eZeeNet™ Software 1.6

eZeeNet API

Reference Manual

Doc. P-EZN-452~02 v.1.6

March 2007

Disclaimer

MeshNetics believes that all information is correct and accurate at the time of issue. MeshNetics reserves the right to make changes to this product without prior notice. Please visit MeshNetics website for the latest available version.

MeshNetics does not assume any responsibility for the use of the described product or convey any license under its patent rights.

MeshNetics warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with MeshNetics standard warranty. Testing and other quality control techniques are used to the extent MeshNetics deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

Trademarks

MeshNetics®, ZigBit, eZeeNet, ZigBeeNet, SensiLink, as well as MeshNetics and ZigBit logos are trademarks of MeshNetics Ltd.

All other product names, trade names, trademarks, logos or service names are the property of their respective owners.

Technical Support

Please e-mail your product-related questions to support@meshnetics.com. If you like to speak to one of our product specialists you can call the phone number listed below.

Development Support

Software customization services can be provided on terms and conditions mutually agreed by MeshNetics and end-user.

Contact Information

MeshNetics

9 Dmitrovskoye Shosse, Moscow 127434, Russia

Tel: +7 (495) 725 8125

Office hours: 8:00am – 5:00pm (Central European Time)

Fax: +7 (495) 725 8116

E-mail: support@meshnetics.com

www.meshnetics.com

Table of Contents

List of Figures

List of Tables

1. Introduction

Purpose of this document

This document specifies the Application Programming Interfaces (API) to the eZeeNet™ Software [1]. The document describes:

- eZeeNet Framework interfaces, chapter 3
- eZeeNet stack interfaces, chapter 4
- HAL interfaces, chapter 5
- MeshBean2 board drivers, chapter 6
- Application development instructions and samples, chapter 7.

Definitions and abbreviations

A/D	analog to digital (conversion)
ADC	analog to digital converter
API	Application Programming Interface
ARQ	Automatic Repeat Request
CS	chip select (signal)
CRC	Cyclic Redundancy Check
DIP	dual in-line package
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	general input-output
HAL	Hardware Abstraction Layer
I^2C	Inter-Integrated Circuit
IO	input/output
IRQ	interrupt request
ISR	interrupt service routine
LED	light emitting diode
LQI	Link Quality Indication
LSB	Least Significant Bit
MAC	Media Access Control layer
MCU	Microcontroller unit
MSB	Most Significant Bit
NWK	Network layer

PAN	Personal Area Network
PC	Personal computer
RSSI	Received Signal Strength Indication
RX	receive
SPI	Serial Peripheral Interface
TX	transmit
UART	Universal Asynchronous Receiver / Transmitter
USART	Universal Synchronous / Asynchronous Receiver / Transmitter
ZDK	ZigBee Development Kit
ZEK	ZigBee Evaluation Kit

References

[1] eZeeNet™ IEEE802.15.4/ZigBee Software. Product Datasheet. MeshNetics Doc. M-251~02

[2] eZeeNet™ Software 1.4. SerialNet™ Reference Manual. AT-Command Set. MeshNetics Doc. P-EZN-452~01

[3] Using the GNU Compiler Collection/ By Richard M. Stallman and the GCC Developer Community.

[4] avr-libc Reference Manual 1.4.3

[5] ZigBit™ OEM Module. Product Datasheet. MeshNetics Doc. M-251~01

[6] Atmel 8-bit AVR Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash. 2549F-AVR-04/06

[7] ZigBit™ Development Kit 1.0. User's Guide. MeshNetics Doc. S-ZDK-451

[8] WinAVR User Manual – 20060125/ By Eric B. Weddington.

[9] AVR Studio. User Guide. http://www.atmel.com/dyn/resources/prod_documents/doc2510.pdf.

[10] JTAGICE mkII User Guide. http://www.atmel.com/dyn/resources/prod_documents/doc2562.pdf

2. System Architecture Overview

2.1. General Software Specification

Software resource usage is summarized in Table 1. These numbers are valid for the following network configuration:

- Maximum child number – 5
- Maximum network depth – 5
- Maximum PAN descriptor numbers (for network discovery) – 5.

Table 1. Firmware storage consumption

Section name	Section size
Code	96K
Data	6K
Stack	1K

NOTE:

The size of code section in the table is accounted for using all functions in the libraries. Real application may be using not all of them. For instance, the Peer-to-Peer application (see Section 7.3.2) is taking 98K in code segment.

2.2. General Limitations on User's Code

The cooperative multitasking model used in the eZeeNet software imposes some limitations on the user's code running under eZeeNet.

- Do not call eZeeNet functions directly from ISRs. Instead, buffer the data and post the TinyOS task by `TOSH_post()` function. Or, organize your semaphores using volatile variables.
- Avoid data processing in the ISR and make ISRs as faster as possible.
- Do not disable interrupts for a long time (more than several tens of microseconds) to provide normal operation of the ZigBee stack and hardware interfaces like UART.
- Avoid calling C-library functions or floating-point manipulations in the ISRs.
- 96 K code section is only available for user.
- 6 K data section is only available for user. It's strongly recommended to follow this size restriction as violation would make stack corrupted.
- Keep the stack size as low as possible.
- Avoid allocation of the memory by `malloc()/calloc()/free()` functions in the working phase of the software due to the unpredictable processing time and possible garbage collection problems if the order of the allocation and freeing is not proper. Normally, allocation should be done at the initialization phase.

- Avoid posting multiple tasks. Instead, organize your code in form of infinite loop and check your status variables in the beginning of the loop.
- Check the code/data size before running the application. To determine the size of an application image file correctly use the `avr-size` utility which comes with AVR Tools [8]. If a part of the code is located higher than `0xFC00` address, then it will not be downloadable with Serial Bootloader. In this case you should load the code via JTAG only in disabling UART booting by proper fuse bits.

2.3. Functional Diagram and Description of eZeeNet Software

eZeeNet is a robust IEEE802.15.4/ZigBee software [1] that forms a wireless network based on the ZigBit module (see Section 2.4). The eZeeNet conforms to ZigBee 1.0 specification. It provides easy to use multi-point networking, with a routing mechanism that optimizes the network traffic and reduces power consumption. The eZeeNet software offers a user-friendly API for network and smart power management, including data exchange, network formation/node join, PAN ID management, channel selection, TX power control, etc.

The eZeeNet layered software is composed by four principal modules: eZeeNet Stack, eZeeNet Framework, SerialNet Module, and Hardware Abstraction Layer (HAL) – see Figure 1.

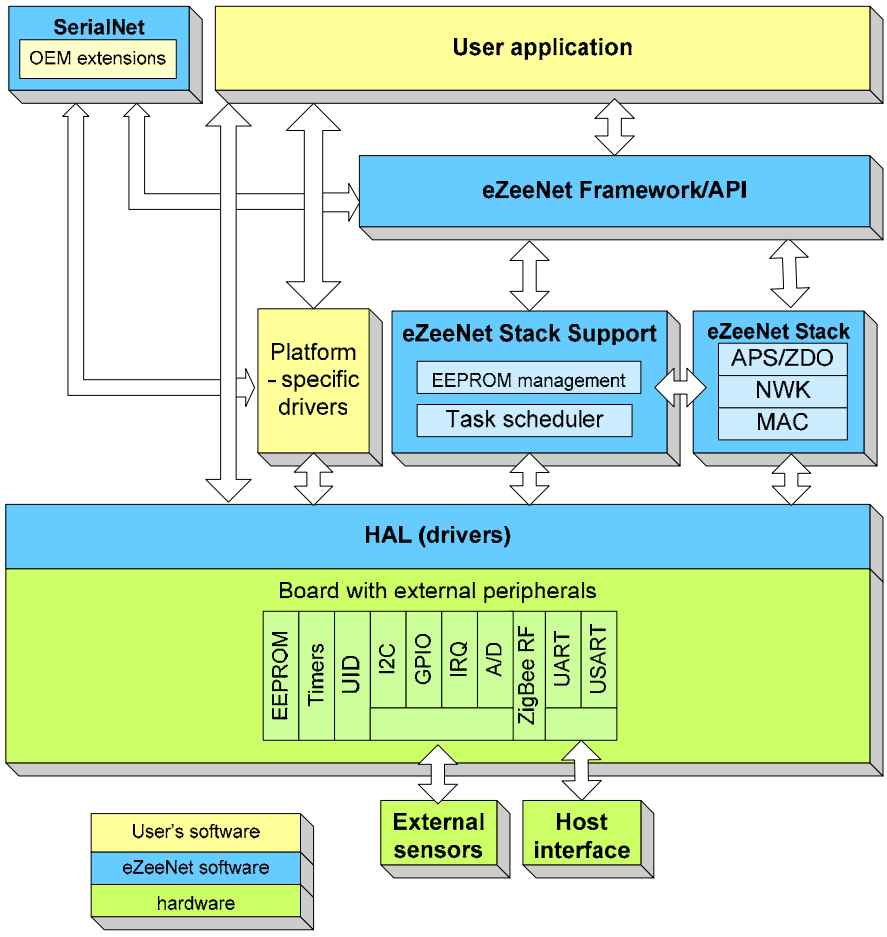

Figure 1. Simplified diagram of the eZeeNet software

The eZeeNet Stack can be easily reconfigured to run on end-device, router or coordinator. Easy-to-use interfaces simplify basic network operations like data exchange, network formation/node join, PAN ID management, channel selection, TX power control and more.

The eZeeNet Framework provides user application with an access to system resources/services (timers, memory, etc.), it implements multitasking for cooperative execution of user code along with networking. Besides, it supports power management mechanism to enable low duty-cycle and low power consumption.

HAL is an interface between ZigBit module and the MCU peripherals giving clear and simple API for user application to access a peripheral easily without conflicting with the eZeeNet Stack.

The SerialNet is an optional module that offers control over the most of ZigBit functionality via UART port or any other communication interface using standardized AT-command set.

2.4. ZigBit Hardware

ZigBit is a low-power, high-sensitivity IEEE802.15.4/ ZigBee OEM module based on Atmel's Z-Link 2.4GHz platform [5]. ZigBit module contains ATmega1281V Microcontroller, AT86RF230 RF Transceiver. See the ZigBit block diagram in Figure 2.

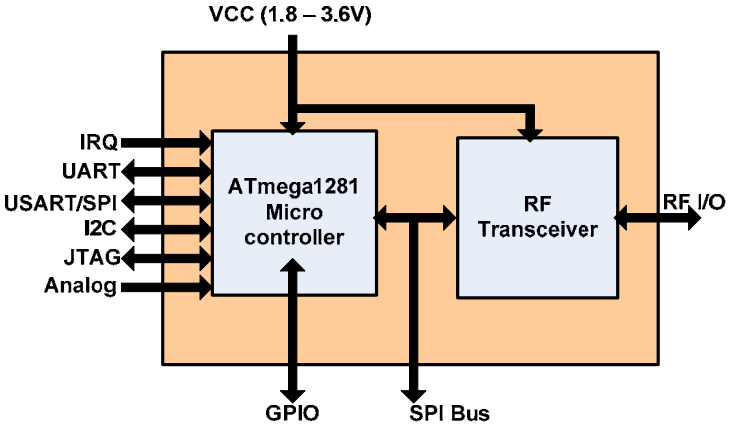

Figure 2. ZigBit block diagram

Details of ZigBit pinout can be found in ZigBit datasheet [5]. Here is the brief summary table that shows, which pins are controlled by which functions of eZeeNet.

Table 2. ZigBit pin assignment

Pin Name	Description	Reference
SPI_CLK, SPI_MOSI, SPI_MISO	Reserved for stack operation	-
GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5, GPIO6, GPIO7, GPIO8, GPIO9	General purpose digital input/output	5.3.1

Pin Name	Description	Reference
OSC32K_OUT	32.768 kHz clock output.	-
RESET	Reset input (active low).	-
D_VCC	Digital supply voltage (Vcc)	-
DGND, AGND	Digital/analog ground	-
CPU_CLK	Clock output.	-
I2C_CLK, I2C_DATA	I^2C bus	5.3.5, 5.3.1
UART_TXD, UART_RXD, UART_RTS, UART_CTS	UART interface	5.3.6, 5.3.1
JTAG_TMS, JTAG_TDI, JTAG_TDO, JTAG_TCK	JTAG	-
BAT, ADC_INPUT_1, ADC_INPUT_2, ADC_INPUT_3	ADC	5.3.3, 5.3.1
A_VREF	Output/Input reference voltage for ADC	5.3.3
UART_DTR	DTR input (Data Terminal Ready) for UART.	5.3.6, 5.3.1
USART0_RXD, USART0_TXD, USART0_EXTCLK	UART/SPI	5.3.6, 5.3.7
IRQ_7, IRQ_6	Digital input interrupt requests / GPIO lines	5.3.2
RF_GND, RFP_IO, RFN_IO	Radio interface	-

2.5. Register and Call Conventions

Functional interface of eZeeNet is C-callable. Mixing with C and user's C++ code is not guarantied.

User's application should follow the register conventions for C-callable functions specified in [3]. Applications should avoid to modify the peripheral registers directly.

eZeeNet HAL drivers are also C-callable, but some functions can be called from Interrupt Service Routines (ISRs).

2.5.1. Callback Interfaces

Some user's functions are defined as callback handlers. They are used to indicate the completion of some process (i.e. data transmission) or they serve as event handlers.

As a rule, eZeeNet does not call user's functions directly from ISRs, so callback functions do not need to save/restore the context. Thus they may use C-library or other eZeeNet functions. There are some exceptions due to the performance reasons, see descriptions of the particular functions below.

2.5.2. State after Reset

The software is capable to check the reason of the reset. As specified in the ATmega1281 datasheet [6], all pins are tri-stated after any kind of reset. But, when the SerialNet runs, it configures some pins during the startup phase accordingly to the configuration values stored in the EEPROM.

Table 3. HAL resources state at startup

Pins	eZeeNet	SerialNet
GPIO lines	tri-state	depends on `S120...S128` registers, see [2]
A/D lines	conversion is disabled	depends on `S100..S108` registers, see [2]
I^2C modes	disabled	disabled
`I2C_CLK`, `I2C_DATA`	tri-state	tri-state
USART	disabled	disabled
`USART0_RXD`, `USART0_TXD`, `USART0_EXTCLK`	tri-state	tri-state
UART	disabled	enabled, the mode and the rate depend on the `+IFC` and `+IPR` commands, see [2]
`UART_TXD`	tri-state	input, no internal pullup
`UART_RXD`	tri-state	output
`UART_CTS`, `UART_RTS`,	tri-state	depends on `+IFC` settings, see [2]
`UART_DTR`	tri-state	tri-state
`IRQ6, IRQ7`	tri-state	tri-state

NOTES:

Serial Bootloader, if enable, delays reset phase by approximately 500 msec. During that time, it waits for data and it can activate pin `UART_RXD` if it receives the valid handshake request from UART. If such handshake signal is not received, bootloader starts user's application. If such behavior is not acceptable then disable the bootloader by corresponding fuse bit.

Use the external pullup resistors on the `UART_CTS` and `UART_RXD` pins to avoid missing UART data during reset. If `UART_DTR` line is used for module wakeup (`%D1` command of SerialNet, see [2]), it is recommended to pull it up.

2.5.3. User-defined ISRs

User application can set own ISR serving some interrupt sources, for example by utilizing the alternative function of some ZigBit pin. For instance, `GPIO6` pin can be used as a timer interrupt source.

ISR management should be done by IRQ functions, see Section 5.3.2.

2.5.4. Use of Power-down Modes

Power-down procedure should be under control of the eZeeNet due to the nature of the ZigBee protocol. It is not recommended to use processor's power-down modes directly by manipulating the processor registers. Instead, use functions `fw_registerSleep()`, `fw_forceToSleep()`, `fw_appReadyToSleep()`, `fw_forceWakeup()` (see Section 3.3.6, Section 3.3.7, Section 3.3.8, Section 3.3.9) to force sleeping or setting active/sleep duty cycle.

NOTE:

Power-down modes should not be used on routers or coordinator.

2.6. eZeeNet Hardware Resource Allocation

eZeeNet allocates some hardware resources for their internal use. They should not be used by user's application. Also, eZeeNet uses specific settings of the MCU which cannot be changed by direct manipulations with the registers or fuse bits. For details, refer to ATmega1281V manual [6] and ZigBit data sheet [5].

The allocated hardware resources are listed in Table 4. They do not include hardware resources like interrupts allocated by particular driver from the HAL.

Table 4. Hardware resources used by eZeeNet

Parameter/Register	Description
Processor main clock	4 MHz
ATmega ports: PA7, PB0, PB1, PB2, PB3, PB4, PE5	Radio interface
ATmega port: PG3, PG4	32.768 kHz clock
Hardware timers (including their interrupts)	Timer/Counter2, Timer/Counter4, Timer/Counter5.
Hardware interrupts	External IRQ5

2.7. General Types

eZeeNet uses a set of standard integer types defined in the `<stdint.h>` file:

Table 5. Standard types

Type	Description
`int8_t`	8-bit signed integer
`uint8_t`	8-bit unsigned integer
`int16_t`	16-bit signed integer
`uint16_t`	16-bit unsigned integer
`int32_t`	32-bit signed integer
`uint32_t`	32-bit unsigned integer
`int64_t`	64-bit signed integer
`uint64_t`	64-bit unsigned integer
`Bool`	See Table 6

Table 6. bool enumeration

Value	Description
`FALSE`	False condition
`TRUE`	True condition

Table 7. result_t enumeration

Value	Description
`SUCCESS`	Operation completed successfully
`FAIL`	Operation failed
`BUSY`	Operation cannot be executed because resource(s) is busy

2.8. Memory Management

It's strongly recommended to avoid using dynamic memory allocation and allocate the data statically.

eZeeNet software itself does not use `malloc/calloc/free` functions from C-library.

2.9. Use of Standard C-libraries

User's application can use standard C-library. Take into account that most functions of this library are not guaranteed to be reentrant. In particular, some functions store local state, they are known to be non-reentrant.Those functions are also non-reentrant that manipulate IO registers, like the EEPROM access routines. Using these functions within interrupt context will result unpredictable behavior.

2.10.TinyOS Functions

TinyOS is a component-based runtime environment designed to provide support for deeply embedded systems which require concurrency intensive operations while constrained by minimal hardware resources. The programming language of TinyOS is stylized C that uses a custom compiler 'NesC', but TinyOS functions are C-callable. A complete documentation on TinyOS is available at http://www.tinyos.net/ .

eZeeNet Software uses a small subset of TinyOS functions. The functions that can be called by user application are shown in the Table 8. They include: TinyOS task management, critical section implementation, and global interrupt management. User's applications should not call any other functions of TinyOS.

Table 8. The callable TinyOS functions

Function	Description
`TOS_post`	Post TinyOS task. Returns `TRUE` if successful. Normally, this function should be used to post signal from interrupt to the software part executing in the non-interrupt context mode.
`ATOMIC_SECTION_ENTER`	Open critical section. This macro temporarily disables interrupts and saves the interrupt context. Critical sections should be used to make correct reading/writing from/to the variables accessible both in interrupt and non-interrupt contexts. And, critical sections should be used in the hardware drivers to enclose the i/o operations.

`ATOMIC_SECTION_LEAVE`	Close critical section.
	
	This macro closes critical section previously opened by `ATOMIC_SECTION_ENTER`.
	
	Example:
	
	`{`
	`  ATOMIC_SECTION_ENTER;`
	`  {`
	`  .... // do something critical`
	`  }`
	`  ATOMIC_SECTION_LEAVE;`
	`}`

Function	Description

TOSH_run_next_task	Normally, there is no need to use this function because eZeeNet main loop already does it. But, this function can be called in user's application to provide waiting for the event from eZeeNet, when some pending operation is executed.
	Example:
	`bool flag; // Flag to indicate when the operation is completed.`
	`...`
	`...`
	`flag = FALSE; // Clear flag.`
	`operation(); // Operation start.`
	`while (!flag) TOSH_run_next_task(); // Wait operation completion.`
	`...`
	`...`
	
	`// Function completing the operation.`
	`complete()`
	`{`
	`...`
	`flag = TRUE;`
	`...`
	`}`

TOSH_interrupt_enable	Enable global interrupts.
TOSH_interrupt_disable	Disable global interrupts.

NOTES:

Instead of global interrupt disabling/enabling it is strongly recommended to use special macros `ATOMIC_SECTION_ENTER, ATOMIC_SECTION_LEAVE.`

Critical sections should be as short as possible. Critical sections should be used mostly to implement semaphores or to read/write to the variables accessible both from interrupt and non-interrupt context. Avoid to call any functions inside.

3. Framework Interfaces

3.1. Function Summary

See Functions controlling Framework functionality in Table 9. These functions provide:

- entry for user HAL and application initialization
- periodical call in organizing a user's loop
- system time management
- eZeeNet parameters management
- EEPROM management
- periodical calibration of the internal RC oscillator.

The interfaces are declared in the `"framework.h"` file.

Table 9. Framework functions

Function	Description	Ref.
`fw_setUserLoop`	Set user's loop	3.3.3
`fw_getSystemTime`	Get system time	3.3.4
`fw_userEntry`	Initialize user's application	3.3.2
`fw_warmReset`	Warm reset	3.3.5
`fw_setParam`	Set eZeeNet parameter	3.3.12
`fw_getParam`	Get eZeeNet parameter	3.3.13
`fw_eepromWrite`	Direct write to EEPROM	3.3.10
`fw_eepromRead`	EEPROM direct reading	3.3.11
`fw_registerSleep`	Register user's handlers for power management	3.3.6
`fw_forceToSleep`	Force the node to sleep	3.3.7
`fw_appReadyToSleep`	Ready-to-sleep indication	3.3.8
`fw_forceWakeup`	Force the node to wake up	3.3.9

3.2. Call Sequences

Program initialization sequence is shown in Figure 3.

Going to sleep sequences are shown in Figure 4 and Figure 5.

Waking-up sequence initiated by user's interrupt handler during sleep is shown in Figure 6.

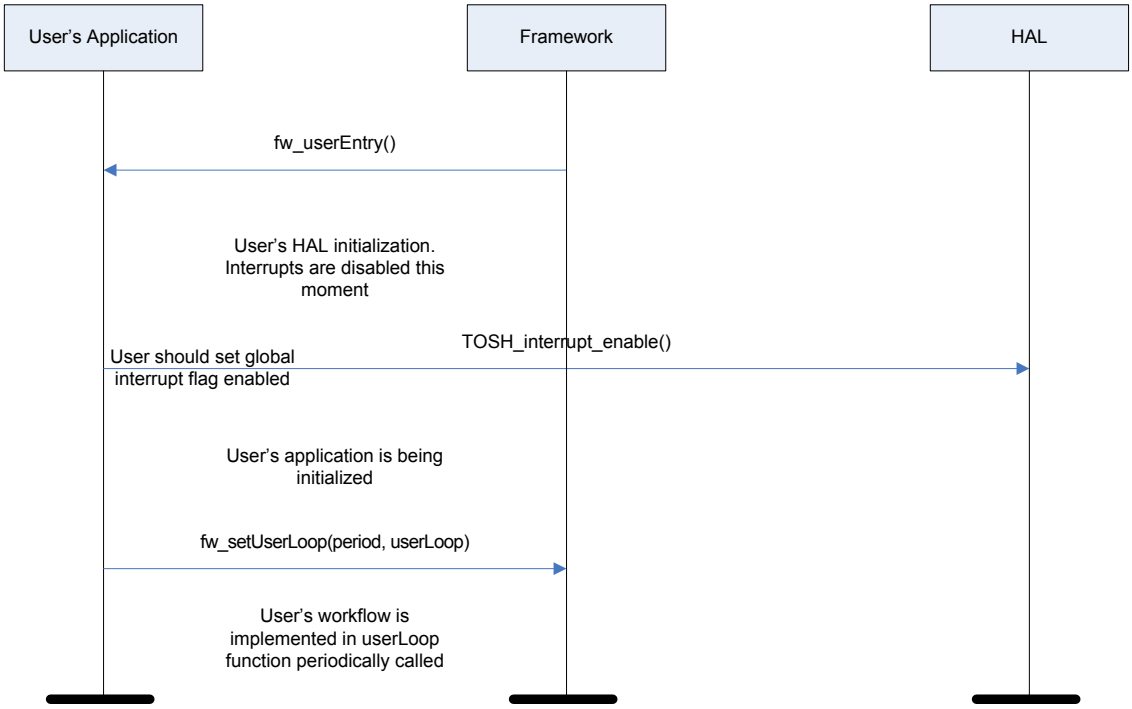

Figure 3. Application initialization sequence

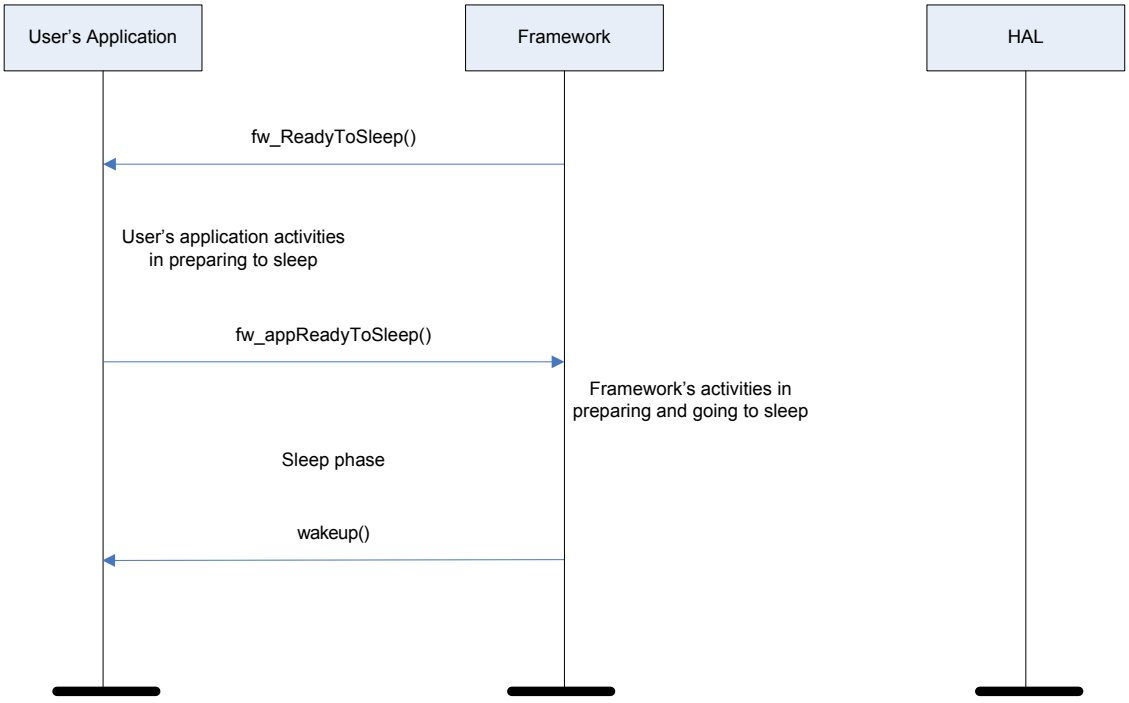

Figure 4. Going to sleep sequence (when initiated by Framework)

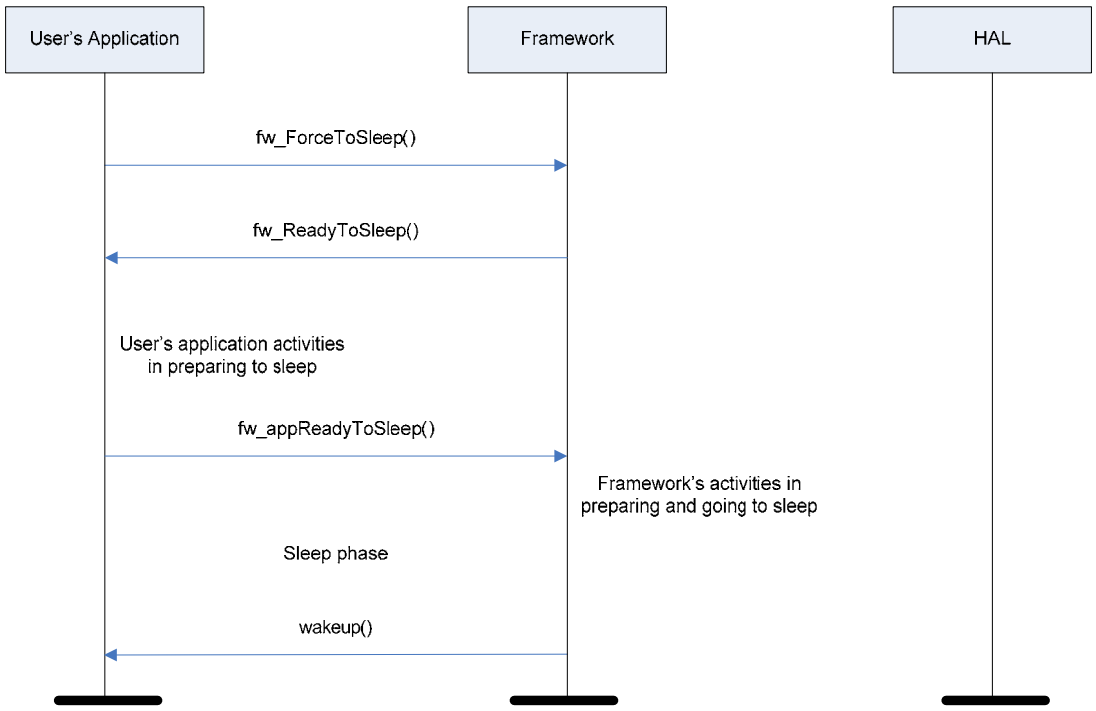

Figure 5. Going to sleep sequence (when initiated by user's application)

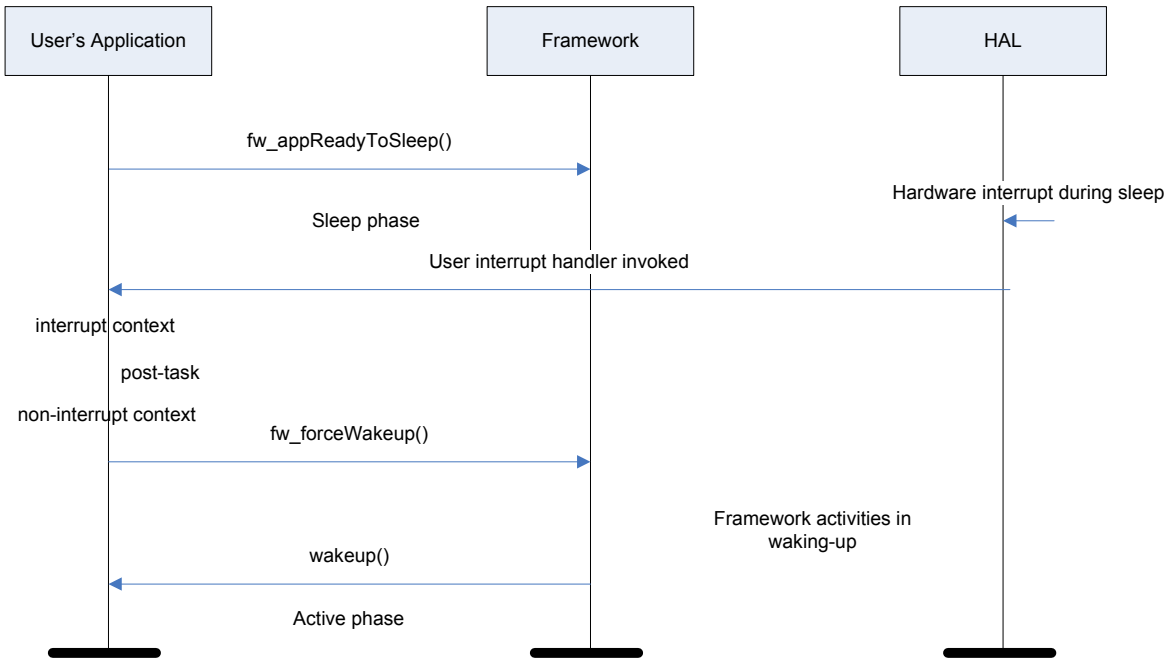

Figure 6. Waking-up sequence, when initiated by user's interrupt handler during sleep

3.3. Detailed Description

3.3.1. Data Structures

There are the following data types frequently used:

Table 10. Framework API data types

Data Type	Description
typedef enum { ZIGBEE_COORDINATOR_TYPE = 0, ZIGBEE_ROUTER_TYPE = 1, ZIGBEE_END_DEVICE_TYPE = 2 } NodeLogicalType_t;	Node role
typedef uint64_t IEEEAddr_t	MAC address
typedef uint16_t PANID_t	PAN ID
typedef struct { bool coordinator; bool router; bool endDevice; } NodeCapability_t	Possible node's roles
typedef uint16_t NodeLogicalAddr_t	Node logical address
typedef uint16_t NWKAddr_t	Node NWK address
typedef struct { NWKAddr_t networkAddr; IEEEAddr_t extAddr; } NodeAddrInf_t	Node address information

Table 11. FW_ResetReason_t type

Value	Description
FW_WARM_RESET	Warm reset has been performed
FW_COLD_RESET	Cold reset occurred
FW_WD_RESET	Watchdog timer caused the reset

3.3.2. fw_userEntry(): Initialize User's Application

Purpose	Initialize user's HAL and application. This function should be implemented in user's code. Main purpose is user's HAL and application initialization.
Function prototype	`void fw_userEntry(FW_ResetReason_t reason)`
Arguments	`reason` – hardware reset reason (warm/cold or watch-dog reset), see Table 11
Return value	none
Calling context	This function is called by eZeeNet Framework during initialization phase, see Figure 3. Interrupts are disabled.
Restrictions	none

3.3.3. fw_setUserLoop(): Set User's Loop

Purpose	Set parameters of user's handler:
Function prototype	`void fw_setUserLoop(uint32_t period, void (*userLoop)(void))`
Arguments	`period` – call interval in milliseconds `userLoop` – user's handler periodically called This function is usually called at the end of `fw_userEntry()` function (see Section 3.3.2). If `period` is set to 0, then `userLoop` will be called by Framework as soon as possible.
Return value	none
Calling context	Should be called during initialization phase only, see Figure 3.
Restrictions	none

3.3.4. fw_getSystemTime(): Get System Time

Purpose	Provide system time in milliseconds.
Function prototype	`uint32_t fw_getSystemTime(void)`
Arguments	none
Return value	system time in milliseconds (passed from the last cold or warm reset)

Calling context	Can be used both in the interrupt and non-interrupt contexts.
Restrictions	none

3.3.5. fw_warmReset(): Warm Reset

Purpose	Perform warm reset. Warm reset procedure reinitializes hardware and software, excluding cleaning the EEPROM data. The node will fall to the same state as it was reset by the hardware `RESET` signal.
Function prototype	`void fw_warmReset(bool factoryDefaults)`
Arguments	`factoryDefaults` – if `TRUE`, set all eZeeNet parameters, stored in the EEPROM, to their default values
Return value	This function never returns any value because it causes hardware reset.
Calling context	non-interrupt context
Restrictions	none

NOTES:

If serial bootloader is enabled by the appropriate fuse bits, it will be called first during the warm reset procedure.

If UART RTS/CTS flow control is enabled by the appropriate parameter, see Table 16, `UART_CTS` line is set as soon as this function starts. But during the MCU reset, `UART_CTS` line is in the tri-state. That is why this line is recommended to be pulled up. `UART_CTS` line will be cleared when eZeeNet will complete initialization procedure that indicates that the module is ready to accept the data.

At startup, eZeeNet configures all the resources listed in Table 4.

3.3.6. fw_registerSleep(): Register User's Handlers for Power Management

Purpose	Register callbacks for: • notification that the Framework is ready to sleep • notification that the node is waken up Node going to sleep sequence is shown in Figure 4 and Figure 5.
Function prototype	`void fw_registerSleep(void (*fwReadyToSleep)(void), void (*wakeup)(void))`
Arguments	`fwReadyToSleep` – pointer to the user's function that is called by eZeeNet when it decides to sleep. This is the right place to prepare all users peripheral resources for sleeping phase (to move them to tri-state, etc.). `wakeup` – pointer to the user's function that is called when eZeeNet is waken up.

Return value	none
Calling context	interrupt and non-interrupt context
Restrictions	Should be used on the end-devices only.

NOTES:

After `fwReadyToSleep` is called back user is yet allowed to transmit data. In this case, Framework will call `fwReadyToSleep` once again.

`fwReadyToSleep` will not be called back when `fw_dataIndicationControl()` function has been called with the enable parameter set to `FALSE`, in other words, when the data indication is disabled within Framework.

3.3.7. fw_forceToSleep(): Force to Sleep

Purpose	Force the node to get sleep. Node going to sleep sequence is shown in Figure 4 and Figure 5.
Function prototype	`void fw_forceToSleep(void)`
Arguments	none
Return value	none
Calling context	non-interrupt context
Restrictions	none

3.3.8. fw_appReadyToSleep(): Ready-to-Sleep Indication

Purpose	Indicate that user's application is ready to get sleep. Node going to sleep sequence is shown in Figure 4 and Figure 5.
Function prototype	`result_t fw_appReadyToSleep(void)`
Arguments	none
Return value	Returns `BUSY` if Framework cannot proceed the sleep procedure (it means, that user calls the function, but `fwReadyToSleep` has not been got yet) or `SUCCESS`, otherwise.
Calling context	Both interrupt and non-interrupt contexts. User's application should call this function in response to the `fwReadyToSleep()` callback (see Section 3.3.6) when it completes the preparations to sleeping.
Restrictions	none

NOTES:

User is not allowed to call `fw_appReadyToSleep` without having received the `fw_ReadyToSleep` callback (see Figure 5). Also, user is not allowed to call `fw_appReadyToSleep`, if data has been transmitted after receiving the `fw_ReadyToSleep` callback.

3.3.9. fw_forceWakeup(): Force the node to wake up

Purpose	Forces the eZeeNet Framework and eZeeNet stack to wake up before the end of sleep period. Active period will start and the user wakeup function will be called. Waking-up sequence is shown in Figure 6.
Function prototype	`void fw_forceWakeup(void)`
Arguments	none
Return value	none
Calling context	Non-interrupt context. Separate task should be posted from the user interrupt handler (CPU is assumed to be woken up by the user interrupt enabled during sleep period). fw_forceWakeup should be called within that task.
Restrictions	Applicable for end-device only.

3.3.10.fw_eepromWrite(): Direct Write to EEPROM

Purpose	Write a block of bytes to EEPROM. User's area starts from the address defined by `FW_EEPROM_START` macro.
Function prototype	`void fw_eepromWrite(EEPROMParams_t *params)`
Arguments	`params` – see Table 12.
Return value	none
Calling context	This function should be called in the non-interrupt context, but interrupts should be enabled.
Restrictions	eZeeNet does not support nested or overlapped calls of this function which can happen potentially because TinyOS events and user-defined tasks are processed when this function is executed.

NOTE:

EEPROM access is a relatively slow process that may take several tens of milliseconds. To avoid real-time problems, this function provides processing the TinyOS events during its execution.

Table 12. EEPROMParams_t struct

Type	Name	Description
`uint16_t`	`addr`	Start address
`uint8_t*`	`data`	Start data pointer
`uint16_t`	`length`	Block size
`EEPROMStatus_t`	`status`	Status of operation, see Table 13 for details

Table 13. EEPROMStatus_t enumeration

Value	Description
`EEPROM_SUCCESS_STATUS`	Operation has been completed successfully.
`EEPROM_ADDR_TOO_FAR_STATUS`	`addr` parameter is out of range.
`EEPROM_LENGTH_TOO_LONG_STATUS`	`length` parameter is out of range.
`EEPROM_HARDWARE_ERROR_STATUS`	Hardware error has occurred.
`EEPROM_OPERATION_OVERFLOW_STATUS`	Previous writing/reading request has not been finished yet.
`EEPROM_INVALID_DATA_POINTER_STATUS`	Start data pointer in parameters is equal to 0.

3.3.11.fw_eepromRead(): EEPROM Direct Reading

Purpose	Read a block of bytes from EEPROM. User's area starts from the address defined by `FW_EEPROM_START` macros.
Function prototype	`void fw_eepromRead(EEPROMParams_t *params)`
Arguments	`params` – see Table 12.
Return value	none
Calling context	This function should be called in the non-interrupt context, but interrupts should be enabled.

Restrictions	eZeeNet does not support nested or overlapped calls of this function that can happen potentially because of TinyOS events and user-defined tasks are processed when this function is executed.

NOTE:

EEPROM access is a relatively slow process that may take several tens of milliseconds. To avoid real-time problems, this function provides processing the TinyOS events during its execution.

3.3.12.fw_setParam(): Set eZeeNet Parameter

Purpose	Set eZeeNet parameter. Most eZeeNet parameters are stored in EEPROM. eZeeNet checks the parameter value before writing down to minimize the number of EEPROM write cycles.
Function prototype	`result_t fw_setParam(const FW_Param_t *param)`
Arguments	`param` – parameter description pointer `FW_Param_t` structure is described in Table 14.
Return value	Returns `FAIL`, if `id` in `param` is out of range `BUSY` if operation should be repeat later as EEPROM is busy `SUCCESS`, otherwise.
Calling context	This function should be called in the non-interrupt context, but interrupts should be enabled.
Restrictions	This function can call implicitly `fw_eepromWrite()` and `fw_eepromRead()` functions; their own restrictions are also imposed.

NOTES:

Setting some parameters may require stack restart to apply the changes, see Table 16. Those parameters are marked with the RW* attribute.

3.3.13.fw_getParam():Get eZeeNet Parameter

Purpose	Get eZeeNet parameter.
Function prototype	`result_t fw_getParam(FW_Param_t *param)`
Arguments	`param` – parameter description pointer `FW_Param_t` structure is described in Table 16.
Return value	Returns `FAIL`, if `id` in `param` is out of range or `SUCCESS`, otherwise.

Calling context	This function should be called in the non-interrupt context, but interrupts should be enabled.
Restrictions	This function can call implicitly `fw_eepromRead()` function; its restrictions are also imposed.

Table 14. FW_Param_t struct

Type	Name	Description
`FW_ParamID_t`	`id`	Parameter id See the description of `FW_ParamID_t` type in Table 15.
`FW_ParamValue_t`	`value`	Parameter value See the description of `FW_ParamValue_t` type in Table 16.

Table 15. FW_ParamID_t type

Value	corresponding member from FW_ParamValue_t union, (see Table 16)
`FW_NODE_ROLE_PARAM_ID`	`role`
`FW_NODE_CAPABILITY_PARAM_ID`	`nodeCapability`
`FW_NODE_LOGICAL_ADDR_PARAM_ID`	`logicalAddr`
`FW_NODE_NWK_ADDR_PARAM_ID`	`NWKAddr`
`FW_MAC_ADDR_PARAM_ID`	`macAddr`
`FW_PANID_PARAM_ID`	`panID`
`FW_ACTUAL_PANID_PARAM_ID`	`actPANID`
`FW_CURRENT_CHANNEL_PARAM_ID`	`channel`
`FW_CHANNEL_MASK_PARAM_ID`	`channelMask`
`FW_LQI_PARAM_ID`	`lqi`
`FW_TX_POWER_PARAM_ID`	`txPower`
`FW_PARENT_ADDR_INFO_PARAM_ID`	`parentAddrInfo`
`FW_CHILDREN_ADDR_INFO_PARAM_ID`	`childrenAddrInfo`
`FW_AUTO_NETWORK_PARAM_ID`	`autoNetwork`
`FW_POWER_DURATION_PARAM_ID`	`powerDuration`
`FW_SECURITY_KEY_PARAM_ID`	`securityKey`
`FW_MANUFACTURER_ID_PARAM_ID`	`manufacturerId`
`FW_MODEL_ID_PARAM_ID`	`modelId`
`FW_HARDWARE_SOFTWARE_ID_PARAM_ID`	`hardsoftId`
`FW_DATA_RETRY_PARAM_ID`	`dataRetryMax`
`FW_DATA_TIMEOUT_PARAM_ID`	`dataDeliveryTimeout`
`FW_SYNC_PERIOD_PARAM_ID`	`syncPeriod`
`FW_GPIO_CONFIG_PARAM_ID`	`gpioConfig`
`FW_UART_DTR_PARAM_ID`	`uartDTR`
`FW_UART_SPEED_PARAM_ID`	`uartSpeed`
`FW_UART_FC_PARAM_ID`	`uartFC`

Value	corresponding member from FW_ParamValue_t union, (see Table 16)
FW_ADC_CONFIG_PARAM_ID	adcConfig
FW_I2C_CONFIG_PARAM_ID	i2cConfig
FW_USART_CONFIG_PARAM_ID	usartConfig
FW_CALIBR_PERIOD_PARAM_ID	calibrPeriod

Table 16. FW_ParamValue_t union

Type	Name	Attribute	Persistence	Description
NodeLogicalType_t	role	RW*	x	Node's role (see Table 10) – coordinator, router or end-device. Default value: depends on the particular library used. If universal library (supporting all types of devices) is used, it will be ZIGBEE_ROUTER_TYPE
NodeCapability_t	nodeCapability	R		Defines the node roles possible in the network (see Table 10).
NodeLogicalAddr_t	logicalAddr	RW	x	Node logical address (see Section 4.3.1). NOTE: The logical address will be used for communications between applications. It should be assigned in some way to make it unique within particular PAN. Default value: 0
NWKAddr_t	NWKAddr	R		Node NWK address (see Section 4.3.1). If node is not in the network, this value is not defined.
IEEEAddr_t*	macAddr	RW*	x	Node MAC address. If eZeeNet detects it defined in EEPROM, the found value is read for use. If undefined (set to zero), eZeeNet checks if MAC address is defined in ZigBit module by hardware, and the detected address is stored in EEPROM. Default value of MAC address is zero. The module will not join the network until user will set MAC address to any value non-zero and non-equal to 0xFFFFFFFFFFFFFFFF or the address is detected in hardware. Memory for the parameter should be allocated by user.

Type	Name	Attribute	Persistence	Description
PANID_t	panID	RW*	x	PAN ID is used for networking. If it is set to 0xFFFF, the module will join the network irrespectively to its PAN ID. Default value: 0xFFFF for router and end-device; 0xFFFE for coordinator
PANID_t	actPANID	R		This parameter contains actual PAN ID that is used for networking. If panID is set to 0xFFFF and the node has joined the network, this parameter will keep PAN ID of the selected network. If the node has not connected, this parameter contains 0xFFFF.
uint8_t	channel	R		Current working channel since network start. If node is not in the network, the value is 0xFF.
uint32_t	channelMask	RW*	x	Channel mask Default value: 0x00000800
FW_LQIParam_t*	lqi	R		Current RSSI and LQI according to connection directed from the specified node. See Table 17 for more details. Memory for the parameter should be allocated by user. NOTE: RSSI/LQI information is retrieved for links within one-hop radius. LQI is not provided for multi-hop links.
int8_t	txPower	RW*	x	Transmission power in dBm. Power level resolution is typically 3 dB. NOTE: This setting will be applied to the radio circuitry during the warm reset procedure or hardware reset. Thus, the accurate setting of TX power requires warm reboot of the module. Default value: 0
NodeAddrInf_t*	parentAddrInfo	R		Parent MAC and network addresses. If the module is not in the connected state (see Section 4.3.6) or it is run as coordinator, 0 will be returned instead of MAC address. Memory for the parameter should be allocated by user.

Type	Name	Attribute	Persistence	Description
FW_ChildrenAddrInfo_t*	childrenAddrInfo	R		Children MAC and network address list. See Table 18 for details. If the module is not in the connected state (see Section 4.3.6) or it is run as an end-device, the list will be empty. Memory for the parameter should be allocated by user.
uint16_t	autoNetwork	RW	x	The parameter controls the node's activity at power-up or when it detects connection loss. This is the interval (in seconds) between two attempts to join the network in case of network loss. Zero value disables automatic joining. Default value: 0
FW_PowerDuration_t*	powerDuration	RW*	x	Sleep and active period for end-device. See Table 19 and Section 3.3.6 for details. Memory for the parameter should be allocated by user. Default value: See Table 19 NOTE: Actual sleep/active periods will be slightly different and their values will depend on multiple circumstances such as the network activity, external interfaces to the sensors, etc. They cannot be used for absolute timing.
uint16_t	syncPeriod	RW*	x	The period (in 100 milliseconds units) for tracking the end-device by its router. Default value: 600
uint8_t*	securityKey[16]	RW	x	Encryption key for data transmission. Setting this key does not mean sending data automatically encrypted or receiving it decrypted. This key is used in explicit call for the aes_encrypt function encrypting the data and the aes_decrypt () function for decrypting one. See Section 4.3.13, Section 4.3.14 for details. Default value: zeros

Type	Name	Attribute	Persistence	Description
uint8_t*	manufacturerId	R		Manufacturer identifier Memory for the parameter should be allocated by user.
uint8_t*	modelId	R		Model identifier Memory for the parameter should be allocated by user.
uint8_t*	hardsoftId	R		Hardware/software revision identifier Memory for the parameter should be allocated by user.
uint16_t	dataDeliveryTimeout	RW*	x	Timeout (in milliseconds) between data resend attempts in case of delivery failure Default value: 1000
uint8_t	dataRetryMax	RW*	x	Max data transmission repetitions in case of delivery failure Default value: 3
FW_GPIOConfig_t*	gpioConfig	RW	x	GPIO configuration. See Table 22. Memory for the parameter should be allocated by user. Default value: Table 22.
bool	uartDTR	RW	x	UART_DTR line behavior Default value: FALSE
UARTBaudRate_t	uartSpeed	RW	x	UART speed Default value: UART_BAUDRATE_38400
bool	uartFC	RW	x	UART flow control Default value: FALSE
uint8_t	adcConfig	RW	x	ADC configuration The configuration is a bit field. Each bit enables or disables the corresponding ADC channel. Usable bits: 0, 1, 2, 3 Default value: 0 (ADC is disabled)

Type	Name	Attribute	Persistence	Description
I2CMode_t	i2cConfig	RW	x	I^2C configuration, see Table 42 and Section 5.3.5 Default value: I2C_CLOCK_RATE_62
FW_USARTConfig_t*	usartConfig	RW	x	USART configuration, see Table 20 Memory for the parameter should be allocated by user. Default value: see Table 20
uint16_t	calibrPeriod	RW	x	RC oscillator calibration period defined in minutes. This parameter can be reset anytime to change calibration periodicity. Setting to 0 (zero) means calibration disabled. For end device, calibration will start when the device wakes up in time after the current calibration period is over. Default value: 60

NOTE:

Attribute field means the following:

R – parameter is read-only

RW – parameter can be read or written

RW* -- parameter can be read or it can be written, but eZeeNet stack should be restarted (see fw_stackRestart(), Section 4.3.11) to apply the changes.

Table 17. FW_LQIParam_t struct

Type	Name	Description
IEEEAddr_t	extAddr	MAC address of the node, from which LQI and RSSI values are requested
uint8_t	lqi	LQI value ranged by 0…255 If the node is not in the network or LQI information is not available, 0 is returned.
int8_t	rssi	RSSI value expressed in dBm If RSSI is not available then -128 is returned.

Table 18. FW_ChildrenAddrInfo_t struct

Type	Name	Description
`uint8_t`	`size`	Children MAC address array size User sets this parameter to max address list size. Framework sets this parameter to the actual number of returned addresses. Maximum number of children is defined by the sum of `NWK_MAX_END_DEVICE` and `MAX_ROUTER_NEIB_NUMBER` (see also Table 25).
`NodeAddrInf_t*`	`addr`	Children MAC and network address array. Allocated by user.

Table 19. FW_PowerDuration_t struct

Type	Name	Description
`uint16_t`	`active`	Active phase duration (in 10 milliseconds units) Default value: `10`
`uint16_t`	`sleep`	Sleep phase (in 100 milliseconds units) duration. Default value: `10`

Table 20. FW_USARTConfig_t struct

Type	Name	Description
`FW_USARTState_t`	`state`	State USART See Table 21 for details. Default value: `FW_USART_MODE_OFF`

`union`
`{`
`  UARTMode_t uart;`
`  SPIMode_t spi;`
`}`
	`mode`	USART mode
		See Section 5.3.6 and Section 5.3.7 for details.

Table 21. FW_USARTState_t type

Value	Description
`FW_USART_MODE_OFF`	USART is not used.
`FW_USART_MODE_UART`	USART is in UART mode.
`FW_USART_MODE_SPI`	USART is in SPI mode.

Table 22. FW_GPIOConfig_t struct

Type	Name	Description
`uint8_t`	`gpio`	GPIO number
`GPIOMode_t`	`mode`	GPIO mode See [6] and Section 5.2 for details. Default value: `GPIO_Z`

4. eZeeNet Stack Interfaces

4.1. Function Summary

Functions described in this chapter are intended for:

- network management (start network, restart network, stop network)
- network monitoring
- data transmission and receiving.

Data can be transmitted in two ways: using logical or network addressing.

As advantage, logical address of a node is not fixed. Logical addressing is preferable when the address of each node is known in advance or when the addresses can be preset during the commissioning procedure. As disadvantage, address conflicting is possible and should be resolved manually or by dedicated software running on the coordinator node.

NWK addresses are allocated and changed dynamically. NWK addressing scheme is only recommended for initial network addressing setup when application receives the data from some unknown node or when several nodes in the network have to use the same logical address. This would be the way to resolve address duplication or provide plug-and-play node installation.

NWK addressing scheme can be also used in wireless network where data is collected at single central point (sink) and no data should be transmitted back. There, logical addressing is not required because NWK address is known for coordinator and it equals zero.

LQI/RSSI values are returned in each data primitive. Although their values correspond to the last hop, this allows to make simplest range estimations based on RSSI. To do it, end-devices can broadcast periodically their specific data like MAC address, some ID or even text name. Application running on the routers will collect data from end-devices, bundle it with measured RSSI/LQI values and send it to central node (coordinator).

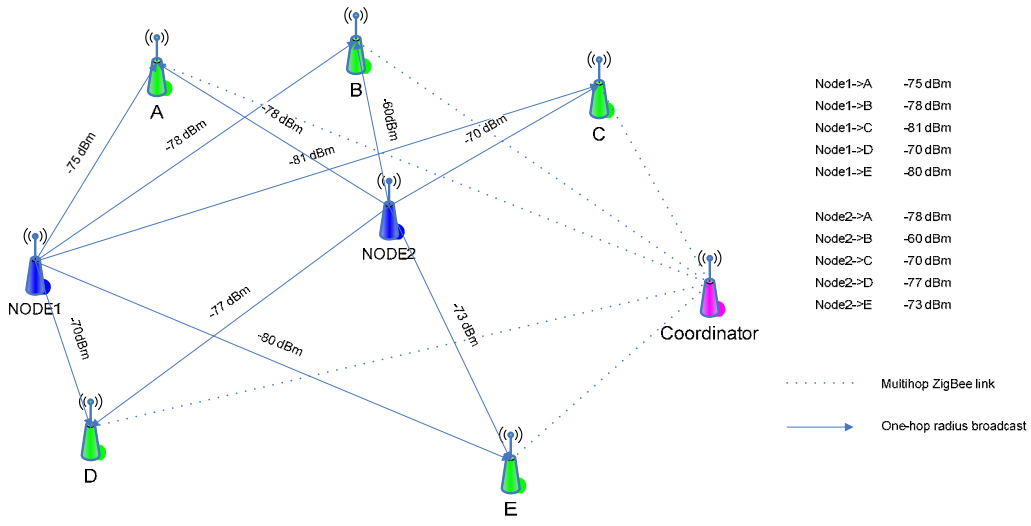

Figure 7. RSSI-based location service

eZeeNet Stack interface functions are declared in the "framework.h" file.

Table 23. eZeeNet stack functions

Function	Description	Ref.
fw_registerNetworkEvents	Register the network event handlers	4.3.3
fw_joinNetwork	Join/start the network	4.3.4
fw_leaveNetwork	Leave the network	4.3.5
fw_isJoined	Check networking status	4.3.6
fw_registerEndpoint	Register the end-point to receive data	4.3.12
Fw_registerLogicalAddress	Register the network event handlers	4.3.7
fw_dataRequest	Data sending	4.3.8
fw_dataIndicationControl	Data indication flow control	4.3.9
fw_delayedDataRequest	Delayed data request	4.3.10
fw_stackRestart	Restart the stack	4.3.11
aes_encrypt	AES128 encryption	4.3.13
aes_decrypt	AES128 decryption	4.3.14

4.2. Call Sequences

The network join-leave sequence is shown in Figure 8. The network join-lost-join sequence is shown in Figure 9 and Figure 10, depending on automatic networking setup.

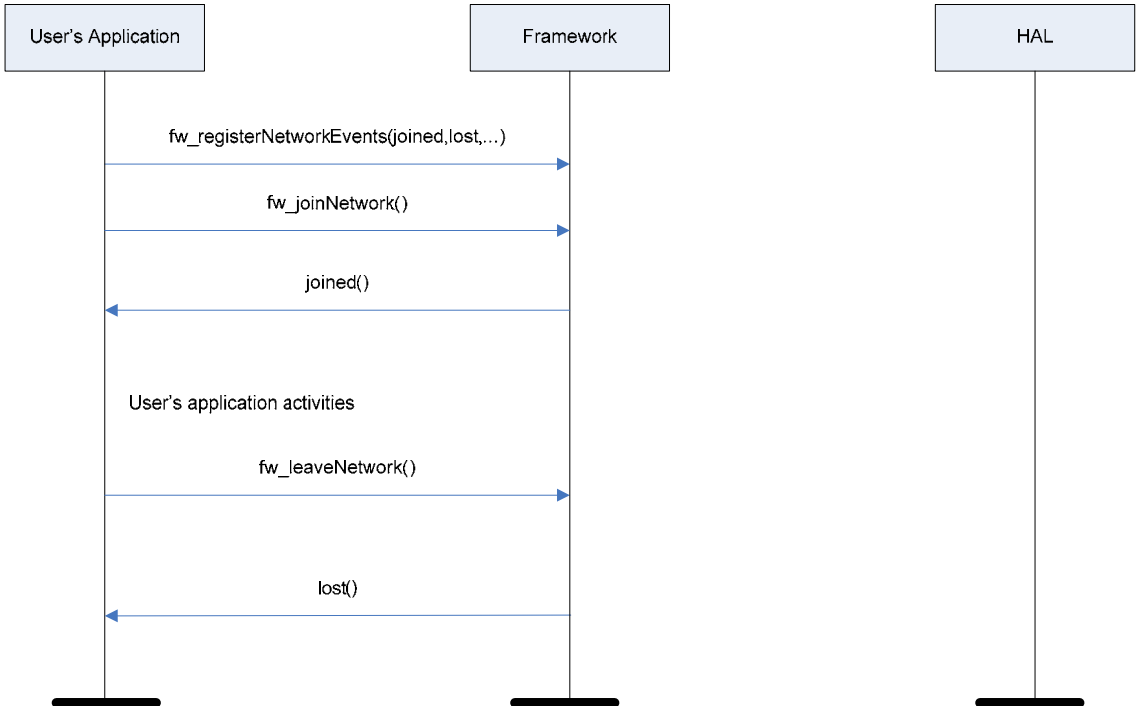

Figure 8. Join-leave sequence

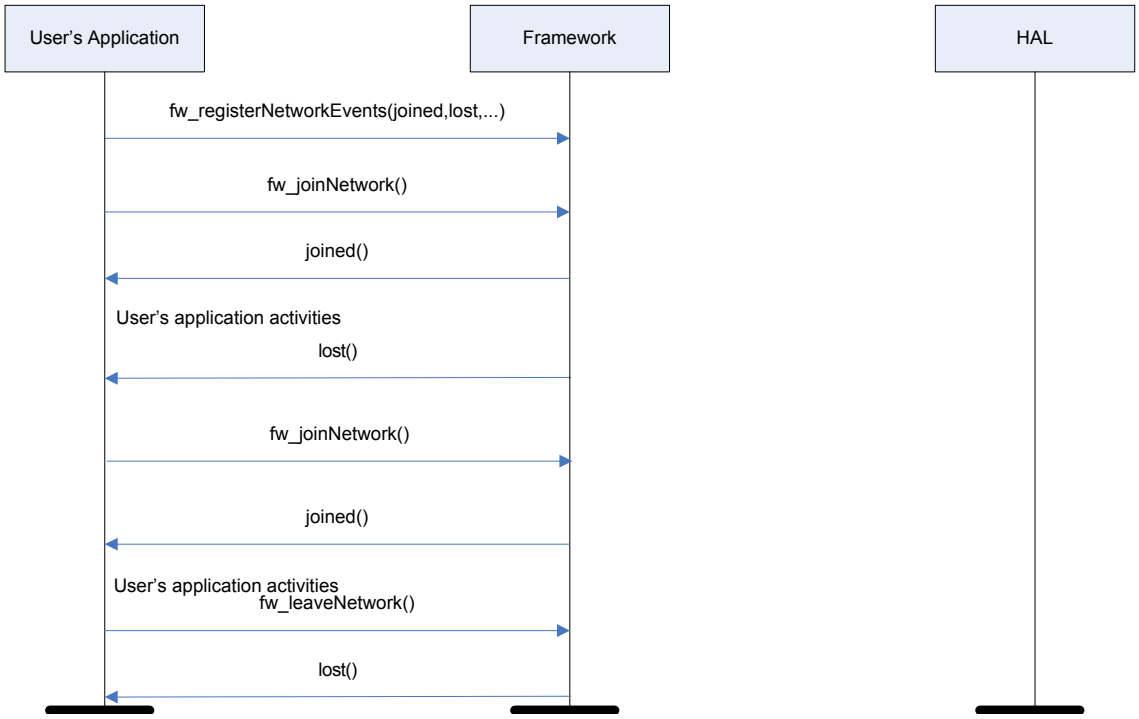

Figure 9. Join-lost-join sequence (automatic networking disabled)

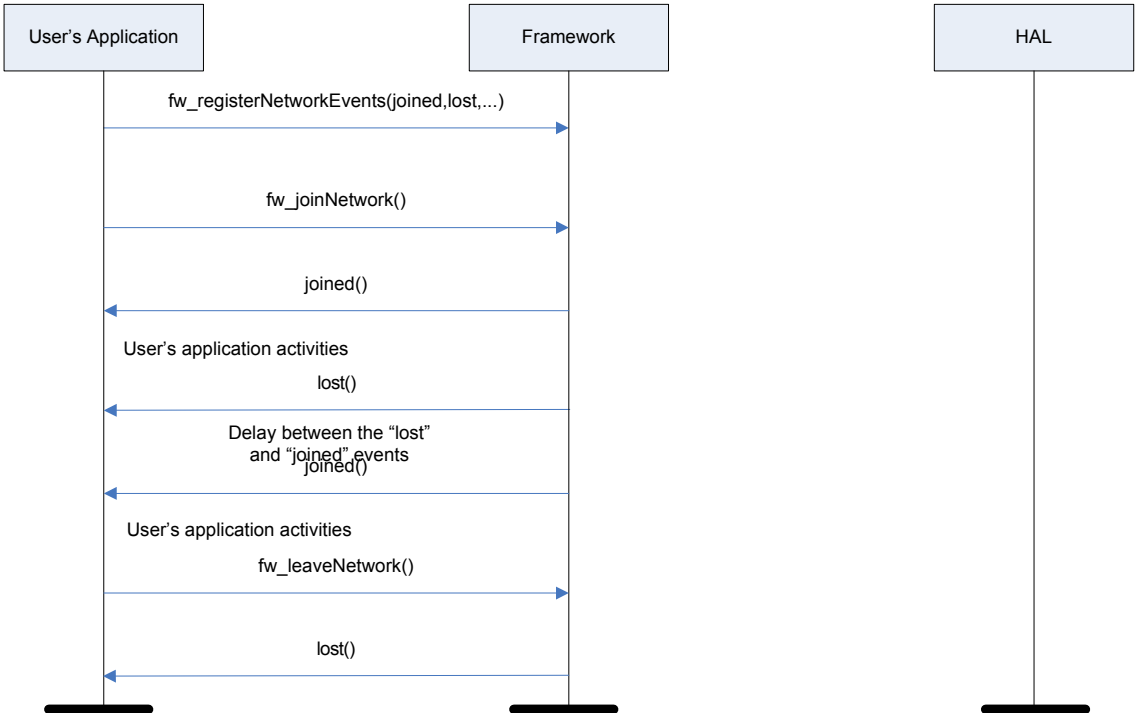

Figure 10. Join-lost-join sequence (automatic networking enabled)

If automatic networking is enabled, the delay time between `lost` and `join` callbacks is depending on automatic networking parameters.

Data transmission sequence is shown in Figure 11.

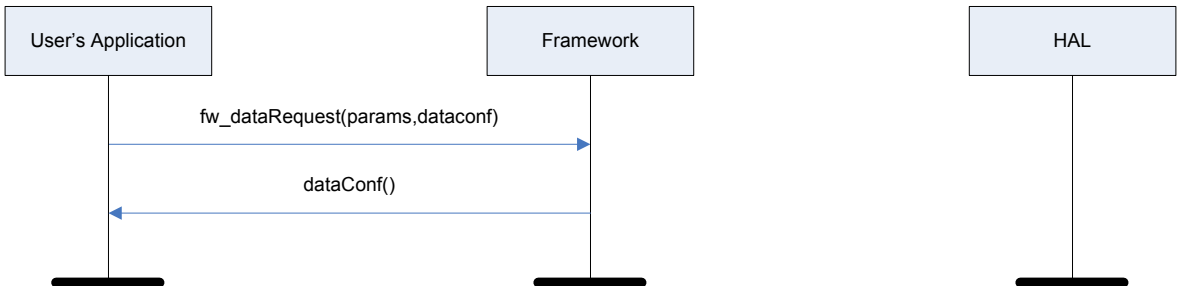

Figure 11. Data transmission sequence

For end-device, data receive sequence has some specifics. See Figure 12.

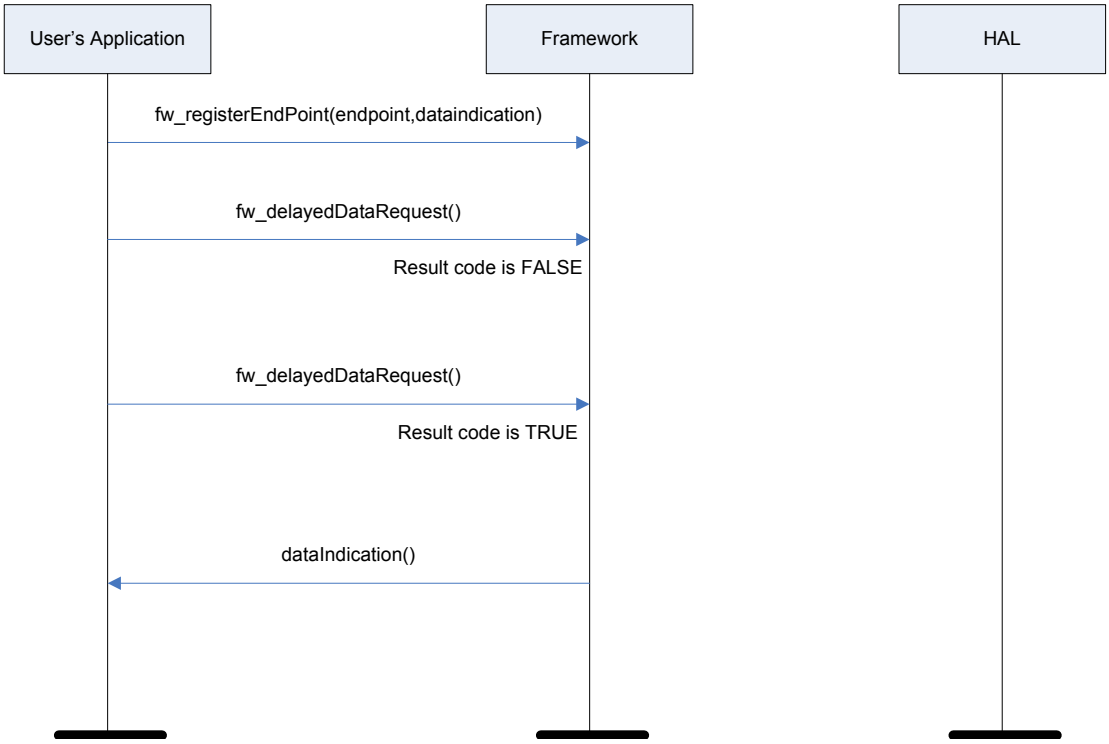

Figure 12. Data receive sequence for end-device

4.3. Detailed description

4.3.1. Data Structures

The following types are used in eZeeNet Stack interfaces: `NodeLogicalAddr_t` and `NWKAddr_t`. See Section 3.3.1 for definitions.

The following types of node address are used (see Table 24):

Table 24. NodeAddrMode_t type

Value	Description
`NODE_NWK_ADDR_MODE`	NWK addressing mode
`NODE_LOGICAL_ADDR_MODE`	Logical addressing mode

4.3.2. eZeeNet Network Configuration

The eZeeNet network configuration parameters are defined in the `".\API\StackSupport\include\Config.h"` file. They can be changed for user's application which should be then recompiled. Override these values if needed in the corresponding makefile, instead of changing them in source code.

NOTE:

Changing the network configuration parameters can result in more RAM required.

Table 25. The eZeeNet network configuration parameters

Name	Description
NWK_MAX_END_DEVICES	Maximum number of end devices that can be associated with coordinator or router. Default value: 5
MAX_ROUTER_NEIB_NUMBER	Maximum number of routers that can be associated with coordinator or router. Default value: 5
NWK_MAX_DEPTH	Maximum depth of network. Default value: 5
MAX_PAN_DESCRIPTORS	Maximum number of PAN descriptors used in network scanning; it is recommended to be equal 3 at least. Default value: 5

4.3.3. fw_registerNetworkEvents(): Register the Network Event Handlers

Purpose	Register the network event handlers. General network events are considered as opposed to those events associated with logical addressing – see Section 4.3.7.
Function prototype	`void fw_registerNetworkEvents(const FW_NetworkEvents_t *handlers)`
Arguments	`handlers` – pointer to network event handlers set. `FW_NetworkEvents_t` type is described in Table 26.
Return value	none
Calling context	non-interrupt context
Restrictions	none

Table 26. FW_NetworkEvents_t struct

Type	Name	Description
`void (*)(void)`	`joined`	The callback function that is called by eZeeNet if a node is joined to the network successfully.
`void (*)(void)`	`lost`	The callback function that is called by eZeeNet if a node lost network connection.

Type	Name	Description
`void (*)(const NodeAddrInf_t*)`	`addNode`	The callback function that is called by eZeeNet if a child node with `addr` address has been added. Note that it is an NWK address (see Section 4.1). Address information should be handled or copied before function return.
`void (*)(const NodeAddrInf_t*)`	`deleteNode`	The callback function that is called by eZeeNet if a child node with `addr` address has been deleted. Note that it is an NWK address (see Section 4.1). Address information should be handled or copied before function return.

NOTES:

If automatic networking is enabled, the `joined` and `lost` functions are only called back after user's call for `fw_joinNetwork`. If opposite, these functions may be called back without calling `fw_joinNetwork`.

`addNode` and `deleteNode` are only called after `joined` is called back by Framework.

4.3.4. fw_joinNetwork(): Join/Start the Network

Purpose	Force a node to start a network (if coordinator) or join (if router or end-device) to the existing network. See Figure 8 for join sequence. In return to this function call, `joined` or `lost` will be issued. If the node joins the network successfully or it has been already joined, then `joined` event will be issued. The `lost` callback will be issued otherwise.
Function prototype	`void fw_joinNetwork(void)`
Arguments	`none`
Return value	none
Calling context	non-interrupt context
Restrictions	none

4.3.5. fw_leaveNetwork(): Leave the Network

Purpose	Force the node to leave the network The node forces all its children to leave and signalize a corresponding event to its parent node. See Figure 8 for leave sequence.

Purpose	Force the node to leave the network The node forces all its children to leave and signalize a corresponding event to its parent node. See Figure 8 for leave sequence.
Function prototype	`void fw_leaveNetwork(void)`
Arguments	none
Return value	none
Calling context	non-interrupt context
Restrictions	none

NOTE:

This function disables the automatic networking function (see `autoNetwork` parameter, Table 16) temporarily. To enable automatic networking, the node should execute `fw_joinNetwork()` function, see Section 4.3.4.

4.3.6. fw_isJoined(): Check Networking Status

Purpose	Request a node for networking status
Function prototype	`bool fw_isJoined(void)`
Arguments	none
Return value	`TRUE` – the node is in the network `FALSE` – the node is not joined to the network or it has been orphaned by its parent
Calling context	both interrupt and non-interrupt context
Restrictions	none

NOTE:

Due to asynchronous nature of the network, if this function returns `TRUE` that means that the node is in the connected state, but it does not mean that subsequent network data transmissions will be successful.

4.3.7. fw_ registerLogicalAddressEvents (): Register Logical Addressing Notification Handlers

Purpose	Registers the handlers for notification events associated to logical addressing (which are opposed to general network events – see Section 4.3.3).
Function prototype	`void fw_registerLogicalAddressEvents(const FW_LogicalAddressEvents_t *handlers)`
Arguments	`handlers` – pointer to the handlers set. `FW_LogicalAddressEvents_t` type is described in Table 27.
Return value	none
Calling context	non-interrupt context
Restrictions	Used for coordinator only

Table 27. `FW_LogicalAddressEvents_t` struct

Type	Name	Description
void (*)(const NodeLogicalInf_t* addr)	addLogicalNodeAddr	the callback function that is called by eZeeNet if a new node with logical address has joined the network
void (*)(const NodeLogicalInf_t* addr)	deleteLogicalNodeAddr	the callback function that is called by eZeeNet if an existing node with logical address has left the network

Table 28. `NodeLogicalInf_t` struct

Type	Name	Description
NodeLogicalAddr_t	logicalAddr	logical address of the node
NWKAddr_t	NWKAddr	NWK address of the node

NOTE:

This function may be used on the coordinator only. Callback functions will be issued for all adding or deleting node events within entire network, not only for coordinator's child devices.

4.3.8. fw_dataRequest(): Data Sending

Purpose	Data transmission request `FW_DataRequest_t` structure is described in Table 30. `FW_DataStatus_t` type is described in Table 29. NOTE: The `FW_DataRequest_t` structure could be formed dynamically or even allocated in stack. There is no need to keep formed request in memory until corresponded acknowledgement will be received.
Function prototype	`result_t fw_dataRequest(const FW_DataRequest_t *params, void (*conf)(uint8_t handle, FW_DataStatus_t status))`
Arguments	`params` – data transmission parameters, see Table 30 `conf` – confirmation for data transmission.
Return value	`FAIL` – if the data to be transmitted is too long (see Table 30) `BUSY` – if the data cannot be transmitted at the moment and transmission can be repeated later `SUCCESS` – otherwise
Calling context	non-interrupt context
Restrictions	This function should be called in the non-interrupt context but interrupts should be enabled.

4.3.9. fw_dataIndicationControl(): Data Indication Flow Control

Purpose	Function to manipulate by Framework data indication flow. `enable` – permission to indicate the received data frame for user application.
Function prototype	`void fw_dataIndicationControl(uint8_t endpoint, bool enable)`
Arguments	`endpoint` – the number of the endpoint for which flow control is changed. Value `0` is used for simultaneous control of all endpoints. `enable` – the indication mode. If it is `TRUE`, the received data will be sent to user application, otherwise they will be stored by eZeeNet until mode is `TRUE`. Atfer the mode is `TRUE` the stored frames will be indicated to user application.
Return value	none
Calling context	interrupt and non-interrupt context
Restrictions	none

NOTE:
Initially, when Framework starts, the indication is enabled.

Table 29. FW_DataStatus_t type

Value	Description
FW_DATA_ACK_STATUS	Data transmission success
FW_DATA_NOACK_STATUS	Data transmission failure

Table 30. FW_DataRequest_t struct

Type	Name	Description
NodeLogicalAddr_t	dstLogicalAddr	Logical address of destination node (if addrMode == NODE_ADDR_LOGICAL)
NWKAddr_t	dstNWKAddr	NWK address of destination node (if addrMode == NODE_ADDR_NWK)
NodeAddrMode_t	addrMode	Type of the destination address used, see Table 24
uint8_t	srcEndPoint	Endpoint of source node
uint8_t	dstEndPoint	Endpoint of destination node
bool	arq	This flag defines data delivery control mode: TRUE – enable ARQ control (application-level acknowledgement will be used) FALSE – disable ARQ control (MAC acknowledgement will be used) ARQ control is defined by the dataDeliveryTimeout and dataRetryMax parameters, see Table 16.
bool	broadcast	This flag defines if broadcast transmission should be done: TRUE – enable broadcasting FALSE – disable broadcasting
uint8_t*	data	Data buffer pointer

Type	Name	Description
uint8_t	length	Data length Maximum length is defined by FW_DATA_MAX_LENGTH macro.
uint8_t	handle	Data frame handle Several data packets can be transmitted concurrently (without waiting for confirmation). Thus, the handle can help to distinguish the packets.

NOTES

One-hop only broadcasting is performed.

NWK address (0XFFFF) is used in broadcasting so that none of addrMode, dstLogicalAddr or dstNWKAddr defined above is required.

Table 31. FW_DataIndication_t struct

Type	Name	Description
NodeLogicalAddr_t	srcLogicalAddr	Source node logical address. This field is valid if isLogicalValid==TRUE, otherwise its value is not defined
NWKAddr_t	srcNWKAddr	NWK address of source node
bool	isLogicalValid	TRUE if logical address is known, otherwise FALSE
uint8_t	srcEndPoint	Source endpoint
uint8_t	dstEndPoint	Destination endpoint
uint8_t*	data	Data buffer pointer
uint8_t	length	Data length
bool	broadcast	Broadcast transmission flag
uint8_t	lqi	LQI value ranged by 0...255
int8_t	rssi	RSSI value expressed in dBm

NOTES:

LQI/RSSI values returned in the data indication primitive are related to the last hop. It is important that they are available both for unicast and broadcast data.

The indication data should be handled or copied before function return.

4.3.10.fw_delayedDataRequest(): Delayed Data Request

Purpose	Request explicitly for the data buffered on the router. Requesting for the data, which is possibly buffered on its router (see Figure 12), is required when the end-device should participate in two-way communications with other node. eZeeNet executes this request automatically on each wakeup, so this function has to be used in rare cases.
Function prototype	`bool fw_delayedDataRequest(void)`
Arguments	none
Return value	`TRUE`, if there is a data on the router and it was extracted `FALSE`, otherwise.
Calling context	non-interrupt context.
Restrictions	This function should be used on an end-device only. TinyOS events and user-defined tasks are processed whenever this function is executed. For this reason eZeeNet does not support nested or overlapped calls of this function that may potentially happen.

4.3.11.fw_stackRestart():Restart the Stack

Purpose	Restart the eZeeNet stack It is required in some cases, for instance when application software needs to apply the changes in some global settings, see Section 3.3.12.
Function prototype	`void fw_stackRestart(void)`
Arguments	none
Return value	none
Calling context	This function should be called in the non-interrupt context, but interrupts should be enabled.
Restrictions	This function enables to execute TinyOS events and user-defined tasks during execution of this function.

NOTE:

It is recommended to leave the network before calling this function.

4.3.12.fw_registerEndPoint(): Register the End-Point to Receive Data

Purpose	Register the end-point for data receive

Purpose	Register the end-point for data receive
Function prototype	`result_t fw_registerEndPoint(uint8_t endpoint, void (*ind)(const FW_DataIndication_t *params))`
Arguments	`endpoint` – endpoint number wish to register User's application should use the endpoint range from 1 to 239. Endpoint 240 is used for Framework purposes. The rest possible 8-bit values are reserved for network management. `ind` – data indication handler associated with the endpoint `FW_DataIndication_t` structure is described in Table 31.
Return value	`FAIL`, if endpoint number is out of range `SUCCESS`, otherwise
Calling context	non-interrupt context
Restrictions	none

NOTE:
If data come to the non-registered end-point they will be lost.

4.3.13.aes_encrypt (): AES128 Encryption

Purpose	Encrypt the input data using the known 128-bit key
Function prototype	`void aes_encrypt(const uint8_t key[], uint8_t buf[], uint8_t temp[]);`
Arguments	`key` – 128-bit key `buf` – 16-byte block to be encrypted `temp` – 176 bytes of temporary data
Return value	none Encryption result replaces input data block.
Calling context	non-interrupt context
Restrictions	none

4.3.14.aes_decrypt (): AES128 Decryption

Purpose	Decrypt input data using the known 128-bit key
Function prototype	`void aes_decrypt(const uint8_t key[], uint8_t buf[], uint8_t temp[])`

Purpose	Decrypt input data using the known 128-bit key
Arguments	`key` – 128-bit key
	`buf` – 16-byte block to be decrypted
	`temp` – 176 bytes of temporary data
Return value	none
	Decryption result replaces input data block.
Calling context	non-interrupt context
Restrictions	none

5. HAL Interfaces

5.1. Functions Summary

HAL is supplied in source code. User can modify it to implement own drivers or to manage specific peripherals. When writing an application code, it should be taken into account that eZeeNet software runs the ZigBit module at 4 MHz. This is important to calculate clock rates for specific interfaces because most interfaces are controlled by the master MCU clock.

Table 32. HAL functions

Function	Description	Ref.
gpio_setConfig	Configure GPIO pins	5.3.1
gpio_getConfig	Read GPIO pins configuration	5.3.1
gpio_getState	Read GPIO pin	5.3.1
gpio_setState	Write to GPIO pin	5.3.1
irq_register	Register the ISR	5.3.2
irq_enable	Enable IRQ	5.3.2
irq_disable	Disable IRQ	5.3.2
irq_unregister	Unregister the ISR	5.3.2
adc_open	Open ADC channel	5.3.3
adc_get	Get ADC sample	5.3.3
adc_close	Close ADC channel	5.3.3
wdt_start	Start Watch-dog	5.3.4
wdt_stop	Stop Watch-dog	5.3.4
wdt_reset	Reset Watch-dog	5.3.4
i2cpacket_open	Open I^2C channel	5.3.5
i2cpacket_close	Close I^2C channel	5.3.5
i2cpacket_write	Write packet to I^2C	5.3.5
i2cpacket_read	Read packet from I^2C	5.3.5
uart_init	Initialize UART	5.3.6
uart_find_free_channel	Search for UART free channel	5.3.6
find_uart_by_descriptor	Search for UART by descriptor	5.3.6
uart_open	Open UART	5.3.6
uart_close	Close UART	5.3.6

Function	Description	Ref.
uart_write	Send data over UART channel	5.3.6
uart_read	Read the received data from UART channel	5.3.6
uart_cts_on	Disable host transmitting over UART	5.3.6
uart_cts_off	Enable host transmitting over UART	5.3.6
uart_rts	Read RTS into a variable	5.3.6
Uart_dtr	Read DTR into a variable	5.3.6
spi_open	Open SPI interface	5.3.7
spi_close	Close SPI interface	5.3.7
spi_readwrite	Read/write byte via SPI	5.3.7
appTimer_open	Open user timer	5.3.8
appTimer_close	Close user timer	5.3.8
appTimer_start	Start user timer	5.3.8
appTimer_stop	Stop user timer	5.3.8

5.2. Call Sequences

See Figure 13 for user's HAL startup sequence.

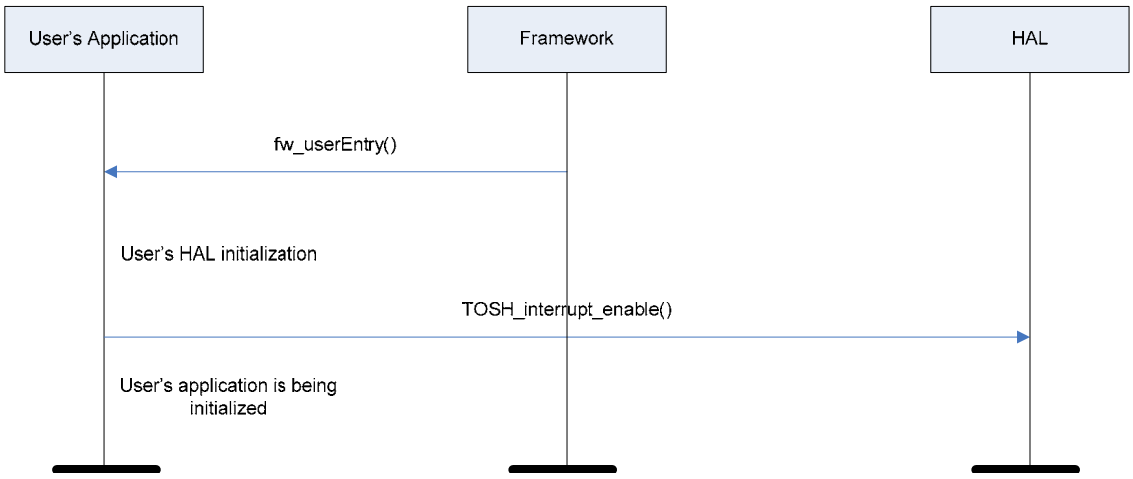

Figure 13. HAL startup sequence

5.3. Detailed description

5.3.1. GPIO Interface

GPIO interfaces are intended to manage peripherals connected to general purpose I/O pins of ZigBit module. The GPI interface functions are summarized in the Table 34. They are defined in "gpio.h" header, and they are implemented in "gpio.c" module.

Use the pin names as specified in Table 33 and GPIOMode_t type as described in Table 35.

All GPIO functions return FAIL if pin number is out of range or SUCCESS, otherwise.

NOTE:

If any of GPIO_0 - GPIO_8 pins is used for alternative function (by UART for instance), it cannot be used for GPIO.

Table 33. GPIO description

GPIO	ZigBit pin name
GPIO_0	GPIO0
GPIO_1	GPIO1
GPIO_2	GPIO2
GPIO_3	GPIO3
GPIO_4	GPIO4
GPIO_5	GPIO5
GPIO_6	GPIO6
GPIO_7	GPIO7
GPIO_8	GPIO8
GPIO_9	GPIO9
GPIO_I2C_CLK	I2C_CLK
GPIO_I2C_DATA	I2C_DATA
GPIO_UART_TXD	UART_TXD
GPIO_UART_RXD	UART_RXD
GPIO_UART_RTS	UART_RTS
GPIO_UART_CTS	UART_CTS
GPIO_ADC_INPUT_3	ADC_INPUT_3
GPIO_ADC_INPUT_2	ADC_INPUT_2
GPIO_ADC_INPUT_1	ADC_INPUT_1
GPIO_BAT	BAT
GPIO_UART_DTR	UART_DTR
GPIO_USART0_RXD	USART0_RXD
GPIO_USART0_TXD	USART0_TXD
GPIO_USART0_EXTCLK	USART0_EXTCLK
GPIO_IRQ_7	IRQ_7
GPIO_IRQ_6	IRQ_6

Table 34. GPIO interface functions

Function	Description
`result_t gpio_setConfig(uint8_t pin, GPIOMode_t mode)`	Configure a particular pin of ZigBit module to the specified mode.
`result_t gpio_getConfig(uint8_t pin, GPIOMode_t *mode)`	Read a particular pin configuration. Configuration is returned in the `mode` parameter.
`result_t gpio_getState(uint8_t pin, uint8_t *value)`	Get state of a pin into the `value` • `0` means logical zero • `1` means logical one. If pin is configured for output or as tri-state, returned value is not defined.
`result_t gpio_setState(uint8_t pin, uint8_t value)`	Set pin into `value` state. The command does not affect any pin configured as input or tri-state.

NOTE:

eZeeNet does not initialize GPIO pins on startup. But GPIO configuration can be saved/restored in EEPROM, see Section 3.3.12.

Table 35. GPIOMode_t type

Value	Description
`GPIO_INPUT_PULLUP_OFF=0`	Input pin, no internal pullup
`GPIO_INPUT_PULLUP_ON=1`	Input pin, internal pullup is turned on
`GPIO_Z=2`	Tri-state
`GPIO_OUTPUT =3`	Output

5.3.2. IRQ Interface

IRQ interfaces are intended to manage interrupts. The IRQ interface functions are listed in the Table 36. They are defined in the `"irq.h"` header and they are implemented in `"irq.c"` module.

Following macros should used as `irqNumber`: `IRQ_6, IRQ_7.`

Table 36. IRQ interface functions

Function	Description
`result_t irq_register(uint8_t irqNumber, IRQMode_t mode, void (*f)(void))`	Register user's `irqNumber` interrupt `mode` – interrupt activation condition, see Table 37. `f` – user's interrupt handler Returns `FAIL` if: • `irqNumber` is out of range • Such interrupt has been already registered; or `SUCCESS`, otherwise. This function does not enable interrupts. After registration, interrupt is disabled. Normally, this function should be called during initialization phase when all the interrupts are disabled, see Figure 13.
`result_t irq_enable(uint8_t irqNumber)`	Enable `irqNumber` interrupt Returns `FAIL`, if `irqNumber` is out of range or has not been registered yet, or `SUCCESS`, otherwise.
`result_t irq_disable(uint8_t irqNumber)`	Disable `irqNumber` interrupt Returns `FAIL`, if `irqNumber` is out of range or has not been registered yet, or `SUCCESS`, otherwise.
`result_t irq_unregister (uint8_t irqNumber)`	Un-registers user's `irqNumber` interrupt IRQ line will be moved to tri-state. Returns `FAIL`, if `irqNumber` is out of range or has not been registered yet, or `SUCCESS`, otherwise.

Table 37. IRQMode_t type: interrupt activation condition

Value	Description
`IRQ_LOW_LEVEL`	The low level generates an interrupt request.
`IRQ_ANY_EDGE`	Any edge generates an interrupt request.
`IRQ_FALLING_EDGE`	Falling edge generates an interrupt request.

Value	Description
IRQ_RISING_EDGE	Rising edge generates an interrupt request.

5.3.3. ADC Interface

ADC interface provides a simplest access to the ATmega1281V on-chip ADC. The ADC interface functions are listed in Table 38. They are defined in "adc.h" header and they are implemented in "adc.c" module.

The following conversion parameters are used:

Sample rate	125KHz
Conversion mode	Single conversion mode
Reference source	A_VREF

The following macros should used as adcNumber: ADC_BAT, ADC_INPUT_1, ADC_INPUT_2, ADC_INPUT_3.

ADC interface functions are defined in the "adc.h" and implemented in "adc.c".

Table 38. ADC interface functions

Function	Description
result_t adc_open(uint8_t adcNumber, void (*f)(uint16_t data))	Open channel adcNumber and register callback for data indication. f – data indication handler Returns FAIL if: • adcNumber is out of range • channel has been already opened; or SUCCESS, otherwise.
result_t adc_get(uint8_t adcNumber)	Get sample from ADC channel adcNumber The result is returned by the callback registered using adc_open() function. Returns FAIL if: • adcNumber is out of range • channel has not been opened yet; or SUCCESS, otherwise.

Function	Description
`result_t adc_close(uint8_t adcNumber)`	Close the `adcNumber` channel After closing the channel the corresponding pin is set to tri-state. If all channels are closed, `A_VREF` pin is moved to tri-state as well. Returns `FAIL` if: • `adcNumber` is out of range • channel has not been opened yet; or `SUCCESS`, otherwise.

NOTES:

eZeeNet does not open ADC channel automatically. But the used channels can be saved/restored in EEPROM, see Section 3.3.12.

When using the ZigBit module installed on the MeshBean2 board, the following restriction is imposed due to the board schematics. You have to consequently configure GPIO 6, GPIO 7 and GPIO 8 for output. Then set GPIO to 0, set GPIO7 and GPIO8 to 1. Now you can configure or read the particular A/D pins.

5.3.4. Watch-dog Interface

The watch-dog interface is intended to prevent the software hanged up due to deadlocks. The watch-dog interface functions are listed in Table 39.

Table 39. 5.1.4.Watch-dog interface functions

Function	Description
`void wdt_start(WDInterval_t interval)`	Start watch-dog timer For the description of `WDInterval_t` see Table 42.
`void wdt_stop(void)`	Stop watch-dog timer
`void wdt_reset(void)`	Reset watch-dog timer

The watchdog timer is disabled at startup. Later, eZeeNet Software does not use watchdog itself, but the following practice is recommended:

- insert the `wdt_reset()` call into user's loop, see Section 3.3.3
- consider user's loop frequency and sleep intervals to calculate correct watch-dog interval.

In such case, watch-dog timer restarts the node if `wdt_reset()` function will not be called during specified timeout.

Table 40. WDInterval_t type: watch-dog interval

Value	Description
WD_INTERVAL_16	16ms
WD_INTERVAL_32	32ms
WD_INTERVAL_64	64ms
WD_INTERVAL_125	125ms
WD_INTERVAL_250	250ms
WD_INTERVAL_500	500ms
WD_INTERVAL_1000	1s
WD_INTERVAL_2000	2s
WD_INTERVAL_4000	4s
WD_INTERVAL_8000	8s

5.3.5. I²C Interface

I²C interface is intended to communicate with I²C-based peripherals. I²C channel works in the master mode only.

I²C interface functions are described in Table 41. They are defined in "i2cpacket.h" header and they are implemented in "i2cpacket.c" module.

Table 41. I²C interface functions

Function	Description
result_t i2cpacket_open(const I2CMode_t *mode)	Open I²C channel I2CMode_t struct is described in Table 42. Returns FAIL if channel has already been opened or SUCCESS, otherwise.
result_t i2cpacket_close(void)	Close I²C channel and set the corresponding ZigBit pins (I2C_DATA, I2C_CLOCK) to tri-state Returns FAIL if channel has not been opened yet or SUCCESS, otherwise.

Function	Description
`result_t i2cpacket_write(uint8_t addr, uint8_t length, const uint8_t *data, void (*f)( bool result))`	Write I²C packet `addr` – device address `length` – packet length `data` – user data pointer The `length` value depends on the device connected to the bus. It should not exceed 256. `f` – the operation completion handler. Returns `FAIL` if previous read/write operation has not been completed yet or `SUCCESS`, otherwise.
`result_t i2cpacket_read(uint8_t addr, uint8_t length, uint8_t *data, void (*f)( bool result))`	Read I²C packet `addr` – device address `length` – packet length `data` – user data pointer `f` – the operation completion handler. Returns `FAIL` if previous read/write operation has not been completed yet or `SUCCESS` otherwise.

NOTES:

eZeeNet does not initialize I²C driver. But, I²C mode can be saved/restored in EEPROM, see Section 3.3.12.

User should allocate buffer for transmitted/received data and keep it reserved until operation is completed, avoiding using the buffer for any other purpose. Buffer size for the received data should be large enough for the requested packet defined by `length`. value.

Table 42. I2CMode_t struct

Type	Name	Description
`I2CClockRate_t`	`clockrate`	I²C clock rate. For more details see Table 43.

Table 43. I2CClockRate_t type: clock rate

Value	Description
`I2C_CLOCK_RATE_250`	250Kb/s
`I2C_CLOCK_RATE_125`	125Kb/se

Value	Description
I2C_CLOCK_RATE_62	62.5Kb/s

5.3.6. UART Interface

UART interface is intended for accessing USART's devices in UART mode.

The UART interface functions are listed in Table 44. They are defined in "uart.h" header and they are implemented in "uart.c" module.

There are two macros defining the channel used in the UART mode:

- UART_CHANNEL_0
- UART_CHANNEL_1.

UART_CHANNEL_0 can also work in SPI mode. For more details see Section 5.3.7.

There are three flow control modes to be selected:

- UART_FLOW_CONTROL_NONE – hardware flow control is not used
- UART_FLOW_CONTROL_HARDWARE – hardware flow control is used
- UART_DTR_CONTROL – hardware terminal readiness control is used.

Table 44. UART interface functions

Function	Description
`int uart_open (uint8_t tty, UARTMode_t* uartmode, uint8_t flags, void (*rx_callback)(uint8_t),void (*tx_callback)() )`	Open UART: • register the handlers for UART events. • perform configuration for UART registers • perform configuration for `RTS`, `CTS` and `DTR` pins. Parameters: • `tty` – the UART referencing number, which can be one of the defines: `UART_CHANNEL_0` or `UART_CHANNEL_1` • `uartmode` – the address of `UARTMode_t` structure • `flags` - one of the defines: `UART_FLOW_CONTROL_NONE` `UART_FLOW_CONTROL_HARDWARE` `UART_DTR_CONTROL`. The defines `UART_FLOW_CONTROL_HARDWARE` `UART_DTR_CONTROL` can be combined via logical OR operation. • `rx_callback` –address pointing to a function which notifies application about receiving each byte of data, otherwise `rx_callback` must be set to `NULL` • `tx_callback` – address pointing to a function which notifies application about completion of data transfer, otherwise `tx_callback` must be set to `NULL` . **Important Note:** `rx_callback, tx_callback` functions are called in an interrupt context. Returns: • -1 indicating at least one of the following errors: – incorrect UART channel – unsupported `flags` for selected UART channel – insufficient resources • otherwise, a positive UART descriptor value is returned.

Function	Description
`int uart_close (int descriptor)`	Close UART. Releases UART channel and pins, if hardware flow control was used. Parameter: `descriptor` – the UART descriptor. Returns: • -1 if descriptor is incorrect for it does not relate to any open channel • otherwise, 0 is returned.
`int uart_write(int descriptor, uint8_t *buffer, uint8_t length)`	Write data to UART channel. The `uart_write` function uses two modes to transmit data – synchronous and asynchronous, depending on parameters which were used for `uart_open`. If `tx_callback` function was not registered, then synchronous mode will be used. In synchronous mode, `uart_write` function will be able to return only after having the last byte transmitted. In asynchronous mode, `tx_callback` function will be used to notify about completion of transmitting `length` bytes. Parameters: • `descriptor` – UART descriptor • `buffer` - pointer to the application data buffer • `length` - a number of bytes to transmit. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – untransmitted data – the pointer to data is `NULL` – `length` is zero • otherwise, 0 is returned.

Function	Description
`int uart_read(int descriptor, uint8_t *buffer, uint8_t length);`	Read data received from UART to the application `buffer`. The `uart_read` function is used only in synchronous mode. Parameters: • `descriptor` – UART descriptor • `buffer` – pointer to the application buffer • `length` – the number of bytes of data attempted to read to `buffer`. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – the pointer to data is `NULL` – `length` is zero – asynchronous mode is used • otherwise, 0 is returned.
`int uart_cts_on(int descriptor)`	Force a connected device to stop sending data via UART. `UART_CHANNEL_1` only can be used for hardware flow control. Parameter: `descriptor` – UART descriptor. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – incorrect UART channel – when opening UART with `uart_open`, the `flags` parameter was not set to `UART_FLOW_CONTROL_HARDWARE` • otherwise, 0 is returned.

Function	Description
`int uart_cts_off(int descriptor)`	Allow a connected device sending data via UART. `UART_CHANNEL_1` only can be used for hardware flow control. If hardware flow control is used in UART asynchronous mode, the application must not close UART within `rx_callback` function. Parameter: `descriptor` – UART descriptor. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – when opening UART with `uart_open`, the `flags` parameter was not set to `UART_FLOW_CONTROL_HARDWARE` – mode is not supported • otherwise, 0 is returned.
`int uart_rts(int descriptor, UARTHardwareControl_t *rts)`	Read RTS pin potential into `rts` variable. `UART_CHANNEL_1` only must be used and hardware flow control mode must be selected. • Parameter: `descriptor` – UART descriptor • `rts` – destination address. `UARTHardwareControl_t` type is defined in Table 50. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – when opening UART with `uart_open`, `flags` parameter was not set to `UART_FLOW_CONTROL_HARDWARE` - `rts` address is NULL – mode is not supported • otherwise, 0 is returned.

Function	Description
`int uart_dtr(int descriptor, UARTHardwareControl_t *dtr)`	Read DTR pin potential into `dtr` variable. `UART_CHANNEL_1` only must be used and hardware flow control mode must be selected.. Parameter: • `descriptor` – UART descriptor. • `dtr` – destination address. `UARTHardwareControl_t` type is defined in Table 50. Returns: • -1 indicating at least one of the following errors: – incorrect UART descriptor – when opening UART with `uart_open`, the `flags` parameter was not set to `UART_DTR_CONTROL` – mode is not supported • otherwise, 0 is returned.

Table 45. UARTMode_t struct

Type	Name	Description
`UARTBaudRate_t`	`baudrate`	UART baud rate. See Table 46 for details.
`UARTData_t`	`data`	UART data length. See Table 47 for details.
`UARTParity_t`	`parity`	UART parity mode. See Table 48 for details.
`UARTStopBits_t`	`stopbits`	UART stop bits number. See Table 49 for details.

Table 46. UARTBaudRate_t type

Value	Description
`UART_BAUDRATE_1200`	1200 baud rate
`UART_BAUDRATE_9600`	9600 baud rate
`UART_BAUDRATE_38400`	38400 baud rate

Table 47. UARTData_t type

Value	Description
`UART_DATA5`	5 bits data length

Value	Description
UART_DATA6	6 bits data length
UART_DATA7	7 bits data length
UART_DATA8	8 bits data length

Table 48. UARTParity_t type

Value	Description
UART_PARITY_NONE	Non parity mode
UART_PARITY_EVEN	Even parity mode
UART_PARITY_ODD	Odd parity mode

Table 49. UARTStopBits_t type

Value	Description
UART_STOPBITS1	1 stop bits mode
UART_STOPBITS2	2 stop bits mode

Table 50. UARTHardwareControl_t type

Value	Description
TERMINAL_READY	Host is ready to receive data
TERMINAL_BUSY	Host is busy

5.3.7. SPI Interface

SPI interface is intended for SPI communication (master mode) using USART (`USART_CHANNEL0` can also work in UART mode) in SPI mode.

The SPI interface functions are described in Table 51. They are defined in `"spi.h"` header they are and implemented in `"spi.c"` module.

NOTE:

If several devices have to share the SPI, chip select management should be done via GPIO pins.

Table 51. SPI interface functions

Function	Description
`result_t spi_open(const SPIMode_t *param)`	Open USART as SPI channel `SPIMode_t` structure is described in Table 52. Returns `FAIL` if USART has been already opened (either as SPI or as UART); or `SUCCESS`, otherwise.

Function	Description
`result_t spi_close(void)`	Close USART that was opened before as SPI channel `FAIL` is returned if USART has not been opened yet (as either SPI or UART) or has been already opened in UART mode. Otherwise, it returns `SUCCESS`.
`result_t spi_readWrite(uint8_t *data, uint8_t length)`	Write the specified data to transmit over SPI. At the same time, function also reads the received data into the same buffer `data` – data buffer pointer `length` – the buffer length in bytes After operation is completed, the buffer contains the received bytes, `length` is the number of bytes actually received, and return code is `SUCCESS`. `FAIL` is returned if: • if USART has not been yet opened as SPI • previous write operation was not completed yet; or `SUCCESS`, otherwise.

NOTE:

The `spi_readWrite` operation works in a poll way and does not use interrupts.

Table 52. SPIMode_t struct

Type	Name	Description
`SPIClockMode_t`	`clockMode`	SPI clock mode. See Table 53 for details.
`SPIClockRate_t`	`clockRate`	SPI clock rate. See Table 54 for details.
`SPIDataOrder_t`	`dataOrder`	SPI data order. See Table 55 for details.

Table 53. SPIClockMode_t type

Value	Description
`SPI_CLOCK_MODE0`	leading edge – sample RX bit (rising) trailing edge – setup TX bit (falling)
`SPI_CLOCK_MODE1`	leading edge – setup TX bit (rising) trailing edge – sample RX bit (falling)

Value	Description
SPI_CLOCK_MODE2	leading edge – sample RX bit (falling) trailing edge – setup TX bit (rising)
SPI_CLOCK_MODE3	leading edge – setup TX bit (falling) trailing edge – sample RX bit (rising)

Table 54. SPIClockRate_t type: clock rate

Value	Description
SPI_CLOCK_RATE_2000	2Mb/s
SPI_CLOCK_RATE_1000	1Mb/s
SPI_CLOCK_RATE_500	500Kb/s
SPI_CLOCK_RATE_250	250Kb/s
SPI_CLOCK_RATE_125	125Kb/s

Table 55. SPIDataOrder_t type

Value	Description
SPI_DATA_LSB_FIRST	Data with LSB first
SPI_DATA_MSB_FIRST	Data with MSB first

5.3.8. IAppTimer Interface

IAppTimer is an interface to the flexible user-defined timers.

The IAppTimer functions are described in Table 56. They are defined in "apptimer.h" header, and they are implemented in "apptimer.c" module.

The number of timers is defined as APP_NUM_TIMERS in "apptimer.h" header and it can be changed by user. But it cannot be less than APP_MIN_NUM_TIMERS.

NOTE:

Formally, the resolution is about 1 msec, but this timer cannot provide very accurate timing because events will be raised in the non-interrupt context. If some process requires accurate timing, user application should use one of ATmega1281V spare timers. In spite of this fact, using of IAppTimer interface is preferable to avoid networking problems.

Table 56. IAppTimer interface functions

Function	Description
`int appTimer_open(void (*fired)(void))`	Register handler for application timer `fired` event `fired` – pointer to a fired event handler Returns positive descriptor if the registration is successful, or negative value, otherwise.
`result_t appTimer_start(int id, TimerMode_t mode, uint32_t delay)`	Start timer `id` – descriptor `mode` – application timer mode, see Table 57 `delay` – delay in milliseconds Returns `FAIL` if there is no such descriptor or `SUCCESS`, otherwise.
`result_t appTimer_stop(int id)`	Stop application timer `id` – descriptor Returns `FAIL` if there is no such descriptor or `SUCCESS`, otherwise.
`result_t appTimer_close(int id)`	Cancel handler that was associated with `id` descriptor Returns `FAIL` if there is no such descriptor or `SUCCESS`, otherwise.

Table 57. TimerMode_t type

Value	Description
`TIMER_REPEAT_MODE`	Timer works in continuous mode.
`TIMER_ONE_SHOT_MODE`	Timer will count only one time.

6. MeshBean Drivers

6.1. Function Summary

This set of interfaces is intended to support the MeshBean2 board. It is not a part of user API, but it can be reused by user's application as samples.

These interfaces support:

- DIP-switches reading
- LEDs manipulation
- buttons
- temperature sensor management and data reading
- light sensor management and data reading
- battery voltage measurement.

These interfaces are declared in the following header files: `"sliders.h"`, `"leds.h"`, `"buttons.h"`, `"sensors.h"`.

Table 58. MeshBean2 driver functions

Function	Description	Ref.
`sliders_read`	read DIP switches	6.2.1
`leds_open`	enable LED control	6.2.2
`leds_on`	turn the LED on	6.2.2
`leds_off`	turn the LED off	6.2.2
`leds_toggle`	toggle the LED	6.2.2
`leds_close`	disable LED control	6.2.2
`buttons_open`	register handlers for buttons events	6.2.3
`buttons_close`	close the handlers	6.2.3
`buttons_readState`	read buttons state	6.2.3
`sensor_open`	open on-board sensors	6.2.4
`sensor_close`	close on-board sensors	6.2.4
`sensor_getData`	get sensor's data	6.2.4

6.2. Detailed description

6.2.1. DIP-switches

DIP-switches interface is described in Table 59. The interface functions are defined in "sliders.h" header, and they are implemented in "sliders.c" module.

Table 59. DIP-switches interface functions

Function	Description
`uint8_t sliders_read(void)`	Return state of 3 on-board DIP-switches.

6.2.2. LEDs

The LED interface functions are described in Table 60. They are defined in "leds.h" header, and they are implemented in "leds.c" module.

Table 60. LED interface functions

Function	Description
`void leds_open(void)`	Switch on the peripheral supporting LEDs control, set up the corresponding MCU pins to outputs.
`void leds_on(uint8_t n)`	Turn on the `n`-th LED.
`void leds_off(uint8_t n)`	Turn off the `n`-th LED.
`void leds_toggle(uint8_t n)`	Change the `n`-th LED state to opposite.
`void leds_close(void)`	Switch off the peripheral, supporting LEDs control, set up the corresponding MCU pins to tri-state.

6.2.3. Buttons

The buttons interface functions are described in Table 61. They are defined in "buttons.h" header and they are implemented in "buttons.c" module.

Table 61. Buttons interface functions

Function	Description
`result_t buttons_open(void (*pressed)(uint8_t bn), void (*released)( uint8_t bn))`	Register handlers for button events, that will be called in non-interrupt context Returns `FAIL` if an open request has been already issued or `SUCCESS`, otherwise. `pressed` – handler to process pressing the button `released` – handler to process releasing the button `bn` – button number. Button identification can be performed by `BUTTON_1` macro and `BUTTON_2` macro.
`result_t buttons_close(void)`	Cancel buttons handlers Returns `FAIL` if there was no open issue or `SUCCESS`, otherwise.
`uint8_t buttons_readState(void)`	Returns buttons current state in binary form: bit 0 defines the state of the button 1, bit 1 defines the state of the button 2. `buttons_open()` should be called before using this function.

6.2.4. Sensors

The sensor interface functions are described in Table 62. They are defined in `"sensors.h"` header, and they are implemented in `"sensors.c"` module.

HAL supports light, temperature sensors and battery voltage as a sensor. Sensor type is defined by `id`. There are the following ids:

- `SENSOR_LIGHT`
- `SENSOR_TEMPERATURE`
- `SENSOR_BATTERY`

Sensors cannot be concurrently accessed. They should be processed in series.

Table 62. Sensors interface functions

Function	Description
`result_t sensor_open(uint8_t id)`	Open `id`-th sensor Returns `FAIL` if there is a hardware error encountered or there is no such sensor; or `SUCCESS`, otherwise.

Function	Description
`result_t sensor_close(uint8_t id)`	Closes `id`-th sensor. Returns `FAIL` if there is a hardware error encountered or there is no such sensor; or `SUCCESS`, otherwise.
`result_t sensor_getData(uint8_t id, void (*dataReady)(bool error, float data))`	Get data from `id`-th sensor `dataReady` – pointer to sensor data handler `error` – hardware error flag `data` – sensor data Returns `FAIL` if there is no such sensor or the recent request has not been finished yet; or `SUCCESS`, otherwise.

7. Application Development

7.1. Directory Structure

The directories containing the files required for application development are listed in Table 63.

Table 63. The eZeeNet Software files used for application development

Directory	Description
./API/Framework ./API/Stack ./API/StackSupport	Header and library files for eZeeNet Framework, Stack and Stack Support
./API/HAL/HAL_R5	Source, header and library files for eZeeNet HAL
./API/TOSLib	Header and library files for TOS
./API/SampleApplication/WSNDemo	Source and project files for WSN Demo application
./API/SampleApplication/Blink	Source, project and image files for Blink minimum application
./API/SampleApplication/lowpower	Source, project and image files for Low Power application
./API/SampleApplication/peer2peer	Source, project and image files for Peer-to-Peer application
./API/SampleApplication/pingpong	Source, project and image files for Ping -Pong application

7.2. Building Custom Application

7.2.1. Target Environment and Tools

Before building custom applications and running them user should become aware of the minimum system requirements and additional tools for application development (see Table 64).

Table 64. System requirements and development tools

Parameter	Value
CPU	Intel Pentium III or higher, 800 MHz
RAM	128 MB
Hard disk free space	50 MB
Operating system	Windows2000/XP
IDE	AVR Studio 4.12 + Service Pack 2 + AVR GCC plug-in
Development tools	AVR GCC compiler
JTAG emulator	AVR JTAGICE mkII

The Atmel's AVR Studio [9] is available at the Atmel's website http://www.atmel.com.

AVR GCC compiler is available within the WinAVR suite of software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform [8]. For the description of GCC compiler, and particularly the command options for compilation and linking, see WinAVR documentation [3].

JTAG emulator is intended to download the application firmware into a board [10].

7.2.2. AVR Studio Projects

In AVR Studio, creating an application is organized under particular project. All the necessary information about a project is kept in project file which has an `*.aps` extension so it opens automatically if double-clicked.

7.2.3. Building Procedure for Making an Application

The easiest way to configure an AVR Studio project is using makefile that is a plain text file which name is just `makefile` without extension. Any makefile specifies the compilation flags and the linking flags. To include header files and to link libraries, a makefile specifies the corresponding directories.

A sample makefile is presented in Section 7.2.5.

7.2.4. Clock sources

There are two clock sources that can be used for application:

- Internal RC-oscillator
- External clock source.

To use the internal RC-oscillator select the `ZigBitInt` library specified in Makefile building the application. Opposite, to use the external clock source select the `ZigBitExt` library.

To use the internal RC-oscillator, fuse bits should be set as follows Check ON the following options in Fuses Tab before downloading the images through JTAG:

```
Brown-out detection disabled; [BODLEVEL=111]

JTAG Interface Enabled; [JTAGEN=0]

Serial program downloading (SPI) enabled; [SPIEN=0]

Boot Flash section size=1024 words Boot start
address=$FE00;[BOOTSZ=10]

Divide clock by 8 internally; [CKDIV8=0]

Int. RC Osc.; Start-up time: 6 CK + 65 ms; [CKSEL=0010
SUT=01]
```

Uncheck the rest of options. Make sure the following hex value string appears at the bottom of Fuses Tab: `0xFF, 0x9D, 0x62.`

`Additionally check ON the following option` if the nodes will be programmed `with Serial Bootloder:`

```
Boot Reset vector Enabled (default address=$0000);
[BOOTRST=0]
```

Make sure the following hex value string appears at the bottom of Fuses Tab: `0xFF, 0x9C, 0x62.`

7.2.5. Minimum Application

This demo application implements the MeshBean2' LED blinking. In user's loop the `GPIO-0` pin state is toggled, using the GPIO interface. The program is coded in C, it is named `blink.c`. The source code is shown below.

```
/*****************************************************
  LED Blinking Implementation Project: C source/
#include "framework.h"
#include "gpio.h"
#include "apptimer.h"
#define LED GPIO_0 // Pin connected to LED.
// Functions' declarations.
void mainLoop();   // Main loop.
void timerFired(); // Blink timer handler.
/*****************************************************
  Users entry.
*****************************************************/
void fw_userEntry(FW_ResetReason_t resetReason)
{
  // Initialize the pin connected to the LED.
  gpio_setConfig(LED, GPIO_OUTPUT);
```

```
  // Open and start blink timer.
  {
    int handle;
    handle = appTimer_open(timerFired);
    appTimer_start(handle, TIMER_REPEAT_MODE, 1000);
  }
  // Start main loop.
  fw_setUserLoop(20, mainLoop);
}
/*********************************************************
  Main loop.
*********************************************************/
void mainLoop()
{
  // Addtional user's activities.
}
/*********************************************************
  Blink timer handler.
*********************************************************/
void timerFired()
{
  static bool on=0;
  gpio_setState(LED, on);
  on = on ? 0:1; // Toggle.
}
// eof blink.c
```

The makefile intended to build this minimum application is given below. It refers the software files as they are structured in Table 63 so that the structure should be kept unchanged when copied to the user's hard drive. Also, for correct referencing in building multiple applications, the `blink` .aps project file, the makefile and the `blink` .c file all should be located in the same directory (`blink`) which, in turn, should be placed to the same location where the `"API"` directory was copied. For example, the `".\SampleApplication"` directory which may encompass different application projects can be set next to `".\API"` directory.

```
############################################################
# LED Blinking Implementation Project: Makefile
############################################################
## General Flags
PROJECT = blink
MCU = atmega1281
TARGET = blink.elf
CC = avr-gcc

## Options common to compile, link and assembly rules
COMMON = -mmcu=$(MCU)
```

```
## Compile options common for all C compilation units.
CFLAGS = $(COMMON)
CFLAGS += -Wall -D_WDM1281_ -Os -g -fsigned-char

## Assembly specific flags
ASMFLAGS = $(COMMON)
ASMFLAGS += -x assembler-with-cpp -Wa,-gdwarf2

## Linker flags
LDFLAGS = $(COMMON)

## Intel Hex file production flags
HEX_FLASH_FLAGS = -R .eeprom

HEX_EEPROM_FLAGS = -j .eeprom
HEX_EEPROM_FLAGS += --set-section-
flags=.eeprom="alloc,load"
HEX_EEPROM_FLAGS += --change-section-lma .eeprom=0

## Framework defines.

## Path to Stack, StackSupport, HAL, TOSLib
SUPPORT_DIR = ../..
#SUPPORT_DIR = ../../API

## Modules directories paths.
APP_DIR = ./
STACK_DIR         = $(SUPPORT_DIR)/Stack
STACK_SUPPORT_DIR = $(SUPPORT_DIR)/StackSupport
TOSLIB_DIR        = $(SUPPORT_DIR)/TOSLib
HAL_DIR           = $(SUPPORT_DIR)/HAL/HAL_R5
FRAMEWORK_DIR     = $(SUPPORT_DIR)/Framework

## Include Directories.
INCLUDES = -I"$(APP_DIR)/include"                  \
           -I"$(STACK_DIR)/include"                \
           -I"$(TOSLIB_DIR)/include"               \
           -I"$(HAL_DIR)/base/include"             \
           -I"$(HAL_DIR)/eZeeNet/include"          \
           -I"$(STACK_SUPPORT_DIR)/include"        \
           -I"$(STACK_SUPPORT_DIR)/include/stack" \
           -I"$(FRAMEWORK_DIR)/include"

## Library Directories
LIBDIRS = -L"$(HAL_DIR)/lib"       \
          -L"$(TOSLIB_DIR)/lib"    \
          -L"$(FRAMEWORK_DIR)/lib" \
```

```
          -L"$(STACK_SUPPORT_DIR)/lib"

## Libraries
LIBS = -ltos          \
       -lFW           \
       -lZigBitInt    \
       -lstackSupport \
       -lc

SRC = $(APP_DIR)/blink.c \
      $(STACK_SUPPORT_DIR)/src/ConfigServer.c \

## Objects explicitly added by the user
LINKONLYOBJECTS = $(STACK_DIR)/lib/NWKMACLibA.o \
                  $(STACK_DIR)/lib/APLLibA.o    \
                  $(HAL_DIR)/lib/wdtinit.o

## Build
all: $(TARGET) blink.srec blink.hex

##Link
$(TARGET): $(SRC)
  $(CC) $(CFLAGS) $(INCLUDES) $(LINKONLYOBJECTS) $(SRC)
-o $(TARGET) $(LIBDIRS) $(LIBS)

%.srec: $(TARGET)
  avr-objcopy -O srec $(HEX_FLASH_FLAGS)  $< $@
%.hex: $(TARGET)
  avr-objcopy -O ihex $(HEX_FLASH_FLAGS)  $< $@
%.lss: $(TARGET)
  avr-objdump -h -S $< > $@

## Clean target
clean:
  -rm -rf $(TARGET) $(PROJECT).srec $(PROJECT).hex
```

Open the `blink.aps` file from the "`.\SampleApplication\Blink`" subdirectory on your hard drive and just execute `Build\Rebuild All` item from the main menu. The `blink.hex` and `blink.srec` image files will be generated there. To get it tested the minimum application should be downloaded into a board.

7.3. Sample Applications

Other examples are provided with the eZeeNet Software API to demonstrate how to develop user's applications. The scenarios of sample applications are presented below. They can be implemented using the ZigBit Development Kit (ZDK) [7].

There are four sample applications:

- Low Power
- Peer-to-peer data exchange
- Ping pong
- WSN Demo.

The applications source code can be found in `./SampleApplication/` subdirectories. In each Makefile, a set of network parameters is defined.

For all the sample applications, fuse bits should give logical value of `FF 9C 62`.

7.3.1. Low-Power Networking Application

This sample shows how to collect data transmitted from low-power devices, employing the simplest power management strategy.

At least 2 nodes are participating, coordinator and end-device, but up to 7 end-devices can be engaged.

For simple node identification, define uniquely logical addresses by DIP-switches. When setting 8 possible DIP combinations take in mind that:

- ON position corresponds to logical 1
- the third DIP switch defines the most significant bit for logical address
- the only coordinator must have zero logical address.

To start the application and to initiate the network push `SW1` button on any node, starting from coordinator. Red LED is getting ON while the network is started successfully. If green LED is getting ON, it means that MAC address is set incorrect, namely it is equal to 0.

Coordinator organizes the network with its own 'unique' PAN ID which is determined by its MAC address (considering the 16 least significant bits). Besides, user can set PAN ID in flash memory or EEPROM. In order to join end-devices scan the network.

End-device measures temperature each 10 seconds and sends data to coordinator if the measurement absolute increment exceeds 0.5°C. Sending data is indicated by yellow LED flashing. On having data sent, an end device falls into sleep. Unconditionally, the current temperature measurement will be also sent if `SW2` button is pressed on end-device, regardless of the node being sleeping before.

Coordinator never sleeps; it keeps sending the received temperature data to UART via USB at 9600 bps rate in text form, without flow control.

To help understanding of the application code easier, two different image files are supplied: one is intended for coordinator, the other – for end-device.

7.3.2. Peer-to-peer Data Exchange Application

This sample shows how to organize the simplest peer-to-peer link. A simple buffering strategy is employed to avoid byte-by-byte data transfer.

At least 2 nodes are participating. Up to 7 nodes can be engaged. In order to define the node logical addresses set DIP switches as described in Section 7.3.1. All of the nodes, which are connected over UART to their own USB, simply exchange over UART with data which is input at their hosts. It is strictly defined that the nodes should communicate in pairs identified corresponding to the following logical addresses:

- node 0 with node 1
- node 2 with node 3
- node 4 with node 5
- node 6 with node 7.

The connection is configured with 9600 bps rate, flow control set on, number of data bits equal to 8, parity – none, and number of stop bits equal to 1.

To organize network pressing `SW1` button on any node, starting from coordinator. LEDs indicate network activity as follows:

- if MAC address is set incorrect, that is equal to 0, yellow LED is getting ON
- when a node is searching for the network red LED is blinking
- red LED is ON when the node joins the network successfully
- green LED is flashing when a node sends data
- yellow LED flashing indicates receiving data.

Initially, all the nodes are configured as routers, so they can route the data from each other, thus extending the links.

7.3.3. Ping-Pong Application

This example shows how to make the nodes to process multiple data requests. Each node expects receiving a message wirelessly, and it passes each of the received messages to the next node. Up to 4 nodes can be engaged forming a circle.

The node logical addresses are defined by two DIP-switches, numbered 1 and 2, which can be set in 4 possible combinations. The third DIP switch turned ON determines the node closing the configured chain top a circle. In configurations with 2, 3 or 4 nodes, it should be set to ON position for the 2nd, for the 3rd or for the 4th node, respectively. DIP-switch configurations are presented as examples in Table 65, Table 66, and Table 67.

Network formation is initiated by pressing `SW1` button on each of the nodes. When network is started successfully, red LED is ON. If green LED is getting ON, it means that MAC address is set incorrect, that is equal to 0. Sending data is accompanied by blinking yellow LED (LED is getting on, when data is sent, or off, when it is acknowledged).

Sending data is initiated by pressing `SW2` button on some of the nodes. Once the third DIP switch on the chain-closing node is in ON position, data will be passed over circularly: 0→1→2→3→0→1... and so forth. You can press `SW2` button on some other node one more time, and then the next data block will pass through the nodes in same direction. If data block fails to be delivered during any transaction (due to communication error, router's overload or some other reason) it gets out of data circulation in the network.

Table 65. DIP-switches in Ping-Pong Application for 2 nodes participating

Node	DIP switches		
	SW4:1	SW4:2	SW4:3
0	OFF	OFF	OFF
1	ON	OFF	ON

Table 66. DIP switches in Ping-Pong Application for 3 nodes participating

Node	DIP switches		
	SW4:1	SW4:2	SW4:3
0	OFF	OFF	OFF
1	ON	OFF	OFF
2	OFF	ON	ON

Table 67. DIP switches in Ping-Pong Application for 4 nodes participating

Node	DIP switches		
	SW4:1	SW4:2	SW4:3
0	OFF	OFF	OFF
1	ON	OFF	OFF
2	OFF	ON	OFF
3	ON	ON	ON

7.3.4. WSN Demo Application

WSN Demo application is intended to demonstrate the WSN full functional operation. End devices and routers update in duty circle the on-board sensor readings (temperature, light and battery level measurements) and send them in packets to coordinator. The data received from WSN nodes displayed with PC GUI application. The WSN demo application is described in details in [7].