UM1459
User manual

STEVAL-IHP002V2: PLM smartplugV2 firmware

Introduction

This document describes the PLM smartplug application in terms of firmware architecture and source code.

The application described refers to the STEVAL-IHP002V2 demonstration board (see related hardware user manual UM1458, for more details).

The firmware application and the related smartplug board is an evaluation system providing guidelines on how to develop a home/building automation subsystem for energy management.

The firmware application is developed for the STM32F103CB microcontroller and implements a PLM node which allows the final user to monitor and manage the plugged load energy consumption.

The current, power, energy and other information related to the electrical load connected to the smartplug board are sent to a PLM data concentrator through the home/building PLM network.

This application is a firmware package which contains a collection of libraries, routines, data structures and macros covering the application features. As a result, using the application firmware as a starting point, it saves significant time that would otherwise be spent in other aspects of system development, while reducing the application development and integration cost.

- *Section 1* describes the document and library rules
- *Section 2* describes the smartplug hardware demonstration board
- *Section 3* describes its modular architecture and highlights each module individually
- *Section 4* describes the application modules running on the OS
- *Section 5* describes the libraries which are OS independent

Contents

List of tables

List of figures

1 Document and library rules

This document uses the conventions described in the sections below.

1.1 Acronyms

Table 1 describes the acronyms used in this document:

Table 1. List of acronyms

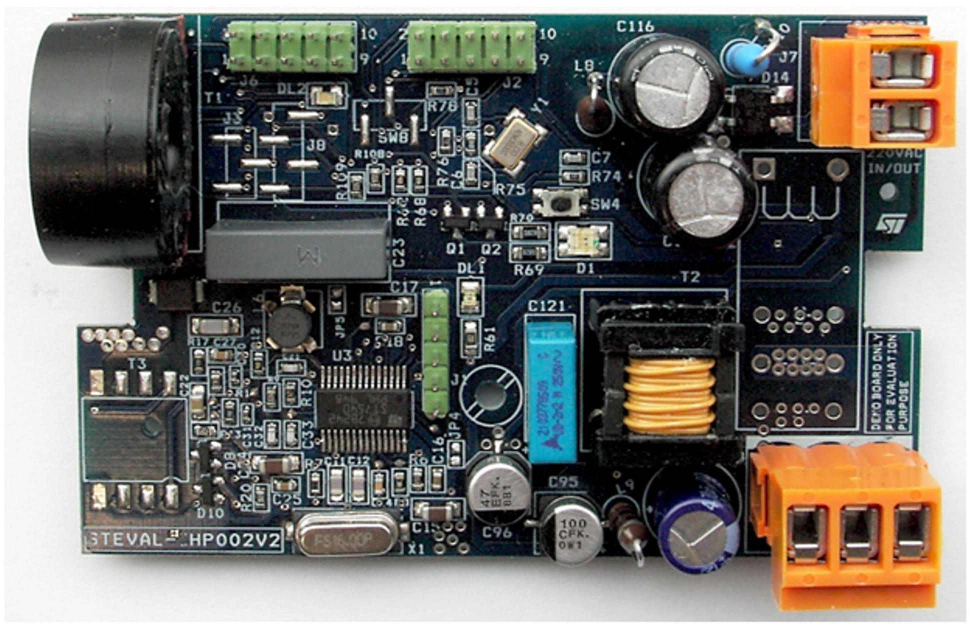
Acronym	Meaning
APP	Application
API	Application programming interface
HAL	Hardware abstraction layer
MCU	Microcontroller unit
SPI	Serial peripheral interface
OOP	Object oriented programming

2 Smartplug demonstration board

2.1 Board introduction

The application firmware described in this document refers to the STEVAL-IHP002V1 demonstration board (see the UM1458 user manual).

Figure 1. STEVAL-IHP002V2 smartplugV2 demonstration board



The board has been developed to provide a guideline to build a home/building automation subsystem for energy management. In a typical home system implementation, the board is plugged into an electrical wall junction box and supplies a home appliance or other generic electrical load through a wall socket. Information about voltage, current, power and energy absorbed by the load are sent to a data concentrator through the PLM network.

Warning: There is no protection against accidental human contact with high voltages. After disconnection of the board from the mains, none of the live parts should be touched immediately because of the energized capacitors. It is mandatory to use a mains insulation transformer to perform any debugging/tests on the board in which debugging and test instruments such as USB-JTAG dongles, spectrum analyzers or oscilloscopes are used. Do not connect any oscilloscope probes to high voltage sections in order to avoid damaging instruments and demonstration tools. STMicroelectronics assumes no responsibility for any consequences which may result from the improper use of this tool.

2.2 Application and HW components

The firmware application running on STM32F10xxx manages the following hardware components assembled on the smartplug demonstration board:

- An STPM01 single-phase energy metering IC: used for load energy consumption measurement
- An ST7540 power line communication B-FSK mode
- 1 relay: used to turn on/off the load
- 1 bi-color status LED
- 1 configuration jumper: used for running mode
- 1 reset button: used to force an MCU reset
- 1 USART communication for configuration purposes

3 Smartplug firmware application

3.1 Introduction

The smartplug application firmware is fully developed in C language. IAR EWARM v5.40 has been used to create the workspace/project and the IAR solution is included in the final application delivery. By opening the IAR project it's possible to change the source code, download the firmware on the STM32F10xxx microcontroller assembled on the smartplug board and debug the code itself.

The application runs with the FreeRTOS real-time operating system for easy application module development and integration, for detailed information please refer to the documentation on www.freertos.org.

The physical and data-link layers of the communication module are based on the firmware related to the AN3046 application note.

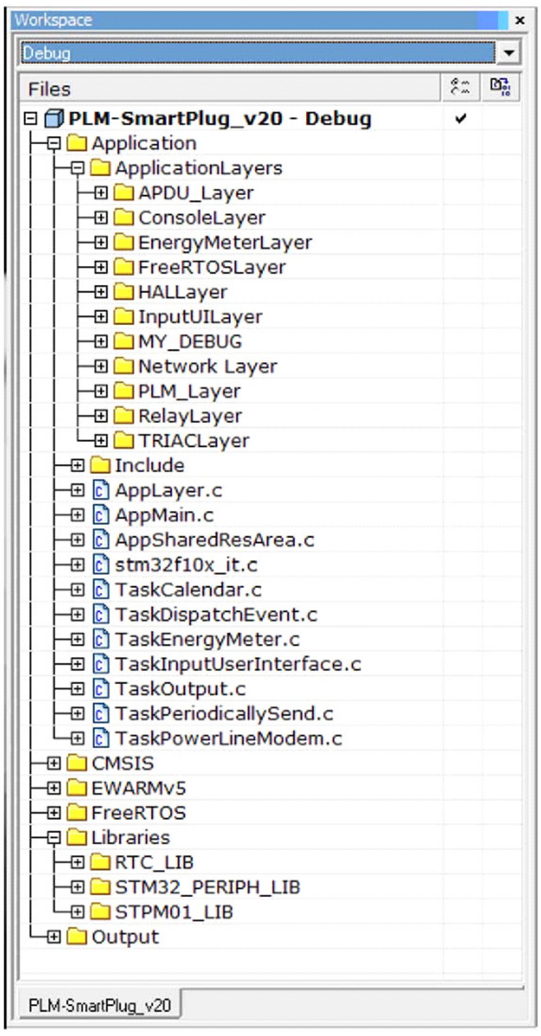
The application includes version 3.1.2 of the STM32 standard library which is CMSIS compliant.

3.2 Application firmware package

The smartplug application firmware, developed using IAR EWARM 5.40, is delivered in a ZIP file and contains all the subdirectories and .h and .c source code files that make up the core of the application. The related IAR workspace/project files are included also.

The IAR project is organized in project folders coherent with file-system folders, (see *Figure 2*):

Figure 2. Project organization



```
Workspace
Debug
Files
⊟ PLM-SmartPlug_v20 - Debug                    ✓
  ⊟ Application
    ⊟ ApplicationLayers
      ⊞ APDU_Layer
      ⊞ ConsoleLayer
      ⊞ EnergyMeterLayer
      ⊞ FreeRTOSLayer
      ⊞ HALLayer
      ⊞ InputUILayer
      ⊞ MY_DEBUG
      ⊞ Network Layer
      ⊞ PLM_Layer
      ⊞ RelayLayer
      ⊞ TRIACLayer
    ⊞ Include
    ⊞ AppLayer.c
    ⊞ AppMain.c
    ⊞ AppSharedResArea.c
    ⊞ stm32f10x_it.c
    ⊞ TaskCalendar.c
    ⊞ TaskDispatchEvent.c
    ⊞ TaskEnergyMeter.c
    ⊞ TaskInputUserInterface.c
    ⊞ TaskOutput.c
    ⊞ TaskPeriodicallySend.c
    ⊞ TaskPowerLineModem.c
  ⊞ CMSIS
  ⊞ EWARMv5
  ⊞ FreeRTOS
  ⊟ Libraries
    ⊞ RTC_LIB
    ⊞ STM32_PERIPH_LIB
    ⊞ STPM01_LIB
  ⊞ Output
PLM-SmartPlug_v20
```

- The application project-folder contains all the application task source files and the related modules files and consists of the following project folders:
  - APDU_Layer: includes the commands implementation related to the application layer of the communication protocol stack. The commands are used to write/read a set of internal variables by remote communication
  - ConsoleLayer: includes the basic function to implement a user I/O console by RS232 using a PC
  - EnergyMeterLayer: contains all the functions for low level initialization of the energy meter IC
  - FreeRTOSLayer: includes only the freeRTOS configuration header file
  - HalLayer: includes some common hardware abstraction functions
  - MY_DEBUG: includes the implementation functions of the I/O terminal commands
  - Network_Layer: is the implementation of the network layer of the communication protocol stack
  - PLM_Layer: implements the physical and data-link layers of the communication protocol stack and includes the low level function for PLC modem management
  - RelayLayer: implements the object to drive the relay.

The libraries project folder contains the libraries layer files directly used by the application. The libraries consist of the following project folders:

  - RTC_Lib: is the library handling the MCU internal RTC used for calendar implementation
  - STM32_PERIPH_LIB: is the STM32 MCU standard library
  - STPM01_LIB: is the library for energy meter IC management including the functions for PowerMeter object handling and HAL layer
  - FreeRTOS project folder contains all the freeRTOS operating system source files.

3.3 Application firmware architecture

Application architecture is planned and developed in separate tasks implemented in separate modules (see *Figure 3*):

- TaskCalendar.c
- TaskEnergyMeter.c
- TaskOutput.c
- TaskPeriodicallySend.c
- TaskPowerLineModem.c

This layers architecture improves the code reusability splitting the application programming interface code (portable and reusable) provided by the libraries layer from the hardware abstraction layer code (hardware dependent and written in the STM32F10xxx libraries). The application layer also includes the modules:

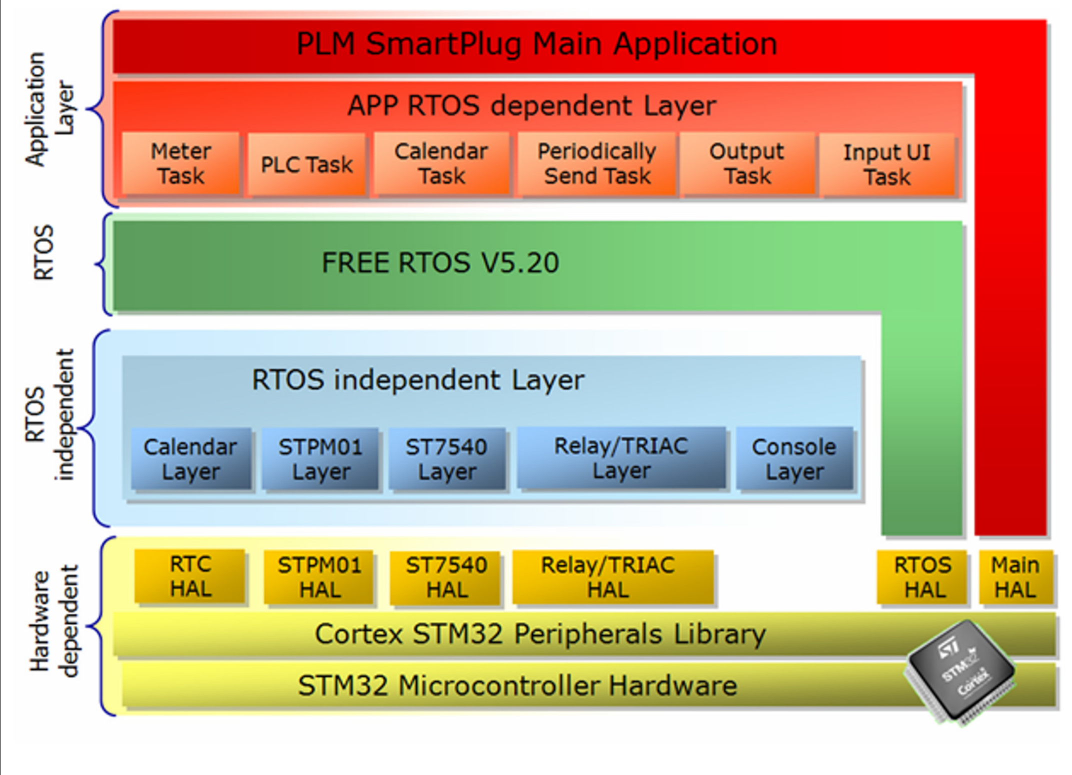
- AppMain.c
- AppLayer.c
- AppSharedResArea.c

The AppMain.c module includes only the main application entry point; the AppLayer.c includes the basic MCU initialization functions and the OS scheduler and tasks startup; the

AppSharedResArea.c includes all the shared variables and the functions used by the tasks for accessing.

The set of the functions exported by the FreeRTOS and the STM32 standard library form a common API layer for the tasks of the application layer.

Figure 3. firmware architecture



4 Application modules

This section describes the functions and the implementation of each application module. The modules are dependent of the FreeRTOS OS and the application functions are implemented as OS tasks. The functions are described in the following format.

Table 2. Function description format

Function	Description
Function name	The name of the peripheral function
Function prototype	Prototype declaration
Behavior description	Brief explanation of how the function is executed
Input parameter {x}	Description of the input parameters
Output parameter {x}	Description of the output parameters
Return value	Value returned by the function
Required preconditions	Requirements before calling the function
Called functions	Other library functions called

4.1 Calendar module

This module simply gets the counter value of the MCU real-time clock and updates the calendar variables stored in the shared resources area. The task is executed every 100 msec with priority 2.

The task is implemented in TaskCalendar.c by the function TaskCalendarFunc and the configurations are included in the TaskCalendar.h header file. The configuration is done by the following macros:

```
#define TASK_CALENDAR_NAME       (signed portCHAR *)"Calendar"
#define TASK_CALENDAR_STACK       configMINIMAL_STACK_SIZE
#define TASK_CALENDAR_PRIORITY    tskIDLE_PRIORITY+2
#define TASK_CALENDAR_RATE_MS     100 /* Rate Delay in ms */
```

4.2 Power line communication module

This module calls all the functions related to PLM communication: the modem initialization functions, the protocol state machine functions, and also the application layer state machine functions. The task is executed every 10 msec with priority 3.

The task is implemented in TaskPowerLineModem.c by the function TaskPowerLineModemFunc and the configuration macros are included in the TaskPowerLineModem.h header file:

```
#define TASK_PLM_COMM_NAME  (signed portCHAR *)"PLM Comm Mgmt"
#define TASK_PLM_COMM_STACK  512
#define TASK_PLM_COMM_PRIORITY  tskIDLE_PRIORITY+3
#define TASK_PLM_COMM_RATE_MS  10 /* Rate Delay in ms */
```

4.3 Periodically send module

This module periodically reads the load power consumption variable from the shared area and in case of variation, all the relevant measurement information for monitoring (mains voltage, load current, power and energy consumption) are read and put into a queue for sending through power line communication. The task is executed every 30 seconds with priority 5.

The task is implemented in TaskPeriodicallySend.c by the function TaskPeriodicallySendFunc and the configuration macros are included in the TaskPeriodicallySend.h header file:

```
#define TASK_PERIODICALLY_SEND_NAME        (signed portCHAR*)"PeriodicallySend"
#define TASK_PERIODICALLY_SEND_STACK       configMINIMAL_STACK_SIZE
#define TASK_PERIODICALLY_SEND_PRIORITY    tskIDLE_PRIORITY+5
#define TASK_PERIODICALLY_SEND_RATE_MS     30000 /* Rate Delay in ms */
```

4.3.1 Exported function

This module exports the following functions:

- CheckPeriodicallyMessage
- StopPeriodicallyMessage

4.3.2 CheckPeriodicallyMessage function

Table 3 describes the CheckPeriodicallyMessage function:

Table 3. CheckPeriodicallyMessage function

Functions	Description
Function name	CheckPeriodicallyMessage
Function prototype	APP_SM_Status_t CheckPeriodicallyMessage(unsigned portCHAR* buff, unsigned portCHAR* len)
Behavior description	Checks if the periodically message queue contains messages
Input parameter {x}	None
Output parameter {x}	Message buffer, buffer length
Return value	Transition action for PLM communication FSM
Required preconditions	TaskPeriodicallySendFunc must be running
Called functions	FreeRTOS API

Example:

```
APP_SM_Status_t  Application_Status;
u8 Network_data[STB_FRAME_MAX_LEN];
u8 APP_ReqDataLen;
...
Application_Status = CheckPeriodicallyMessage( &Network_data[5], &APP_ReqDataLen );
```

```
if (Application_Status == APP_CONN_REQUEST){
     /* send data */
…
}
```

4.3.3 StopPeriodicallyMessage function

Table 4 describes the StopPeriodicallyMessage function:

Table 4. StopPeriodicallyMessage function

Functions	Description
Function name	StopPeriodicallyMessage
Function prototype	Void CheckPeriodicallyMessage(void)
Behavior description	Stop the scheduling of the TaskPeriodicallySendFunc
Input parameter {x}	None
Output parameter {x}	None
Return value	None
Required preconditions	TaskPeriodicallySendFunc must be running
Called functions	FreeRTOS API

Example:

```
…
StopPeriodicallyMessage();
…
```

4.4 Output module

This module initializes the output object, reads the status variables from the shared resource area and drives the output devices. The task is executed every 1 second with priority 6.

The task is implemented in TaskOutput.c by the function TaskOutputFunc and the configuration macros are included in the TaskOutput.h header file:

```
#define TASK_OUTPUT_NAME         (signed portCHAR *)"Output"
#define TASK_OUTPUT_STACK        configMINIMAL_STACK_SIZE
#define TASK_OUTPUT_PRIORITY  tskIDLE_PRIORITY+6
#define TASK_OUTPUT_RATE_MS  1000 /* Rate Delay in ms */
```

4.5 Input UI module

This module manages the user interface through the RS232 interface; it checks for a valid input command and puts it into a queue for processing. The task is executed every 50 msec with priority 2.

The task is implemented in TaskInputUserInterface.c by the function TaskInputUserInterfaceFunc and the configuration macros are included in the TaskInputUserInterface.h header file:

```
#define TASK_INPUT_USER_INTERFACE_NAME       (signed portCHAR *)"Input UI"

#define TASK_INPUT_USER_INTERFACE_STACK      configMINIMAL_STACK_SIZE

#define TASK_INPUT_USER_INTERFACE_PRIORITY   tskIDLE_PRIORITY+2

#define TASK_INPUT_USER_INTERFACE_RATE_MS  50 /* Rate Delay in ms */
```

4.5.1 Exported function

This module exports the following function:

- CheckConsoleMessage

4.5.2 CheckConsoleMessage function

Table 5 describes the CheckConsoleMessage function:

Table 5. GetTriacObj function

Functions	Description
Function name	CheckConsoleMessage
Function prototype	u8 CheckConsoleMessage(USER_IN_T** buff)
Behavior description	Checks if the UI command queue contains messages for processing
Input parameter {x}	None
Output parameter {x}	Reference to UI command if available
Return value	Returns 0 if no messages are pending for processing, otherwise 1.
Required preconditions	The TaskInputUserInterfaceFunc task must be running.
Called functions	FreeRTOS APIs, console layer APIs.

Example:

```
USER_IN_T* input;

…

if ( CheckConsoleMessage( &input ) ){

/* process input */

…

}
```

4.6 Meter module

This module initializes the power meter object and periodically reads the measurements from STPM01 and puts them into metering data variables inside the shared resource area. The task is executed every 120 msec with priority 4. After energy meter initialization this task resumes the out module task.

The task is implemented in TaskEnergyMeter.c by the function TaskEnergyMeterFunc and the configuration macros are included in the TaskEnergyMeter.h header file:

```
#define TASK_ENERGY_METER_INTERFACE_NAME  (signed portCHAR *)"Energy Meter"

#define TASK_ENERGY_METER_INTERFACE_STACK    configMINIMAL_STACK_SIZE

#define TASK_ENERGY_METER_INTERFACE_PRIORITY     tskIDLE_PRIORITY+4

#define TASK_ENERGY_METER_INTERFACE_RATE_MS  120  /* Rate Delay in ms */
```

5 OS independent library modules

This section describes the libraries used by the application modules. These modules are independent from the FreeRTOS OS and can be considered as the libraries used by the application modules. The API functions are described in the following format:

Table 6. Function description format

Functions	Description
Function name	The name of the peripheral function
Function prototype	Prototype declaration
Behavior description	Brief explanation of how the function is executed
Input parameter {x}	Description of the input parameters
Output parameter {x}	Description of the output parameters
Return value	Value returned by the function
Required preconditions	Requirements before calling the function
Called functions	Other library functions called

5.1 RTC library

This library module implements a calendar using the STM32F embedded RTC and it is used by the TaskCalendar.c module. The date is calculated on the basis of the RTC counter by the function UpdateDate which must be periodically called. The date is maintained by the following data structure:

```
typedef struct{
  s32 year;
  s32 month;
  s32 day;
  s32 hh;
  s32 mm;
  s32 ss;
} DATE_T;
```

5.1.1 Exported function

This module exports the following functions:

- GetDate
- GetQuarter
- UpdateDate
- Time_Show
- Calendar_Init
- Time_Adjust

5.1.2 GetDate function

Table 7 describes the GetDate function:

Table 7. GetDate function

Functions	Description
Function name	GetDate
Function prototype	Void GetDate (DATE_T* date)
Behavior description	Copy the current date variable into the user variable referenced by the parameter
Input parameter {x}	None
Output parameter {x}	Current date
Return value	None
Required preconditions	RTC and calendar initialized
Called functions	None

Example:

```
DATE_T tmpDate;

…

GetDate(&tmpDate);

…

}
```

5.1.3 GetQuarter function

Table 8 describes the GetQuarter function:

Table 8. GetQuarter function

Functions	Description
Function name	GetQuarter
Function prototype	U8 GetQuarter (DATE_T date)
Behavior description	Calculates and provides the year quarter on the basis of the input date

Table 8. GetQuarter function (continued)

Functions	Description
Input parameter {x}	Date
Output parameter {x}	None
Return value	Quarter
Required preconditions	RTC and calendar initialized
Called functions	None

Example:

```
u8 tmpQuarter;

DATE_T tmpDate;

…

tmpQuarter = GetQuarter(tmpDate);

…

}
```

5.1.4 UpdateDate function

Table 9 describes the UpdateDate function:

Table 9. UpdateDate function

Functions	Description
Function name	UpdateDate
Function prototype	Void UpdateDate (u32 counter)
Behavior description	Calculates the current date using the input counter which is read from RTC and updates the internal variable.
Input parameter {x}	Counter
Output parameter {x}	None
Return value	None
Required preconditions	RTC and calendar initialized
Called functions	None

Example:

```
…

UpdateDate(RTC_GetCounter());

…

}
```

5.1.5 Time_Show function

Table 10 describes the Time_Show function:

Table 10. Update_Show function

Functions	Description
Function name	Time_Show
Function prototype	Void Time_Show (DATE_T date)
Behavior description	Shows the date passed as input parameter. This API is not used because the system doesn't include any display.
Input parameter {x}	Date
Output parameter {x}	None
Return value	None
Required preconditions	RTC and calendar initialized
Called functions	Time_Display internal function

Example:

```
DATE_T tmpDate;
…
Time_Show(tmpDate);
…
}
```

5.1.6 Calendar_Init function

Table 11 describes the calendar_init function:

Table 11. Calendar_Init function

Functions	Description
Function name	Calendar_Init
Function prototype	Void Time_Show (void)
Behavior description	Initializes the RTC
Input parameter {x}	Date
Output parameter {x}	None
Return value	None
Required preconditions	None
Called functions	RTC standard library functions

Example:

```
DATE_T tmpDate;

…

Time_Show(tmpDate);

…

}
```

5.2 STPM01 library

This library module implements the API function related to the management of the STPM01 energy meter IC. This module has been built on the basis of the library published with the AN2799 application note.

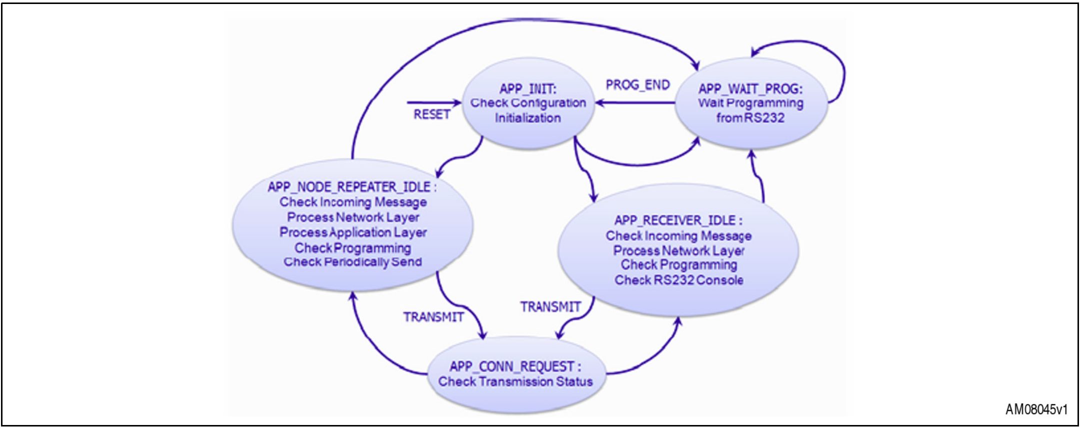
5.3 PLM layer

This module implements the basic communication function with the ST7540 PLM, for configuration and data transmit/receive, and also all remote communication protocol stack layers. The physical layer and data link layer are based on the protocol stack described in the AN3046 application note already mentioned. The main features are:

- PHY layer
  - CENELEC 50065 band C
  - FSK modulation with low frequency deviation
  - Centre frequency 132.5 kHz ± 0.2 % (± 0.25 kHz)
  - Bit rate 2400 bits/s.
  - Forward error correction (FEC) implemented to correct phase synchronous disturbances with a duration up to 1 ms
- Data link layer
  - Fixed length (telegrams) messages embedding 100 byte payload
  - ACK and Timeout data transmission services
  - CSMA/CA: Back-off min. (> ACK time) + random part (2 to 150 ms). The back-off algorithm is not exponential and it is recalculated each time the band is in use before the transmission
  - CRC16 protection

For further details on the implementation please refer to AN3046. The application layer described in AN3046 has been modified in order to add a network layer and the specific smartplug application layer on top of the data link layer. The state machine of this layer is implemented in application.c module and shown in *Figure 4*.

Figure 4. Application layer FSM


APP_INIT: Check Configuration Initialization
PROG_END
APP_WAIT_PROG: Wait Programming from RS232
RESET
APP_NODE_REPEATER_IDLE: Check Incoming Message Process Network Layer Process Application Layer Check Programming Check Periodically Send
APP_RECEIVER_IDLE: Check Incoming Message Process Network Layer Check Programming Check RS232 Console
TRANSMIT
TRANSMIT
APP_CONN_REQUEST: Check Transmission Status
AM08045v1

5.4 Network layer

This layer implements a network with a tree topology for communication between each smartplug inside the network and a data concentrator. The network has a static architecture so the communication path for communication from each smartplug to the data concentrator is fixed and configured at startup by RS232 and stored in the STM32F embedded flash using the following data structure:

```
typedef struct


{
  u16 DeviceType;
  u16 LocalID;
  u16 PrimaryRemoteID;
  u16 SecondaryRemoteID;
  u16 TestRemoteID;
  u16 TestTimeout;


}DEVICE_Data_t;
```

The device type (REPEATER/NODE/RECEIVER) and the local address are stored in the same structure too. A smartplug can be a normal node and also a repeater in order to allow the smartplugs far from the data concentrator to reach it even if they cannot communicate directly. The path for communication from the data concentrator to each smartplug is built at run-time with a back-routing algorithm described below. There are two kinds of network frames identified by the following enumerator:

```
enumerator:
enum { //Packet Direction
  NTWK_DATA = 0, //UPSTREAM
  NTWK_CMD   //DOWNSTREAM


};
```

The NTW_DATA type (data frame) is originated from the smartplugs and the NTW_CMD (command frame) from the data concentrator; the network frame also includes a “behalf” and a “for” address used to build the routing table of the network repeaters. Each entry of the routing table has the following structure:

```
typedef struct{
  u8 Status;
  u16 NodeAddr;
  u16 RouteAddr; }T_ROUTE_TBL_ENTRY;
```

The entry puts in relation a node address with the router address that is able to reach it; when a data frame is received by a repeater, it updates the table with a new entry including the frame source address field as the router address and the “behalf” address field as the node address. When a command frame is received, the repeaters check the routing table and read the next hop destination address to reach the final destination. *Figure 5* shows the frame structure:

Figure 5. Frame structure

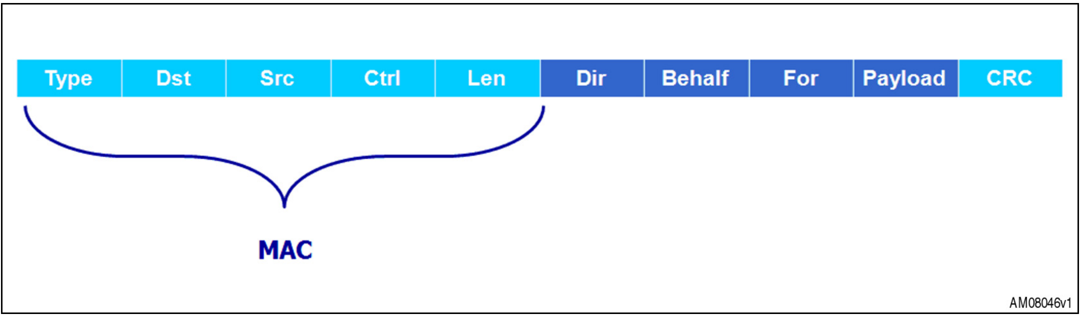
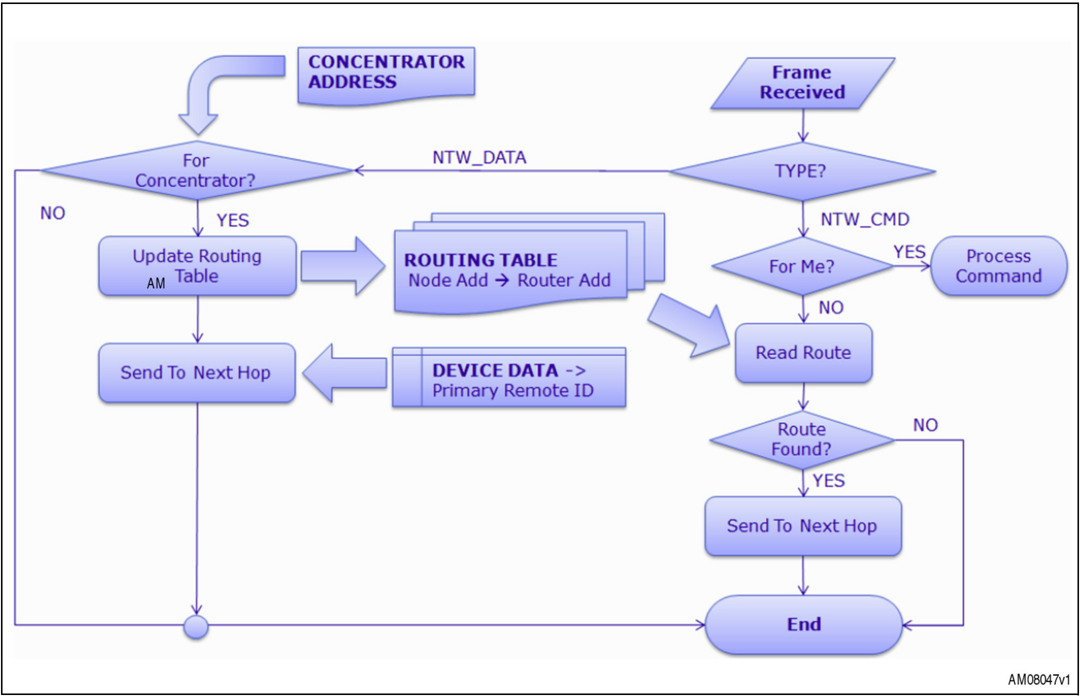


Figure 6 shows the routing algorithm.

Figure 6. Back routing algorithm



This module also implements the network functions used by the data concentrator which maintains a list of the addresses of all network nodes too. This list is implemented by the following data structure:

```
typedef struct{
  u16 NodesList[NODES_TBL_SIZE];
  u16 NodesCount;

} NODES_TABLE_T;
```

5.4.1 Exported function

This module exports the following functions:

- NTWK_Data
- NTWK_CheckFrame
- NTWK_BuildFrame
- NTWK_ReadAdd
- NTWK_UpadateRouteTbl
- NTWK_ResolveAdd
- NTWK_ReadRoute
- NTW_UpdateNodesTable
- NTWK_FindRoute
- NTWK_GetNodesTable
- NTWK_GetRoutingTable

5.4.2 NTWK_Data

Table 12 describes the NTWK_Data function:

Table 12. NTWK_Data function

Functions	Description
Function name	NTWK_Data
Function prototype	u8* NTWK_Data(u8* p_dl_frame, u8* len)
Behavior description	Returns the pointer to the data payload of the input frame and provides the payload length.
Input parameter {x}	Received frame
Output parameter {x}	Payload length
Return value	Pointer to the data payload
Required preconditions	None
Called functions	None

Example:

```
u8* pPayload;

u8 Network_data[STB_FRAME_MAX_LEN];

u8 Len;

…

pPayload = NTWK_Data( Network_data, &Len );

…
```

5.4.3 NTWK_CheckFrame function

Table 13 describes the NTWK_CheckFrame function:

Table 13. NTWK_CheckFrame function

Functions	Description
Function name	NTWK_CheckFrame
Function prototype	u8 NTWK_CheckFrame(u8* NtwkData)
Behavior description	Verifies the destination address of the input frame. Returns TRUE if the destination address is the local address, zero, the concentrator address, or the broadcast address
Input parameter {x}	Received frame
Output parameter {x}	None
Return value	True/false
Required preconditions	None
Called functions	None

Example:

```
u8 Network_data[STB_FRAME_MAX_LEN];

…

if ( NTWK_CheckFrame( Network_data ) )

{

…

}

…
```

5.4.4 NTWK_BuildFrame function

Table 14 describes the NTWK_BuildFrame function:

Table 14. NTWK_BuildFrame function

Functions	Description
Function name	NTWK_BuildFrame
Function prototype	Void NTWK_BuildFrame(u8* pdata, u8* len, u16 add, u8 type)
Behavior description	Updates the network frame and the length with source and destination addresses and the with the network frame type (NTWK_DATA/NTWK_CMD)
Input parameter {x}	Frame to transmit, reference to frame length, destination address, frame type
Output parameter {x}	Updated frame, updated length
Return value	None
Required preconditions	None
Called functions	None

Example:

```
u8 Network_data[STB_FRAME_MAX_LEN];

u8 Len;

u16 Add

…

NTWK_BuildFrame(Network_data, &Len, Add, NTWK_DATA);

…
```

5.4.5 NTWK_ReadAdd function

Table 15 describes the NTWK_ReadAdd function:

Table 15. NTWK_ReadAdd function

Functions	Description
Function name	NTWK_ReadAdd
Function prototype	u16 NTWK_ReadAdd(u8* pdata)
Behavior description	Extracts the “behalf” network address from the input frame
Input parameter {x}	Received frame
Output parameter {x}	None
Return value	“Behalf” address
Required preconditions	None
Called functions	None

Example:

```
u8 Network_data[STB_FRAME_MAX_LEN];

u16 Add

…

Add = NTWK_ReadAdd(Network_data);

…
```

5.4.6 NTWK_UpadateRouteTbl function

Table 16 describes the NTWK_UpadateRouteTbl function:

Table 16. NTWK_UpadateRouteTbl function

Functions	Description
Function name	NTWK_UpadateRouteTbl
Function prototype	Void NTWK_UpadateRouteTbl(u16 SrcAddr, u8* NtwkData)
Behavior description	Updates the routing table with the source address and the “behalf” address extracted from the input frame
Input parameter {x}	Source address, received frame
Output parameter {x}	None
Return value	None
Required preconditions	None
Called functions	None

Example:

```
u8 Network_data[STB_FRAME_MAX_LEN];

u16 Add

…

NTWK_UpadateRouteTbl(Add, Network_data);

…
```

5.4.7 NTWK_ResolveAdd function

Table 17 describes the NTWK_ResolveAdd function:

Table 17. NTWK_ResolveAdd function

Functions	Description
Function name	NTWK_ResolveAdd
Function prototype	u16 NTWK_ResolveAdd(u16 id);
Behavior description	Reads the network address of a node identified by the application ID from the node tables
Input parameter {x}	Node ID

Table 17. NTWK_ResolveAdd function (continued)

Functions	Description
Output parameter {x}	None
Return value	Network address
Required preconditions	It is used only by the master node
Called functions	None

Example:

```
u8 id;
u16 Add
…
Add = NTWK_ResolveAdd(id);
…
```

5.4.8 NTWK_ReadRoute function

Table 18 describes the NTWK_ReadRoute function:

Table 18. NTWK_ReadRoute function description

Functions	Description
Function name	NTWK_ReadRoute
Function prototype	u8 NTWK_ReadRoute(u8* NtwkData, u16* add)
Behavior description	Finds the router address to reach the node specified in the “for” address of the input frame
Input parameter {x}	Received frame
Output parameter {x}	Router address
Return value	HOP_FOUND/ HOP_NOT_FOUND
Required preconditions	None
Called functions	NTWK_FindRoute

Example:

```
u8 Network_data[STB_FRAME_MAX_LEN];
u16 Add
…
if ( NTWK_ReadRoute(Network_data, &Add) == HOP_FOUND )
{
…
}
…
```

5.4.9 NTW_UpdateNodesTable function

Table 19 describes the NTW_UpdateNodesTable function:

Table 19. NTW_UpdateNodesTable function

Functions	Description
Function name	NTW_UpdateNodesTable
Function prototype	Void NTW_UpdateNodesTable(u16 add)
Behavior description	Updates the node tables
Input parameter {x}	Node’s network address
Output parameter {x}	None
Return value	None
Required preconditions	None
Called functions	None

Example:

```
u16 Add
…
NTW_UpdateNodesTable(Add);
…
```

5.4.10 NTWK_FindRoute function

Table 20 describes the NTWK_FindRoute function:

Table 20. NTWK_FindRoute function

Functions	Description
Function name	NTWK_FindRoute
Function prototype	u8 NTWK_FindRoute(u16 dest, u16* add)
Behavior description	Finds the router address to reach the node specified in the destination address
Input parameter {x}	Destination address
Output parameter {x}	Router address
Return value	HOP_FOUND/ HOP_NOT_FOUND
Required preconditions	None
Called functions	None

Example:

```
u16 APP_TargetDevice;
u16 Add
```

```
…
if (NTWK_FindRoute(Add, &APP_TargetDevice) )
{
…
}
```

5.4.11 NTWK_GetNodesTable function

Table 21 describes the NTWK_GetNodesTable function:

Table 21. NTWK_GetNodesTable function

Functions	Description
Function name	NTWK_GetNodesTable
Function prototype	NODES_TABLE_T* NTWK_GetNodesTable(void)
Behavior description	Returns the pointer to the nodes table
Input parameter {x}	None
Output parameter {x}	None
Return value	Pointer to nodes table
Required preconditions	None
Called functions	None

Example:

```
NODES_TABLE_T* pNodes;
…
pNodes = NTWK_GetNodesTable();
…
```

5.4.12 NTWK_GetRoutingTable function

Table 22 describes the NTWK_GetRoutingTable function:

Table 22. NTWK_GetRoutingTable function

Functions	Description
Function name	NTWK_GetRoutingTable
Function prototype	T_ROUTE_TBL_ENTRY* NTWK_GetRoutingTable(void)
Behavior description	Returns the pointer to the routing table.
Input parameter {x}	None
Output parameter {x}	None
Return value	Pointer to routing table

Table 22. **NTWK_GetRoutingTable function (continued)**

Functions	Description
Required preconditions	None
Called functions	None

Example:

```
T_ROUTE_TBL_ENTRY* pRoute;

…

pRoute = NTWK_GetRoutingTable();

…
```

5.5 APDU layer

On top of the communication protocol network layer, a set of commands has been implemented to monitor and control the smartplug board by a remote data concentrator. A SET/GET model has been implemented: four types of application messages have been defined as shown in *Table 23*.

Table 23. **Commands description**

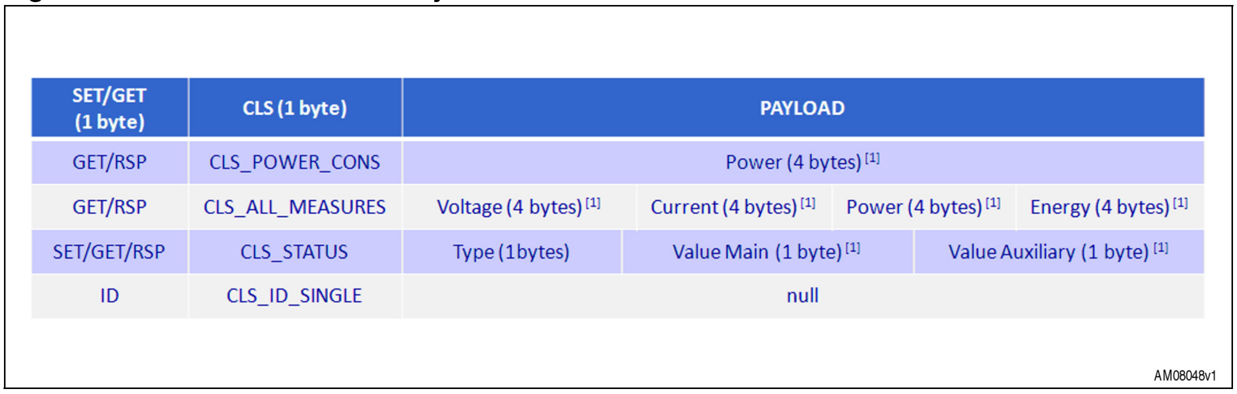
Command name	Command code	Description
GET	0x00	Used to read data from the smartplug
SET	0x01	Used to write data to the smartplug
RSP	0x02	Data response to a GET request
ID	0x03	Used to identify a smartplug inside a network

The database variables are identified by the classes described in *Table 24*.

Table 24. **Commands classes description**

Class name	Class code	Description
POWER_CONS	0x00	Used to read data from the smartplug
ALL_MEASURE	0x01	Used to write data to the smartplug
STATUS	0x02	Data response to a GET request
ID	0x03	Used to identify a smartplug inside a network

Figure 7. APDU format summary



SET/GET (1 byte)	CLS (1 byte)	PAYLOAD			
GET/RSP	CLS_POWER_CONS	Power (4 bytes) [1]			
GET/RSP	CLS_ALL_MEASURES	Voltage (4 bytes) [1]	Current (4 bytes) [1]	Power (4 bytes) [1]	Energy (4 bytes) [1]
SET/GET/RSP	CLS_STATUS	Type (1bytes)	Value Main (1 byte) [1]	Value Auxiliary (1 byte) [1]	
ID	CLS_ID_SINGLE	null			

AM08048v1

Note: The “Type” and “Value Auxiliary” fields are not used.

6 References

- STM32F10xxx; STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx datasheets
- RM0008 reference manual
- STM32F10xFWLib 3.1.0, help file
- UM1458 user manual
- AN3287 application note
- STPM01 datasheet
- ST7540 datasheet
- http://www.freertos.org/

7 Revision history

Table 25. Document revision history

Date	Revision	Changes
11-Nov-2011	1	Initial release.