APPLICATION NOTE

# Atmel AVR2130: Lightweight Mesh Developer Guide

## 8-bit Atmel Microcontrollers

## Features

- Atmel® Lightweight Mesh stack specification and APIs
- Lightweight Mesh Software Development Kit (SDK)

## Description

This document describes the specification for Lightweight Mesh – the easy to use proprietary low power wireless mesh network protocol from Atmel. This document can be considered a full and complete specification of the protocol and related APIs. This document also describes a reference implementation of the protocol.

The Lightweight Mesh target audience is system designers, embedded programmers, and hardware engineers, evaluating, prototyping, and deploying wireless solutions and products. The Lightweight Mesh stack is delivered as a software development kit, which includes complete source code for the stack components, as well as sample applications.

The audience is assumed to be familiar with the C programming language. Some knowledge of embedded systems is recommended, but not required.

## Table of Contents

# 1. Introduction

## 1.1 Target applications

Lightweight Mesh was designed to address the needs of a wide range of wireless connectivity applications. Some of these applications include:

- Remote control
- Alarms and security
- Automatic Meter Reading (AMR)
- Home and commercial building automation
- Toys and educational equipment

## 1.2 Hardware requirements

Lightweight Mesh is designed to work with all Atmel IEEE® 802.15.4 transceivers and SoCs. Currently the stack works with AVR®-based MCUs, but given extreme portability and low resource requirements, it can be run on almost any Atmel MCU.

Currently applications include project files for the following platforms:

- ZigBit® modules (ATZB-24-B0, ATZB-24-A2)
- STK®600-mega128rfa1 board (ATAVR128RFA1-EK1)
- XMEGA®-B1 Xplained and RZ00 radio modules (ATXMEGAB1-XPLD and ATAVRRZ600)
- RCB128RFA1 board
- RCB231 board

## 1.3 Abbreviations and terminology

| | | |
|---|---|---|
| API | – | Application Programming Interface |
| APL | – | Application layer |
| Device, Node | – | Physical device acting as a part of the network |
| GPIO | – | General Purpose Input/Output |
| LQI | – | Link Quality Indicator |
| MAC | – | Medium Access Control |
| MCU | – | Microcontroller Unit |
| MIC | – | Message Integrity Code |
| NWK | – | Network layer |
| PAN | – | Personal Area Network |
| PHY | – | Physical layer |
| RAM | – | Random Access Memory |
| RSSI | – | Received Signal Strength Indicator |
| SDK | – | Software Development Kit |
| SoC | – | System-on-Chip |
| User, Customer | – | Entity using Lightweight Mesh in a final product |

# 2. Lightweight Mesh protocol overview

## 2.1 Features

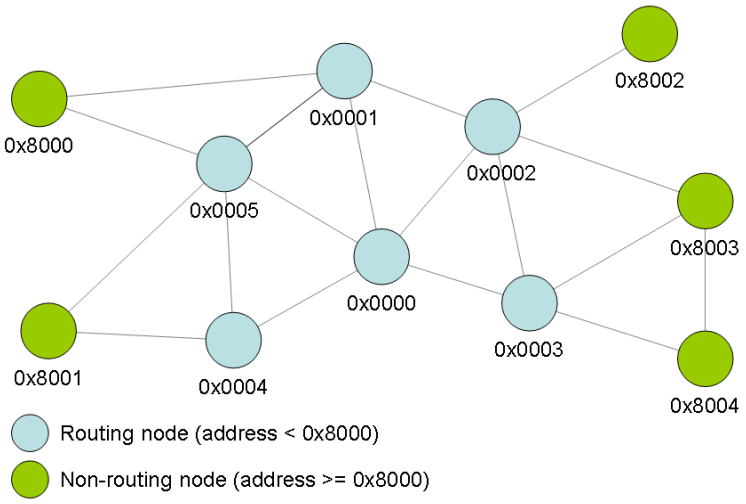
The current implementation of the Lightweight Mesh protocol has the following features:

- Simplicity of configuration and use
- Up to 65535 nodes in one network (theoretical limit)
- Up to 65535 separate PANs on one channel
- Up to 15 independent application endpoints
- No dedicated node is required to start a network
- No periodic service traffic occupying bandwidth
- Two distinct types of nodes:
  - Routing (network address < 0x8000)
  - Non-routing (network address >= 0x8000)
- Once powered on node is ready to send and receive data; no special joining procedure is required
- No child-parent relationship between the nodes
- Non-routing nodes can send and receive data to/from any other node (including non-routing nodes), but they will never be used for routing purposes
- Route discovery happens automatically if route to the destination is not known
- Routing table is updated automatically based on the data from the received and transmitted frames
- Duplicate frames (broadcast or multipath unicast) are rejected
- Small footprint (less than 8kB of Flash and 4kB of RAM for a typical application)

## 2.2 Network topology

Network topology and possible device types are illustrated by Figure 2-1. Nodes shown in blue are routing nodes; they form a core of the typical network and expected to be mains-powered. Nodes shown in green are non-routing nodes; they are part of the network and they can send and receive data as long as they are in range and have the radio turned on, but they are not expected to be available all the time (they can be sleeping nodes, mobile nodes going out of range, etc). Non-routing nodes will not be used for routing purposes, so they cannot act as range extenders, and typically will be located at the edge of the network.

**Figure 2-1. Network topology and device types.**




## 2.3 Provided services

Lightweight Mesh provides the following core services:

- Basic data services (send and receive data)
- Acknowledgments
- Routing
- Basic security
- Power management of the radio transceiver
- Interface to the advanced features of the radio transceiver (encryption, energy detection, random number generation, etc)

An application running Lightweight Mesh is responsible for:

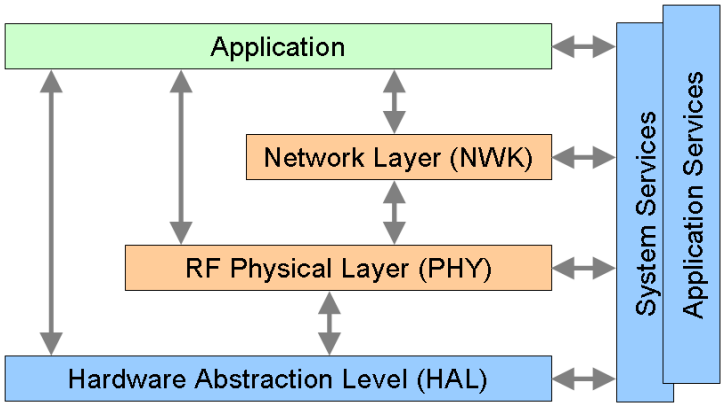
- Network management (discovery, joining, commissioning, etc)
- Advanced network operation scenarios (sleeping routers, parent-child relationship, data delivery to the sleeping nodes, etc)
- Retries to send data in case of failures
- Defining message payload format
- Advanced security
- Power management of the MCU
- Interfacing hardware peripherals (ADC, PWM, EEPROM, etc)

# 3. Lightweight Mesh architecture

## 3.1 Architecture highlights

High level architecture of the Lightweight Mesh stack is presented on the Figure 3-1. Lightweight Mesh code is separated into a number of logical levels each providing a set of APIs accessible for the user. The Stack is designed to provide only functions absolutely necessary for wireless communication, and it is expected that the rest will be created by the user, or provided by third-party libraries, if required.

**Figure 3-1. Lightweight Mesh architecture.**




- Hardware abstraction layer (HAL) provides basic hardware dependent functionality, like hardware timer, sleep control, GPIO access for the radio interface
- Radio physical layer (PHY) provides functions for radio transceiver access. Some of them are accessible only by the network layer (request to send data, data indication); some of them can be used from the application (channel selection, random number generation, energy detection, etc.)
- Network layer (NWK) provides core stack functionality, which is described in detail throughout this document
- System services provide common functions for all layers, which are necessary for normal stack operation. System services include basic types and definitions, software timers, default configuration parameters, encryption module access, etc.
- Application services include modules that are not required by the stack, but are common for most applications. Currently the only service of this type is over-the-air upgrade (OTA). Application services are out of the scope of this document

## 3.2 Naming conventions

Lightweight Mesh uses a well defined set of naming conventions that make it very easy to read the source code and reduce application development time. Here are the basic rules:

- Each API function name is prefixed by the name of the layer where the function resides. For example, the *NWK_SetAddr()* function is contributed by the network layer of the stack
- Each function name prefix is followed by an underscore character separating the prefix from the descriptive function name
- The descriptive function name may have a *Req*, *Ind* or *Conf* suffix, indicating the following:
  - *Req* corresponds to the requests from the user application to the stack (for example, *NWK_DataReq()*)
  - *Ind* corresponds to the asynchronous indication of events propagated to the user application from the stack (for example, *NWK_DataInd()*)

- *Conf* corresponds to the user-defined callbacks for the asynchronous requests, which confirm the request's execution

- Each structure and type name carries a *_t* suffix, standing for type
- Enumeration and macro variable names are in capital letters

It is recommended that the application developer adhere to the aforementioned naming conventions in the user application.

## 3.3 File system layout

The file system layout of the SDK closely reflects the architecture of the Lightweight Mesh stack.

**Table 3-1. Lightweight Mesh SDK file system layout.**

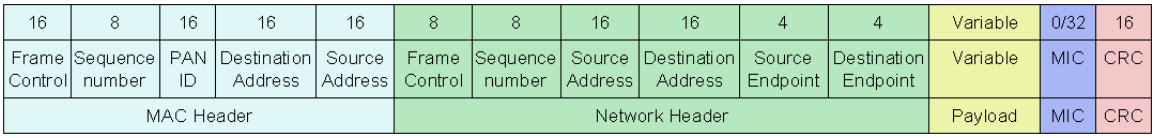
| Directory | Table Text |
|---|---|
| apps | Sample applications |
| doc | Documentation |
| hal | Hardware abstraction layer |
| nwk | Network layer |
| phy | Radio physical layer |
| service | Application services |
| sys | System services |
| tools | Supporting tools and PC applications |

# 4. Network layer specification

## 4.1 General Lightweight Mesh frame format

The Lightweight Mesh network header and application payload are encapsulated inside the standard IEEE 805.15.4 data frame payload (see [1]), but the stack itself does not adhere to the standard, so it will not receive and process IEEE 805.15.4 command frames. Figure 4-1 illustrates a general frame format composed of an IEEE 805.15.4 MAC header, network header, application payload, optional message integrity code (MIC) and a check sum (CRC).

**Figure 4-1. General Lightweight Mesh frame format.**




The following settings are used for a MAC Frame Control Field as described in [1]:

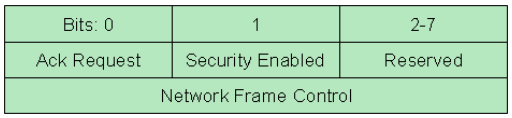
- Frame Type: Data
- Security Enabled: False
- Frame Pending: False
- Acknowledgment Request: True for unicast frames and False for broadcast frames
- PAN ID Compression: True
- Destination Addressing Mode: 16-bit short address
- Frame Version: 0
- Source Addressing Mode: 16-bit short address

This gives two possible values for the MAC Frame Control Field that can be generated and recognized by the stack: 0x8841 for broadcast frames, and 0x8861 for unicast frames.

### 4.1.1 Frame Control field

The Frame Control field is 1 byte in length and contains control information for the frame. The Frame Control field shall be formatted as illustrated in the Figure 4-2.

**Figure 4-2. General Lightweight Mesh frame Control Field format.**




#### 4.1.1.1 Acknowledgment Request subfield

The Acknowledgment Request subfield is 1 bit in length and specifies whether an acknowledgment is required from the destination node. If this subfield is set to one, the destination node shall send an acknowledgment command if frame was received and processed. If this subfield is set to zero, the destination device shall not send an acknowledgment frame (for exceptions see Section 4.3).

#### 4.1.1.2 Security Enabled subfield

The Security Enabled subfield is 1 bit in length, and it shall be set to one if the frame payload is encrypted, and shall be set to zero otherwise. Message Integrity Code (MIC) should be present if the Security Enabled subfield is set to one.

#### 4.1.1.3 Reserved subfield

The Reserved subfield is 6 bits in length, and it shall be set to 0.

### 4.1.2 Sequence Number field

The Sequence Number field is 1 byte in length and specifies the sequence identifier for the frame. The Sequence Number field shall be increased by 1 for every outgoing frame.

### 4.1.3 Source Address field

The Source Address field is 2 bytes in length and specifies the network address of the node originating the frame.

### 4.1.4 Destination Address field

The Destination Address field is 2 bytes in length and specifies the network address of the destination node. The Destination Address field can be set to 0xffff for broadcast frames.

### 4.1.5 Source Endpoint field

The Source Endpoint field is 4 bits in length and specifies the source endpoint identifier. Value of 0 is reserved for a stack command endpoint.

### 4.1.6 Destination Endpoint field

The Destination Endpoint field is 4 bits in length and specifies the destination endpoint identifier. Value of 0 is reserved for a stack command endpoint.

## 4.2 Format of individual command frames

### 4.2.1 Acknowledgment Command frame format

Figure 4-3 illustrates the Acknowledgment Command frame format.

**Figure 4-3. Acknowledgment Command frame format.**

| Bytes: 1 | 1 | 1 |
|---|---|---|
| Command ID (0x00) | Sequence Number | Control Message |

#### 4.2.1.2 Command ID field

The Command ID field is 1 byte in length, and it contains a constant value (0x00).

#### 4.2.1.3 Sequence Number field

The Sequence Number field is 1 byte in length, and it contains a network sequence number of a frame that is being acknowledged.

#### 4.2.1.4 Control Message field

The Control Message field is 1 byte in length, and it contains an arbitrary value that can be set on the sending side. This filed can be used to provide additional instruction to the receiving side.

### 4.2.2 Route Error Command frame format

Figure 4-4 illustrates Route Error Command frame format.

**Figure 4-4. Route Error Command frame format.**

| Bytes: 1 | 2 | 2 |
|---|---|---|
| Command ID (0x01) | Source Address | Destination Address |

#### 4.2.2.2 Command ID field

The Command ID field is 1 byte in length, and it contains a constant value (0x01).

#### 4.2.2.3 Source Address field

The Source Address field is 2 bytes in length, and it contains a source network address from the frame that cannot be routed.

#### 4.2.2.4 Destination Address field

The Destination Address field is 2 bytes in length, and it contains a destination network address from the frame that cannot be routed.

## 4.3 Routing

### 4.3.1 Overview

The Routing algorithm uses routing tables for its operation. A routing table consists of routing records. Each routing record includes the fields described in the Table 4-1.

**Table 4-1. Routing record fields.**

| Field Name | Description |
|---|---|
| Network destination address | Network address of the last device in the route |
| Next hop MAC address | MAC address of the first node in the route. For direct link this field will be equal to the Network destination address field |
| Score | Record score is used to remove stale records |
| LQI | Link Quality Indicator of the last received frame from the node who's address is in the Next hop MAC address field |

When frames have to be sent:

1. If there is a record in the routing table for the destination network address of the frame, then destination MAC address of the frame is set to the Next Hop field of the corresponding route record.
2. If there is no record in the routing table for the destination network address of the frame, then the destination MAC address of the frame is set to 0xffff (broadcast).

Setting the destination MAC address to the broadcast address, while the destination network address is unicast, initiates route discovery, is described in Section 4.3.2.

A Received frame has to be routed if the MAC destination address is equal to the node's own address and the network destination address is not equal to the node's own address.

When routing is needed, the following conditions are evaluated:

1. If there is a record in the routing table for the destination network address, then the destination MAC address of the frame is replaced by the next hop address from the routing table, and the frame is sent.
2. If there is no record in the routing table for the destination network address, then a Route Error command is sent to the originator of the frame.

### 4.3.2 Route discovery and establishment

In Lightweight Mesh there is no special route discovery procedure; routes are discovered as part of normal data delivery. This way, the penalty of not having a route is very low and comparable to the cost of sending a regular broadcast frame.

Route discovery algorithm is illustrated below. Nodes marked “1”, “2” and “3” are routing nodes. This example makes the following assumptions:

- Node 1 wants to send data to node 3
- Routing tables on all nodes are empty
- There is no direct path between node 1 and node 3

Initial network configuration is shown on the Figure 4-5.

**Figure 4-5. Initial network configuration.**

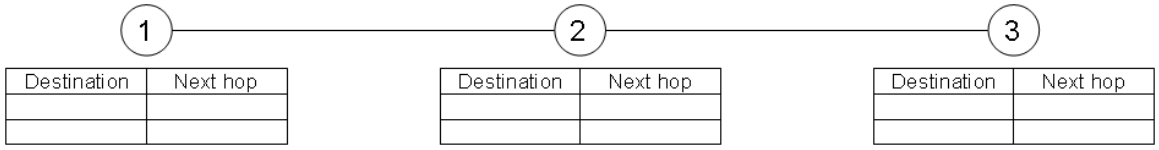
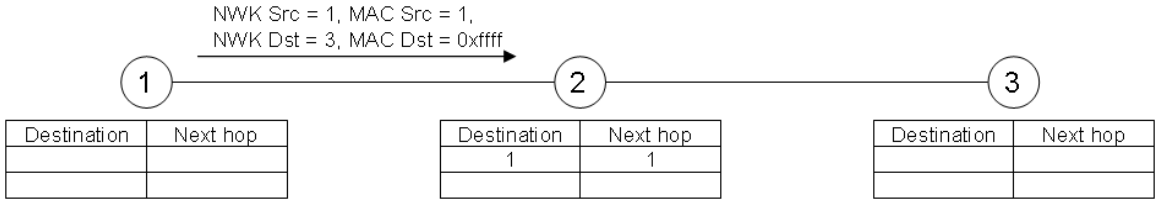
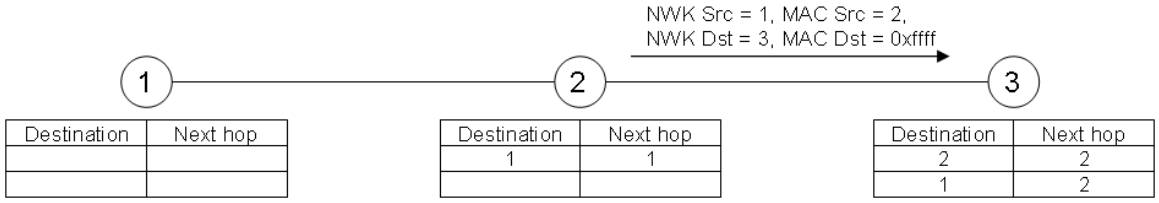



**Figure 4-6. First step of a data transfer involving route discovery.**



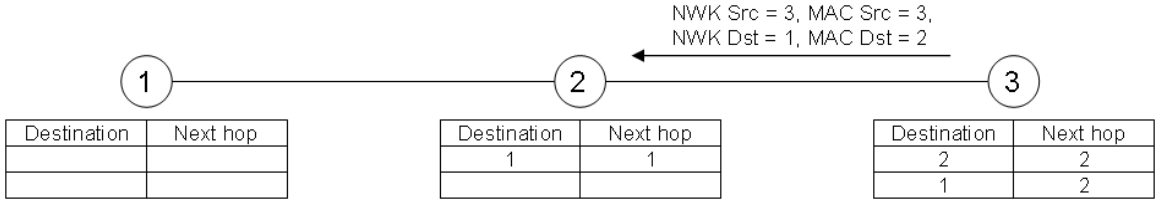

1. Node 1 sends a frame with the Network Destination Address set to 3, and the MAC Destination Address set to 0xffff.
2. Node 2 receives this frame, and adds the record for node 1 to its routing table.

**Figure 4-7. Second step of a data transfer involving route discovery.**



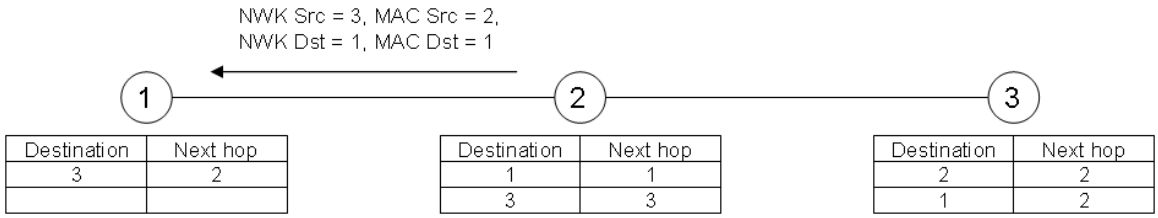

3. Node 2 broadcasts the frame (because MAC Destination Address is set to 0xffff).
4. Node 3 receives this frame, and adds the record for node 2 to its routing table.
5. Node 3 adds record for node 1 (from a Network Source Address).

**Figure 4-8. Third step of a data transfer involving route discovery.**




6. Node 3 handles the frame and sends an Acknowledgment frame, even if one was not requested. This is done to establish a reverse route. Node 3 now knows the route to node 1, so a unicast frame is sent.

**Figure 4-9. Final step of a data transfer involving route discovery.**




7. Node 2 receives the frame and adds the route to node 3 to its routing table.
8. Node 2 has a route record for node 1, so it routes the received frame to its final destination.
9. Node 1 receives the frame and adds the route to node 3 to its routing table.

**Figure 4-10. Final network configuration after a data transfer involving route discovery.**


Now a route between node 1 and node 3 is established and it will be used for the following frames. Note that during route discovery, all nodes along the route learned how to route data to the destination node. Those route records will be used for routing purposes without further route discovery. Eventually, given big enough routing tables, all nodes in the network will discover all possible (used) routes.

Also note that for some nodes discovery of one route created more than one record in a routing table. This is a common property of the routing algorithms and it should be kept in mind when selecting route table sizes for nodes that are expected to route a lot of traffic. At the same time there is an upper bound on the number of records in the routing table (the maximum number of nodes in the network).

### 4.3.3 Route maintenance and utilization

The routing table is updated with every sent or received frame.

When a frame is received, the routing table is updated in the following way:

1. No action is performed if the source MAC address indicates that the sending device is a non-routing node (it cannot be used as part of the route).
2. If there is a record for the source network address, and this record contains a next hop address different from the source MAC address of the received frame, and the LQI of the received frame is better than the LQI in the routing record, then the record is updated to use the source node as the next hop. Record score is set to NWK_ROUTE_DEFAULT_SCORE.
3. If there is no record for the source network address, then a new record is created in the routing table.
4. LQI of the record (existing or newly created) is set to the LQI of the received frame.

When a frame is sent the routing table is updated in the following way:

1. If there is a record for the destination network address, then
    1. If the frame was sent successfully, then the record score is set to NWK_ROUTE_DEFAULT_SCORE.
    2. Otherwise the record score is decreased. If the record score drops to 0, then the record is removed from the table.
2. If there is no record for the destination network address, then no action is performed.

Essentially this algorithm performs local optimizations, so it establishes and maintains routes with the best link quality on each link, but it does not guarantee shortest routes.

### 4.3.4 Route invalidation and removal

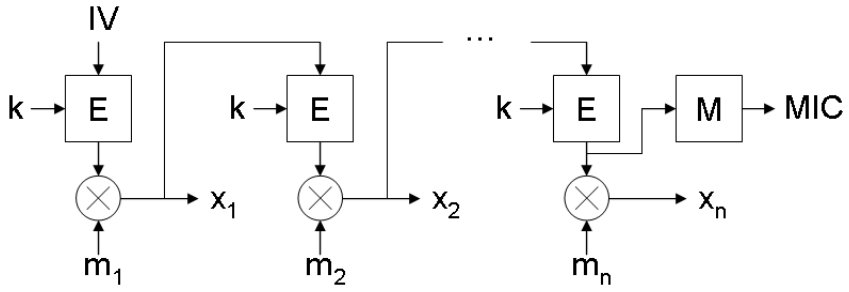
Route records never expire or timeout, but there are a few ways a record can be altered:

1. Route record is replaced by a new record if there are no free records, and a new record has to be placed in the routing table. The oldest and the least active record are replaced first.
2. Route record is removed if its score drops to 0.
3. Route record is removed when a Route Error command, with the Destination Address field equal to the route record destination address, is received.

## 4.4 Security

Lightweight Mesh support two base encryption algorithms: hardware accelerated AES-128 (where supported by the hardware) and software XTEA (all platforms). Both AES and XTEA are block-based algorithms, so messages that exceed the size of the block have to be split into a set of blocks. After splitting and encryption, blocks are chained in order to derive a message integrity code (MIC) that covers the entire message. The chaining method is illustrated in the Figure 4-11. Only the encryption operation is utilized for both encryption and decryption of messages; symmetry of the binary exclusive-or operation is used to achieve this.

**Figure 4-11. Encryption block chaining.**

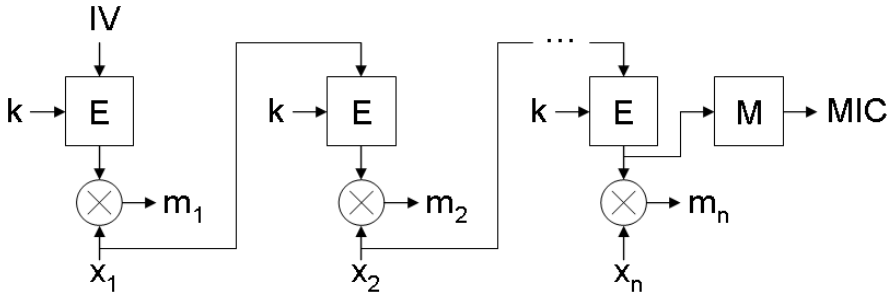



Here:

- *IV* – initialization vector, which is constructed based on the information from the frame header
- *K* – 128-bit key
- *E* – base encryption block (AES or XTEA)
- *M* – message integrity code transform
- *MIC* – message integrity code (32 bits)
- $m_1, m_2, \ldots, m_n$ – 128-bit blocks of the unencrypted message
- $x_1, x_2, \ldots, x_n$ – 128-bit blocks of the encrypted message

Decryption is performed in a similar way. It is illustrated in Figure 4-12.

**Figure 4-12. Decryption block chaining.**




Message integrity code is included in all encrypted frames. It allows verification of the message integrity, validation of the key, and also protection against altering the network header of the message.

Few notes about security:

- There is no protection against replay attacks; it should be implemented on the application layer if required
- The entire network uses the same shared encryption key, so if additional protection is required, it should be implemented on the application layer as well
- Although theoretically AES is stronger than XTEA, for external radio chips, encryption keys are sent in plain text over the SPI bus, so if there is the possibility of physical access to the devices, software XTEA implementation might offer stronger overall protection

# 5. Application programming

## 5.1 Typical application structure

Typical standalone Lightweight Mesh application has a structure as shown below.

Typical standalone Lightweight Mesh application

```
static void APP_TaskHandler(void)
{
  // Put your application code here
}

int main(void)
{
  SYS_Init();

  while (1)
  {
    SYS_TaskHandler();
    APP_TaskHandler();
  }
}
```

On the other hand, if Lightweight Mesh is used inside another environment or task scheduler, it is not necessary to follow this structure. The only requirement is that *SYS_Init()* is called before any other Lightweight Mesh function and *SYS_TaskHandler()* is called as often as possible. See [2] for a description of good, event-driven, application design practices.

## 5.2 Basic network configuration

Below is a list of parameters that should be set in order to send and receive data.

### 5.2.1 Network address

The network address of the node is set via *NWK_SetAddr()* function. Parameter *addr* cannot take value of 0xffff as it is reserved for broadcast frames.

Setting the network address

```
NWK_SetAddr(0x0001);
```

### 5.2.2 Network Identifier

The network identifier (PAN ID) of the node is set via *NWK_SetPanId()* function. Parameter *panId* cannot take value of 0xffff as it is reserved.

Setting the network identifier

```
NWK_SetPanId(0x1234);
```

### 5.2.3 Frequency channel

The frequency channel of the node is set via *PHY_SetChannel()* function. Valid range of values for the *channel* parameter on 2.4 GHz radios is 11 – 26 (0x0b – 0x1a). For sub-GHz radios this parameter represents channel number with a valid range of 0 – 10 (0x00 – 0x0a) if frequency band is set to 0, or frequency index otherwise. Refer to the description of the *CC_BAND* and *CC_NUMBER* settings in [3].

Setting the frequency channel

```
PHY_SetChannel(0x0f);
```

### 5.2.4 Frequency band

The frequency band of the node is set via *PHY_SetBand()* function. This function is only available for sub-GHz radios.

Frequency bands and corresponding frequencies are described in [3].

Setting the frequency band

```
PHY_SetBand(2);
```

### 5.2.5 Modulation mode

The modulation mode of the node is set via *PHY_SetModulation()* function. This function is only available for sub-GHz radios. See the description of *TRX_CTRL_2* register in [3] for details on various modulation modes.

Setting the modulation mode

```
PHY_SetModulation(0x35);
```

### 5.2.6 Receiver state

The transceiver state of the node is set via *PHY_SetRxState()* function.

Setting the transceiver state

```
PHY_SetRxState(true);
```

### 5.2.7 Security key

The security key of the node is set via *NWK_SetSecurityKey()* function. Size of the security key is 16 bytes.

Setting the security key

```
NWK_SetSecurityKey((uint8_t *)"Security12345678");
```

### 5.2.8 Application endpoints

In order to receive data, the application should register a data indication callback associated with the endpoint identifier. *NWK_OpenEndpoint()* function is used to register an endpoint. Endpoint identifier should be less than NWK_MAX_ENDPOINTS_AMOUNT. Section 5.4 further explains how to process received data.

Registering the endpoint indication callback

```
static bool appDataInd(NWK_DataInd_t *ind)
{
  // process the frame
  return true;
}

NWK_OpenEndpoint(1, appDataInd);
```

## 5.3 Sending the data

In order to perform data transmission, the application first needs to create a data transmission request of *NWK_DataReq_t* type that specifies the data payload, data payload size, sets various transmission parameters, and defines the callback function to be executed to inform the application about transmission results. Function *NWK_DataReq()* is used to send the data. Fields of the *NWK_DataReq_t* structure, accessible to the user, are listed in Table 5-1.

**Table 5-1. Network data request parameters.**

| Name | Description |
|---|---|
| dstAddr | Network address of the destination device |
| dstEndpoint | Endpoint number on the destination device |
| srcEndpoint | Local endpoint number |
| options | Data request options. It may be any combination of the following constants (combined using bitwise OR operator “|”):<br>• NWK_OPT_ACK_REQUEST<br>• NWK_OPT_ENABLE_SECURITY<br>• NWK_OPT_BROADCAST_PAN_ID<br>• NWK_OPT_LINK_LOCAL |
| data | Pointer to the payload data |
| size | Size of the payload data |
| confirm | Pointer to the confirmation callback. It should have the following prototype:<br>“void confirm(NWK_DataReq_t *req)” |
| status | This field is filled by the stack and can be accessed from the confirmation callback. It can have one of the following values:<br>• NWK_SUCCESS_STATUS<br>• NWK_ERROR_STATUS<br>• NWK_OUT_OF_MEMORY_STATUS<br>• NWK_NO_ACK_STATUS<br>• NWK_PHY_CHANNEL_ACCESS_FAILURE_STATUS<br>• NWK_PHY_NO_ACK_STATUS |
| control | This field is filled by the stack and can be accessed from the confirmation callback. It contains a value from the Control field of the Acknowledgment command frame |

```
static uint8_t message;
static NWK_DataReq_t nwkDataReq;

static void appDataConf(NWK_DataReq_t *req)
{
  if (NWK_SUCCESS_STATUS == req->status)
    // frame was sent successfully
  else
    // some error happened
}

static void sendFrame(void)
{
  nwkDataReq.dstAddr = 0;
  nwkDataReq.dstEndpoint = 1;
  nwkDataReq.srcEndpoint = 5;
  nwkDataReq.options = NWK_OPT_ACK_REQUEST | NWK_OPT_ENABLE_SECURITY;
  nwkDataReq.data = &message;
  nwkDataReq.size = sizeof(message);
  nwkDataReq.confirm = appDataConf;
  NWK_DataReq(&nwkDataReq);
}
```

## 5.4 Receiving the data

If the application has registered a data indication callback, as described in Section 5.2.8, it will be able to receive data. When a frame is received and processed by the stack it is indicated to the application via a registered callback function. This function has the following prototype *"bool appDataInd(NWK_DataInd_t *ind)"*. Fields of the *NWK_DataInd_t* structure are described in Table 5-2. The callback function must return a boolean value telling the stack whether to send an acknowledgment frame. This feature allows application data flow control.

**Table 5-2. Network data indication structure fields.**

| Name | Description |
|---|---|
| srcAddr | Network address of the source device |
| dstEndpoint | Destination endpoint number (local) |
| srcEndpoint | Source endpoint number (remote) |
| options | Data indication options. It may be any combination of the following constants (combined using bitwise OR operator "\|"):<br>• NWK_IND_OPT_ACK_REQUESTED<br>• NWK_IND_OPT_SECURED<br>• NWK_IND_OPT_BROADCAST<br>• NWK_IND_OPT_LOCAL<br>• NWK_IND_OPT_BROADCAST_PAN_ID<br>• NWK_IND_OPT_LINK_LOCAL |
| data | Pointer to the payload data |
| size | Size of the payload data |
| lqi | LQI of the received frame |
| rssi | RSSI of the received frame |

The application can set one byte of data to be sent in the acknowledgment frame. This is done using the *NWK_SetAckControl()* function. This byte can be used to pass additional information to the sending side, for example a parent can tell a sleeping device not to sleep for a while, and wait for additional data.

Receiving the data

```
static bool appDataInd(NWK_DataInd_t *ind)
{
  if (!appReadyToReceive)
    return false;
  // process ind->size bytes of the data pointed by ind->data
  NWK_SetAckControl(APP_DO_NOT_SLEEP);
  return true;
}
```

## 5.5 Network layer power management

The network layer provides an API to manage the radio transceiver power state. This API is represented by the following functions:

- *NWK_Busy()* – check if stack is ready to sleep (no frames are being processed at the moment)
- *NWK_SleepReq()* – request to switch radio transceiver into sleep mode. This function should be called only if function *NWK_Busy()* returned *true*. Radio transceiver is asleep at the function return; there is no special confirmation callback
- NWK_WakeupReq() – request to switch radio transceiver into active mode. Radio transceiver is awake at the function return; there is no special confirmation callback

Radio transceiver power management

```
case APP_STATE_PREPARE_TO_SLEEP:
{
  if (!NWK_Busy())
  {
    NWK_SleepReq();
    appState = APP_STATE_SLEEP;
  }
} break;

...

case APP_STATE_WAKEUP:
{
  NWK_WakeupReq();
  appState = APP_STATE_SEND;
} break;
```

## 5.6 System services

### 5.6.1 Initialization and task scheduling

Before any Lightweight Mesh APIs can be used the system must be initialized. Initialization is done using the *SYS_Init()* function. This function also performs low level hardware initialization, so it is recommended to call this function as early as possible in the application.

Lightweight Mesh uses cooperative multitasking; in order to run stack internal tasks, the application must call the *SYS_TaskHandler()* function as often as possible. Generally this should be done from the main “while (1)” loop.

Typical use of SYS_init() and SYS_TaskHandler() functions

```
int main(void)
{
  SYS_Init();

  while (1)
  {
    SYS_TaskHandler();
  }
}
```

### 5.6.2 Software timers

Lightweight Mesh system environment provides support for software timers. Software timers have a low hardware overhead (they all run from a single hardware timer) and the application can start any number of software timers.

A software timer is described by the *SYS_Timer_t* structure. Fields of this structure are described in Table 5-3.

**Table 5-3. Software timer parameters.**

| Name | Description |
|---|---|
| interval | Timer interval in milliseconds |
| mode | Timer operation mode. One of the following:<br>• SYS_TIMER_INTERVAL_MODE – timer *handler* will be called once after *interval* milliseconds<br>• SYS_TIMER_PERIODIC_MODE – timer *handler* will be called every *interval* milliseconds until stopped by the application |
| handler | Timer event handler. This function should have following prototype: “void handler(SYS_Timer_t *timer)” |

Software timer API is represented by the following functions:

- *SYS_TimerStart()* – start a timer
- *SYS_TimerStop()* – stop a timer
- *SYS_TimerStarted()* – check if timer is started

Using software timers

```
static SYS_Timer_t appTimer;

static void appTimerHandler(SYS_Timer_t *timer)
{
  // handle timer event
  If (timeToStop)
    SYS_TimerStop(timer);
}

static void startTimer(void)
{
  appTimer.interval = 1000;
  appTimer.mode = SYS_TIMER_PERIODIC_MODE;
  appTimer.handler = appTimerHandler;
  SYS_TimerStart(&appTimer);
}
```

## 5.7 Advanced transceiver features

### 5.7.1 Random number generator

The random number generator can be enabled by defining PHY_ENABLE_RANDOM_NUMBER_GENERATOR in the configuration file. This setting only has an effect if the radio transceiver supports this feature.

A Random number can be requested using the *PHY_RandomReq()* function. A 16-bit random value will be returned via the *PHY_RandomConf()* function.

Using the random number generator

```
void PHY_RandomConf(uint16_t rnd)
{
  srand(rnd);
}

void randomize(void)
{
  PHY_RandomReq();
}
```

### 5.7.2 Energy detection measurement

An energy detection measurement can be enabled by defining PHY_ENABLE_ENERGY_DETECTION in the configuration file. This feature is available on all radio transceivers.

A channel energy measurement can be requested using the *PHY_EdReq()* function. The measured channel energy value (in dB) will be returned via *PHY_EdConf()* function.

Using energy detection

```
void PHY_EdConf(int16_t ed)
{
  if (ed < -50)
    // stay on this channel
  else
    // scan another channel
}

void scanChannelEnergy(void)
{
  PHY_SetChannel(0x0d);
  PHY_EdReq();
}
```

## 5.8 Configuration parameters

Every application should provide a file named *config.h*. This file should contain any overrides of the default parameters, and it may be empty. It is also recommended that application parameters are stored in this file with prefix *APP_*. A complete list of Lightweight Mesh stack configuration parameters and their description's are provided below.

- NWK_BUFFERS_AMOUNT – Number of buffers reserved for stack operation. These buffers are used to send, receive, and route frames. The minimum useful value is 1; this will allow the stack to send and receive frames. However, sending a frame with an acknowledgment request requires two buffers – one to send a frame and another to receive an acknowledgment. The minimum recommended value is 3

- NWK_MAX_ENDPOINTS_AMOUNT – maximum number of source endpoints supported by the stack and application. Endpoint 0 is reserved for the stack internal use, so this parameter should be set to at least 1. Maximum value is limited by the protocol specification and equals to 16
- NWK_DUPLICATE_REJECTION_TABLE_SIZE – number of entries in the duplicate rejection table. This table is used to detect and reject frames that already have been received and processed by the stack. This table is used to resolve:
  - Loops in the network topology. Sometimes the routing algorithm may create a situation where a frame is routed in a loop between a few nodes
  - Already processed broadcast frames, received from the neighboring nodes
- Entries in this table remain active for NWK_DUPLICATE_REJECTION_TTL milliseconds and are never replaced before this timeout, so when a frame is received and there are no free entries in the duplicate rejection table, the frame will not be processed
- NWK_DUPLICATE_REJECTION_TTL – duplicate rejection table entry life time (in milliseconds). This parameter should be set to at least the maximum anticipated frame propagation time across the network, multiplied by two
- NWK_ROUTE_TABLE_SIZE – number of entries in the routing table. Each entry contains a next hop address for the destination network address. It is recommended to set this parameter to the expected number of nodes in the network, but if it is impossible (due to memory constraints, for example) then this parameter should be set as high as possible
- NWK_ROUTE_DEFAULT_SCORE – default score assigned to a new entry in the routing table. This score defines how many failed attempts to use this route will be performed, before this entry is removed from the routing table. Routing algorithm is further described in Section 4.3
- NWK_ACK_WAIT_TIME – network acknowledgment wait time (in milliseconds). After this timeout expires, request to send data is confirmed with the status of NWK_NO_ACK_STATUS. Generally this parameter should be set to at least the maximum anticipated frame propagation time across the network, multiplied by two. But it can be reduced in some cases
- NWK_ENABLE_ROUTING – If defined, will enable support for routing on the network layer. Disabling routing for nodes that are not expected to route data (sleepy devices, for example) helps to reduce memory footprint
- NWK_ENABLE_SECURITY – If defined, will enable support for data encryption on the network layer. See also SYS_SECURITY_MODE
- SYS_SECURITY_MODE – selects encryption algorithm if NWK_ENABLE_SECURITY is defined. Possible values are:
  - 0 – Hardware accelerated AES-128 (only on platforms with hardware AES engine)
  - 1 – Software XTEA (all platforms)

# 6. References and suggested literature


[1] IEEE Std 802.15.4-2006: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs).

[2] Atmel AVR2050: Atmel BitCloud® Developer Guide

[3] AT86RF212 Complete Datasheet

# 7. Revision History

| Doc. Rev. | Date | Comments |
|---|---|---|
| 42028A | 09/2012 | Initial document release |

Atmel | Enabling Unlimited Possibilities®

**Atmel Corporation**
1600 Technology Drive
San Jose, CA 95110
USA
**Tel:** (+1)(408) 441-0311
**Fax:** (+1)(408) 487-2600
www.atmel.com

**Atmel Asia Limited**
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
**Tel:** (+852) 2245-6100
**Fax:** (+852) 2722-1369

**Atmel Munich GmbH**
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
**Tel:** (+49) 89-31970-0
**Fax:** (+49) 89-3194621

**Atmel Japan G.K.**
16F Shin-Osaki Kangyo Bldg.
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
**Tel:** (+81)(3) 6417-0300
**Fax:** (+81)(3) 6417-0370