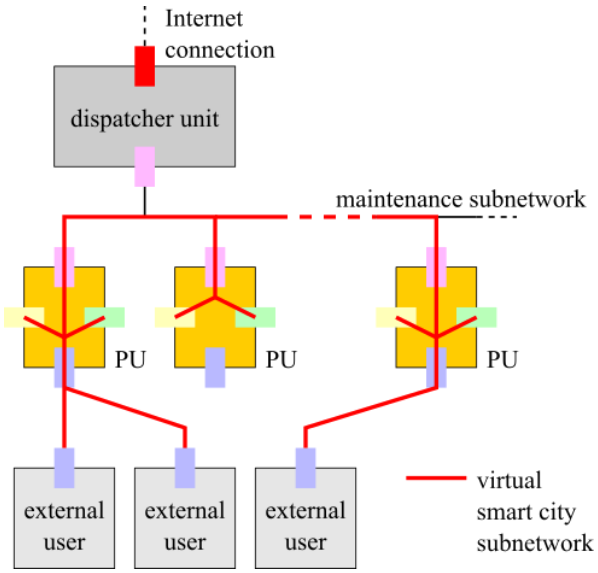


Fig. 3. The smart city subnetwork is a virtual one, physically realized by the maintenance subnetwork resources.

3.2. Node architectures, interfaces, communication protocols

3.2.1. Dispatcher Unit

As may be seen in Fig. 1, the maintenance subnetwork is formed by multiple Pylon Units (PU) and a single Dispatcher Unit (DU) interconnected in a tree topology. The DU represents

the root of this network, through which the whole maintenance subnetwork has access to the Internet. Accordingly, the DU has two external interfaces:

- **if-DU-IN**: Through this interface, the DU may connect to the internet (IN).
- **if-DU-MS**: Through this interface, the DU may connect to the leaves of the maintenance subnetwork (MS).

The details of the external interfaces are presented in Section 3.2.3.
The DU is comprised of two submodules, namely the Central Computing Unit (CCU) and the Maintenance Subnetwork Modem (MSM) (see Fig. 4).

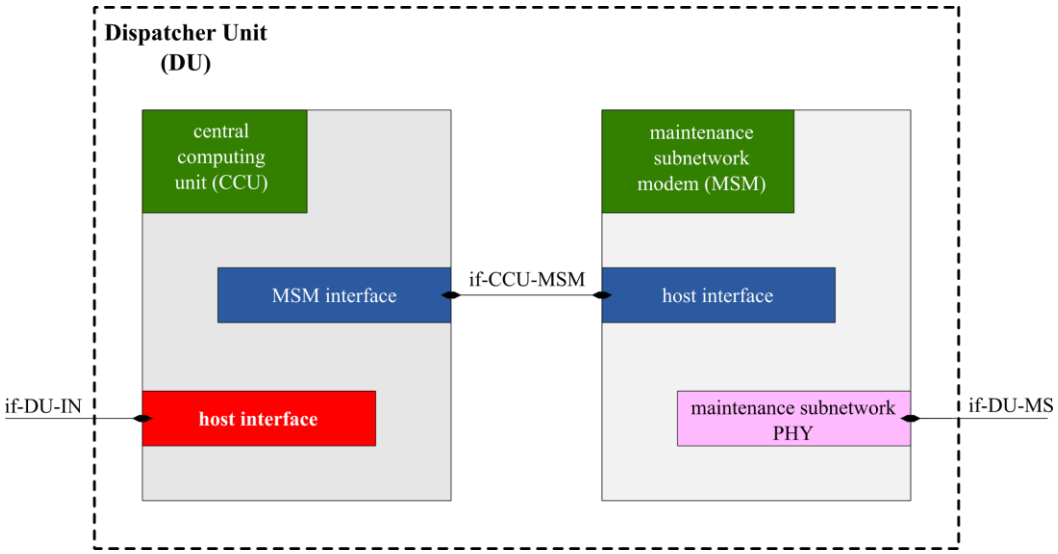


Fig. 4. The internal structure and interfaces of the DU.

The CCU implements the application level of the maintenance subnetwork communication protocol stack and the system level application logic, while the MSM implements the network layer, data link layer and physical layer of the maintenance subnetwork communication protocol stack. The DU includes a single internal interface:

- **if-CCU-MSM**: This interface realizes the communication between two layers of the maintenance subnetwork protocol stack. This abstraction is needed because the application level and the lower levels (from network level down to PHY) of this protocol stack are implemented by two physically separated hardware units (see the details in Section 4.1.). The network level services (thus indirectly all lower level services) of the MSM are accessible for the application level through a UART-based link defined in Appendix A.

3.2.2. Pylon Unit and ancillary modules

The hardware structure of the PU is similar to the DUs structure. It consists of a CCU and an MSM. It implements a communication interface, through which external devices may be connected to it (smart city subnetwork interface, if-PU-SS), and another interface making it possible for the PU to connect to the maintenance subnetwork as a leaf (if-PU-MS). Fig. 5 shows the internal structure and interfaces of the PU.

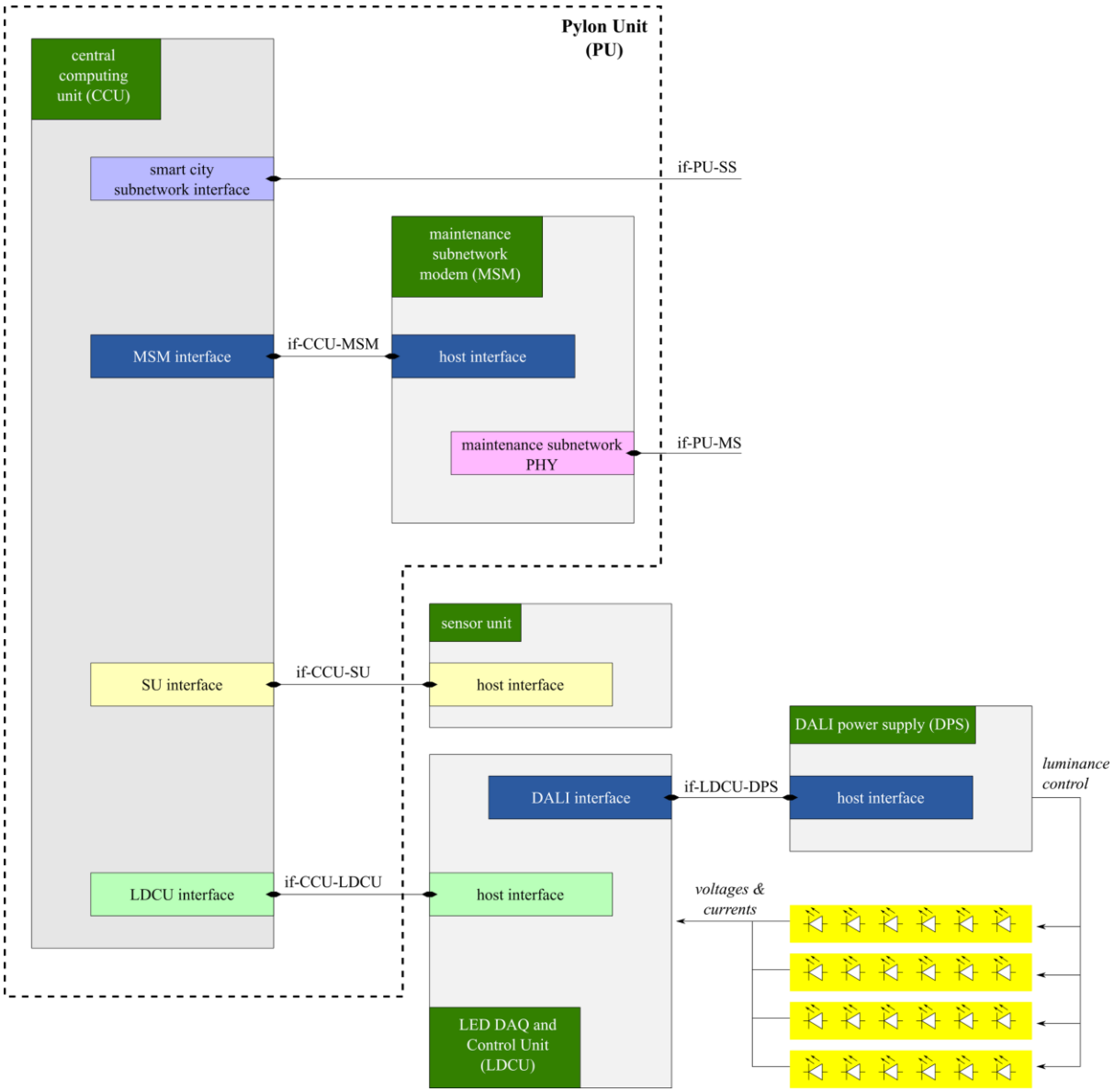


Fig. 5. The submodules and interfaces of the PU and the ancillary devices providing it with housekeeping data.

Just as the DU, the PU is also connected to the maintenance subnetwork through an MSM:

- **if-CCU-MSM**: From functional point of view, this interface is equivalent to if-CCU-MSM in the DU: It forms a link between the application level logic implemented by the CCU and the network level implemented by the MSM. However, the application level logic implemented by the CCU is different, because of the different roles of the design units in the maintenance subnetwork. The DU is a root, while the PUs are leaves of the tree-like network.

The CCU of the PU implements two additional communication interfaces through which ancillary modules may be connected to it:

- **if-CCU-LDCU**: Interface between the CCU and the LED DAQ and Control Unit (LDCU).
- **if-CCU-SU**: Interface between the CCU and the Sensor Unit (SU).

The LDCU is able to control the luminance of the LED strings making intelligent optimizations regarding power consumption possible. Additionally, the LDCU measures the states of the LED strings and provides the CCU with the readings through this interface. if-CCU-LDCU is based on UART. The link speed is 9600 bps, the format is 8N1. The link utilizes the packet encapsulation method defined by PPP. Fig. 6 shows the frame structure applied.

start/stop frame flag	payload	start/stop frame flag
1 byte	variable length	1 byte

Fig. 6. The frame structure used between the CCU and the LDCU.

The ‘start/stop frame flag’ is defined as 0x7E. When this byte occurs in the payload data (DATA), an ESC character (0x7D) is inserted before it and (0x20 XOR DATA) is inserted into the payload. The ESC character itself is escaped in the same way. Fig. 7 shows the payload structure.

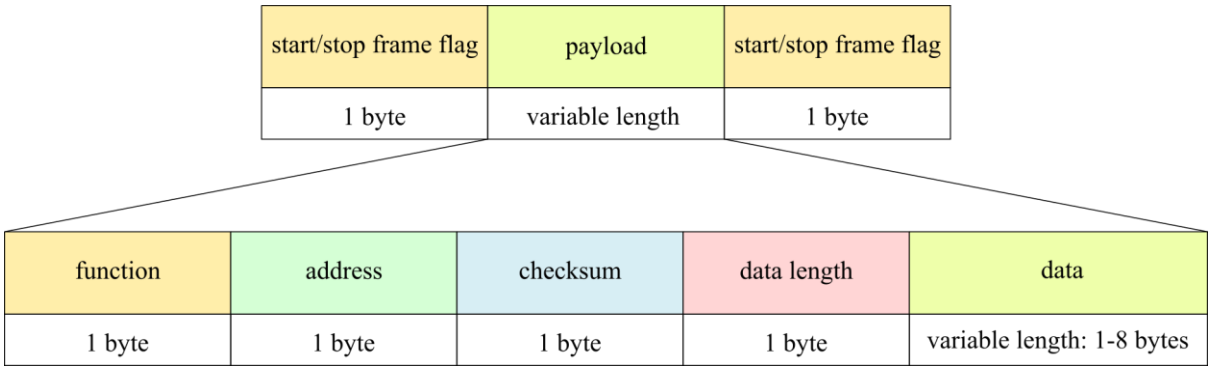


Fig. 7. The payload of a PPP frame between the CCU and the LDCU.

The payload of the PPP frame includes the following fields:

- **Function**
  - Query: 0x01; The CCU sends a query to the LDCU.
  - Command: 0x09; The CCU sends a command or data to the LDCU.
  - Response: 0x19; The LDCU sends data back to the CCU.
- **Address**: The address of the register to read/write. In case of acknowledgement, ‘connection OK’, and error messages, it is set to 0xFF.
  - 0x00: 1-byte DALI luminous power (logarithmic)
  - 0x01: 1-byte DALI fade speed.
  - 0x02...0x07: reserved for future use
  - 0x08; 0x09: U0L; U0H: Voltage of the first LED string * 10 in Volts.
  - 0x0A; 0x0B: I0L; I0H: Current of the first LED string in mAs.
  - 0x0C; 0x0D: U1L; U1H: Voltage of the second LED string * 10 in Volts.
  - 0x0E; 0x0F: I1L; I1H: Current of the second LED string in mAs.
  - 0x10; 0x11: U2L; U2H: Voltage of the third LED string * 10 in Volts.
  - 0x12; 0x13: I2L; I2H: Current of the third LED string in mAs.

        - 0x14; 0x15: U3L; U3H: Voltage of the fourth LED string * 10 in Volts.
        - 0x16; 0x17: I3L; I3H: Current of the fourth LED string in mAs.
    - **Checksum**: 8-bit checksum covering the ‘function’, the ‘address’, the ‘data length’, and the ‘data’ fields.
    - **Data length**: The number of bytes sent in the ‘data’ field of the payload.
    - **Data**: the data itself. If address is 0xFF, the data field contains acknowledgement, ‘connection OK’, or error messages:
        - 0xF0: LDCU is connected
        - 0xF1: command acknowledged
        - 0xF2: invalid data (e.g. DALI luminous power greater than 254)
        - 0xF3: invalid address
        - 0xF4: invalid command
        - 0xF5: checksum error

The LED string luminances are indirectly controlled by the LDCU through a DALI-capable Power Supply (DPS). **if-LDCU-DPS** implements a minimal subset of the DALI protocol [3]. This implementation is capable of adjusting LED string luminances and fading characteristics.

The Sensor Unit (SU) measures the ambient temperature and humidity and provides the CCU with the readings through the **if-CCU-SU** interface. It is a standard I2C link implemented based on the sensor specification (see Section 4.1.3.).

3.2.3. External interfaces of the design units

3.2.3.1. Internet connection of the DU

The DU is able to connect to the Internet using the TCP/IP protocol stack, which is a part of the Linux operation system running on the DU. WiFi is used as physical layer protocol.

3.2.3.2. Smart city subnetwork interface of the PU

The PU is able to connect to external devices using the TCP/IP protocol stack, which is a part of the Linux operation system running on the PU. WiFi is used as physical layer protocol.

3.2.3.3. Maintenance subnetwork

The maintenance subnetwork is a Power Line Communication (PLC) network, which makes it possible to use the already installed power lines for communication purposes. The MSM’s network layer (see the description of the applied Yitran IT700 modem in Section 4.1.2.) is designed to create a network with a tree type topology, allowing efficient connectivity between all nodes in the network. The tree consists of a Network Coordinator (NC) and multiple Remote Stations (RSs).

- **NC**: It functions as the concentrator of the network. Once a network is created, it assigns a network ID to the NC. The node ID of the NC is 1. As a concentrator of the network, the NC is the root of the tree. The network topology is derived from the NC’s tables. In our application, the DU represents the NC.

- **RS**: The RS is a general name for all nodes in the tree, which are not the root, i.e. body nodes and leaves. An RS can function as both an initiating as well as a repeater and maintains both functions, or may be one or the other. In our application, the PUs represent the RSs.

Fig. 8 shows an example for the PLC network topology.

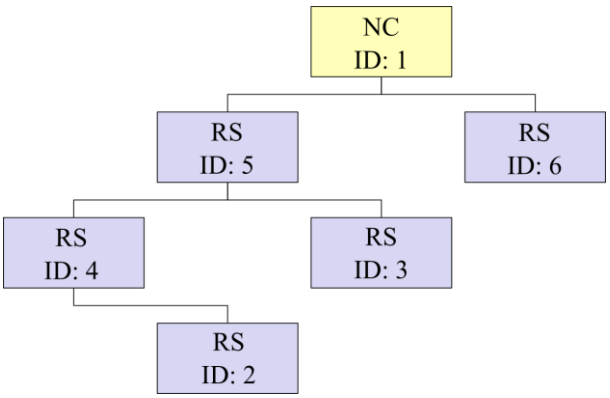


Fig. 8. Exemplary PLC network topology.

In our application, the exact structure of the PLC network is slightly simpler, because there are no repeaters. All RSs (PUs) connect directly to the NC (DU) as leaves (see Fig. 9).

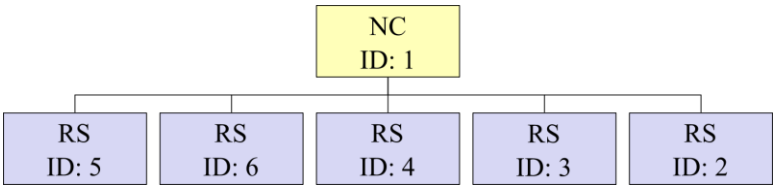


Fig. 9. PLC network topology in our application.

4. Implementation details

4.1. Hardware components

4.1.1. CCU hardware

The central data processing tasks of the PUs and the DU are implemented by Intel Edison single board computers. Intel Edison is an ultra-small computing platform designed for industrial IoT applications [4], [5]. The CCU is comprised of an Intel® Atom™ SoC dual-core CPU, 1 GB DDR and 4 GB flash memory, an integrated WiFi, Bluetooth LE, and a 70-pin connector to attach shield-like “blocks” which can be stacked on top of each other. It’s low power and small footprint make it ideal for projects that need a lot of processing power, but don’t have the ability to be near a larger power source or have a large footprint. Fig. 10 shows the Intel Edison compute module. For more technical details see Appendix B.

Fig. 10. The Intel Edison Compute module.

Multiple different breakout boards were developed for the Intel Edison compute module. Because of its small form factor, CCU uses the Intel Edison Mini Breaout Kit (see Fig. 11). For more technical details see Appendix C. For pin configurations see Appendix H.

Fig. 11. The Intel Edison compute module and the Mini Breakout Kit.

The Intel Edison compute module runs a Yocto Linux distribution named Poky as its operating system. The Poky reference distribution is a modern, fully-fledged Linux allowing to run user applications connected to the Internet. User applications may be developed in several languages and using different development frameworks that are built in the system. One can create user applications with the following software frameworks, IDEs and languages:

- Intel® XDK IoT Edition
- NodeJS – Javascript
- Arduino IDE
    - Arduino language
- Intel® System Studio IoT Edition
    - C/C++ with MRAA [REFMRAA] and UPM [REF9fromPaper] libraries to control GPIO interfaces with high level APIs.
- MRA and UPM with Python bindings
- MRA and UPM with Java bindings

4.1.2. MSM hardware

The MSM is realized by the IT700 PLC modem in conjunction with the so-called Plug-In Module (PIM) manufactured by Yitran (see Fig. 12). IT700 is a highly integrated System-on-Chip (SoC) power-line communication modem. It incorporates Yitran's extremely reliable

physical layer, high performance data link layer and Yitran Network Layer (Y-Net) protocol. An integrated microcontroller with extended 8051 core, 256 KB flash memory, 16 KB RAM, and 24 general purpose I/Os implements the protocol stack and offers the required flexibility to implement various protocols and applications. The microcontroller's UART interface provides the connection to an external host and application controller.

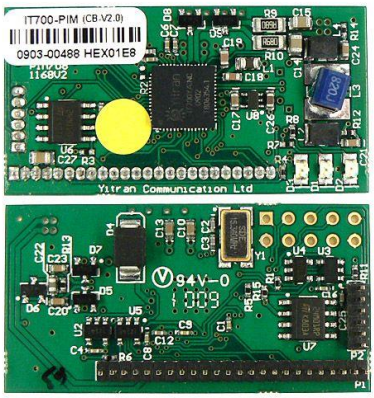


Fig. 12. The YITRAN PIM.

The IT700 modem core uses Yitran's patented Differential Code Shift Keying (DCSK) advanced spread spectrum modulation technique. DCSK enables extremely robust communication over existing electrical wiring with data rates up to 7.5 Kbps. In addition to the inherent interference immunity provided by DCSK modulation, the device utilizes several mechanisms for enhanced communication robustness, such as a patented forward short-block soft-decoding error-correction algorithm and special synchronization algorithms.
The integrated Analog Frontend provides differential inputs and line driver outputs to connect via an external line filter and coupler to the power transmission lines. An integrated PLL circuit allows the operation of the IT700 with a choice of different crystal oscillators. An integrated POR circuitry eliminates the need for any external reset components and provides an autonomous, safe power-up and power-down reset to the chip. The integrated 1.8V regulator allows the IT700 to operate from a single 3.3V supply voltage.
The IT700 complies with worldwide regulations (FCC part 15, ARIB, and CENELEC bands) and is an ideal solution for a variety of "No New Wires" narrowband PLC applications.
The IT700 is available in two versions:

- The **Protocol Control Architecture** version has Yitran's Y-Net network layer protocol pre-programmed into the 8051 microcontroller's flash memory. A UART interface and a simple command language provide seamless connection to an external host controller and simplify application development. In this version the user has no access to the microcontroller's unused memory space, peripheral functions and general purpose I/Os.
- The **Open Solution Architecture** version allows you to utilize the IT700 microcontroller's peripheral functions, such as timers, interrupts, communication interfaces, A/D, spare memory resources, and general purpose I/Os to implement your own application code, thereby eliminating the requirement for an external host controller. An API enables the easy integration of your application code with Yitran's network layer code.

Our application uses the **Protocol Control Architecture** version of IT700 to implement application logic.

4.1.3. SU hardware

SU functionalities are implemented by the TH02 sensor module manufactured by HopeRF Electronic (see Fig. 13). This monolithic CMOS IC integrates temperature and humidity sensor elements, an A/D converter, signal processing, calibration data, and an I2C host interface. The patented use of industry-standard low-K polymeric dielectrics for sensign humidity enables the construction of a low-power, monolithic CMOS sensor IC with low drift and hysteresis and excellent long-term stability.

Fig. 13. TH02 temperature and humidity sensor.

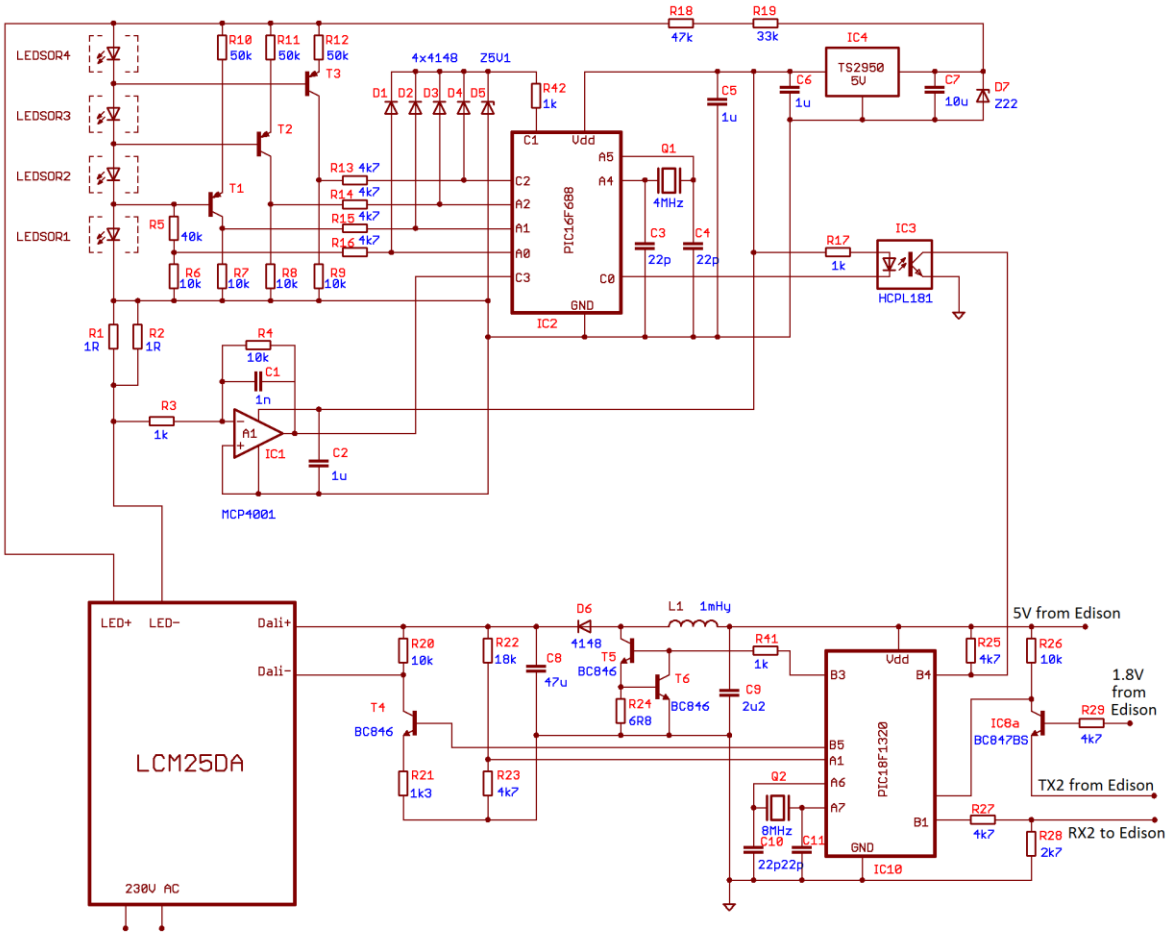
Both the temperature and humidity sensors are factory-calibrated and the calibration data are stored in the on-chip non-volatile memory. This ensures that the sensors are fully interchangable, with no recalibration or software changes required. For more technical details see Appendix D.

4.1.4. LDCU hardware

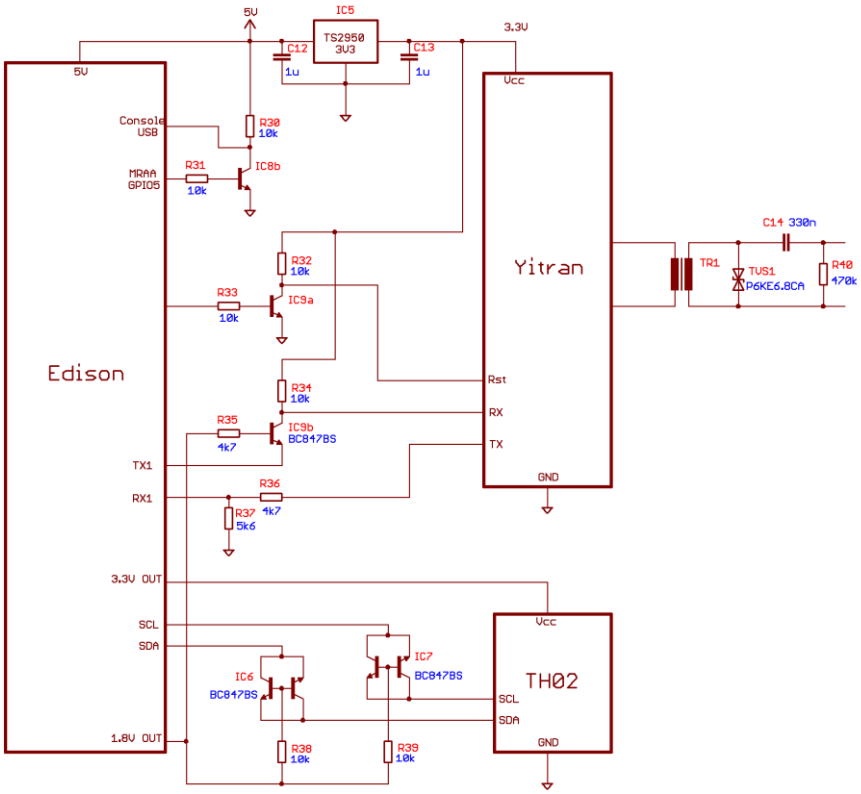
The LDCU is realized by an application-specific PCB including the following components:

- power supply modules
- voltage and current measurement and A/D converter circuits for LED string voltages and currents
- a microcontroller unit managing all the LDCU tasks as follows
    - the LCDU side of the if-CCU-LDCU interface
    - DALI master
    - DC/DC converter control
    - LED string measurement tasks
- level shifter circuitry for the Intel Edison compute module

Fig. 14 shows the detailed schematic of the LDCU module while Fig 15 shows its PCB layout. The LDCU PCB carries a set of level shifter circuits too, which are not directly related to the LDCU functionalities, however, the are necessary to connect the 1.8V voltage level domain of Intel Edison to the 3.3 V voltage level domain of the TH02 sensor unit and the Yitran IT700 PLC modem (Fig. 14 b). See the high-resolution schematic and the detailed BOM in Appendix E. For pin configurations see Appendix H.


LEDSOR4
LEDSOR3
LEDSOR2
LEDSOR1
R10 50k
R11 50k
R12 50k
T3
T2
T1
R5 10k
R6 10k
R7 10k
R8 10k
R9 10k
4x4148
Z5V1
D1 D2 D3 D4 D5
R42 1k
R13 4k7
R14 4k7
R15 4k7
R16 4k7
PIC16F688
IC2
C1 Vdd
A5
A4
A2
A1
A0
C3
C0
GND
Q1
4MHz
C3 22p
C4 22p
C5 1u
R18 47k
R19 33k
IC4
TS2950 5V
C6 1u
C7 10u
D7 Z22
R17 1k
IC3
HCPL181
R1 1R
R2 1R
R4 10k
C1 1n
R3 1k
A1
IC1
C2 1u
MCP1001
LED+ LED-
Dali+
Dali-
LCM25DA
230V AC
R20 10k
R22 18k
C8 47u
D6 4148
L1 1mHy
T5 BC846
T6
R24 6R8
C9 2u2
T4 BC846
R21 1k3
R23 4k7
R41 1k
PIC18F1320
Vdd
B3
B4
B5
A1
A6
A7
B1
GND
IC10
Q2
8MHz
C10 22p
C11 22p
5V from Edison
R25 4k7
R26 10k
IC8a
BC847BS
R29 4k7
1.8V from Edison
TX2 from Edison
R27 4k7
R28 2k7
RX2 to Edison

(a)


5V
IC5
TS2950 3V3
C12 1u
C13 1u
3.3V
Edison
Console USB
R30 10k
MRAA GPIO5
R31 10k
IC8b
R32 10k
R33 10k
IC9a
R34 10k
R35 4k7
IC9b
BC847BS
TX1
RX1
R36 4k7
R37 5k6
3.3V OUT
SCL
SDA
1.8V OUT
Yitran
Vcc
Rst
RX
TX
GND
C14 330n
TR1
TVS1
P6KE6.8CA
R40 470k
IC6
BC847BS
IC7
BC847BS
TH02
Vcc
SCL
SDA
GND
R38 10k
R39 10k

(b)

Fig. 14. The schematic of the LDCU module.

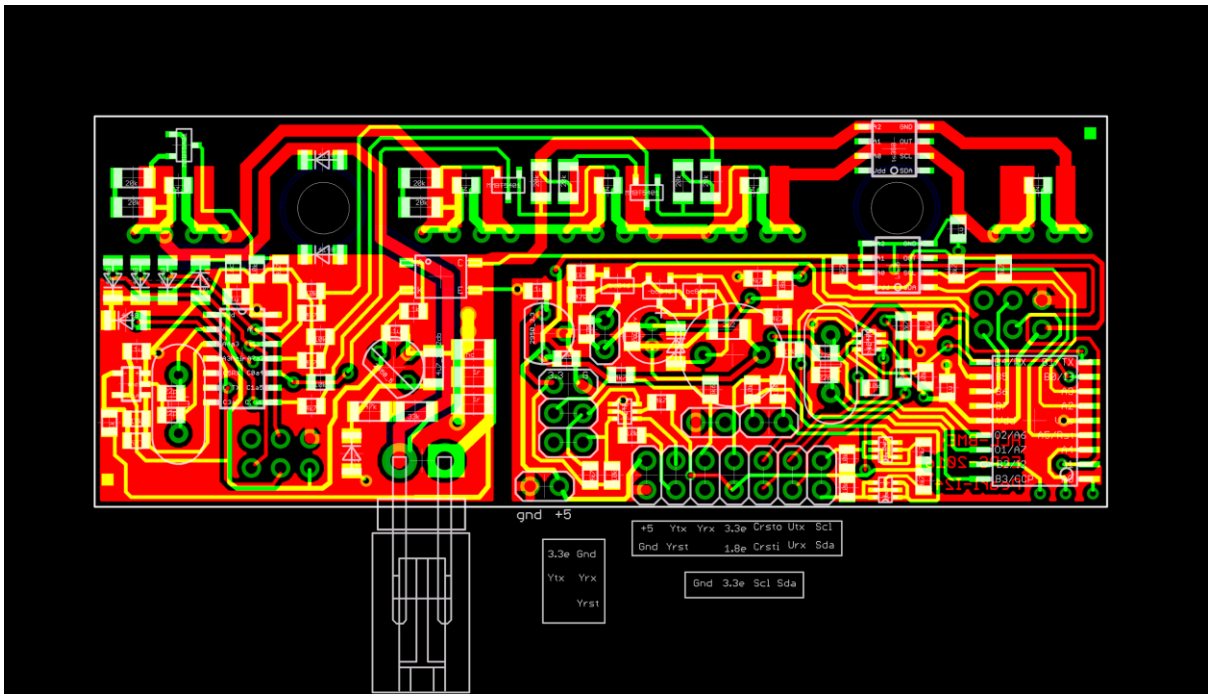


Fig. 15. The detailed layout of the LDCU PCB.

4.1.5. DPS hardware

The DPS is realized by the LCM-25 DA power supply module manufactured by MEAN WELL (see Fig. 16).

Fig. 16. LCM-25DA DALI-compatible power supply.

LCM-25 DA is a 25W multiple-stage output current LED power supply. One single unit supplies multiple current levels (350mA, 500mA, 600mA, 700mA, 900mA, 1050 mA). The current levels are able to be easily switched by adjusting the built-in DIP switch. LCM-25 DA also provides the dimming function that is controlled by push dimming or DALI signal. Moreover, the synchronization design allows the dimming for up to 10 units of LCM-25 DA to be controlled simultaneously. For more technical details see Appendix F.

4.2. Software components

In this section a brief overview of the software components of the CCU is presented. The low-level APIs are written in C++.

4.2.1. Low-level APIs

4.2.1.1. TH02 I2C API (if-CCU-SU)

There is a single class implementing the if-CCU-SU interface functionalities. This class, called ‘th02’ uses the class library mraa [6] to access the I2C bus of the Intel Edison compute module. The naming of this class implies that it is specifically developed for the TH02 temperature and humidity sensor. The class provides the user with the following functions:

- **Constructor parameters**: none
- **check_sensor()**
    - Function: The function checks whether there is a TH02 sensor connected to the I2C bus by sending an ID read request. If the bus is not responding or invalid device ID is read, the function throws a ‘sensor_not_responding_exception’ exception.
    - Parameters: none
    - Return value: void
- **heater_on()**
    - Function: The function sends a command through the I2C bus, which turns the heater of the TH02 sensor on.
    - Parameters: none
    - Return value: void
- **heater_off()**
    - Function: The function sends a command through the I2C bus, which turns the heater of the TH02 sensor off.
    - Parameters: none
    - Return value: void
- **get_rh()**
    - Function: The function sends a relative humidity query through the I2C bus.
    - Parameters
        - unsigned timeout_ms: If the sensor does not respond within a time period defined in the parameter, the function throws a ‘conversion_timed_out_exception’ exception.
    - Return value: double: The temperature-compensated relative humidity reading.
- **get_temp()**
    - Function: The function sends a temperature query through the I2C bus.
    - Parameters

        - unsigned timeout_ms: If the sensor does not respond within a time period defined in the parameter, the function throws a ‘conversion_timed_out_exception’ exception.
    - Return value: double: The temperature reading.

The ‘th02’ class defines exceptions which are derived from the ‘exception’ class defined in the Standard Template Library. The ‘th02’ class defines the following exceptions:

- sensor_not_responding_exception: There are no sensors connected to the I2C bus, or the device connected does not respond, or the response does not contain the device ID expected.
- conversion_timed_out_exception: The I2C device does not respond to a query within the predefined timeout period.

4.2.1.2. LED interface API (if-CCU-LDCU)

The two-layer interface called if-CCU-LDCU in Fig. 2 is implemented by two classes in the CCU software. The class called ‘ppp_link’ implements packet encapsulation/decapsulation according to the PPP standard, while the class called ‘led_interface’ uses ‘ppp_link’ services to embed LED luminance control and measurement data into the PPP frames’ payload fields. The ‘ppp_link’ class uses the mraa [6] class library to access UART modules, and it provides the following interface:

- **Constructor parameters**
    - use_console_port: There are two UART modules inside the Intel Edison compute module. One of them is shared between the user applications and the operating system’s console. This parameter determines, which one is initialized for use as a PPP link. If the console-UART is used, the service of the operating system forwarding the console information to the UART module should be disabled. This is done by disabling the corresponding systemd service in the CCU that provides console access to the device via the UART channel. This change is permanent across restarts of the CCU.
- **reset()**
    - Function: The function resets the link, including receive and transmit FIFOs, and status information.
    - Parameters: none
    - Return value: void
- **receive_a_packet()**
    - Function: The function checks the UART’s input FIFO for data. If any data are present, the function starts to decapsulate them according to the frame format presented in Section 3.2.2. If no data is available, the function throws an ‘uart_timeout_exception’ exception as the timeout period defined in parameter ‘timeout’ expires.
    - Parameters
        - unsigned timeout: A timeout period for UART packet frame reception given in milliseconds. If no data is available in the UART receive FIFO, the function throws an ‘uart_timeout_exception’ exception as this timeout period expires.

        - char packet_buffer[32]: A 32-byte buffer for the data read from the UART input FIFO.
    - Return value: unsigned char: Number of bytes in the PPP payload.
- **transmit_a_packet()**
    - Function: The function encapsulates the content of the input data according to the PPP frame structure presented in Section 3.2.2. and sends the encapsulated byte stream to the UART transmission buffer.
    - Parameters
        - unsigned number_of_bytes: The number of bytes for encapsulating.
        - unsigned packet_buffer[32]: A 32-byte buffer containing the raw payload data intended to be encoded and sent through the link.
    - Return value: void

The 'ppp_link' class defines exceptions which are derived from the 'exception' class defined in the Standard Template Library. The 'ppp_link' class defines the following exceptions:

- uart_timeout_exception: No data appears in the receive FIFO within a predefined time period.
- receive_fifo_overflow_exception: The PPP-encoded byte stream includes more than 32 bytes.
- transmit_buffer_overflow_exception: The number of bytes to encode and transmit exceeds 32.

The 'led_interface' class provides the following functions:

- **Constructor parameters**: none
- **check_connection()**
    - Function: The function checks whether there is another link node at the end of the UART channel, which responds to the 'check connection' request. If no response is received within 10 seconds or the format of the response is invalid, a 'connection_error_exception' exception is thrown by the function.
    - Parameters: none
    - Return value: void
- **set_luminance()**
    - Function: The function sends the 'set_luminance' command through the link.
    - Parameters
        - unsigned char luminance: The new value of the DALI luminance. If the luminance value is greater than 254, the function throws an 'invalid_luminance_exception' exception.
    - Return value: void
- **get_luminance()**
    - Function: The function sends a luminance query through the link.
    - Parameters: none
    - Return value
        - unsigned char: The current luminance value.
- **set_fade()**
    - Function: The function sends the 'set_fade' command through the link.
    - Parameters

        - unsigned char fade: The new value of the DALI fade speed.
    - Return value: void
- **get_u()**
    - Function: The function sends a LED string voltage query through the link.
    - Parameters
        - unsigned char uid: The identifier of the LED string to be queried.
    - Return value: double: The value of the LED string voltage in Volts.
- **get_i()**
    - Function: The function sends a LED string current query through the PPP link.
    - Parameters
        - unsigned char iid: The identifier of the LED string to be queried.
    - Return value: double: The value of the LED string current in mAs.

The ‘led_interface’ class defines exceptions which are derived from the ‘exception’ class defined in the Standard Template Library. The ‘led_interface’ class defines the following exceptions:

- connection_error_exception: The device on the other end of the link does not respond or its response cannot be decoded.
- command_not_acknowledged_exception: No acknowledge received to the command sent by the CCU within 10 seconds.
- invalid_luminance_exception: The luminance value posted for transmission is greater than 254.
- checksum_error_exception: An error encountered during transmission and the checksum field of the received frame is not equal to that calculated by the CCU based on the received data.
- checksum_error_response_exception: The LDCU sends a ‘checksum error’ response back, indicating that a transmission error occurred.
- invalid_response_exception: The LDCU sends an invalid response.
- invalid_channel_id_exception: The if-CCU-LDCU interface supports only 4 LED strings. The string identifiers used in the queries must be between 0 and 3.

4.2.1.3. Yitran host API (if-CCU-MSM)

The aim of the Yitran host API is to provide the user with C++ classes hiding the UART-based interface of the Yitran modem’s network layer, making it possible to develop applications over the PLC network in a more efficient manner. This UART-based host interface protocol, which is a baseline for this API, is defined in Appendix A.
The Yitran host API consists of multiple classes, whose relations are indicated by the UML class diagram in Fig. 17.

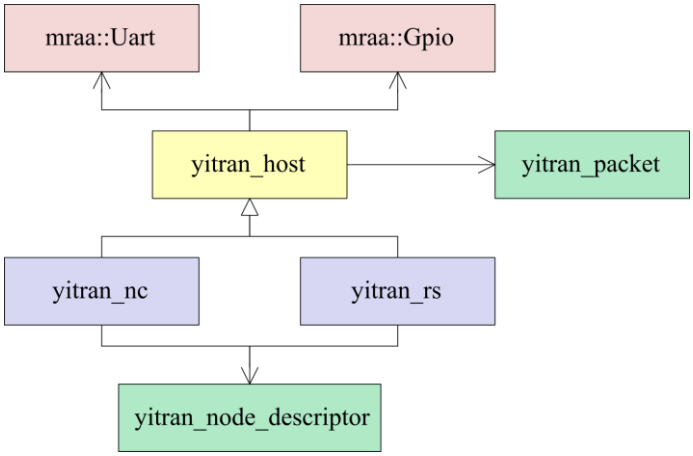


Fig. 17. UML class diagram indicating the relations among the Yitran host API constructs.

The central element is the ‘yitran_host’ base class responsible for concentrating the functionalities, which are the same in case of the NC and the RS nodes. It uses an ‘mraa::Uart’ class to connect to the 8051 microcontroller of the Yitran IT700 modem. This connection practically represents the link between our application logic and the network layer of the Y-Net protocol stack implemented by the 8051 microcontroller. The ‘mraa::Gpio’ class is needed because the Yitran IT700 modem shall be reset eventually. The resetting is done by pulling it’s reset pin low for a given time period. The reset pin is connected to a general purpose I/O pin of the Intel Edison compute module. This I/O pin may be accessed through the ‘mraa::Gpio’ class from the user applications running on the compute module.
The packet reception is realized in a blocking form in ‘yitran_host’. If a blocking receive function is called, with a certain timeout period, the eventually received packet is stored into a temporary variable with a type of ‘yitran_packet’. This class is a special container class carrying the information encapsulated in a yitran packet sent by the Y-Net network layer:

- **Command start symbol**: Yitran packets start with a special command start symbol (0xCA), which may be used to resynchronize the communication with the network layer.
- **Payload size**: The number of bytes encapsulated into the yitran packet.
- **Packet type**: The type of the packet received (indication from a lower layer, packet carrying user data, or a response packet after a command).
- **Opcode**: There are several types of indications and user data packets. The subcategories are identified by this byte.
- **Payload**: The payload field of the yitran packet. In this API, the maximum size of a packet is 256 bytes (including header).

There are additional fields of the ‘yitran_packet’ class, which are only interpretable in case of packets carrying user information (instead of indications from lower levels of the protocol stack). These member data are accessible after a decoding step:

- **Origin ID**: The sender of the packet.
- **Final ID**: The destination of the packet.
- **User payload size**: The size of the user payload data (header and addressing information excluded from the yitran packet).

- **User payload**: The user payload encapsulated into a yitran packet (without the addressing information).

There are two additional classes derived from ‘yitran_host’, namely the ‘yitran_nc’ and the ‘yitran_rs’ classes. They are responsible for the tasks performed specifically on the NC and the RS nodes respetively. There are several indication types, which carry information about new RS nodes connecting to the network and another ones disconnecting from it. These indication provide information about nodes and their connection status. The class named ‘yitran_node_descriptor’ is a container for this information:

- **Node logical ID**: The identifier of the node in the network.
- **Parent logical ID**: The identifier of the node’s parent in the network.
- **Connection status**: The new status of the node:
    - Disconnected
    - Connected (good quality)
    - Connected (poor quality)

The ‘yitran_host’ class defines the following data members:

- **public yitran_packet recently_received_packet**: The blocking receive functions store the received packets into this temporary variable. It may be decoded thereafter to get user information or other data encapsulated in a Yitran packet.
- **protected mraa::Uart* yitran_uart**: A pointer to the object, which through the UART channel connected to the 8051 microcontroller of the Yitran modem may be accessed.
- **protected mraa::Gpio* yitran_reset_pin**: A pointer to an object, which through the reset pin of the Yitran modem may be accessed.

The ‘yitran_host’ class provides the following public functions:

- **Constructor parameters**
    - bool use_console_port: The Intel Edison compute module includes two UART channels. One of them is shared between a service of the operating system sending console information to this channel and the user applications. If this UART channel is intended to be used in a user application, the corresponding service shall be disabled in the operating system. The constructor parameter defines, which UART channel shall be initialized for the user application.
- **reset()**
    - Function: The function resets the Yitran IT700 modem by pulling its reset pin to LOW for 1 second.
    - Parameters: none
    - Return value: void
- **send_packet()**
    - Function: The function sends a single Yitran packet. The checksum is calculated automatically.
    - Parameters
        - const yitran_packet& packet_2_send: The prepared Yitran packet to send.

    - Return value: void
- **get_packet()**
    - Function: The function receives a single packet and it checks the checksum. The bytes received before the command start symbol are discarded. The decoded packet is stored into the ‘recently_received_packet’ member variable of the ‘yitran_host’ object.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: void
- **wait_for_runtime_indication()**
    - Function: The function waits for the following indications:
        - ‘connectivity_status_with_rs’: The NC may receive an indication that a connection with an RS became valid/invalid.
        - ‘new_connection_to_nc’: The NC may receive an indication that a new RS has connected to it.
        - ‘disconnected_from_nc’: An RS may receive an indication that it has been disconnected from the NC.
        - ‘connected_to_nc’: An RS may receive an indication that it has been connected to the NC.
        - ‘rx_intranetworking_packet’: The NC and an RS may receive an indication that a new RX intranetworking packet has been received.

      The function returns the type of the indication received. If any other type of indication is received, the function returns ‘unknown_indication’. In case of timeout, the function throws a ‘packet_reception_timeout’ exception. The decoded packet is stored into the ‘recently_received_packet’ member variable of the yitran_host object.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.
    - Return value: runtime_indication: The indication type received (enumerated type).
- **decode_rx_intranetworking_packet()**
    - Function: After receiving an RX intranetworking packet indication, this function may be used to get the relevant information stored into the payload (command data) field of the indication. Relevant information:
        - Origin ID
        - Final ID
        - User payload
    - Parameters: none
    - Return value: void
- **wait_for_reset_response()**
    - Function: The function receives a single Yitran packet. If the received packet is a reset response, the function returns true, otherwise it returns false. If no packet is received within the timeout period, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters

        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.
    - Return value: void
- **wait_for_go_online_response()**
    - Function: The function receives a single Yitran packet. If the received packet is a go online response, the function returns true, otherwise it returns false. If no packet is received within the timeout period, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.
    - Return value: void
- **acknowledged_nop()**
    - Function: The function sends a ‘NOP’ request and waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.
    - Return value: void
- **acknowledged_set_network_size()**
    - Function: The functions sends a ‘Set Device Parameters’ request setting the network size to the given value. Thereafter, the function waits for the response and checks its status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char size: The new network size to set.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **verified_set_network_size()**
    - Function: The functions sends a ‘Set Device Parameters’ request setting the network size to the given value. Thereafter, the function waits for the response and checks its status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the status field. If the status field indicates command error, the function returns false immediately. If the status field indicates a successful command, the function sends a ‘Get Device Parameters’ request, reading the network size. Thereafter, the function waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the network size value.
    - Parameters
        - unsigned char size: The new network size to set.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ is thrown.

    - Return value: bool: If the network size is set to size, the function returns true, otherwise it returns false.
- **acknowledged_set_node_id()**
    - Function: The functions sends a ‘Set Device Parameters’ request setting the node ID to the given value. Thereafter, the function waits for the response and checks its status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char size: The new node ID to set.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **verified_set_node_id()**
    - Function: The functions sends a ‘Set Device Parameters’ request setting the node ID to the given value. Thereafter, the function waits for the response and checks its status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the status field. If the status field indicates command error, the function returns false immediately. If the status field indicates a successful command, the function sends a ‘Get Device Parameters’ request, reading the node ID. Thereafter, the function waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the node ID value.
    - Parameters
        - unsigned char size: The new node ID to set.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: If the node ID is set to size, the function returns true, otherwise it returns false.
- **get_operation_mode()**
    - Function: The function sends a ‘Get Device Parameters’ request addressing operation mode. Thereafter, the function waits for the response for timeout_ms. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. If the response is successfully received, the response is stored into ‘recently_received_packet’.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the operation mode is set to NC, false otherwise.
- **get_network_size()**
    - Function: The function sends a ‘Get Device Parameters’ request addressing network size. Thereafter, the function waits for the response for timeout_ms. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. If the response is successfully received, the response is stored into ‘recently_received_packet’.
    - Parameters

        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: unsigned char: The network size.
- **get_node _id()**
    - Function: The function sends a ‘Get Device Parameters’ request addressing node ID. Thereafter, the function waits for the response for timeout_ms. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. If the response is successfully received, the response is stored into ‘recently_received_packet’.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: short: The node ID.
- **acknowledged_save_device_parameters()**
    - Function: The function sends a ‘Save Device Parameters’ request and it waits for response for timeout_ms. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **acknowledged_go_online()**
    - Function: The function sends a ‘Go Online’ request and it waits for response for timeout_ms. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **acknowledged_packet_tx()**
    - Function: The function sends a ‘Packet TX’ command. If the user payload size is greater than 245, the function throws a ‘invalid_user_payload_size_exception’ exception. The function waits for ‘Packet TX Admission’ and ‘Packet TX Transmission’ responses. Any other types of packets are discarded while waiting for these. In case of timeout, the function throws a ‘packet_reception_timeout_exception’ exception. Packet TX command parameters:
        - Data Service Type: Intranetworking Unicast
        - Priority: Normal
        - Ack Service: Ack required
        - Hops: 8
        - Gain: 7
        - Tag: 0
        - Encrypt: 0
        - Destination port: 0
        - target ID LSB: destination_id LSB

        - target ID MSB: destination_id MSB
        - user payload: user_payload
    - Parameters
        - short destination_id: The logical ID of the target node.
        - unsigned char* user_payload: The message to send.
        - unsigned char user_payload_size: The size of the message to send.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: If any of the responses indicates an error (the packet tx request is rejected or the transmission fails), the function returns false. If the transmission is successful, the function returns true.
- **non_acknowledged_packet_tx()**
    - Function: The function sends a ‘Packet TX’ command. If the user payload size is greater than 245, the function throws a ‘invalid_user_payload_size_exception’ exception. Packet TX command parameters:
        - Data Service Type: Intranetworking Unicast
        - Priority: Normal
        - Ack Service: Ack required
        - Hops: 8
        - Gain: 7
        - Tag: 0
        - Encrypt: 0
        - Destination port: 0
        - target ID LSB: destination_id LSB
        - target ID MSB: destination_id MSB
        - user payload: user_payload
    - Parameters
        - short destination_id: The logical ID of the target node.
        - unsigned char* user_payload: The message to send.
        - unsigned char user_payload_size: The size of the message to send.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: void.
- **print_recently_received_packet()**
    - Function: The function prints the recently received packet to the standard output.
    - Parameters: none
    - Return value: void

The ‘yitran_host’ class implements the above described public functions using the following protected functions:

- **get_byte()**
    - Function: The function receives a single byte from the UART channel.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.

        - char* buffer: The received byte is stored into this buffer.
    - Return value
        - bool: True, if the byte reception is successfull, false otherwise.
- **send_nop()**
    - Function: The function sends a ‘NOP’ request.
    - Parameters: none
    - Return value: void
- **set_nc_operation_mode()**
    - Function: The function sends a ‘Set Operation Mode’ request setting the operation mode to NC.
    - Parameters: none
    - Return value: void
- **set_rs_operation_mode()**
    - Function: The function sends a ‘Set Operation Mode’ request setting the operation mode to RS.
    - Parameters: none
    - Return value: void
- **set_nc_database_size()**
    - Function: The functions sets the NC database size.
    - Parameters
        - unsigned char size: The new NC database size.
    - Return value: void
- **set_network_size()**
    - Function: The functions sets the network size.
    - Parameters
        - unsigned char size: The new network size.
    - Return value: void
- **set_node_id()**
    - Function: The functions sets the node ID.
    - Parameters
        - short size: The new node ID.
    - Return value: void
- **save_device_parameters()**
    - Function: The function sends a ‘Save Device Parameters’ request.
    - Parameters: none
    - Return value: void
- **go_online()**
    - Function: The function sends a ‘Go Online’ request.
    - Parameters: none
    - Return value: void

The ‘yitran_host’ class defines exceptions which are derived from the ‘exception’ class defined in the Standard Template Library. The ‘yitran_host’ class defines the following exceptions:

- checksum_error_exception: A checksum error is detected during packet reception.
- packet_reception_timeout_exception: A packet reception timed out.
- reset_response_timeout_exception: The expected reset response timed out.

- invalid_nc_database_index_exception: The NC database index given to the function is invalid.
- invalid_user_payload_size_exception: The user payload size exceeds the maximum value.
- packet_tx_admission_failed_exception: The packet transmission process failed.

The ‘yitran_nc’ class defines the following data members:

- **public short rs_address_table[256]**: If the ‘one-shot network formation’ method is used for network management (see Section 4.2.2.1.), this vector stores the logical addresses of the RS nodes connected to the NC node.
- **unsigned number_of_connected_rss**: If the ‘one-shot network formation’ method is used for network management (see Section 4.2.2.1.), this variable stores the number of already connected RS nodes.

The ‘yitran_nc’ class provides the following public functions:

- **Constructor parameters**
  - bool use_console_port: The Intel Edison compute module includes two UART channels. One of them is shared between a service of the operating system sending console information to this channel and the user applications. If this UART channel is intended to be used in a user application, the corresponding service shall be disabled in the operating system. The constructor parameter defines, which UART channel shall be initialized for the user applicartion.
- **send_set_luminance_command()**
  - Function: The function sends a 'data exchange request' to the RS node with logical id of 'destination_id'. The 'data exchange request' indicates that the NC node wants to access some data provided by the RS node or to adjust the luminance of it. The 'set luminance commad' consists of the following bytes:
    - byte #0 (sent first): 0x80: command header
    - byte #1: flag indicating whether the luminance shall be adjusted or not
    - byte #2: new luminance value

    This function sends the data exchange request with byte #1 set to 0x01.
  - Parameters
    - short destination_id: The logical address of the RS node addressed.
    - unsigned char new_luminance: The new luminance value.
  - Return value: void
- **send_full_hk_request_command()**
  - Function: The function sends a 'data exchange request' to the RS node with logical id of 'destination_id'. The 'data exchange request' indicates that the NC node wants to access some data provided by the RS node or to adjust the luminance of it. The 'set luminance commad' consists of the following bytes:
    - byte #0 (sent first): 0x80: command header
    - byte #1: flag indicating whether the luminance shall be adjusted or not
    - byte #2: new luminance value

    This function sends the data exchange request with byte #1 set to 0x00.
  - Parameters
    - short destination_id: The logical address of the RS node addressed.

    - Return value: void
- **decode_data_exchange_reply()**
    - Function: When an RS node receives a 'data exchange request', it'll send a set of HK data back to the NC node, and it'll adjust the luminance, if it's needed (see byte #1 in 'data exchange request'). When the NC node receives the reply packet to a 'data exchange request', it shall decode it by calling this function. This function processes the 'recently_received_packet' of the yitran_nc object (publicly inherited from yitran_host). The function returns a complete 'rs_hk_data_set' struct with the recently received HK data.
    - Parameters: none
    - Return value
        - rs_hk_data_set: A structure containing the recently received HK data.
- **reset_and_go_online()**
    - Function: The function initializes the Yitran modem as an NC node, saves the device parameters into the non-volatile memory, resets the modem and sends a go-online request.
    - Parameters
        - unsigned char network_size: The number of expected RSs in the network.
    - Return value: bool: The function returns true, if the initialization is successful, it returns false otherwise.
- **initialize_and_establish_a_network()**
    - Function: The function initializes the NC node and the PLC network automatically. If ‘initial_number_of_rss’ nodes are connected, the function returns true. In this case, ‘rs_address_table’ is ready to use. The network_size stores the number of nodes in the PLC network. In case of a timeout, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char initial_number_of_rss: The number of the initial expected RS nodes in the PLC network. After ‘initial_number_of_rss’ RSs have connected to the NC, the NC may send the ‘traffic_enabled_indication’ to the connected RSs.
        - unsigned network_size: The number of expected RS nodes in the PLC network.
    - Return value: bool: If ‘initial_number_of_rss’ nodes are connected, the function returns true.
- **send_data_traffic_enabled_indication()**
    - Function: After the PLC network is established, this function may be used to send an indication to the RSs that they can start to send and receive user packets.
    - Parameters: none.
    - Return value: void
- **decode_connectivity_status_with_rs_indication()**
    - Function: After receiving a ‘Connectivity Status with RS’ indication, this function may be used to get the relevant information stored in the payload (command data) field of the indication. Relevant information:
        - node ID: The ID of the node sending the indication.

        - Connectivity status: The new connectivity status of the node.
    - Parameters
        - yitran_node_descriptor& nd: A temporary variable storing the relevant data of the recently received indication.
    - Return value: void.
- **decode_new_connection_to_nc_indication()**
    - Function: After receiving a ‘New Connection to NC’ indication, this function may be used to get the relevant information stored in the payload (command data) field of the indication. Relevant information:
        - node ID: The logical ID of the node sending the indication.
        - parent ID: The logical ID of the node's parent.
    - Parameters
        - yitran_node_descriptor& nd: A temporary variable storing the relevant data of the recently received indication.
    - Return value: void.
- **wait_for_network_id_assigned_indication()**
    - Function: The function receives a single packet. If the received packet is a ‘Network ID Assigned’ indication, the function returns true, otherwise it returns false.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if network ID assigned indication has been successfully received, false otherwise.
- **wait_for_new_connection_to_nc_indication()**
    - Function: The function receives a single packet. If the received packet is a ‘New Connection to NC’ indication, the function returns true, otherwise it returns false. If the received packet is a ‘New Connection to NC’ indication, the parameters of the new node are stored into nd as follows:
        - node ID: received
        - parent ID: received
        - serial number: 0
        - connectivity status: 1
    - Parameters
        - yitran_node_descriptor& nd: A temporary variable storing the relevant data of the recently received indication.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if new connection to NC indication has been successfully received, false otherwise.
- **acknowledged_set_nc_operation_mode()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the operation mode to NC. Thereafter, the function waits for the response and checks the status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.

    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **verified_set_nc_operation_mode()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the operation mode to NC. Thereafter, the function waits for the response and checks the status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the status field. If the status field indicates command error, the function returns false immediately. If the status field indicates a successful command, the function sends a ‘Get Device Parameters’ request, reading the operation mode. Thereafter, the function waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the operation mode value. If the operation mode is set to NC, the function returns true, otherwise it returns false.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the operation mode is successfully set to NC, false otherwise.
- **acknowledged_set_nc_database_size()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the NC database size to size. Thereafter, the function waits for the response and checks the status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char size: The new size of the NC database.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **verified_set_nc_database_size()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the NC database size to size. Thereafter, the function waits for the response and checks the status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the status field. If the status field indicates command error, the function returns false immediately. If the status field indicates a successful command, the function sends a ‘Get Device Parameters’ request, reading the NC database size. Thereafter, the function waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the NC database size value.
    - Parameters
        - unsigned char size: The new size of the NC database.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.

    - Return value: bool: If the NC database size is set to size, the function returns true, otherwise it returns false.
- **get_nc_database_size()**
    - Function: The function sends a ‘Get Device Parameters’ request addressing NC database size. Thereafter, the function waits for the response for timeout_ms. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. If the response is successfully received, the response is stored into ‘recently_received_packet’.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: unsigned char: NC database size
- **get_current_nc_database_size()**
    - Function: The function sends a ‘Get NC Database Size’ request addressing. Thereafter, the function waits for the response for ‘timeout_ms’. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. If the response is successfully received, the response is stored into ‘recently_received_packet’.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: unsigned char: Current number of RSs connected to the NC.
- **get_node_information()**
    - Function: The function sends a ‘Get Node Information’ request. The requested node is identified based on the NC database index. If the NC database index is invalid (greater than ‘NC database size’), the function throws an ‘invalid_nc_database_index_exception’ exception. In case of timeout, a ‘packet_reception_timeout_exception’ exception is thrown. The node information is stored into nd.
    - Parameters
        - unsigned char node_index: The logical ID of the node, which information is needed from.
        - yitran_node_descriptor& nd: A temporary variable storing the relevant data of the recently received indication.
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exceptionis thrown.
    - Return value: bool: If the response's status field indicates a successful command, the function returns true, otherwise it returns false.

The ‘yitran_rs’ class provides the following public functions:

- **Constructor parameters**
    - bool use_console_port: The Intel Edison compute module includes two UART channels. One of them is shared between a service of the operating system sending console information to this channel and the user applications. If this UART channel is intended to be used in a user application, the corresponding service shall be disabled in the operating system. The constructor parameter defines, which UART channel shall be initialized for the user applicartion.

- **reset_and_go_online()**
    - Function: The function initializes the Yitran modem as an RS node, saves the device parameters into the non-volatile memory, resets the modem and sends a go-online request.
    - Parameters
        - unsigned char network_size: The number of expected RSs in the network.
    - Return value: bool: The function returns true, if the initialization is successful, it returns false otherwise.
- **initialize_and_login()**
    - Function: The function initialized the RS node and after the address reception is done, it sends a login message to the NC. In case of a timeout, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char network_size: The number of expected RSs in the network.
    - Return value: bool: If the RS initialization, address reception, and login message transmission is successful, the function returns true, otherwise it returns false.
- **wait_for_data_traffic_enabled_indication()**
    - Function: The function waits for the ‘data traffic enabled’ indication from the NC. In case of a timeout, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned char network_size: The number of expected RSs in the network.
    - Return value: bool: If the indication received is a valid enable data traffic indication, the function returns true, otherwise it returns false.
- **decode_disconnected_from_nc_indication()**
    - Function: After receiving a ‘Disconnected from NC’ indication, this function may be used toget the relevant information stored in the payload (command data) field of the indication. Relevant information: Reason of disconnection.
    - Parameters: none
    - Return value
        - disconnect_reason (enumerated type) (see Appendix A for exact definitions)
            - parent_unstable
            - nvr_nack
            - infinity
            - init
            - cant_start_timer
            - nvr_refused
            - nvr_enq
            - invalid_node_id
            - disconnected_by_application_request
            - unknown_reason
- **decode_connected_to_nc_indication()**

    - Function: After receiving a ‘Connected to NC’ indication, this function may be used to get the relevant information stored in the payload (command data) field of the indication. Relevant information: parent ID: The logical ID of the node's parent.
    - Parameters
        - yitran_node_descriptor& nd: A temporary variable storing the relevant data of the recently received indication.
    - Return value: void
- **wait_for_connected_to_nc_indication()**
    - Function: The function receives a single packet. If the received packet is a ‘Connected to NC’ indication, the function returns true, otherwise it returns false. Note: ‘Connecting to NC’ timeout recommendation: 30 sec
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if Connected to NC indication has been successfully received, false otherwise.
- **acknowledged_set_rs_operation_mode()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the operation mode to RS. Thereafter, the function waits for the response and checks the "status" field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the status field of the response indicates that the command was executed successfully, false otherwise.
- **verified_set_rs_operation_mode()**
    - Function: The function sends a ‘Set Device Parameters’ request setting the operation mode to RS. Thereafter, the function waits for the response and checks the status field. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the status field. If the status field indicates command error, the function returns false immediately. If the status field indicates a successful command, the function sends a ‘Get Device Parameters’ request, reading the operation mode. Thereafter, the function waits for the response. If no response comes back, the function throws a ‘packet_reception_timeout_exception’ exception. If a response is received, the function checks the operation mode value. If the operation mode is set to RS, the function returns true, otherwise it returns false.
    - Parameters
        - unsigned timeout_ms: If no packet is received within this time period, a ‘packet_reception_timeout_exception’ exception is thrown.
    - Return value: bool: True, if the operation mode is successfully set to RS, false otherwise.

4.2.2. Yitran host API use cases

The above described Yitran host API may be used in different ways to implement user applications. In this section, two possible solutions are presented, aiming different purposes. They differ mainly in the ways they separate Yitran network management and application logic tasks.

The nominal PLC network operation from the viewpoint of the NC node and an RS node is shown in Fig. 18 and Fig. 19 respectively.

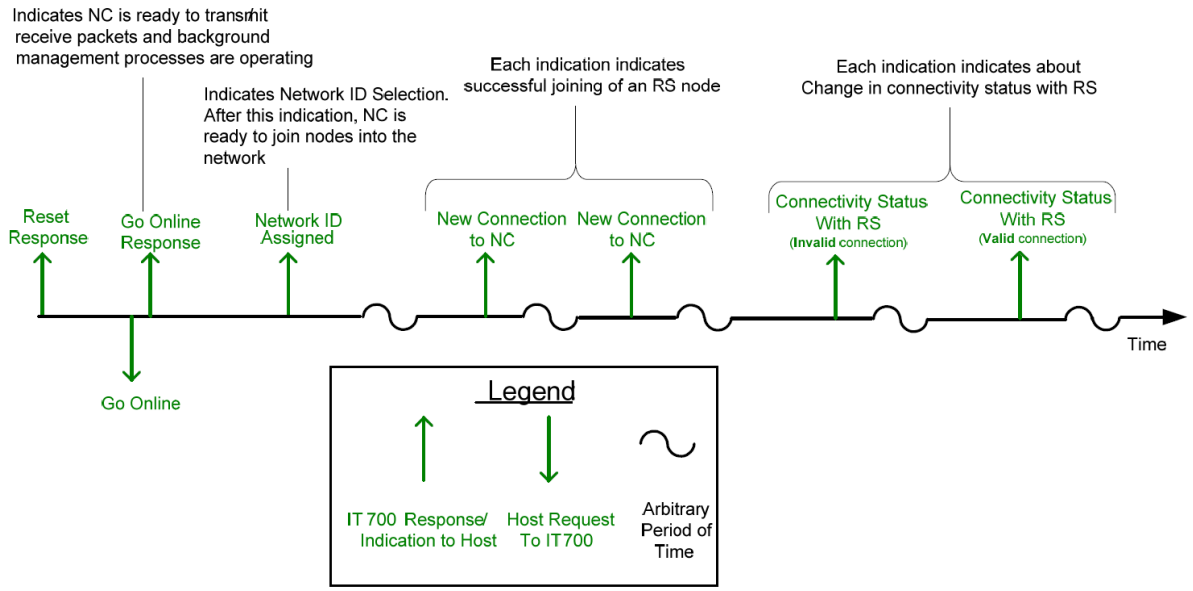


Fig. 18.Opearation of the PLC network from the viewpoint of the NC node.

Fig. 18 indicates that the user application running on the NC node shall only initialize the NC node modem and wait for indications, which may carry connectivity information or user data.

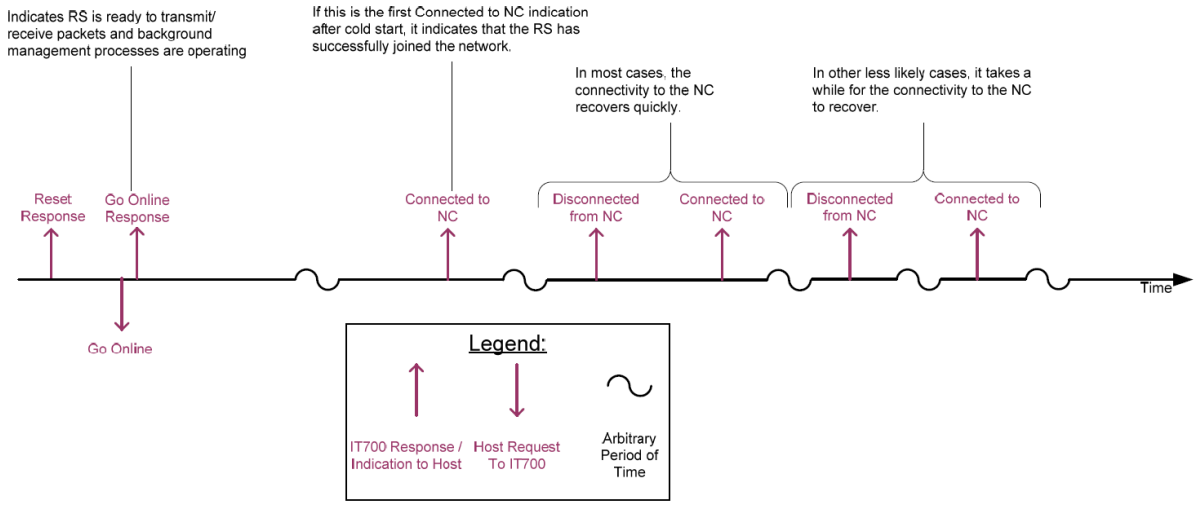


Fig. 19.Opearation of the PLC network from the viewpoint of an RS node.

Based on Fig. 19, the user application running on the RS node may be very similar to that one running on the NC node; It shall initialize the modem, and wait for indications carrying connection information or user data.

However, during development, some bugs have been found in the applied Yitran IT700 modem (or its documentation). The problem can be summarized as follows: The user application running on the NC node does not receive any indications from the IT700 SoC's 8051 microprocessor regarding RS connections ('New Connection to NC', 'Connectivity Status with RS'). However, only the indications are missing, the lower layers seem to establish the connections between the nodes. Therefore, a subset of network management tasks had to be mapped onto the application layer instead of the network layer provided by the IT700 modem. That means that the routing table carrying the node IDs of the already connected RS nodes are managed by the user application. Two solutions have been developed to perform this task:

- **One-shot network formation**: Using this method, the PLC network with all its RS nodes is formed before any user application logic activities. This solution is developed for testing only, it assumes a very reliable connection between the nodes. There is no chance to reconnect a node, if it is disconnected. The advantage is that the application logic and the PLC network management can be completely separated from each other, which makes this solution very easy to apply and favorable during application logic development and debugging.
- **Continouos network management:** Using this method, the application logic and the network management tasks are running in a round-robin manner. The advantage of this solution is that it very robust, both the NC and the RSs handle disconnections, so in case of hardware errors and noisy interconnection medium, the network remains stable because of the connection status detection and correction algorithms implemented by the nodes. The disadvantage is that in this case the user application logic itself is embedded in a pre-defined, strict round-robin architecture, which makes the development and debugging more difficult, and it results in a slower user application logic and debug cycles.

In the following subsections, detailed presentations are provided, how a user application can be created with the different network management methods.

4.2.2.1. One-shot network formation

The one-shot network formation method is developed for testing purposes. In this solution, the PLC network is established before any user application logic activities. Fig. 20 shows a simplified flow chart of the user applications applying one-shot network formation method.

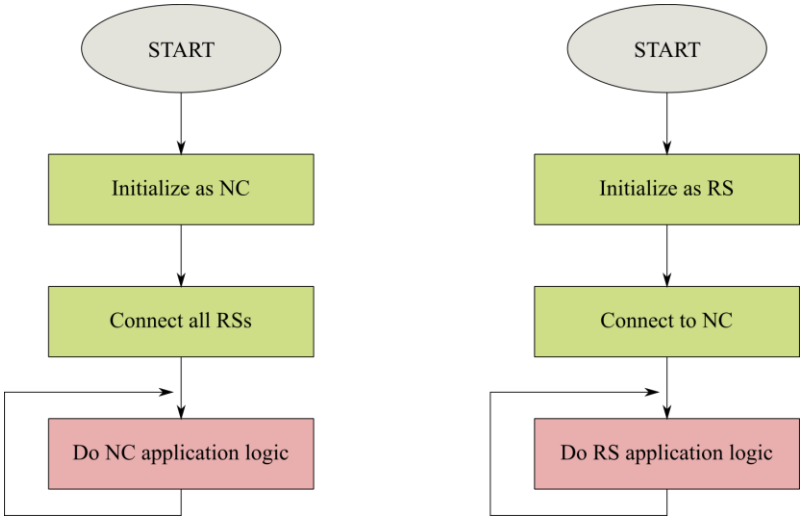


Fig. 20. One-shot network formation mechanism from the viewpoint of the NC node (left) and an RS node (right).

Since the connection indications on the NC side (see Fig. 18) seem to be missing, another hand-shake mechanism is implemented to make it possible for the NC node to detect new RSs and build its own routing table. As the connection between the NC and the new RS is established on the network level, the application level host of the RS node is signalled that it is connected to the NC. Since the NC does not know anything about this connection, the RS node sends a simple intranetworking packet to the NC with the node ID it got from its network layer.

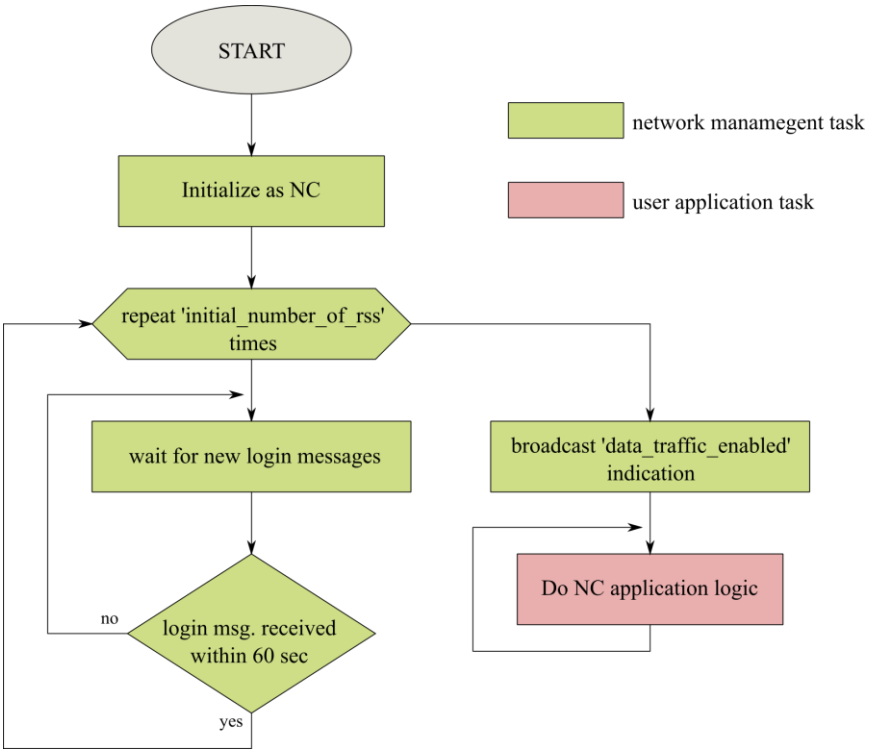


Fig. 21. A more detailed scheme of the one-shot network formation mechanism from the viewpoint of the NC node.

As the NC node receives this intranetworking packet with the special ‘login request’ mark in the packet header, the NC can store the recently received node ID in its routing table.

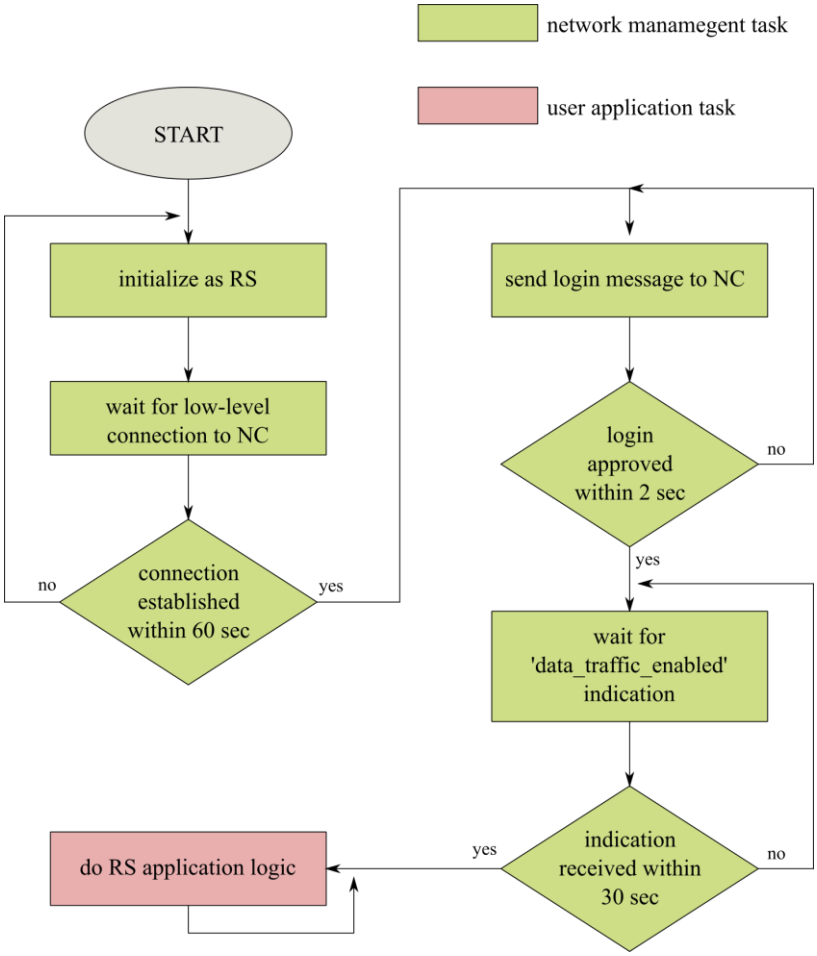


Fig. 22. A more detailed scheme of the one-shot network formation mechanism from the viewpoint of the RS node.

As all RSs have sent a login request to the NC and the NC stored all of them into its routing table, the network is considered established. The NC sends a ‘data_traffic_enabled’ indication (another special intranetworking packet) to all the registered RS nodes and starts its user application logic. As the RS nodes receive the ‘data_traffic_enabled’ indication, they also start their user application logics. Fig. 21 and Fig. 22 show a more detailed flow diagram of the one-shot network formation mechanism in case of the NC and the RS nodes respectively.

The obvious disadvantage of this network formation method is that there is no chance for a disconnected RS node to reconnect. Actually, any hardware errors or noisy medium can cause complete network collapse. However, the API described above makes it possible to form a network very fast and clear (using a single API function on either side). In case of application logic testing, when the PLC network is based on a stable, low-noise physical medium (realized by short wiring without high voltages applied on them), this method is favorable.

4.2.2.2. Continouos network management

In the continouos network management method, the user application logic and the PLC network management tasks are performed in a round-robin manner. Every cycle in the application superloop contains subtasks regarding the connections of the PLC network and the user application logic itself (see Fig. 23).

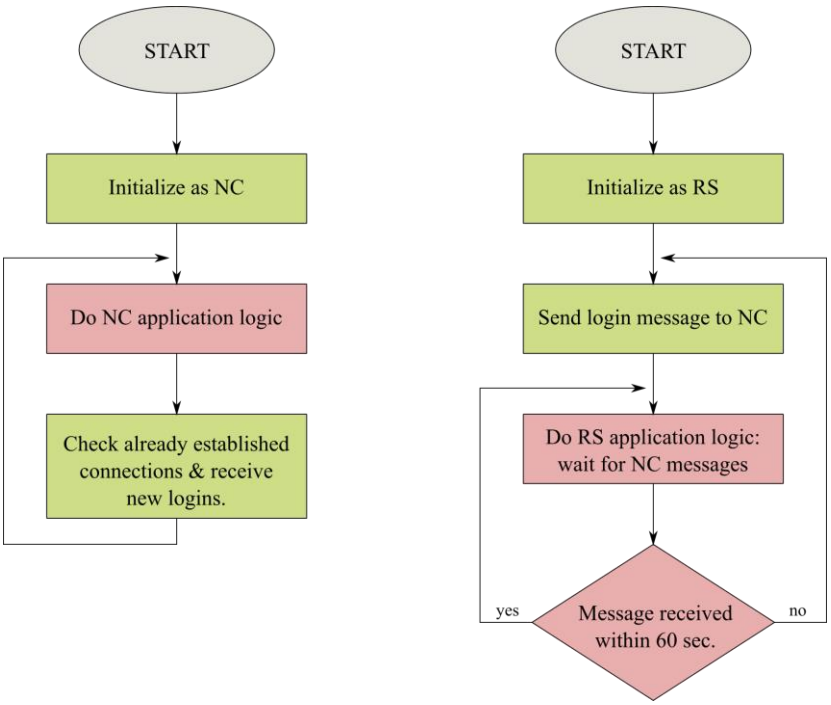


Fig. 23. The continouos network management mechanism from the viewpoint of the NC node (left) and an RS node (right).

The connections between the NC and the RSs are established in a similar way as seen in case of the one-shot network formation method. The same login message mechanism is used to notify the NC node about the RSs' logical ID got from the network layer. The difference is that in the continouos network management method, these connections are reconsidered cyclically. Fig. 24 and Fig. 25 show a more detailed flow diagram of the continouos network management method.

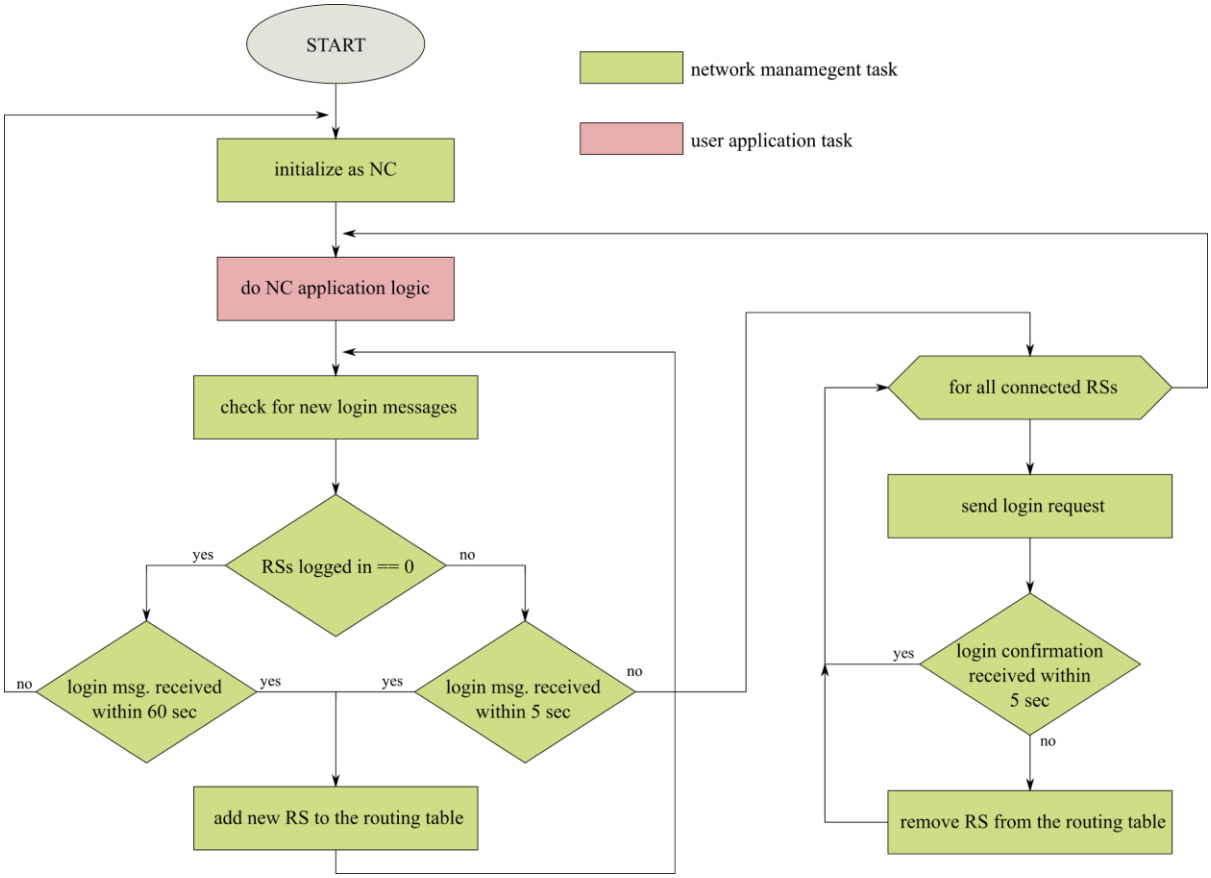


Fig. 24. A more detailed scheme of the continouos network management mechanism from the viewpoint of the NC node.

The main cycle of the NC node's application consists of three main parts:

- **Application logic**: The user application performing its tasks using the already connected RS nodes.
- **New connection detection**: The NC is looking for login messages from RS nodes, which are not part of the PLC network yet.
- **Connection confirmation**: The NC node sends login confirmation requests to the RS nodes, which are already connected to the PLC network.

Based on the responses received during the last two steps, the NC node cyclically refreshes its routing table.

Fig. 25. A more detailed scheme of the continouos network management mechanism from the viewpoint of the RS node.

The continouos network management method may be used in every circumstances regarding hardware errors and the quality of the physical medium. If an RS node disconnects for some reason, its application will eventually detect the problem and it will restart its PLC modem. On the other hand, the NC is also able to detect the missing link and it can manage its routing table accordingly.
Additionally, the scheme presented above is also able to handle modem errors. Both the NC and the RS application layer is able to detect “suspicious” silence on the PLC network. In such cases, the applications will restart their modems to ensure that the PLC network remains operable.

4.2.3. Application logic

The description of the application logic is not part of this documentation.

4.2.4. Software tests

4.2.4.1. Standalone API tests

- SU API test (th02_test-cpp): The test verifies the functionality of the TH02 sensor API.

- LDCU API test (led_interface_test.cpp): The test verifies the functionality of the 'ppp_link' and the 'led_interface' classes. The test covers the functionalities of the LED DAQ and control unit and the interface denoted as if-CCU-LDCU.

- MSM standalone tests: These tests verify the collaboration between the Yitran modem API functions created for the NC side and the RS side.
    - NC side (yitran_nc_app.cpp)
        - The Yitran modem is reset and initialized as NC.
        - A network creation is initialized according to the one-shot network formation method described in Section 4.2.2.1.
        - After successful network creation, the NC node broadcasts the 'data traffic enabled' indication.
    - RS side (yitran_rs_app.cpp)
        - The Yitran modem is reset and initialized as RS.
        - After initialization, the RS node wait for the 'data traffic enabled' indication from the NC node.

4.2.4.2. Low-level integration tests

- Intelligent LED control test (temp_feedback.cpp): The test verifies the collaboration between the SU and the LDCU by implementing a control-loop keeping the LEDs in an optimal operation point. The optimal LED string current is determined by the temperature measured by the SU (TH02 temperature sensor). The actuation is performed indirectly through the LDCU API, which adjusts the LED luminance (thus the current) by controlling the DALI-capable LED driver (see Section 4.1.5).
    - if-CCU-LDCU is initialized (the connection between the Intel Edison compute module and the LED DAQ and control unit is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
    - if-CCU-SU is initialized (the connection between the Intel Edison compute module and the TH02 sensor is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
    - A luminance query is sent to the LDCU. The recent luminance is stored.
    - A temperature query is sent to the SU. The recent temperature is stored.
    - A current query is sent to the LDCU. The recent LED string current is stored.
    - The optimal operation point of the LED strings is calculated based on the recent temperature read from the SU.
    - If the recent operating point current is less than the optimal value, the luminance is increased by sending a luminance adjustment query to the LDCU. If the recent operating point current is greater than the optimal value, the luminance is decreased by sending a luminance adjustment query to the LDCU.
    - The above described measurement-adjustment cycle is repeated infinitely.

- One-shot network formation and data exchange tests
    - Data exchange test - NC side (full_test_nc.cpp)
        - The Yitran modem is initialized as NC.
        - A PLC network is initialized using the one-shot network formation method described in Section 4.2.2.1.
        - After the PLC network is formed, the NC broadcasts the ‘data traffic enabled’ indication.
        - The luminance value is defined and a data exchange request is sent to the RS nodes connected to the PLC network.
        - The replies sent back by the RS nodes include all HK information. After packet reception, the NC node decodes the packet and displays the HK data set on the standard output.
        - The luminance adjustment - HK data query cycle is repeated infinitely.
    - Data exchange test - RS side (full_test_rs.cpp)
        - if-CCU-LDCU is initialized (the connection between the Intel Edison compute module and the LED DAQ and control unit is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
        - if-CCU-SU is initialized (the connection between the Intel Edison compute module and the TH02 sensor is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
        - The Yitran modem is initialized as RS.
        - The RS waits for the ‘data traffic enabled’ indication from the NC node.
        - After receiving the ‘data traffic enabled’ indication, the RS waits for data exchange requests.
        - If a data exchange request is received by the RS, it adjusts the LED luminance values through the LDCU interface according to the request packet sent by the NC.
        - After luminance adjustment, the RS constructs a reply packet including all HK data and it sends them back to the NC node.
        - After sending the respone packet, the RS waits for another data exchange requests from the NC node.

- Continouos PLC network management and exemplary PLC application logic tests
    - Exemplary application logic test - NC side (application_nc.cpp)
        - The Yitran modem is reset and initialized as NC.
        - The NC side of the network management process described in Section 4.2.2.2 is started with the following application logic functionality:
            - A random luminance value is defined and a data exchange request is sent to the RS nodes connected to the PLC network.
            - The replies sent back by the RS nodes include all HK information. After packet reception, the NC node decodes the packet and displays the HK data set on the standard output.
    - Exemplary application logic test - RS side (application_rs.cpp)

- if-CCU-LDCU is initialized (the connection between the Intel Edison compute module and the LED DAQ and control unit is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
- if-CCU-SU is initialized (the connection between the Intel Edison compute module and the TH02 sensor is initialized and checked). The connection is re-initialized in case of connection error wihin 1 second.
- The Yitran modem is reset and initialized as RS.
- The RS side of the network management process described in Section 4.2.2.2 is started with the following application logic functionality:
    - The RS waits for data exchange requests.
    - If a data exchange request is received by the RS, it adjusts the LED luminance values through the LDCU interface according to the request packet sent by the NC.
    - After luminance adjustment, the RS constructs a reply packet including all HK data and it sends them back to the NC node.
    - After sending the respone packet, the RS waits for another data exchange requests from the NC node.

4.2.4.3. Application logic tests

The description of the application logic is not part of this documentation.

5. References

1. “IEEE Standard for Broadband over Power Line Networks: Medium Access Control and Physical Layer Specifications”, IEEE Std 1901-2010
2. “Waze Social GPS, Map and Navigation”, https://www.waze.com Visited on: 11/07/2016
3. DALI: http://www.dali-ag.org/discover-dali/dali-standard.html
4. “Intel edison, one tiny platform, endless possibility,” http://www.intel.com/content/www/us/en/do-it-yourself/edison.html Visited on: 07/07/2016
5. http://hackerboards.com/edison-iot-module-ships-with-atom-plus-quark-combo-soc/ Visited on: 07/07/2016
6. MRAA documentation - http://iotdk.intel.com/docs/master/mraa/

6. Appendix

Appendix A - Yitran IT700 Host Interface Command Set User Guide
Appendix B - Intel Edison Compute Module
Appendix C - Intel Edison Breakout Board
Appendix D - Digital I2C Temperature and Humidity Sensor
Appendix E - The detailed schematic and BOM of LDCU PCB
Appendix F - LCM-25 DA specification
Appendix G - Photo documentation
Appendix H - Connector pin configurations