Atmel AVR2025: IEEE 802.15.4 MAC Software Package - User Guide

Features

- **Portable and highly configurable MAC stack based on IEEE® 802.15.4**
- **Atmel® MAC architecture and implementation introduction**
- **Support of several microcontroller families includes ATxmega, AT32UC3A and SAM4L**
- **Support of Atmel IEEE 802.15.4 transceivers and single chips, includes Atmel's AT86RF212, AT86RF230, AT86RF231, AT86RF233 and ATmega256RFR2 SOC.**
- **Example applications description**

1 Introduction

This document is the user guide for the Atmel MAC software for IEEE 802.15.4 transceivers. The mechanisms and functionality of the IEEE 802.15.4 standard is the basis for the entire MAC software stack implementation. Therefore it is highly recommended to use it as a reference. Basic concepts that are introduced by the IEEE standard are assumed to be known within this document.

The user guide describes about:

- The Atmel AVR®2025 MAC software package release 3.0.0

  The software contains the 2nd generation MAC, which:

  - Allows a highly flexible firmware configuration to adapt to the application requirements
  - Supports different microcontrollers and platforms/boards
  - Supports different IEEE 802.15.4 based transceivers and single chips
  - Allows easy and quick platform porting
  - Provides project files for two supported IDEs (IAR Embedded Workbench®, Atmel Studio®) and gcc support as well.
  - Supports star networks and peer-to-peer communication
  - Supports non-beacon and beacon-enabled networks

  The MAC software package is a reference implementation demonstrating the use of the Atmel IEEE 802.15.4 transceivers. It follows a generic approach and is not optimized to any specific application requirement. The user needs can be adapted to its specific application requirements.

- The general software architecture followed, package directory structure, understanding the stack, MAC power management and stack configuration.
- Example applications provided in the release, supported platforms, supported toolchains, integrated development environments and references for help.

Atmel
MCU Wireless Solutions

Application Note

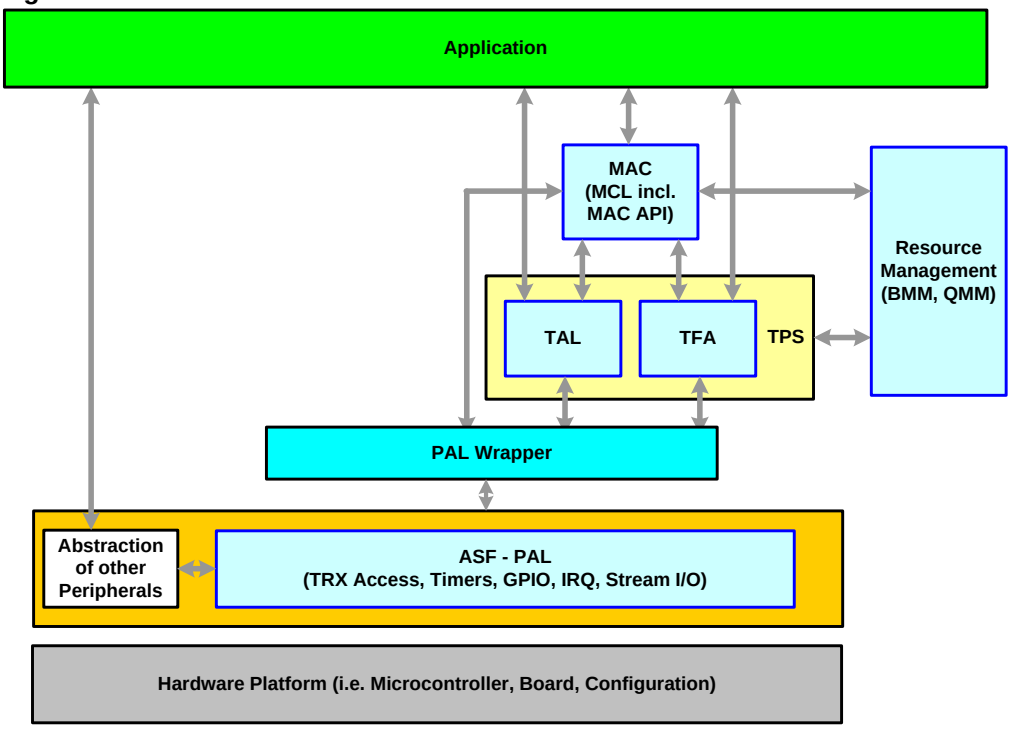
Rev. 8412A-AVR-02/13

2 General architecture

The MAC software package follows a layered approach based on several stack modules and applications. Figure 2-1 shows the stack's architecture. The stack modules are from the bottom up:

- Platform Abstraction Layer (PAL) (see Section 2.1.1)
- Transceiver Abstraction Layer (TAL) (see Section 2.1.2) and Transceiver Feature Access (TFA) (see Section 2.2.4)
- MAC including MAC Core Layer and MAC-API (see Section 2.1.3)
- Resource Management including Buffer and Queue Management (BMM and QMM) (see Section 2.2.1)

Figure 2-1. MAC architecture.



Complete description of each layer and component are provided in the succeeding sections of this document.

2.1 Main stack layers

The main MAC stack software consists of three layers starting from the bottom up:

- Platform Abstraction Layer - PAL
- Transceiver Abstraction Layer - TAL
- MAC Core layer – MCL

For other stack layers please refer to Section 2.2.

2.1.1 Platform abstraction layer (PAL)

The Platform Abstraction Layer (PAL) contains wrapper functions, which provides a seamless interface between the MAC software and ASF-PAL modules.

PAL provides interface to the following components:

1. Base timer for running software timer.
2. GPIO interrupt for External transceiver access.
3. SPI access for External transceiver access.
4. Access to persistent storage (for example, Flash or NVM or EEPROM).

Services such as Timer, SPI, and persistent storage are mapped to common\services of ASF directory to the extent possible to reduce the porting efforts. All the GPIO related configurations has to be available in corresponding <board>.h or conf_pal.h.

For each microcontroller a separate implementation exists within the ASF modules. The board and application needs are adapted via a board configuration file (conf_board.h). This board configuration file exists exactly once for each supported hardware platform.

These components are implemented as software blocks and are ported based on the target microcontroller. The timer module implements software timer functionality used by the MAC, TAL, and Application layer. The function prototypes for all PAL API functions we use are included in file PAL/Inc/pal.h.

2.1.2 Transceiver abstraction layer (TAL)

The Transceiver Abstraction Layer (TAL) contains the transceiver specific functionality used for the 802.15.4 MAC support and provides interfaces to the MAC Core Layer which is independent from the underlying transceiver. Besides that, the TAL API can be used to interface from a basic application. There exists exactly one implementation for each transceiver using transceiver-embedded hardware acceleration features. The TAL (on top of PAL) can be used for basic applications without adding the MCL.

The following components are implemented inside the TAL:

- Frame transmission unit (including automatic frame retries)
- Frame reception unit (including automatic acknowledgement handling)
- State machine
- TAL PIB storage
- CSMA module
- Energy detect scan
- Power management
- Interrupt handling
- Initialization and reset

The Transceiver Abstraction Layer uses the services of the Platform Abstraction Layer for its operation. The Frame Transmission Unit generates and transmits the frames using PAL functionality. The Frame Reception Unit reads/uploads the incoming frames and pushes them into the TAL-Incoming-Frame-Queue. The TAL handles the Incoming-Frame-Queue and invokes the receive callback function of the MCL. The operation of the TAL is controlled by the TAL state machine. The CSMA-

CA module is used for channel access. The PIB attributes related to the TAL are stored in the TAL PIB storage.

The function prototypes for the TAL features are provided in file TAL/Inc/tal.h. The implementation of a TAL is located in a separate subdirectory for each transceiver.

2.1.3 MAC core layer (MCL)

The MAC Core Layer (MCL) abstracts and implements IEEE 802.15.4-2006 compliant behavior for non-beacon enabled and beacon-enabled network support. The implemented building blocks are:

- MAC Dispatcher
- MAC Data Service
- MAC Management Service (like start, association, scan, poll, etc.)
- MAC Beacon Manager
- MAC Incoming Frame Processor
- MAC PIB Module
- MAC-API
- MAC stack task functions

The MAC Core layer provides an API that reflects the IEEE 802.15.4 standard (4).

2.1.3.1 Stack task functionality

The stack (consisting of PAL, TAL, and MCL) task functionality consists of the following API:

- Initialization
  The function wpan_init() initializes all stack resources including the microcontroller and transceiver using functions provided by the TAL and the PAL.
- Task handling
  The function wpan_task() is the stack task function and is called by the application. It invokes the corresponding task functions of the MCL, TAL, and PAL. Using the MAC software package it is required to call this function frequently supporting a round robin approach. This ensures that the different layers' state machines are served and their queues are processed.

2.1.3.2 MAC-API

The application interfaces the MAC stack via the MAC-API (see file mac_api.h in directory MAC/Inc).

It sends requests and responses to the stack by calling the functions provided by the MAC-API. The MAC-API places these requests and responses in the NHLE-MAC-Queue. It also invokes the confirmation and indication callback functions implemented by the user.

2.1.3.3 MAC core layer functionality

The MAC Dispatcher reads the NHLE-MAC-Queue and passes the requests or responses to the MAC Data Service or the MAC Management Service. The MAC Dispatcher also reads the internal event queue (TAL-MAC-Queue) and calls the corresponding event handler.

The MAC Data Service transmits data using the frame transmission services of the Transceiver Abstraction Layer and invokes the confirmation function

mcps_data_conf(), which is implemented in the MAC-API. This function in turn calls the usr_mcps_data_conf() callback function implemented by the application. The indirect data requests are queued into the Indirect-Data-Queue, where the frames are re-fetched from when a corresponding data request (poll request) is received from a device.

Receiving a data frame from the TAL through MAC Incoming Frame Processor, the MAC Data Service invokes the indication function mcps_data_ind(), which is implemented by the MAC-API. This function calls the usr_mcps_data_ind() callback function implemented by the application.

The MAC Management Service processes the management requests and responses through TAL and PAL and if applicable invokes the respective confirm function implemented by the MAC-API. This function in turn calls the usr_mlme_xyz_conf() callback function implemented by the application.

Receiving a command frame from the TAL through the MAC incoming frame processor, the MAC Management Service invokes the indication function mlme_xyz_ind(), which is implemented by the MAC-API if required. The mlme_xyz_ind() function calls the usr_mlme_xyz_ind() callback function implemented by the application.

The MAC Incoming Frame Processor receives frames from the TAL and depending on the type of the frame, passes it to the MAC Data Service or the MAC Management Service for further processing.

The MAC PIB attributes are stored in the MAC PIB and are accessed by the MAC Data Service, the MAC Management Service and the Beacon Manager. PIB attributes that are used by the TAL module are stored within the TAL.

The Beacon Manager generates the beacon frames which are transmitted using the TAL. The beacon manager is also responsible for beacon reception at the start of a superframe and its synchronization. The received beacons are processed based on the current state of the MAC and if required indications or notifications are given to the MAC-API.

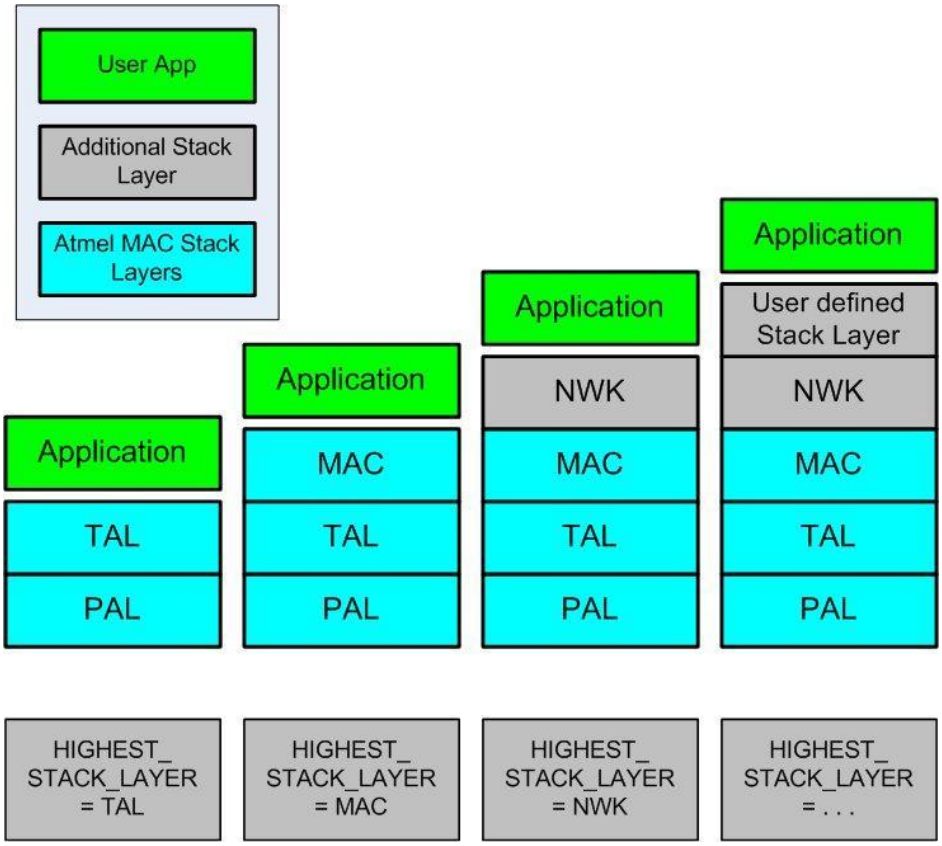
2.1.4 Usage of the stack

An application can use any layer as desired depending on the required functionality. An application that is based on a standard IEEE 802.15.4 MAC uses the MAC-API based on the stack built by PAL, TAL, and MCL. Another application (for example, a simple data pump) may want to use only the basic channel access mechanism, automatic handling of Acknowledgments, etc. In this case potentially only the TAL API based on a stack consisting of PAL and TAL will be used. What kind of stack is actually being used by the application is always depending on the end user needs and the available resources.

In order to specify which layer of the stack the application is actually based on (that is, which API it is using), the build switch HIGHEST_STACK_LAYER needs to be set properly. Depending on this switch only the required resources from the stack are used for the entire application. For further information about the usage of HIGHEST_STACK_LAYER please see Section 7.1.1.1 HIGHEST_STACK_LAYER.

The following picture shows which layers of the stack are available for the application depending on the build switch HIGHEST_STACK_LAYER. Obviously a trade-off needs to be found between required functionality on one hand and the footprint on the other hand.

Figure 2-2. Stack usage.



2.2 Other stack components

2.2.1 Resource management

The Resource Management provides access to resources to the stack or the application. These resources are:

1. Buffer Management (large and small buffers): provides services for dynamically allocating and freeing memory buffers.
2. Queue Management: provides services for creating and maintaining the queues.

The following queues are used by the software:

3. Queue used by MAC Core Layer:
    a. NHLE-MAC-Queue
    b. TAL-MAC-Queue
    c. Indirect-Data-Queue
    d. Broadcast-Queue
4. Queue used by TAL:
    a. TAL-Incoming-Frame-Queue
5. Additional queues and buffers can be used by higher layers, like application such as the MAC-NHLE-Queue.

2.2.2 Security abstraction layer

The SAL (Security Abstraction Layer) provides an API that allows access to low level AES engine functions abstraction to encrypt and decrypt frames. These functions are

actually implemented dependent on the underlying hardware, for example, the AES engine of the transceiver. The API provides functions to set up the proper security key, security scheme (ECB or CBC), and direction (encryption or decryption).

For more information about the SAL-API see file sal/inc/sal.h.

2.2.3 Security toolbox

The STB (Security Toolbox) is a high level security abstraction layer providing an easy-to-use crypto API for direct application access. It is placed on top of the SAL and abstracts and implements transceiver or MCU independent security functionality that encrypts or decrypts frames using CCM* according to 802.15.4 / ZigBee®.

For more information about the STB-API see file stb/inc/stb.h.

2.2.4 Transceiver feature access

2.2.4.1 Introduction

The current 802.15.4 stack is designed to be fully standard compliant. On the other hand Atmel transceivers provide a variety of additional hardware features that are not reflected in the standard. In order to have a clear design separation between the standard features and additional features, a new software block has been introduced – TFA (Transceiver Feature Access).

If the TFA shall be used within the application a special build switch needs to be set in order to get access to these specific features (see Section 7.1.4.2).

2.2.4.2 Features

The following features have been implemented within the TFA:

- Additional PIB attribute handling
    - Function for reading or writing special PIB attributes (not defined within 4) are provided
    - Example: Transceiver Rx Sensitivity (see the Data Sheets of the transceivers for more information about the Transceiver Rx Sensitivity)
- Single CCA
    - Based on 4 a function is implemented to initiate a CCA request to check for the current state of the channel
    - The result is either PHY_IDLE or PHY_BUSY
    - Allows for CCA measurements independent from the MAC-based CSMA-CA algorithm
- Single ED measurement
    - Based on 4 a function is implemented to initiate a single ED measurement separate from the cycle of a full ED scanning
- Reading transceiver's supply voltage
    - The battery or supply voltage reading can be enabled separately without enabling the entire TFA. If only the reading (function tfa_get_batmon_voltage()) of the supply voltage is needed, the build switch TFA_BAT_MON needs to be set. See also Section 7.1.4.2 for further information about build switches for the TFA

- Continuous transmission
    - For specific measurements a continuous transmission on a specific is required
    - In order to support this feature functions are implemented to initiate or stop a continuous transmission
- Temperature Measurement (Single Chip transceivers only)
    - A function is implemented to read the temperature value from the integrated temperature sensor in degree Celsius

For more information about the TFA implementation see file TFA/Inc/tfa.h and the source code for the various transceivers (tfa/tal_type/src/tfa.c).

For further explanation of applications and the included example applications please refer Chapter 11.

3 Understanding the software package

The following chapter describes the content of the AVR2025_MAC software package, common asf modules and drivers which are required for this MAC software to work and explains some general guidelines how the various software layers are structured.

3.1 MAC package directory structure

The AVR2025_MAC package directory structure looks as shown in Figure 3-1. This software package strategically placed under the thirdparty/wireless folder in the ASF directory.

Figure 3-1. avr2025_mac package directory Structure

```
wireless
├──addons
├──avr2025_mac
      ├──addons
      │   ├──api_parser
      │   ├──serial_bridge
      │   │   ├──sam4lc4c_sam4l_xplained_pro
      ├──apps
      │   ├──MAC
      │   │   ├──beacon
      │   │   │   ├──coord (Coordinator)
      │   │   │   │   ├──host
      │   │   │   │   │   ├──sam4lc4c_sam4l_xplained_pro
      │   │   │   │   │   ├──. . .
      │   │   │   │   │   ├──Config files
      │   │   │   │   ├──ncp
      │   │   │   │   │   ├──at32uc3a3256s_rz600_at86rf212
      │   │   │   │   │   ├──. . .
      │   │   │   │   │   ├──Include files
      │   │   │   │   └──main.c – source file
      │   │   │   └──dev (Device)
      │   │   │   │   ├──host
      │   │   │   │   │   ├──sam4lc4c_sam4l_xplained_pro
      │   │   │   │   │   ├──. . .
      │   │   │   │   │   ├──Config files
      │   │   │   │   ├──ncp
      │   │   │   │   │   ├──at32uc3a3256s_rz600_at86rf212
      │   │   │   │   │   ├──. . .
      │   │   │   │   │   ├──Include files
      │   │   │   │   └──main.c – source file
      │   │   ├──no_beacon
      │   │   │   ├──coord (Coordinator)
      │   │   │   │   ├──host
```

```
|   |   |   |   |   ├──sam4lc4c_sam4l_xplained_pro
|   |   |   |   |   ├──. . .
|   |   |   |   |   ├──Config files
|   |   |   |   ├──ncp
|   |   |   |   |   ├──at32uc3a3256s_rz600_at86rf212
|   |   |   |   |   ├──. . .
|   |   |   |   |   ├──Include files
|   |   |   |   |   └──main.c - source file
|   |   |   |   └──dev (Device)
|   |   |   |   ├──host
|   |   |   |   |   ├──sam4lc4c_sam4l_xplained_pro
|   |   |   |   |   ├──. . .
|   |   |   |   |   ├──Config files
|   |   |   |   ├──ncp
|   |   |   |   |   ├──at32uc3a3256s_rz600_at86rf212
|   |   |   |   |   ├──. . .
|   |   |   |   |   ├──Include files
|   |   |   |   |   └──main.c - source file
|   |   |   ├──no_beacon_sleep
|   |   |   |   ├──. . .
|   |   |   |   ├──Include files
|   |   |   |   └──main.c - source file
|   |   |   ├──serial_if
|   |   |   |   ├──bcn_ffd
|   |   |   |   ├──bcn_rfd
|   |   |   |   ├──no_bcn_ffd
|   |   |   |   └──no_bcn_rfd
|   |   |   |
|   |   ├──sio_helper
|   |   |   ├──module_config
|   |   |   ├──uart
|   |   |   ├──usb
|   |   ├──TAL
|   |   |   ├──Performance_analyzer
|   |   |       ├──at32uc3a3256s_rz600_at86rf212
|   |   |       ├──. . .
|   |   |       ├──inc
|   |   |       └──src
|   ├──Doc
|   |   └──User_Guide
|   ├──Include
|   ├──Source
|   |   ├──mac
|   |   |   ├──inc
|   |   |   ├──src
|   |   |   ├──unit_tests
|   |   |   |   ├──at32uc3a3256s_rz600_at86rf212
```

```
|   |   |   ├──...
|   ├──pal
|   |   ├──common_hw_timer
|   |   |   ├──example
|   |   |   ├──mega
|   |   |   ├──module_config
|   |   |   ├──sam
|   |   |   ├──uc3
|   |   |   ├──xmega
|   |   ├──common_sw_timer
|   |   |   ├──example
|   |   |   ├──module_config
|   ├──resources
|   |   ├──buffer
|   |   |   ├──inc
|   |   |   ├──src
|   |   ├──queue
|   |   |   ├──inc
|   |   |   ├──src
|   ├──sal
|   |   ├──at86rf2xx
|   |   |   ├──src
|   |   ├──  ...
|   |   ├──inc
|   ├──stb
|   |   ├──src
|   |   ├──inc
|   ├──tal
|   |   ├──at86rf212
|   |   |   ├──inc
|   |   |   ├──src
|   |   ├──atmegarf...
|   |   |   ├──...
|   |   |   ├──...
|   |   ├──inc
|   |   ├──src
|   |   ├──unit_tests
|   |   |   ├──at32uc3a3256s_rz600_at86rf212
|   |   |   ├──...
|   ├──tfa
|   |   ├──at86rf212
|   |   |   ├──inc
|   |   |   ├──src
|   |   ├──atmegarf...
|   |   |   ├──...
|   |   |   ├──...
|   |   ├──inc
```

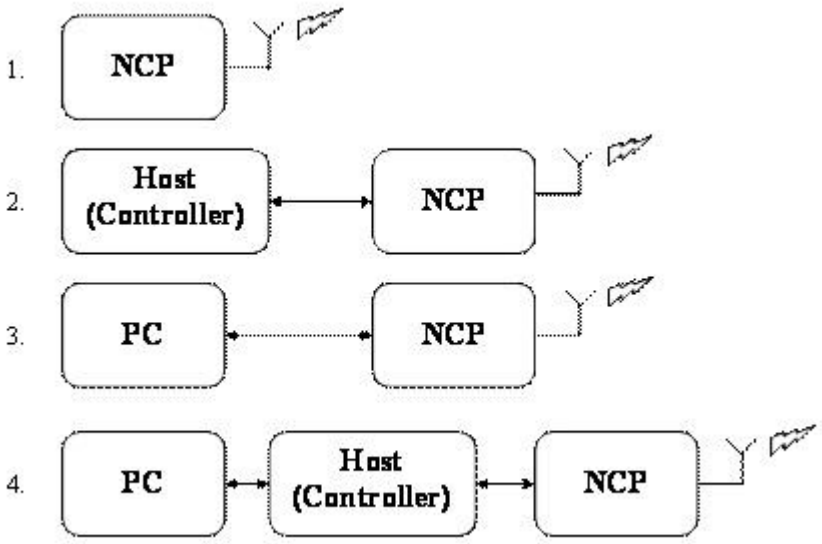
```
|   |   ├──src
|   |   ├──unit_tests
```

These directories contain the following items (in alphabetical order):

- Addons (wireless\addons):

  This addons directory placed under the wireless directory (wireless\addons) contains Serial I/O functionalities for sio2host and sio2ncp (serial I/O to Network Co-Processor) for combinations shown in below picture.

Figure 3-2. Host NCP approach



- Addons (avr2025_mac):

  This addons directory place under the wireless directory (avr2025_mac/addons) contains the source files for following addons supported by avr2025_mac.

  - api_parser : This module is responsible for encoding and decoding the serial bytes from NCP into APIs to provide seamless working for application similar to stack running in same processor or on NCP.
  - bootloader : Consists of hex file for UART based bootloader along with PC utility for upgrading the firmware. Also supports NCP firmware upgrade and the steps are explained in \thirdparty\wireless\avr2025_mac\bootloader\ readme.txt.
  - serial_if : This module is responsible for serial interface to the host.

  The above modules can co-exist in same application as described in the below architecture with proper handling in the application main.c for individual tasks.

- Examples:

  - The MAC package comes with a variety of examples applications which comprise MAC examples (using the MAC-API on top of the MAC Core Layer), TAL examples (using the TAL API)
  - The provided makefiles and project files can be used as quick start
- Doc:
  - This directory contains the AVR2025_MAC software package user guide.
- Include:
  - This directory contains header files that are of general interest both for example-applications and for all layers of the stack, such as IEEE constants, data types, return values, etc.
- Source :
  - This directory holds the source sub-directories for MAC, PAL, TAL, SAL, STB layers and other resources required.
  - MAC: This directory contains the MAC Core Layer (MCL) and the MAC-API
  - PAL: This directory contains only wrapper functions which communicate with ASF-PAL drivers (located outside this AVR2025_MAC software package) for each microcontroller family.
  - Resources: This directory contains the buffer and queue management implementation used internally inside MCL and TAL. Also hooks for application usage are provided
  - SAL: This directory contains the Security Abstraction Layer providing specific security implementations based on available hardware support
  - STB: This directory contains the Security Toolbox implementing an independent crypto API
  - TAL: This directory contains the Transceiver Abstraction Layer with subdirectories for each supported transceiver providing specific implementations addressing the specific needs of each transceiver
  - TFA: This directory contains the Transceiver Feature Access with sub-directories for each supported transceiver providing access to unique transceiver features, like receiver sensitivity configuration, etc.

3.2 Header file naming convention

The different modules or building blocks of the stack are structured very similar. Once the reader is familiar with the provided file structure, it becomes very easy to find any required information.

Each stack layer directory or building block has a directory named Inc:

- mac/inc
- pal/inc
- sal/inc
- stb/inc
- tal/inc
- tfa/inc

These directories contain basic header files that are generic for the entire block (independent from the specific implementation) or required for the upper layer.

Additionally there are further Inc subdirectories designated to specific implementations. Each transceiver implementation inside the TAL has its own Inc directories (for example, tal/at86rf231/inc).

Generally the following header file naming conventions are followed:

6. layer.h:
    - This file contains global information that forms the layer or building block API such as function prototypes, global variables, global macros, defines, type definitions, etc.
    - Each upper layer that wants to use services from a lower layer needs to include this file
    - Examples: mac.h, tal.h, pal.h, stb.h, sal.h, tfa.h
7. layer_internal.h:
    - This file contains stack internal information only. No other layer or building block shall include such a file
    - Examples: MAC/Inc/mac_internal.h, TAL/AT86RF212/Inc/tal_internal.h
8. layer_types.h:
    - This file contains the definitions for the supported types of each category that can be used with Makefiles or project files to differentiate between the various implementations and make sure that the proper code is included
    - Whenever a new type of this category is introduced (for example a new hardware board type), the corresponding file needs to be updated
    - Examples: tal_types.h, sal_types.h, vendor_boardtypes.h
9. layer_config.h:
    - This file contains definitions of layer specific stack resources such as timers or buffers.
    - For further information see Section 5.3
    - Examples: mac_config.h, tal_config.h, pal_config.h

4 Brief about ASF

The following chapter gives a brief explanation about the Atmel Software Framework.

The Atmel Software Framework (ASF) is a MCU software library providing a large collection of embedded software for Atmel flash MCUs: megaAVR, AVR XMEGA, AVR UC3 and SAM devices.

4.1 ASF directory structure

For more details on ASF directory structure please refer to Atmel Software Framework documentation [9]

5 Understanding the stack

The following chapter explains how an end user application is configured. Generally the stack is formed by every software portion logically below the application.

The stack can comprise:

- The TAL based on PAL, or
- The MAC based on TAL and PAL, or
- A network layer (NWK) based on MAC, TAL, and PAL
- Any other layer residing below the application

For configuring the stack appropriately please refer to Chapter 7.

5.1 Frame handling procedures

5.1.1 Frame transmission procedure

This section shall explain the stack layer interworking for the transmission of a MAC data frame. The payload of such a frame requires special treatment, since it is handed over from the higher layer or application, whereas other MAC frames are generated inside the MAC layer itself.

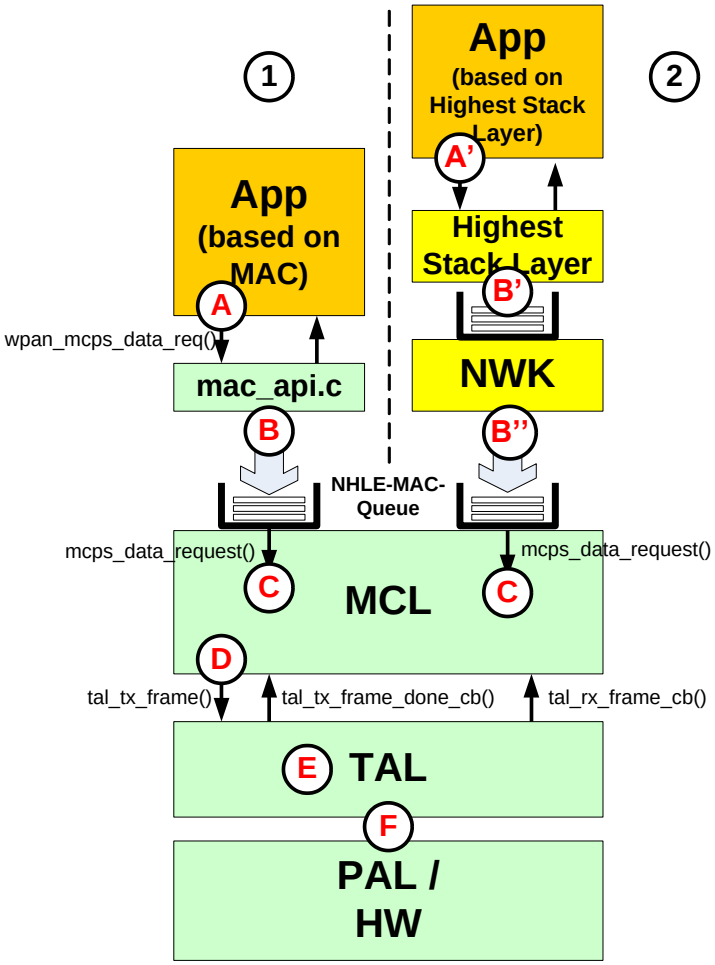
The stack is always separated into a stack domain and an application domain. The application resides on the stack layer called Highest Stack Layer (see Section 2.1.4). The AVR2025 software package can be utilized in the following two different architectures:

(**1**) An Application residing on top of the MAC layer: The application interacts with the MAC Layer by means of functions call (residing in file mac_api.c) and callbacks (residing in files usr_*.c). The MAC-API in return interacts with the MAC Core Layer (MCL) by means of messages handled with an internal queue.

(**2**) An application residing on top of another layer (above the MAC layer): The application interacts with the “Highest Stack Layer” by means of function calls and callback to be implemented within the highest stack layer and/or the application. The stack layer above the MAC (that is, the Network Layer – NWK) interacts with the MAC by means of messages handled with an internal queue (similar to (**1**)).

5.1.1.1 Part 1 – Data frame creation and transmission

Figure 4-1. Data frame transmission procedure – Part 1.



How is the procedure for a MAC Data frame which shall be transmitted?

(**A**) In case the MAC application wants to initiate a frame transmission, it call the MAC-API function wpan_mcps_data_req() function with the corresponding parameters (see file *MAC/Inc/mac_api.h*). As part of the parameter list the application needs to specify the proper MAC addressing information and the actual application payload.

In case the application resides on another higher layer than the MAC, the Highest Stack Layer needs to provide a similar API than the MAC and should handle the request of the application for a frame transmission similarly (**A'**).

(**B**) Within file mac_api.c the corresponding MAC message is generated and queued into the NHLE-MAC-Queue (which handles all MAC layer request and response messages). During this process the actual application payload is copied once into the proper position of the MCPS message. This is actually the only the data payload is copied during the entire frame transmission process. During the further processing of the frame, the payload is not copied further (except for the utilization of MAC security).

In case the application resides on another higher layer than the MAC, the Highest Stack Layer needs to generate the corresponding message accordingly and queued

this into the proper queue. Here the application payload is also copied only once at the interface of the Highest Stack Layer (**B'**). If the application is already at the right position, is it not necessary to copy the application payload again during the further process of the frame in all lower layers down to the MAC layer (**B''**).

The subsequent handling of the frame transmission attempt is identical independent from the stack layer the application is actually residing on.

(**C**) Within the MAC Core Layer (MCL) the dispatcher reads the message form the NHLE-MAC-Queue and call the corresponding function mcps_data_request() (see file *MAC/Src/mac_mcps_data.c*). The following functions are performed:

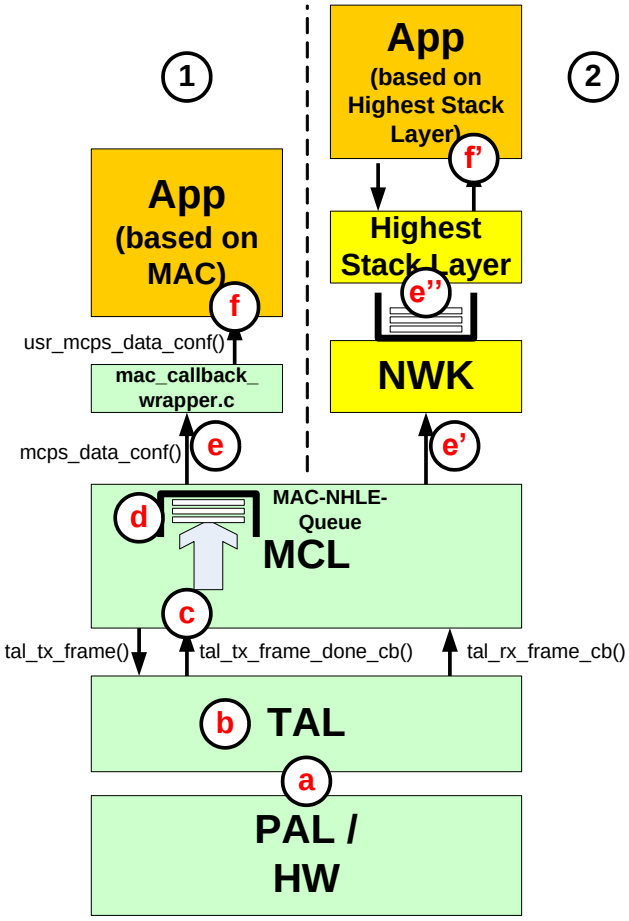
- Parsing of MAC address information
- Creation of the actual MAC frame by filling the information structure (structure ***frame_info_t*** – see file *TAL/Inc/tal.h*)
- The frame_info_t structure for the data frame contains a fully formatted MAC frame including the MAC Header information and the MSDU (that is, the MAC payload of the frame); the MSDU is not copied again during this process of the MAC frame creation

(**D**) Once the MAC frame is properly formatted, the corresponding TAL-API function is called in order to initiate the actual frame transmission (see function tal_tx_frame(); declaration in *TAL/Inc/tal.h*). The TAL functions required the frame_info_t structure as input.

(**E**) Inside the TAL no further formatting of the MAC frame is done. The frame is transmitted using the requested CSMA-CA scheme and retry mechanism. This is done by means of using PAL functions and the provided hardware (**F**). For further information, check function tal_tx_frame() in file *TAL/tal_type/tal_tx.c*.

5.1.1.2 Part 2 – Data frame clean-up and confirmation

Figure 5-1-. Data frame transmission procedure – Part 2.



(a)(b) Once the MAC frame has been transmitted (either successfully or unsuccessfully) by means of using PAL functions and the provided hardware, the TAL calls the frame transmission callback function tal_tx_frame_done_cb() residing inside the MAC (see *MAC/Src/mac_process_tal_tx_frame_status.c*) (c).

(d) Inside the MCL the corresponding callback message is generated including the frame transmission status code for the MAC DATA frame and queued into the MAC-NHLE-Queue (which handles all MAC layer confirmation and indication messages).

(e) The dispatcher extracts the confirmation message and calls the corresponding callback function (mcps_data_conf() in file *MAC/Src/mac_callback_wrapper.c*) if the application is residing on top of the MAC layer.

In case the stack utilizes another stack on top of the MAC layer, the callback functions are implemented inside the higher stack layers (e')(e'').

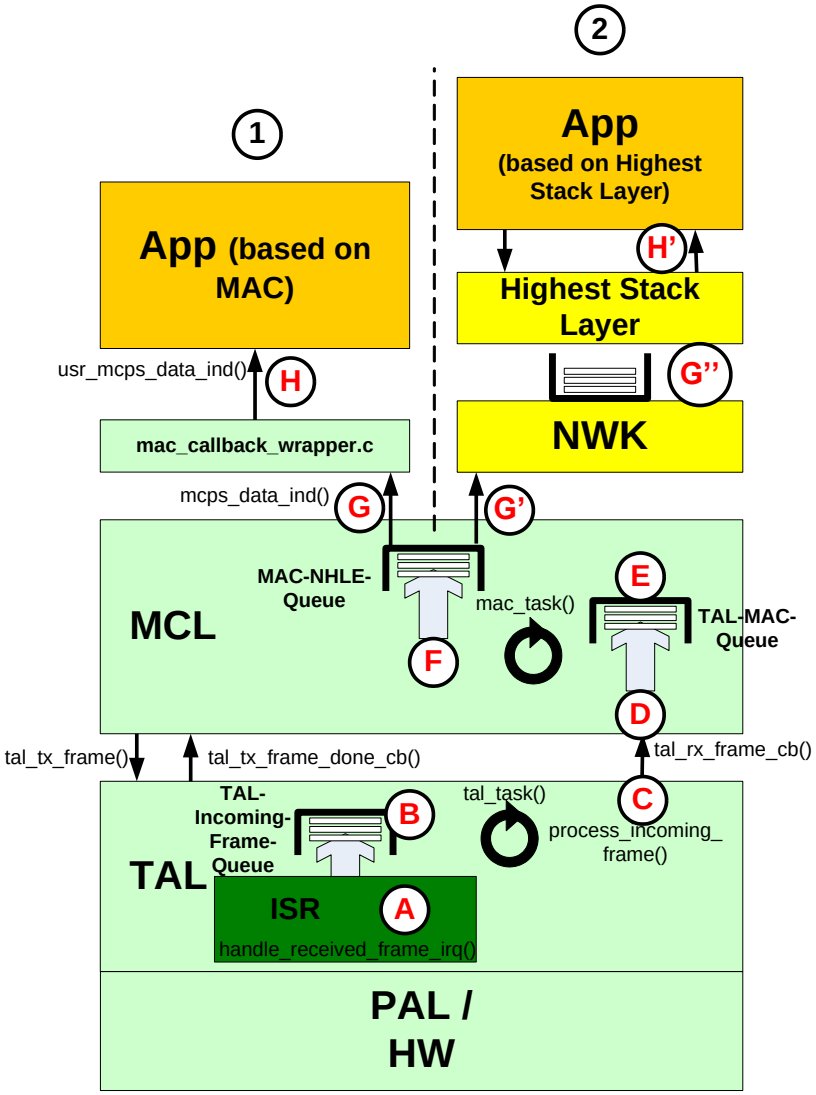
(f)(f') Finally the application is notified about the status of the attempt to transmit a data frame by means of the callback function (usr_mcps_data_conf() if the application resides on top of the MAC layer) to be implemented inside the application itself.

5.1.2 Frame reception procedure

This section shall explain the stack layer interworking by for the reception of a MAC data frame.

As already explained in Section 5.1.1 the stack is always separated into a stack domain and an application domain.

Figure 5-2. Data frame reception procedure.



How is the procedure for a MAC Data frame which is received?

(A) Once the frame has been received by the hardware the ISR is invoked and function handle_received_frame_irq() (located in file *TAL/tal_type/Src/tal_rx.c*) is called within the ISR contect. In this function the following tasks are performed:

- Reading of the ED value of the current frame
- Reading of the frame length
- Uploading of the actual frame including the LQI octet appended at the end of the frame

- Constructing the “mdpu” array of the frame_info_t structure for the received frame by additionally appending the ED value after the LQI value (for more information about the structure of the received frame see Section 5.2.1.2)
- Reading of the timestamp of received frame if required
- Queuing the received frame into the TAL-Incoming-Frame-Queue for further processing in the main context

(**B**) During the subsequent call to tal_task() (see file *TAL/tal_type/Src/tal.c*) the frame is extracted from the TAL-Incoming-Frame-Queue and function process_incoming_frame() (see file *TAL/tal_type/Src/tal_rx.c*) is called

(**C**) Within function process_incoming_frame() further handling of the frame is performed (such as calculation of the normalized LQI value based on the selected algorithm for LQI handling) and the callback function tal_rx_frame_cb() residing inside the MAC (see file *MAC/Src/mac_data_ind.c*) is called

(**D**) The callback function tal_rx_frame_cb() pushes the TAL frame indication message into the TAL-MAC-Queue for further processing inside the MCL

(**E**) During the subsequent call to mac_task() (see file *MAC/Src/mac.c*) the TAL indication message is extracted from the TAL-MAC-Queue and function mac_process_tal_data_ind() (see file *MAC/Src/mac_data_ind.c*) is called

(**F**) Within MCL the following task are performed once function mac_process_tal_data_ind() is executed

- Depending on the current state of the MCL the frame type is derived and the function handling the specific frame type is invoked
- In case of a received MAC Data Frame received during regular state of operation (that is, no scanning is ongoing, etc.) the corresponding function is mac_process_data_frame() residing in *MAC/Src/mac_mcps_data.c*
- Within function mac_process_data_frame() the MAC Header information is extracted from the received frame and the corresponding MCPS-DATA.indication primitive message is assembled
- The formatted MCPS-DATA.indication message is pushed into the MAC-NHLE-Queue

(**G**) The dispatcher extracts the indication message and calls the corresponding callback function (mcps_data_conf() in file *MAC/Src/mac_callback_wrapper.c*) if the application is residing on top of the MAC layer.

In case the stack utilizes another stack on top of the MAC layer, the callback functions are implemented inside the higher stack layers (**G’**)(**G’’**).

(**H**)(**H’**) Finally the application is notified about the reception of a Data frame data by means of the callback function (usr_mcps_data_ind() if the application resides on top of the MAC layer) to be implemented inside the application itself.

Once the received frame content is uploaded from the hardware into software, during the further process of the reception of a MAC Data frame, the actual payload of the Data frame only needs to be copied once within the receiving application on top of the MAC layer (or on top of another Highest Stack Layer). Within the stack itself the payload handling is very efficient and the content never needs to be copied.

5.2 Frame buffer handling

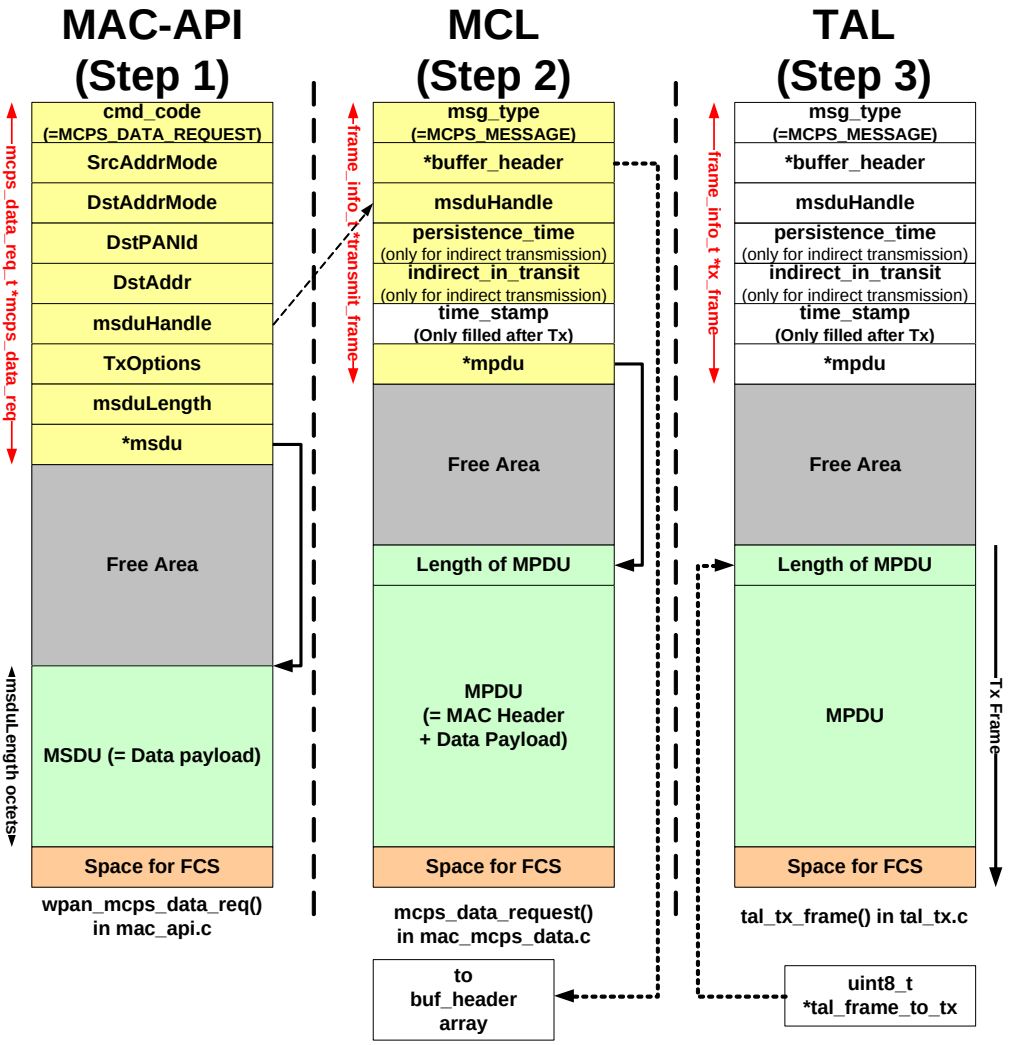
5.2.1 Application on top of MAC-API

This section explains the buffer handling for applications residing on top of the MAC-API (HIGHEST_STACK_LAYER = MAC).

5.2.1.1 Frame transmission buffer handling

The following section describes how the buffers are used inside the stack during the procedure of the transmission of a MAC Data Frame.

Figure 5-3. Frame buffer handling during data frame transmission – part 1.



Step 1:
If an application based on the MAC layer as Highest Stack Layer shall transmit a frame to another node, the MAC needs to generate a MAC Data frame. Initially the application calls function wpan_mcps_data_req() (located in file mac_apic.). In this function a new (large) buffer is requested from the Buffer Management Module (BMM) by means of the function bmm_buffer_alloc(). After the successful allocation of

the buffer the structure of type *mcps_data_req_t* is overlaid over the actual buffer body:

```
mcps_data_req =
    (mcps_data_req_t *)BMM_BUFFER_POINTER(buffer_header);
```

For more information about the *mcps_data_req_t* structure see file *MAC/Inc/mac_msg_types.h*.

The mcps_data_req_t structure is filled according to the parameters passed to function wpan_mcps_data_req() and the Data frame payload (MSDU) is copied to the proper place within this buffer. This is the only time the actual payload is copied during frame transmission inside the entire stack. The MSDU will reside right at the end of the buffer (with additional space for the FCS). The size of such a buffer fits the maximum possible payload (according to 4). Also the parameter *msdu* is updated to point right at the beginning of the MSDU content.

The entire frame buffer is then queued as an MCPS_DATA_REQUEST message into the NHLE-MAC-Queue.

Step 2:
Once the MCPS_DATA_REQUEST message has been de-queued and the dispatcher has called the corresponding function mcps_data_request() (see file *MAC/Src/mac_mcps_data.c*), the structure of type *frame_info_t* is overlaid over the actual buffer body:

```
frame_info_t *transmit_frame =
    (frame_info_t *)BMM_BUFFER_POINTER((buffer_t *)msg);
```

For more information about the *frame_info_t* structure see file *MAC/Inc/mac_msg_types.h*.

Afterwards the corresponding elements of the *frame_info_t* structure are filled accordingly:

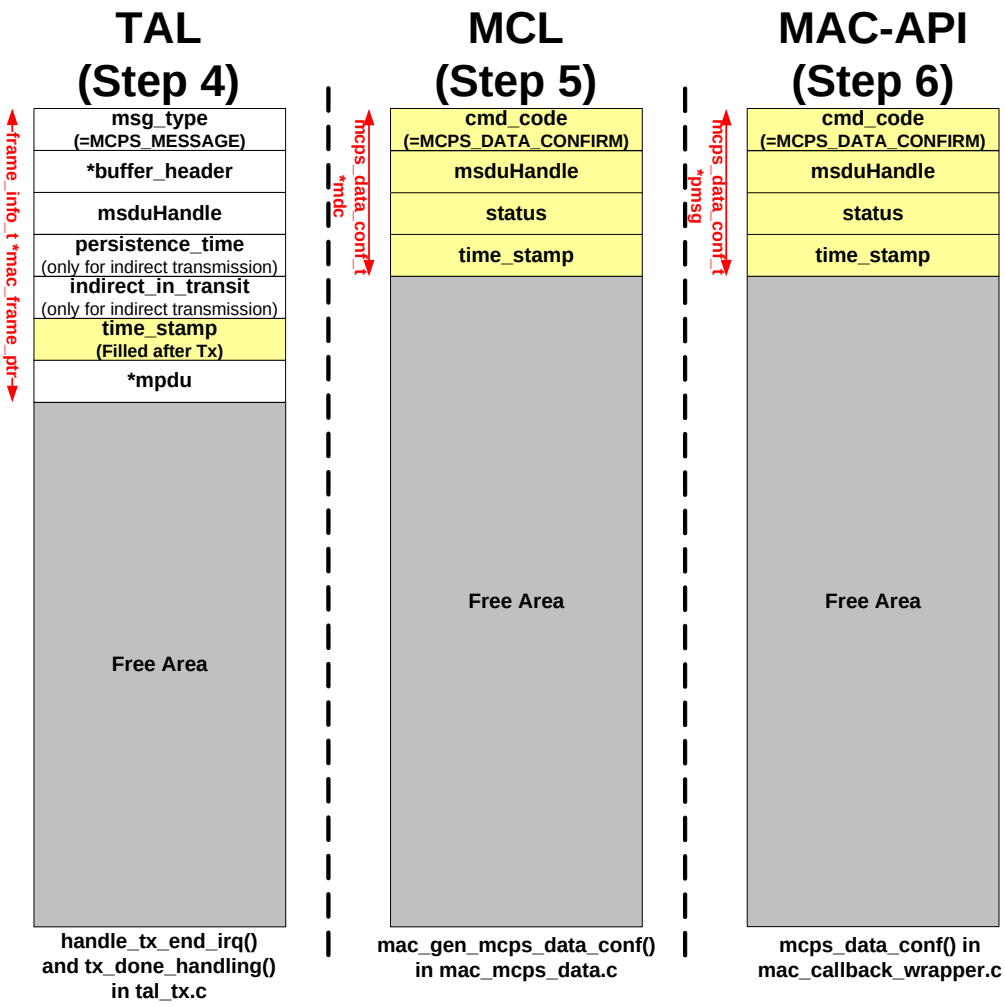
- The message_type parameter is set to MCPS_MESSAGE
- The MSDU handle is copied to the proper place
- The parameter in_transit (only utilized during indirect transmission) is set to the default value
- The buffer_header parameter is set to point to the actual buffer header; this is required once the transmission has finished to free the buffer properly

As the last step the complete frame (that is, the MPDU) is formatted. This is done by simply adding the required MAC Header information fields at the correct location in front of the MSDU (that is, the Data payload). The first element of the MPDU fill then contain the length of the entire MPDU to be transmitted, and the *mpdu* pointer within the *frame_info_t* structure is updated to point to the beginning of the frame.

This step within the MCL is finalized by initiating the actual frame transmission by calling the TAL function tal_tx_frame().

Step 3:
Within the TAL in function tal_tx_frame() (see file *TAL/tal_type/Src/tal_tx.c*) a pointer is set to the location of the actual MPDU inside the frame buffer (member *mpdu* of structure *frame_info_t*). This pointer is used for initiating the frame transmission (by means of function send_frame() with the appropriate parameters for CSMA-CA and frame retry.

Figure 5-4. Frame buffer handling during data frame transmission – part 2.



Step 4:
Once the transmission of the frame has been finished (either successfully or unsuccessfully), the transceiver generates an interrupt indicating the end of the transmission. This interrupt is handled in function handle_tx_end_irq() located in file *TAL/tal_type/Src/tal_tx.c*. In case timestamping is enabled, the time_stamp parameter is written into the proper location of the frame_into_f structure of the frame buffer. This happens in the context of the Interrupt Service Routine.

Afterwards function tx_done_handling() (located in *TAL/tal_type/Src/tal_tx.c*) is called in the main execution context. Here the timestamp is updated and the corresponding callback function tal_tx_frame_done_cb() inside the MCL is called.

Step 5:
Function tal_tx_frame_done_cb() (residing in file *MAC/Src/mac_process_tal_tx_frame_status.c*) calls a number of other functions inside the MCL, which (in the case of a processed Data frame) finally will end up in function mac_gen_mcps_data_conf() in file *mac_mcps_data.c*.

Here the structure of type *mcps_data_conf_t* is overlaid over the actual buffer body:

```
mcps_data_conf_t *mdc =
    (mcps_data_conf_t *)BMM_BUFFER_POINTER(buf);
```

For more information about the *mcps_data_conf_t* structure see file *MAC/Inc/mac_msg_types.h*.

The mcps_data_conf_t structure is filled accordingly and the entire buffer is then queued as an MCPS_DATA_CONFIRM message into the MAC-NHLE-Queue.

Step 6:
Once the MCPS_DATA_ CONFIRM message has been de-queued and the dispatcher has called the corresponding function mcps_data_conf() (see file *MAC/Src/mac_callbcak_wrapper.c*), the structure of type *mcps_data_conf_t* is again overlaid over the message (which is the actual buffer body):

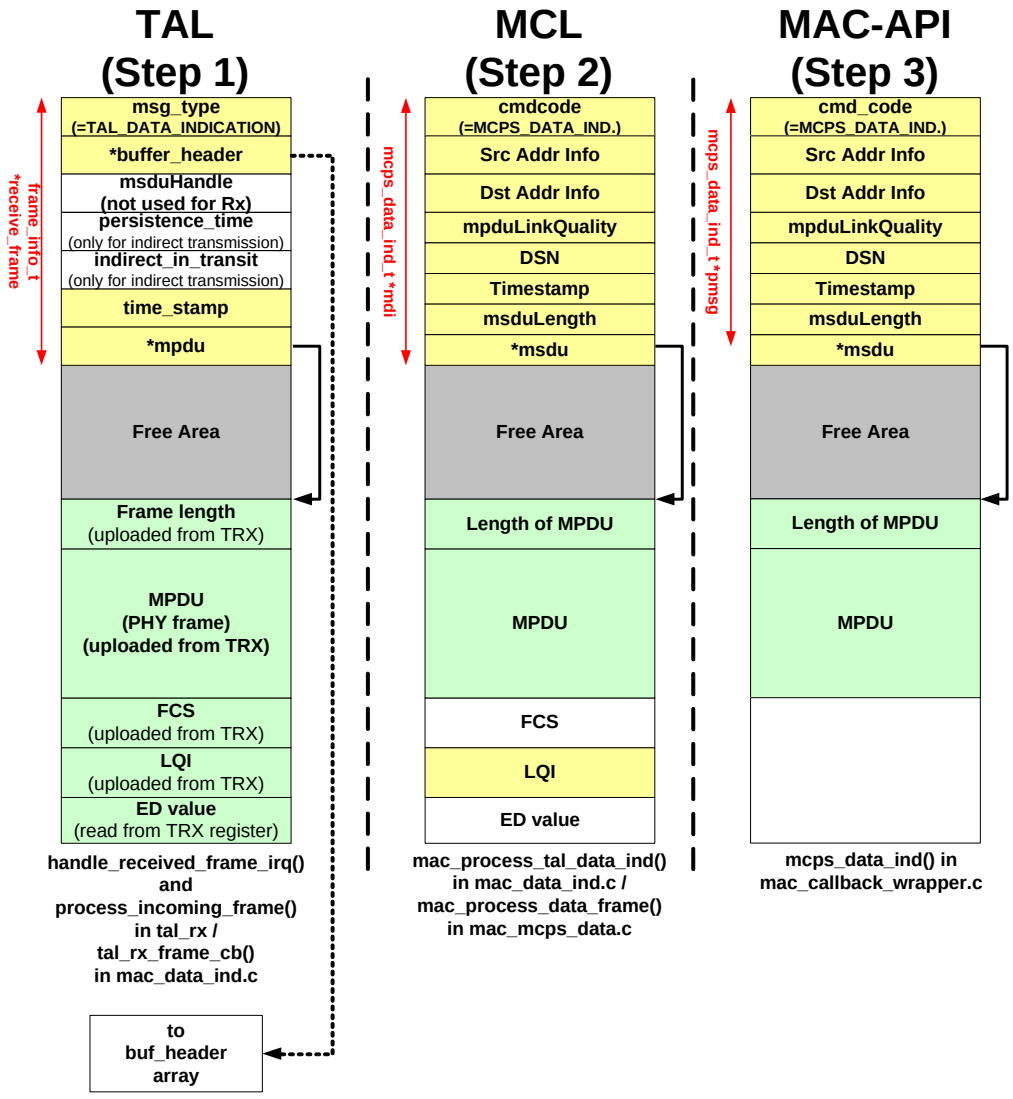
```
pmsg =
    (mcps_data_conf_t *)BMM_BUFFER_POINTER(((buffer_t
*)m));
```

Finally the corresponding parameters of the callback function inside the application (function usr_mcps_data_conf()) are filled by the corresponding members of the *mcps_data_conf_t* structure and the buffer is freed again by calling function bmm_buffer_free(). The buffer is now free for further usage.

Through all steps from (1) to (6) the same buffer is used.

5.2.1.2 *Frame reception buffer handling*

Figure 5-5. Frame buffer handling during data frame reception.



Step 1:
Once the transceivers raises an interrupt indicating the reception of a frame, function handle_received_frame_irq() (located in file *TAL/tyl_type/Src/tal_rx.c*) is called in the context of an ISR. Here a structure of type *frame_info_t* is overlaid over the current receive buffer body:

```
frame_info_t *receive_frame;
…
receive_frame =
    (frame_info_t*)BMM_BUFFER_POINTER(tal_rx_buffer);
```

After reading the ED value of the current frame and the frame length, the entire frame is uploaded from the transceiver and the ED value is stored at the location after the LQI (which automatically was uploaded from the transceiver). The mpdu pointer of the frame_info_t structure points to the proper location where the actual frame starts

within the buffer. Afterwards the entire buffer is pushed into the TAL-Incoming_Frame-Queue for further processing outside the ISR context.

After removing the buffer from the TAL-Incoming-Frame-Queue function process_incoming_frame() (also located in file *TAL/tal_type/Srctal_rx.c*) is called. Here again a structure of type *frame_info_t* is overlaid over the receive buffer body:

```
frame_info_t *receive_frame =
    (frame_info_t*)BMM_BUFFER_POINTER(buf_ptr);
```

Before the callback function inside the MAC is called, the proper buffer header is stored inside the buffer_header element of structure *frame_info_t*:

```
receive_frame->buffer_header = buf_ptr;
```

The processing inside the TAL is done once tal_rx_frame_cb() is called. Although this function resides inside the MCL (see file *MAC/Src/mac_data_ind.c*), the functionality is considered here being logically part of the TAL. Here the msg_type of the frame residing in the current buffer is specified as TAL_DATA_INDICATION and the buffer is pushed into the TAL-MAC-Queue.

Step 2:
Once the TAL_DATA_INDICATION message has been de-queued from the queue the dispatcher calls the corresponding function mac_process_tal_data_ind() (see file *MAC/Src/mac_data_ind.c*). In this function the received frame is parsed and eventually the dedicated function handling the particular frame type is invoked, which is mac_process_data_frame() in file *MAC/Srcmac_mcps_data.c*.

Here a structure of type *mcps_data_ind_t* is overlaid over the receive buffer body:

```
mcps_data_ind_t *mdi =
    (mcps_data_ind_t *)BMM_BUFFER_POINTER(buf_ptr);
```

For more information about the *mcps_data_ind_t* structure see file *MAC/Inc/mac_msg_types.h*.

The members of the mcps_data_ind_t structure are filled based on the information within the received MAC Data frame. The message is identified as a MCPS_DATA_INDICATION message and is queued into the MAC-NHLE-Queue.

Step 3:
Once the dispatcher removes the MCPS_DATA_INDICATION from the queue, the corresponding function for handling this message is called - mcps_data_ind() in file *MAC/Src/mac_callback_wrapper.c*. Here the structure of type *mcps_data_ind_t* is overlaid again over the actual buffer body:

```
pmsg =
    (mcps_data_ind_t *)BMM_BUFFER_POINTER(((buffer_t *)m));
```

Finally the corresponding parameters of the callback function inside the application (function usr_mcps_data_ind()) are filled by the corresponding members of the *mcps_data_ind_t* structure and the buffer is freed again by calling function bmm_buffer_free(). The buffer is now free for further usage.

Through all steps from (1) to (3) the same buffer is used.

5.2.2 Application on top of TAL

While an application on top of the MAC-API is logically decoupled from the actual buffer handling inside the entire stack (such an application does neither need to allocate nor free a buffer), an application on top of the TAL requires more interworking

with the stack in regards of buffer handling and internal frame handling structures. This is explained in the subsequent section.

As an example for an application residing on top of the TAL (HIGHEST_STACK_LAYER = TAL) is described in Section 11.2.2.1.

5.2.2.1 Frame transmission buffer handling using TAL-API

The following section describes how buffers are used inside the stack during the procedure of the transmission of a Frame using the TAL-API.

While the application on top of the TAL needs to free buffers received by the frame reception callback function (since these buffers are always allocated inside the TAL automatically), for frame transmission two different approaches are available:

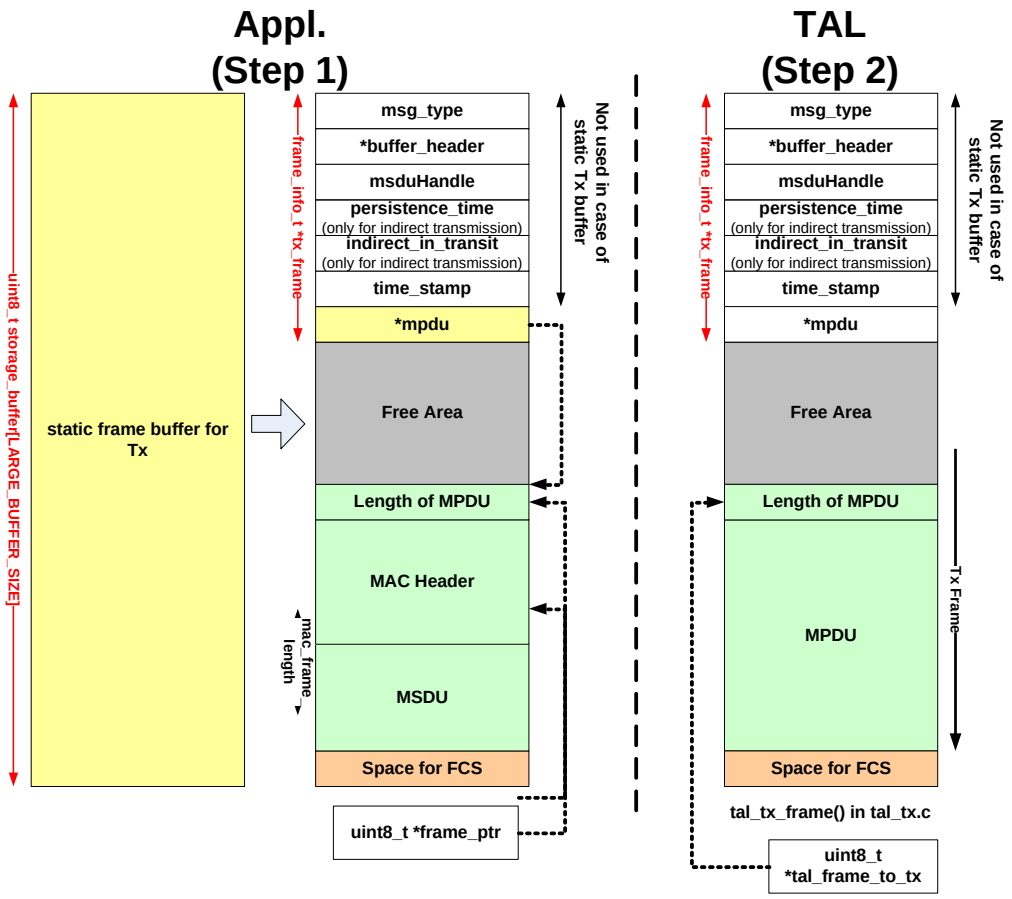
10. Application uses buffer management module provided by stack including allocation and freeing of buffers for frame transmission, or
11. Application uses static frame transmission buffer without the need for allocating or freeing buffers dynamically.

Approach (2) will be used subsequently (similar to the source code based on example application in Section 11.2.2.1).

IMPORTANT

Independent from the selected approach regarding the buffer management, it is important that the frame finally presented to the TAL for transmission follows the scheme in Figure 5-6. The frame needs to be stored at the end of the buffer (right in front of the space for the FCS). This is required in order to fit a frame using the maximum frame length according to 4 into a buffer of size LARGE_BUFFER_SIZE. If this scheme is not proper applied, memory corruption may occur.

Figure 5-6. Frame buffer handling during frame transmission using TAL-API.



Step 1:
The application (not using dynamic buffer management for frame transmission) requires a static buffer for frame to be transmitted, such as:

```
static uint8_t storage_buffer[LARGE_BUFFER_SIZE];
```

A large buffer is big enough to incorporate the longest potential frame to be transmitted based on 4.

Later a structure of type *frame_info_t* is overlaid over the static transmit buffer:

```
tx_frame_info = (frame_info_t *)storage_buffer;
```

For more information about the *frame_info_t* structure see file T*AL/Inc/tal.h*.

The static frame buffer is filled with the MSDU (that is, the actual application payload) and the MAC Header information as required by this frame. Note that also a free format frame not compliant with 4 can be created within the application. The octet in front of the MAC header needs to store the actual length of the frame.

Afterwards the *mpdu* pointer (member of the *frame_info_t* structure) is updated to point at the start of the MPDU (that is, the octet containing the length of the actual frame). Since dynamic buffer management is not used for frame transmission, the other members of the *frame_info_t* structure are not used in this frame transmission approach.

Once the frame formatting is completed, the frame is handed over to the TAL for transmission by calling function tal_tx_frame().

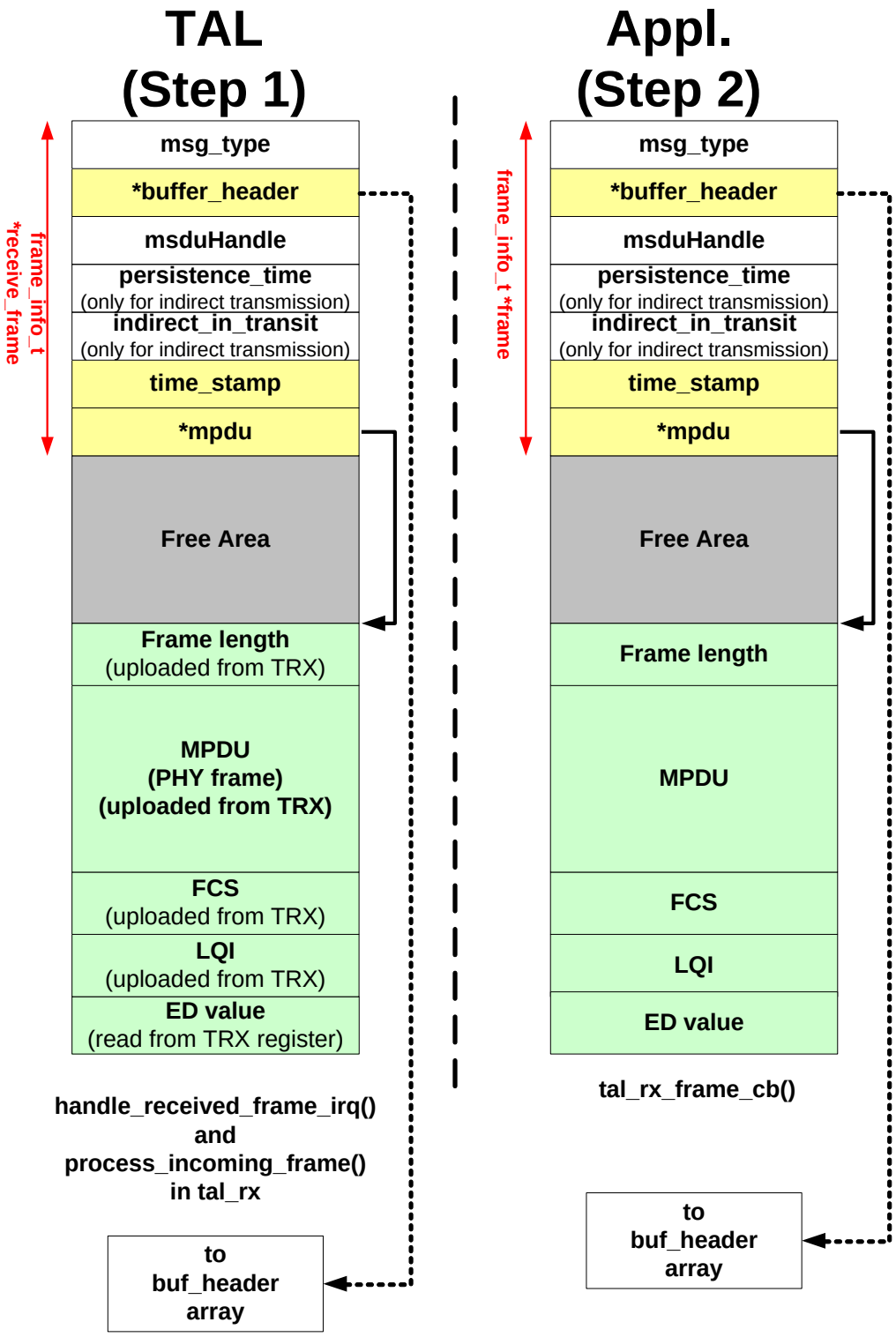
Step 2:
Within the TAL in function tal_tx_frame() (see file *TAL/tal_type/Src/tal_tx.c*) a pointer is set to the location of the actual MPDU inside the frame buffer (member *mpdu* of structure *frame_info_t*). This pointer is used for initiating the frame transmission (by means of function send_frame() with the appropriate parameters for CSMA-CA and frame retry.

Step 3:
After the frame has been transmitted the TAL acts as similar at described in Section 5.2.1.1. Once the TAL frame transmission callback function tal_tx_frame_done_cb() (residing inside the application) is called, no further handling is required in case of the usage of a static frame transmission approach. The static frame buffer can immediately be re-used for further transmission attempts. In case of dynamic buffer handling the Tx frame buffer needed to be freed additionally by calling function bmm_buffer_free().

5.2.2.2 Frame reception buffer handling using TAL-API

The following section describes how buffers are used inside the stack during the procedure of the reception of a Frame using the TAL-API.

Figure 5-7. Frame buffer handling during frame reception using TAL-API.



Step 1:

The processing of a received frame in side the TAL is independent from the layer residing on top of the TAL. The same mechanisms as described in Section 5.2.1.2 apply within in the TAL layer.

Step 2:
Once the TAL frame reception callback function tal_rx_frame_cb() (implemented inside the application) is called, the application can access the frame buffer via a *frame_inof_t* structure. At the end it is necessary to free the receive buffer by calling the function bmm_buffer_free(). A new buffer for frame reception is automatically allocated inside the TAL itself, so the application does not need to take for Rx buffer allocation.

5.3 Configuration files

The stack contains a variety of configuration files, which allow:

- The stack to configure the required stack resources according to the application needs based on the required functionality, and
- The application to configure its own resources

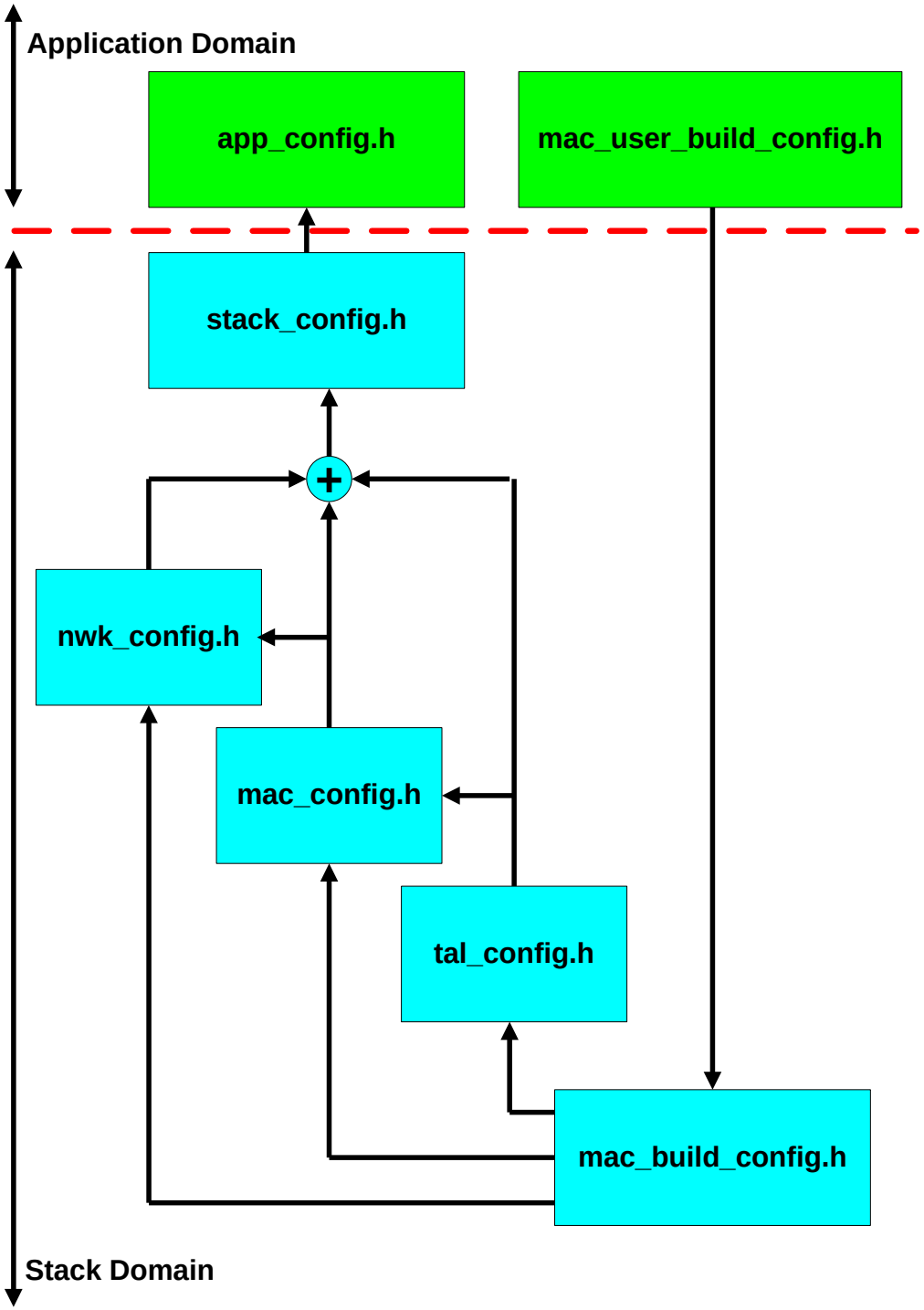
Throughout the various layers and thus directories within the software package the following configuration files are available:

- app_config.h
- stack_config.h
- tal_config.h
- mac_config.h
- mac_build_config.h
- mac_user_build_config.h

The meaning of these configuration files are described in more detail in the following sections.

The following picture shows the “#include”-hierarchy (`#include "file_name.h"`) for these configuration files.

Figure 5-8. Configuration file #include-hierarchy.



5.3.1 Application resource configuration – app_config.h

Each application is required to provide its own configuration file *app_config.h* usually located in *Inc* directory of the application.

This configuration file defines the following items:

- Timers required only within the application domain (independent from the timers used within the stack): Here the number of application timers and their timer IDs are defined
- Large and small buffers required only within the application domain (independent from the buffers used within the stack)
- Additional settings regarding the buffer size of USB or UART buffers
- Any other resources as required

In order to allow for proper resource configuration (for example, to calculate the overall number of timers) app_config.h includes the file stack_config.h which contains resource definitions from the stack domain (without the application).

This file can be adjusted by the end user according to its own needs.

5.3.2 Stack resources configuration – stack_config.h

The stack uses its own configuration file *stack_config.h* located in directory Include.

This configuration file defines the following items:

- IDs of the currently known stack layers (PAL up to NWK)
- Size of large and small buffers
- Total number of buffers and timers

Depending on the setting of the build switch HIGHEST_STACK_LAYER the configuration file of the highest layer of the stack (tal_config.h, mac_config.h, etc.) is included in order to calculate the resource requirements at compile time.

IMPORTANT This file must not be changed by the end user.

5.3.3 TAL resource configuration – tal_config.h

The TAL layer uses its own configuration file *tal_config.h* located in directory TAL/*trx_name*/Inc, that is, each transceiver (and thus each TAL implementing code for a specific transceiver) has its own TAL configuration file:

- TAL/ ATMEGARFR2/Inc
- TAL/AT86RF212/Inc
- TAL/AT86RF230B/Inc
- TAL/AT86RF231/Inc
- TAL/AT86RF233/Inc
- Etc.

These configuration files define the following items:

- Transceiver dependent values required by any upper layer (radio wake-up time)
- Timers and their IDs used within this particular TAL implementation
- The capacity of the TAL-Incoming-Frame-Queue

If the build switch HIGHEST_STACK_LAYER is set to TAL, the proper tal_config.h file (depending on build switch TAL_TYPE) is directly included into file stack_config.h since there are no further stack layers defined.

IMPORTANT These files must not be changed by the end user.

5.3.4 MAC resource configuration – mac_config.h

The MAC layer uses its own configuration file mac_*config.h* located in directory MAC/Inc.

This configuration file defines the following items:

- Timers and their IDs used within the MAC layer based on the current build configuration
- The capacity of certain MAC specific queues

If the build switch HIGHEST_STACK_LAYER is set to MAC, mac_config.h is directly included into file stack_config.h since there is no upper stack layer defined.

IMPORTANT This file must not be changed by the end user.

5.3.5 NWK resource configuration – nwk_config.h

Once a network layer (NWK) is provided as part of the stack on top to the MAC, the network layer uses its own configuration file *nwk_config.h* located in directory NWK/Inc.

If the build switch HIGHEST_STACK_LAYER is set to NWK, nwk_config.h is directly included into file stack_config.h since there is no upper stack layer defined.

IMPORTANT This file must not be changed by the end user.

5.3.6 Build configuration file – mac_build_config.h

File mac_build_config.h located in directory /*Include* defines the MAC features required for specific build configurations. See Section 7.2.1 for more information about mac_build_config.h.

IMPORTANT This file must not be changed by the end user.

5.3.7 User build configuration file – mac_user_build_config.h

Each application may provide its own user build configuration file mac_user_build_*config.h* usually located in Inc directory of the application, although this is not required. This configuration file defines the actual MAC components used for the end user application and can actually reduce resource requirements drastically.

If the application wants to use its own user build configuration the build switch MAC_USER_BUILD_CONFIG needs to be set. See Section 7.2.2 for more information about mac_user_build_config.h.

This file can be adjusted by the end user according to its own needs.

For more information about user build configurations and its utilization please refer to Section 5.4 and Section 7.2.2.

5.4 MAC components

The MAC is implemented to be fully compliant to the IEEE 802.15.4-2006 standard.

The MAC components are clustered in essential components and supplementary components.

Essential components are required for a minimum reasonable application based on the MAC and are thus always included in a build. These components are:

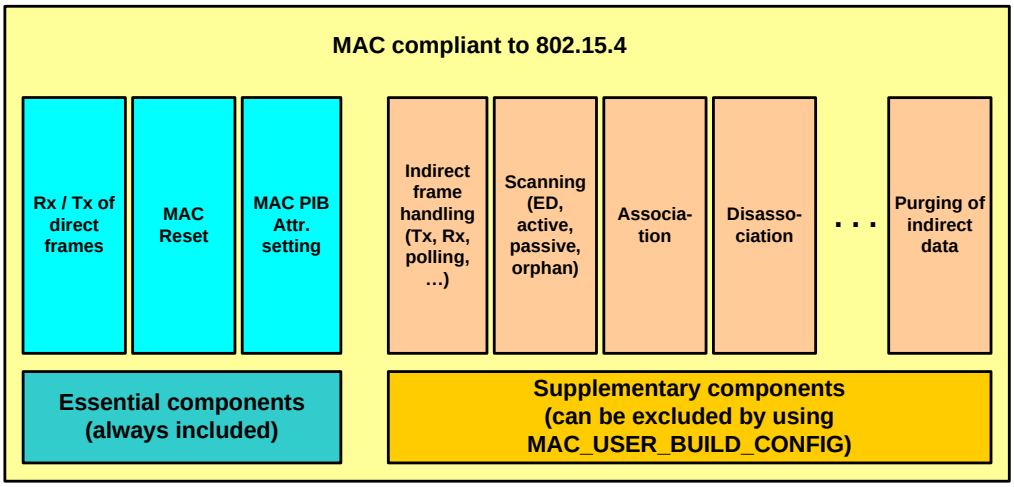
- MAC reset

- Direct data transmission and reception
- Writing MAC PIB attributes

Supplementary components are components that provide standard MAC functionality that might not be required for some applications. This is example association, indirect data transmission, scanning, etc. These components are also included in the standard build and can be used by any applications, so the end application does not have to worry about the inclusion of any functionality.

On the other hand all supplementary components can be removed from the build in order to drastically reduce footprint. For more information about how to add or remove components from the build please see Section 7.2 (using build switch MAC_USER_BUILD_CONFIG).

Figure 5-9. Essential and supplementary MAC components.



The following sections describe some of these supplementary components (especially the more complex ones) in more detail.

5.4.1 MAC_INDIRECT_DATA_BASIC

This feature is usually required for any node (both RFD and FFD) that wants to receive indirect data. This is for instance helpful, if a node is usually in power save mode and thus cannot receive direct frames from its parent. The node could then periodically wake-up and poll its parent for pending data.

This feature includes the following functionality:

- Initiation of explicit polling for pending of indirect data (usage of wpan_mlme_poll_req() / usr_mlme_poll_conf())
- Transmission of data request frames to its parent
- Reception of indirect data frames
- Initiation of implicit polling for indirect data (that is, transmission of data request frame without an explicit call of function wpan_mlme_poll_req()):
    - Polling for an association response frame during the association procedure
    - Polling for more pending data once a received frame from its parent has indicated more pending data at the parent

- Polling for pending data in case a received beacon frame from its parent has indicated pending data at the parent

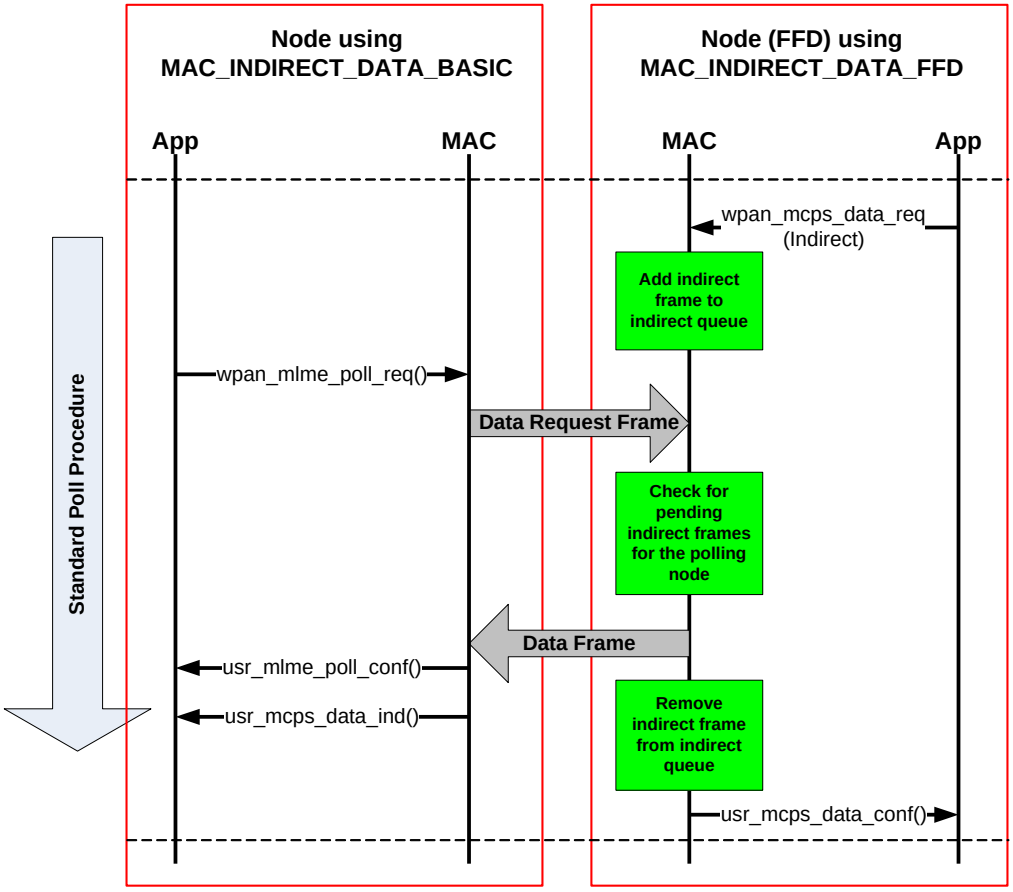
5.4.2 MAC_INDIRECT_DATA_FFD

This feature is a further extension of the feature MAC_INDIRECT_DATA_BASIC (that is, in order to use MAC_INDIRECT_DATA_FFD also MAC_INDIRECT_DATA_BASIC is required). It is designed for FFDs (PAN Coordinators or Coordinators) to allow the handling of transmitting indirect data frames.

This feature includes the following functionality:

- Initiation of indirect data transmission (usage of wpan_mcps_data_req() with TxOption = Indirect Transmission)
- Handling of the Indirect-Data-Queue
  - Adding and removing of indirect frames
  - Handling of a persistence timer in order to check for expired transactions
- Transmission of association response frame or indirect disassociation notification frames
- Handling of received data request frames and the proper responses (either with pending frames or a data frame with zero length payload)
- Setting of Frame Pending bit in the Frame Control field
- Adding of address of nodes with pending frames in the beacon frame payload

Figure 5-10. Example of provided functionality for MAC_INDIRECT_DATA_BASIC and MAC_INDIRECT_DATA_FFD.



5.4.3 MAC_PURGE_REQUEST_CONFIRM

This feature is a typical FFD feature allows a node to purge pending indirect frames from its Indirect_Data_queue by means of using functions wpan_mcps_purge_req() / usr_mcps_purge_conf().

Since purging of pending data requires handling of transmitting indirect frames, the feature MAC_INDIRECT_DATA_FFD is also required.

5.4.4 MAC_ASSOCIATION_INDICATION_RESPONSE

This feature is a typical FFD feature that allows a node to receive and process association request frames and handle them properly. In case the network uses short addresses, a short address may be selected and returned to the initiating device by means of association response frame.

Since the association procedure is perform using indirect traffic and the node using MAC_ASSOCIATION_INDICATION_RESPONSE has to transmit the association response frame indirectly also the components MAC_INDIRECT_DATA_BASIC and MAC_INDIRECT_DATA_FFD are required.

5.4.5 MAC_ASSOCIATION_REQUEST_CONFIRM

This feature allows a node (both RFD and FFD) to associate to a parent (PAN Coordinator or Coordinator) to initiate an association procedure (by transmitting an

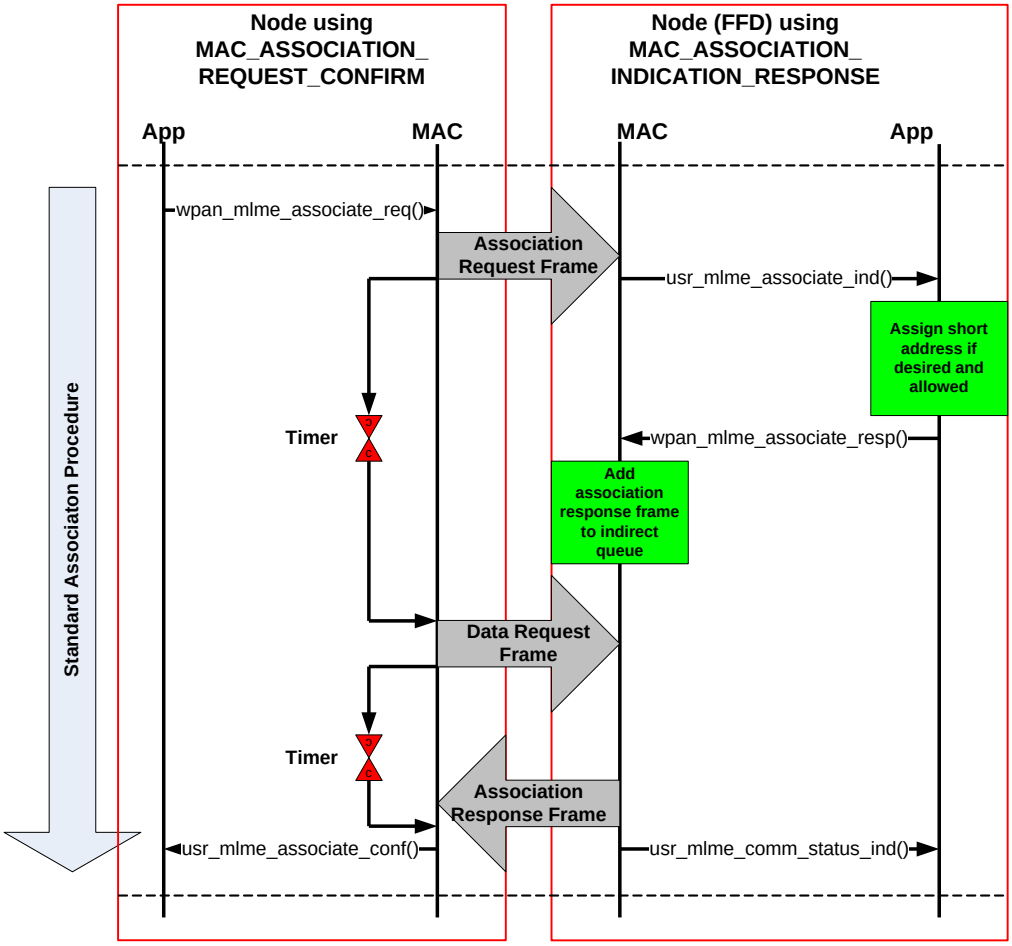
association request frame) and handle the reception of an association response frame.

In case a short address is desired this will be requested by the parent if allowed. All required timer for the association process are handled as well.

Since the association response frame is received indirectly, also the feature MAC_INDIRECT_DATA_BASIC is required.

The node is able accept and process a request from its upper layer (for example, the network layer) to associate itself to another node (that is, its parent).

Figure 5-11. Provided functionality for MAC_ASSOCIATION_INDICATION_RESPONSE and MAC_ASSOCIATION_REQUEST_CONFIRM.



5.4.6 MAC_DISASSOCIATION_BASIC_SUPPORT

This components allows

- A node (both RFD and FFD) to initiate a disassociation procedure from its parent (PAN Coordinator or Coordinator),
- a node (both RFD and FFD) to handle a received disassociation notification frame from its parent,
- a node (both RFD and FFD) to poll for a pending indirect disassociation notification frame,

- a node (FFD only) to initiate a disassociation procedure to its child

Since the disassociation notification frame may be received indirectly, also the feature MAC_INDIRECT_DATA_BASIC is required.

5.4.7 MAC_DISASSOCIATION_FFD_SUPPORT

This feature is a typical FFD feature that allows a node to transmit an indirect disassociation notification frame to one of its children.

The following components are required as well:

- MAC_DISASSOCIATION_BASIC_SUPPORT
- MAC_INDIRECT_DATA_BASIC
- MAC_INDIRECT_DATA_FFD

5.4.8 MAC scan components

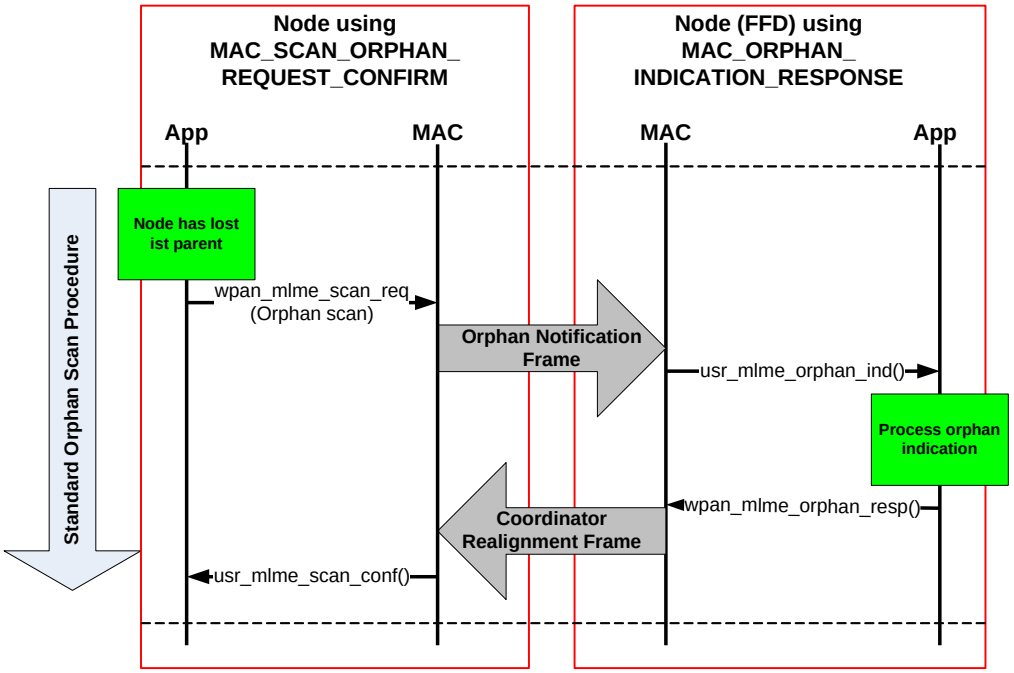
These components allow a node to perform a specific type of scanning.

- MAC_SCAN_ACTIVE_REQUEST_CONFIRM
  The node is able to perform an active scan to search for existing networks
- MAC_SCAN_ED_REQUEST_CONFIRM
  The node is able to perform an energy detect scan
- MAC_SCAN_ORPHAN_REQUEST_CONFIRM
  The node is able to perform an orphan scan in case it has lost its parent
- MAC_SCAN_PASSIVE_REQUEST_CONFIRM
  The node is able to perform a passive scan to search for existing networks. This feature is only available if beacon-enabled networks are supported

5.4.9 MAC_ORPHAN_INDICATION_RESPONSE

This feature is a typical FFD feature that allows a node to process a received orphan notification frame from any of its children (initiated via an orphan scan request at the children) and process them properly. In response a realignment frame may be returned.

Figure 5-12. Provided functionality for MAC_ORPHAN_INDICATION_RESPONSE and MAC_SCAN_ORPHAN_REQUEST_CONFIRM (orphan scan procedure).



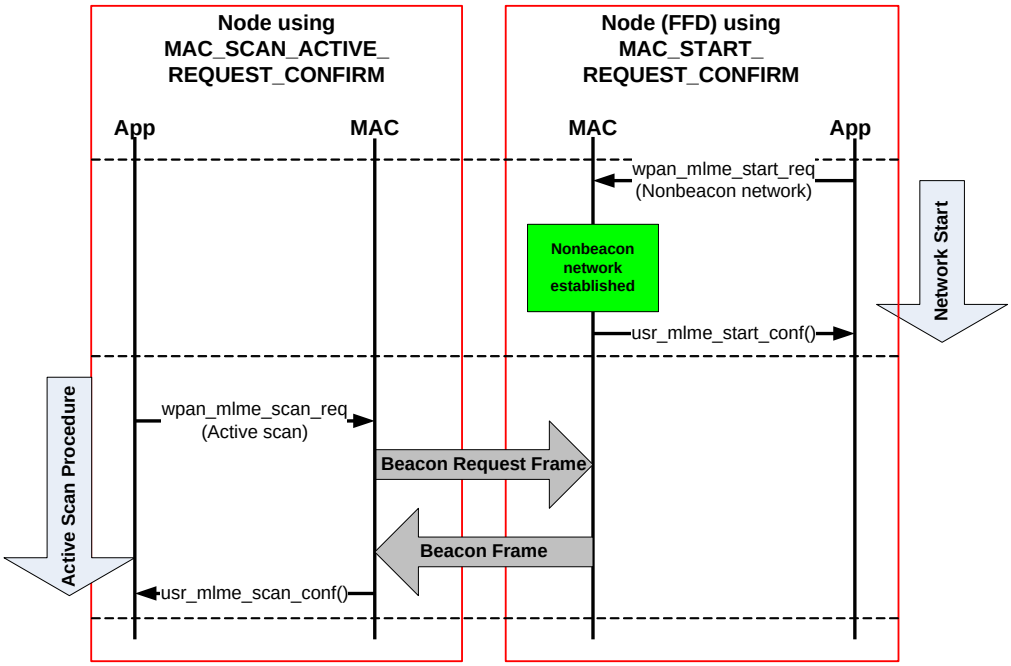
5.4.10 MAC_START_REQUEST_CONFIRM

This feature is a typical FFD feature that allows a node to start a new PAN (network) by means of using functions wpan_mlme_start_req() / usr_mlme_start_conf(). Depending on the setting of BEACON_SUPPORT this can be either only a non-beacon enabled network or also a beacon-enabled network.

Consequently this also enables the ability of the node to:

- Transmitting beacon frames (in case beacon-enabled networks are supported)
- Respond to beacon request frames (active scan by another node) with proper beacon frames
- Perform network realignment and transmit coordinator realignment frames (initiated by calling function wpan_mlme_start_req() with parameter CoordinatorRealignment = true)

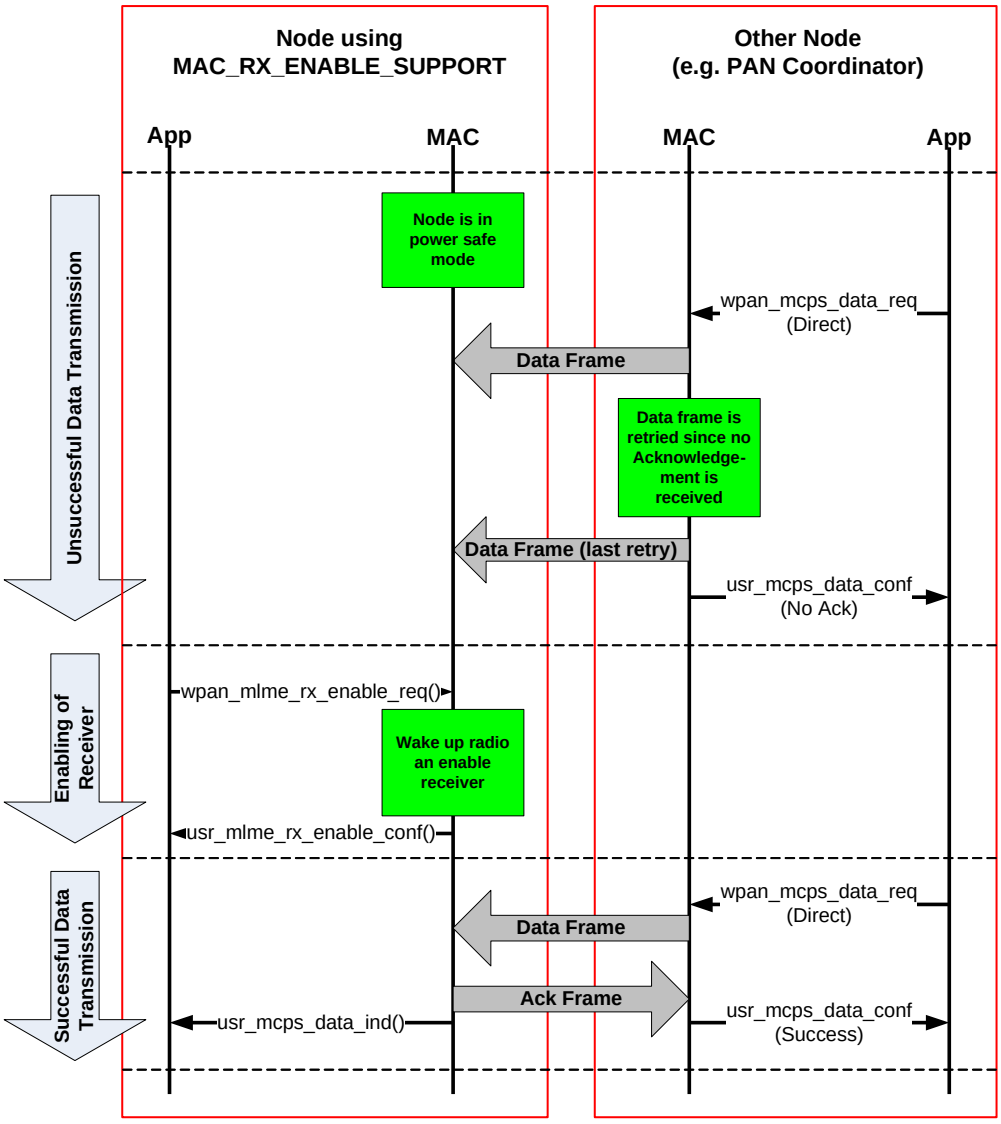
Figure 5-13. Start of non-beacon network and active scan.



5.4.11 MAC_RX_ENABLE_SUPPORT

This feature is usually required for any node (both RFD and FFD) that wants to enable its receiver for a certain amount of time or disable its receiver. Most commonly it is utilized at an RFD that goes to sleep mode during idle periods to save as much power as possible. In order to periodically listen to the channel or frames to be received, the application can initiate a wpan_mlme_rx_enable_req() with proper parameters (see MAC Example Basic_Sensor_Network in Section 11.2.1.1).

Figure 5-14. Enabling of receiver and proper data reception.



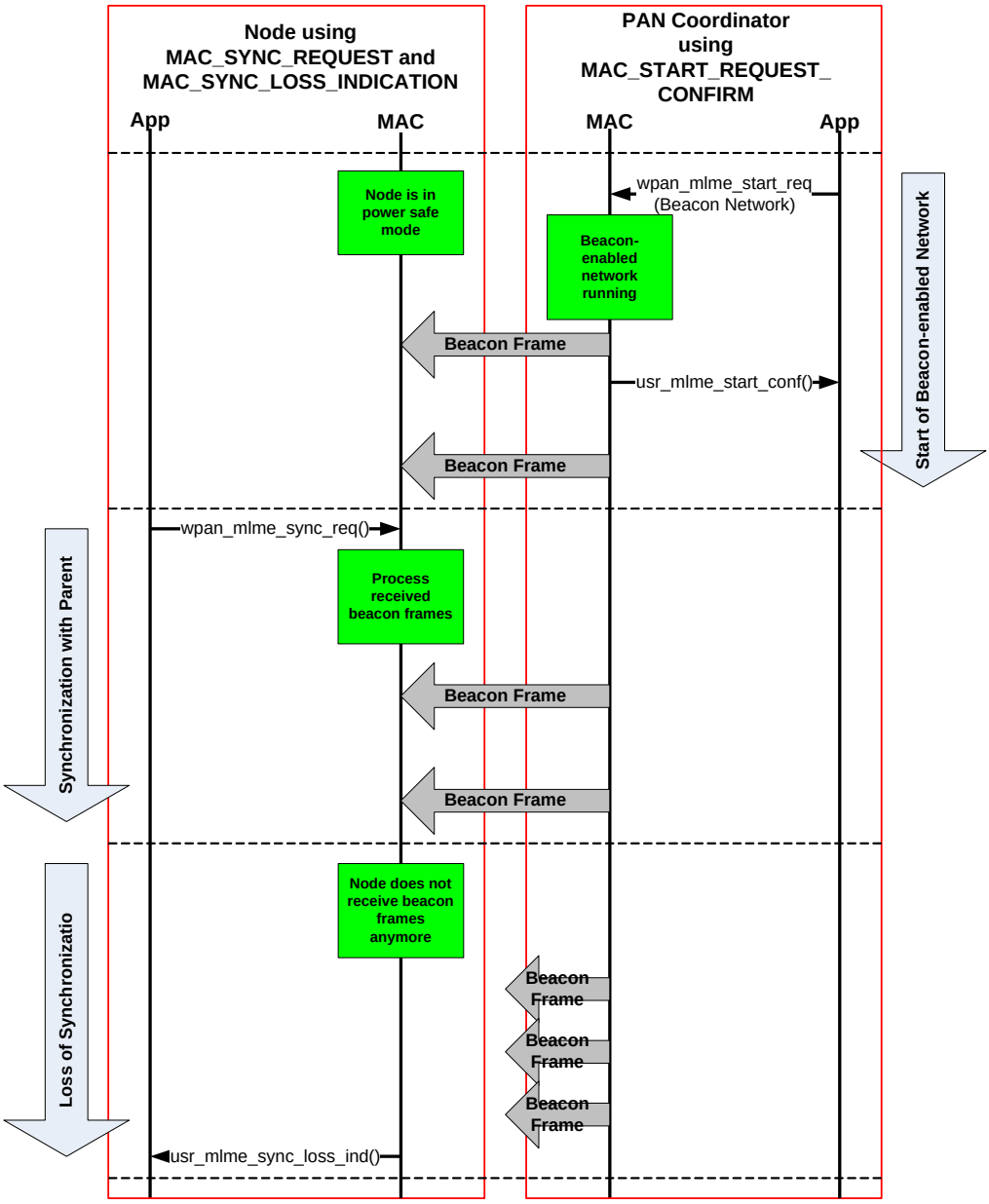
5.4.12 MAC_SYNC_REQUEST

This feature is usually required for any node (both RFD and FFD) that wants to synchronize with its beacon-enabled network tracking beacon frames from its parent by means of using function wpan_mlme_sync_req().

5.4.13 MAC_SYNC_LOSS_INDICATION

This feature is usually required for any node (both RFD and FFD) that needs to be able to report a sync loss condition to its upper layer. This can be either the reception of a coordinator realignment frame from its parent, or caused by the fact that a synchronized node has not received beacon frames from its parent for a certain amount of time.

Figure 5-15. Synchronization and loss of synchronization.



5.4.14 MAC_BEACON_NOTIFY_INDICATION

This feature is usually required for any node (both RFD and FFD) that may need to present received beacon frame to its upper layer. This could be caused by the fact that the received beacon frame contains a beacon payload or the MAC PIB attribute macAutoRequest within the node is set to false.

5.4.15 MAC_GET_SUPPORT

This feature allows reading the current values of MAC PIB attributes by means of using function wpan_mlme_get_req().

5.4.16 MAC_PAN_ID_CONFLICT_AS_PC

This feature is a typical FFD feature that allows a PAN Coordinator node to detect a PAN-Id conflict situation and report this to its higher layer, allowing the higher layer or application to initiate the proper PAN-Id conflict resolution. The node is able to detect a PAN-Id conflict situation while acting as a PAN Coordinator by checking received beacon frames from other PAN Coordinators and being able to act upon the reception of PAN-Id Conflict Notification Command frames from its children.

The following components are required as well:

- MAC_START_REQUEST_CONFIRM
- MAC_SYNC_LOSS_INDICATION

5.4.17 MAC_PAN_ID_CONFLICT_NON_PC

This feature is usually required for any node (both RFD and FFD) that may need to detect a PAN-Id conflict situation while acting not as a PAN Coordinator node. The node is able to detect a PAN-Id conflict situation while NOT acting as a PAN Coordinator by checking received beacon frames from other PAN Coordinators and being able to initiate the transmission of PAN-Id Conflict Notification Command frames from its parents if required.

The following components are required as well:

- MAC_SYNC_LOSS_INDICATION
- Either MAC_ASSOCIATION_REQUEST_CONFIRM or MAC_SYNC_REQUEST

5.5 High-density network configuration

The IEEE standard 4 provides knobs to adjust the MAC layer to the application needs. These knobs are the PIB attributes that allow configuring the behavior of the MAC. In particular, in high-density networks where many nodes access the network at the same time it might be necessary to tweak the MAC to achieve better performance. This applies to non-beacon enabled and beacon-enabled networks. The following PIB attributes can be used to tweak the MAC in this regard:

- Back off exponent: The PIB attributes *macMinBE* and *macMaxBE* determine the potential length of the back off exponent used for the CSMA algorithm. By default macMinBE = 3 and macMaxBE = 5. In order to reduce the probability that different nodes use the same back off period, it is recommend changing the PIB attribute values from the default values. Example: macMinBE = 6 and macMaxBE = 8. The macMaxBE value needs to be increased before the macMinBE is increase.
- Frame retries: The PIB attribute *macMaxFrameRetries* defines the maximum number of re-transmissions if a requested ACK is not received. The default value (defined by the IEEE standard and used by the MAC implementation) is three. The IEEE standard allows increasing the number of retries up to seven.
- Maximum CSMA back offs: The PIB attribute *macMaxCSMABackoffs* defines the number of back offs that are allowed before the channel is finally determined as busy and no transmission happens. The MAC implementation uses the default value of four. Increasing the value to five also increases the probability of successful transmission of a big number of nodes trying to access the channel at the same time.

5.6 High data rate support

The Atmel transceivers (except for the AT86RF230) are capable of transmitting frames at higher data rates than the standard data rates within the given band. The supported data rates are currently up to 2Mbit/s. These higher data rates are rates not defined within the IEEE standard (see 4). For more information about high data rate support please refer to the data sheet for the corresponding transceiver.

In order to enable higher data rates than the standard rates, the following two items needs to be done:

12. Enable the build switch HIGH_DATA_RATE_SUPPORT within the corresponding Makefile or project file (see Section 7.1.4.3).
13. Set the PIB attribute phyCurrentPage to the corresponding value (for example, phyCurrentPage = 17 for 2Mbit/s support); after setting the correct channel page all frames will be transmitted using the corresponding data rate belonging to this channel page.

Table 5-1 shows which data rate can be selected (by setting a specific channel page) using a particular transceiver. The table entries with yellow background refer to Atmel proprietary channel pages for non-standard high rates. Please note that the standard channel page is always channel page 0.

Table 5-1. Channel pages vs. data rates.

Frequency band/ transceiver	**MAC-2003 compliant channel page 0**	**MAC-2006 compliant**	**High data rate mode 1**	**High data rate mode 2**
Sub-1 GHz **Channel 0** **Channel 1-10** **AT86RF212**	20kb/s @ -110dBm 40kb/s @ -108dBm (2)	Channel Page 2 100kb/s 250kb/s	Channel Page 16 200kb/s 500kb/s (1)(3)(4)(5)(6)	Channel Page 17 400kb/s 1000kb/s (1)(3)(4)(5)(6)
Chinese Band **Channel 0-3** **AT86RF212**	N/A	Channel Page 5 250kb/s	Channel Page 18 500kb/s (1)(3)(4)(5)(6)	Channel Page 19 1000kb/s (1)(3)(4)(5)(6)
2.4GHz **Channel 11-26** **AT86RF231, ATmega256RFR2, …**	250kb/s @ -101dBm	Channel Page 2 500kb/s (1)(3)(5)	Channel Page 16 1000kb/s (1)(3)(5)	Channel Page 17 2000kb/s (1)(3)(5)

Frequency band/ transceiver	MAC-2003 compliant channel page 0	MAC-2006 compliant	High data rate mode 1	High data rate mode 2

Notes:
1. PSDU data rate.
2. BPSK.
3. Reserved channel pages are used to address the appropriate mode (non-compliant).
   Marked w/ yellow background color. The Atmel AT86RF212's sensitivity values for the proprietary modes are based on PSDU length of 127 bytes.
4. Scrambler enabled.
5. Reduced ACK timing. Proprietary channel pages can be enabled using build configuration switch HIGH_DATA_RATE_SUPPORT.
6. Proprietary channel pages 18 and 19 used for Chinese frequency band. Pages 18 and 19 support channels 0-3.

For a TAL example using a high data rate of 2Mbit/s please refer to Section 11.2.2.1.

6 MAC power management

The MAC stack provide built-in power management that allows to put the transceiver into power save state as often as possible in order to save as much energy as possible. This allows for example End Devices, which are usually battery powered, to use a sleeping state of the transceiver as default state. The entire power management is inherent in the MAC itself and works without any required interaction from the application. On the other hand there exist means for the application to control the power management scheme in general as desired.

6.1 Understanding MAC power management

The following section is only valid for MAC applications (or applications on higher layers), but not for TAL applications. For TAL applications please refer to Section 6.4.

Once a node (running an application on top of the MAC stack) has finished its initialization procedure, the MAC layer decides whether the node stays awake or enters SLEEP state. This is controlled by the MAC PIB attribute macRxOnWhenIdle.

Whenever this PIB attribute is set to True, the MAC keeps the radio always in a state where the default state of the receiver is on, that is, the transceiver is able to receive incoming frames whenever there is nothing else to do for the stack. Since this is a non-sleeping state for the transceiver and requires much more energy than a power save state, this behavior is usually applied for all mains powered nodes, such as PAN Coordinators or Coordinators/Routers (nodes built using build switch FFD, see Section 7.2.1.1).

As mentioned MAC power management works automatically within the stack. Once a node is started, it is automatically in power save mode. This is handled by the MAC PIB attribute macRxOnWhenIdle. The default value for macRxOnWhenIdle is False, that is, the radio shall be off in case the node is idle, meaning that the radio will be in sleep mode. This is valid for all nodes including End Devices, Coordinators and PAN Coordinators.

Any node that shall not be in sleep as the default mode of operation needs to be put in listening mode by setting the PIB attribute macRxOnWhenIdle to True. Usually this is only done for mains powered nodes, such as Coordinator or PAN Coordinators. This will be explained more in detail in the subsequent sections.

For battery powered nodes (usually End Devices) the default state shall be sleeping, since otherwise the battery would be emptied too fast. Since the PIB attribute macRxOnWhenIdle is per default set to False by default, such nodes will automatically enter SLEEP state when there is nothing to be done for the transceivers.

The transceiver of such nodes will be woken up automatically whenever a new request from the upper layer (for example, the application on top of the MAC or the Network layer), which requires the cooperation of the transceiver, is received. This could be the request to transmit a new frame (wpan_mcps_data_req()), the request to set a PIB attribute that is mirrored in transceiver registers, the request to poll for pending data at the node's parent, etc.

Once such a request has been received at the MAC-API, the MAC performs a check whether the radio is currently in SLEEP state and wakes up the transceiver if required. Afterwards the MAC can us the transceiver to perform whichever action is required. After the ongoing transaction is finished, the MAC will put the transceiver back to SLEEP if allowed.

This complete MAC controlled power management implies that a node being in SLEEP state will be not able to receive any frame during, that is, a PAN Coordinator would not be able to send a direct frame to its sleeping child being an End Device. This can be reproduced using the provided MAC example applications nobeacon (see Section 11.2.1.1). In both of these applications all directed traffic goes from the End Device to the PAN Coordinator. If these applications where changed so that the PAN Coordinator sent traffic to the End Device, the End Device would not accept these frames, since it is in SLEEP most of the time.

Of course there is variety of means defined within IEEE 802.15.4 to allow a reasonable communication between mains powered nodes and battery powered nodes (applying power management) in both directions. This is explained in more detail in the following section. One way to allow communication from the PAN Coordinator to the End Device is to use a timer which wakes up the End Device periodically, allowing this device to receive frames during this time period from other nodes. This is implemented in MAC example nobeacon_sleep (see Section 11.2.1.3).

6.2 Reception of data at nodes applying power management

When data shall be received by an End Device (that is, a node applying MAC power management as default), we have several options as described in the subsequent sections.

6.2.1 Setting of macRxOnWhenIdle to true

One option is to leave the receiver of the node always on, so the device is able to always receive frames. This can be done be setting the MAC PIB Attribute macRxOnWhenIdle to True (1) (by using the API function wpan_mlme_set_req()) at any time. This will immediately wake up the radio and enable the receiver of the node.

If this scheme is applied, the node will not enter SLEEP state anymore and thus uses it’s battery power very extensively. So for battery powered End Devices this is not recommended, although it might be an easy solution for mains powered End Devices.

6.2.2 Enabling the receiver

If a PAN Coordinator wants to send data to an End Device periodically, the application of the Device can be implemented as such, that the Device maintains a timer with the same time interval that the Coordinator wants to transmit its data to the Device.

Upon expiration of this timer the application of the Device can then enable its receiver for a certain amount of time or until it receives data from the Coordinator, and turn off the receiver again. The receiver can be enabled or disabled by using the MAC-API function “wpan_mlme_rx_enable_req”.

The parameter RxOnDuration contains the value (number of symbols, for example, one symbols is 16µs for 2.4GHz networks) that the receiver shall be enabled. If this parameter is zero (0), the receiver of the node will be disabled.

The parameters DeferPermit and RxOnTime shall be set to 0 for a nonbeacon-enabled network, since these parameters are obsolete for nonbeacon-enabled networks.

If this scheme is used, handling of power management is done by the device itself. When the RxEnable timer expires, or if parameter RxOnDuration is zero, the MAC will initiate its standard power management scheme and put the transceiver again into SLEEP if allowed (that is, in case macRxOnWhenIdle is False).

6.2.3 Handshake between end device and coordinator

Another scheme is based on a combination of enabling the receiver in conjunction with a handshake scheme between End Device and the PAN Coordinator.

The End Device enables its receiver periodically (as in Section 6.2.2), and sends a data frame to the Coordinator indicating it is alive for a certain amount of time. The Coordinator in return either answers with direct data to the Device directly (in case it has something to deliver) to the Device, or simply does nothing.

After the device has received a frame from the PAN Coordinator the End Device disables its receiver again. This can be done by simply letting the RxEnable timer expire (depending on the original value of RxOnDuration" at the first call of wpan_mlme_rx_enable_req") by directly calling wpan_mlme_rx_enable_req" with parameter RxOnDuration" set to zero.

If the End Device does not receive a frame from its Coordinator, the device automatically goes to sleep depending on the original value of RxOnDuration" at the first call of wpan_mlme_rx_enable_req".

If this scheme is used, handling of power management is done by the device itself. When the RxEnable timer expires, or if parameter RxOnDuration is zero, the MAC will initiate its standard power management scheme and put the transceiver again into SLEEP if allowed (that is, in case macRxOnWhenIdle is False).

6.2.4 Indirect transmission from coordinator to end device

Another option is to use indirect data transmission from the Coordinator to the End Device and let the Device poll the Coordinator periodically for pending data. When this scheme is applied the Coordinator sets the parameter TxOptions of the API function:

```
bool wpan_mcps_data_req(uint8_t SrcAddrMode,
                        wpan_addr_spec_t *DstAddrSpec,
                        uint8_t msduLength,
                        uint8_t *msdu,
                        uint8_t msduHandle,
                        uint8_t TxOptions)
```

to WPAN_TXOPT_INIDIRECT_ACK (indirect, but acknowledged Transmission; value 5), instead of WPAN_TXOPT_ACK (direct, acknowledged Transmission value 1) as it is currently (see file mac_api.h in directory MAC/Inc).

NOTE This requires an additional check which device type the node is currently, since data from the Device to the Coordinator is still only transmitted directly.

Additionally the Device needs to implement a polling scheme in its application, during which it periodically calls function wpan_mlme_poll_req(). This will initiate a data request frame to the Coordinator. In case the Coordinator does have pending data for the Device, it initiates the direct transmission of those frames. Otherwise Coordinator sends a null data frame (data frame with empty payload). The Device both receives the response from the Coordinator and, if there is no further action to be done, returns to standard power management procedures. For more information see Sections 5.4.1 and 5.4.2.

If this scheme is used, handling of power management is done by the device itself.

6.3 Application control of MAC power management

As indicated throughout the previous sections there are several means for the application to control the general power management scheme applied by the MAC.

These are setting the MAC PIB attribute macRxOnWhenIdle and the MAC primitive MLME_RX_ENABLE.request.

6.3.1 MAC PIB attribute macRxOnWhenIdle

Setting the MAC PIB attribute to a specific value controls the handling of MAC power management. Whenever a transaction within the MAC has finished (for example, transmitting a frame, setting of PIB attributes residing within the transceiver, etc.) the MAC checks this PIB attribute. If the corresponding value is False, the radio enters SLEEP mode again, otherwise the transceiver stays awake.

Any node will always enter SLEEP mode after each finished transition, since the PIB attribute macRxOnWhenIdle is False as default.

If this behavior shall be altered (especially for Coordinators or PAN Coordinators), the application needs to change the value of macRxOnWhenIdle to True after whenever this shall be applied.

The current value of the MAC PIB attribute macRxOnWhenIdle can be altered by calling function wpan_mlme_set_req() with the appropriate value (see file *main.c* of example in Section 11.2.1.2):

```
/* Switch receiver on to receive frame. */
wpan_mlme_set_req(macRxOnWhenIdle, true);
```

or

```
/* Switch receiver off. */
wpan_mlme_set_req(macRxOnWhenIdle, false);
```

Please note that the receiver will be immediately enabled or disabled.

6.3.2 Handling the receiver with wpan_rx_enable_req()

While setting of the PIB attribute macRxOnWhenIdle is a more “globally” or “statically” applied means to change the standard MAC power management scheme, the MAC primitive MLME_RX_ENABLE.request can be used to change the behavior more temporarily.

The receiver can be enabled by calling function wpan_mlme_rx_enable_req() with (the 3rd parameter) RxOnDuration larger than zero:

```
/* Switch receiver on for 10000 symbols. */
wpan_mlme_rx_enable_req(false, 0, 10000);
```

This wakes up the transceiver (independent from the current value of the PIB attribute macRxOnWhenIdle) if required and switches the receiver on.

After the timer with the specified time (RxOnDuration symbols) expires, the receiver is disabled automatically if the current value of macRxOnWhenIdle is false, or remains in receive mode if macRxOnWhenIdle is true.

The receiver can be disabled explicitly by calling wpan_mlme_rx_enable_req() with (the 3rd parameter) RxOnDuration equal to zero:

```
/* Disable receiver now. */
```

```
wpan_mlme_rx_enable_req(false, 0, 0);
```

This disables the receiver if the current value of the PIB attribute macRxOnWhenIdle is false and puts the transceiver to SLEEP mode.

For more information about function wpan_mlme_rx_enable_req() see files *mac_api.c* and *mac_rx_enable.c* in directory mac/src.

For more information about the interaction of MLME_RX_ENABLE and macRxOnWhenIdle see file *mac_misc.c* in directory mac/src and check the following function mac_sleep_trans().

6.4 TAL power management API

The MAC power management mechanisms as described in this chapter are only valid if the application is residing on top of the MAC layer (or a higher layer on top of the MAC). In case the application is residing on top of the TAL layer, the application needs to take care for transceiver power management itself.

Therefore the TAL provides a Power Management API which is generally used by the MAC layer but can also be used by an application residing on top of the TAL. Nevertheless an application residing on top of any higher layer than the TAL must not use this TAL API explicitly, otherwise this may lead to undefined behavior.

The TAL Power Management API consists of two functions:

- tal_trx_sleep()
- tal_trx_wakeup()

For more information see file *tal.h* in directory tal/inc and the various implementations for each transceiver in file *tal_pwr_mgmt.c* in directories tal/*TAL_TYPE_NAME*/src.

7 Application and stack configuration

The MAC and its modules are highly configurable to adapt to the application requirements. The utilized resources can be configured and adjusted according to the application needs. This allows a drastic footprint reduction.

During build process the required features are included depending on the used build switches and basic configuration type.

Qualitative configuration includes:

- Features or modules can be included to or excluded from the firmware using build switches
- The build configuration is controlled by switches within Makefiles or the project files
- Primitives as defined within IEEE 802.15.4 (and thus features within the MAC) can be included or excluded depending on the required degree of standard compliance or application needs
- Two generic profiles are provided: RFD and FFD primitive configuration

Quantitative configuration includes:

- Resources can be adjusted to the application needs
- The file app_config.h (usually located in the inc path of the applications) provides hooks for the application configuration to configure its own resources
- Each demonstration application comes with its own configuration file (app_config.h) that can be re-used for own application design

7.1 Build switches

The stack and application based on the stack can be highly configured according to the end user application needs. This requires a variety of build switches to be set appropriately. The following section describes that build switches may be used during the build process.

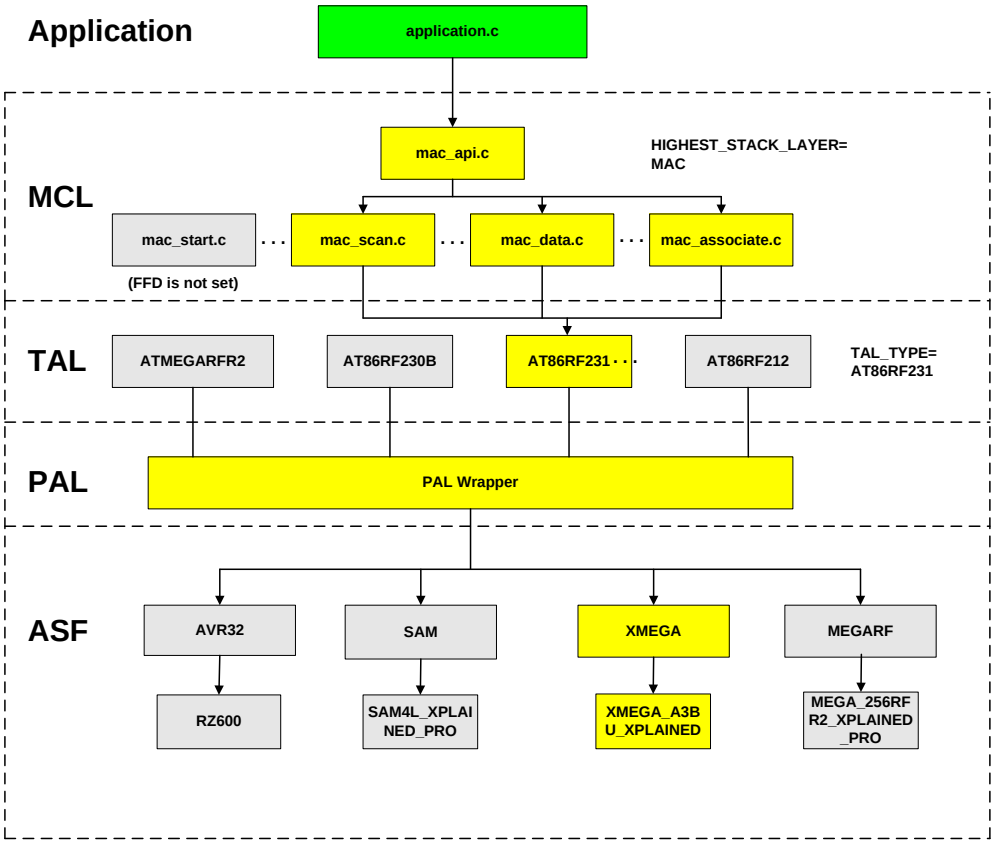
The switches may be categorized as follows:

1. Global stack switches
    - HIGHEST_STACK_LAYER
    - REDUCED_PARAM_CHECK
    - PROMISCUOUS_MODE
    - ENABLE_TSTAMP
2. Standard or user build configuration switches
    - BEACON_SUPPORT
    - FFD
    - MAC_USER_BUILD_CONFIG
3. Platform switches
    - BOARD (Values provided in ASF common sources)
    - ENABLE_TRX_SRAM
    - EXTERN_EEPROM_AVAILABLE
    - NON_BLOCKING_SPI
    - VENDOR_STACK_CONFIG

4. Transceiver specific switches
    - TAL_TYPE
    - ENABLE_TFA
    - TFA_BAT_MON
    - HIGH_DATA_RATE_SUPPORT
    - CHINESE_BAND
    - RSSI_TO_LQI_MAPPING
    - ENABLE_FTN_PLL_CALIBRATION
    - DISABLE_IEEE_ADDR_CHECK
5. Test and Debug switches
    - _DEBUG_
    - TEST_HARNESS

The following picture shows an example how the various build switches lead to a particular configuration. For simplicity reasons only the basic building blocks are included. More advance blocks (for example, TFA or STB have been omitted). The used build switches are explained in the subsequent sections.

Figure 7-1. Build configuration example.



7.1.1 Global stack switches

7.1.1.1 HIGHEST_STACK_LAYER

This build switch defines the layer that the end user application is actually based on. The MAC software package comprises of three real layers (from bottom: PAL, TAL, and MAC Core Layer). All these layers forms the stack, although not necessarily all layers are always part of the actual binary. Depending upon the required functionality (full blown MAC versus simple data pump), the user application needs to define which layer it shall be based on (that is, which API it is using). If an application for instance only uses the PAL and TAL layers (that is, MAC is not used), all resources required for MAC are not part of the final application. This reduces code size and SRAM utilization drastically.

Also, if a Network Layer (NWK) will be part of the stack and residing on top of the MAC layer, a number of resources are used differently compared to an application on top of the MAC.

Example: Reading of PIB attributes residing in TAL layer.

If the HIGHEST_STACK_LAYER is MAC (HIGHEST_STACK_LAYER = MAC), the function tal_pib_get() in file tal_pib.c is not included in the binary, because the MAC reads all PIB attributes residing in TAL directly by accessing the global variables. On the other hand, if the HIGHEST_STACK_LAYER is TAL (HIGHEST_STACK_LAYER = TAL) this function is available, because an application (being not part of the stack) shall not access global variables of the stack directly.

Conclusion:

If the application sits on top of the MAC layer, this build switch shall be set to „HIGHEST_STACK_LAYER = MAC“.

If the application sits on top of the TAL layer, this build switch shall be set to „HIGHEST_STACK_LAYER = TAL“.

Usage in Makefiles:

```
CFLAGS += -DHIGHEST_STACK_LAYER=MAC
```

or

```
CFLAGS += -DHIGHEST_STACK_LAYER=TAL
```

Usage in IAR ewp-files:

```
HIGHEST_STACK_LAYER=MAC
```

or

```
HIGHEST_STACK_LAYER=TAL
```

For more information check file include/*stack_config.h*. This shows how the resources are used and included in the end application depending upon the highest stack used.

7.1.1.2 REDUCED_PARAM_CHECK

Whenever an application or a higher layer accesses an API function of a lower layer usually a variety of parameter checks are done in order to ensure proper usage of the desired functionality. This leads to a more robust application. On the other hand, if the higher layer is designed to always call an API function with reasonable parameter values, this build switch might be omitted. This will reduce the code size.

Please refer the file mac\src\mac_mcps_data.c how a build switch is used. In function mcps_data_request() a number of additional checks are performed if this switch is not set.

It is strongly recommended to disable the following build switch at least during the development cycle of the application.

Usage in Makefiles:

```
CFLAGS += -DREDUCED_PARAM_CHECK
```

disables the additional parameter checking.

Usage in IAR ewp-files:

```
REDUCED_PARAM_CHECK
```

reduces the additional parameter checking.

7.1.1.3 PROMISCUOUS_MODE

This build switch allows for the creation of a special node that can be put into promiscuous mode thus allowing it to act as a very simple frame sniffer. It can be used as a very simple network diagnostic tool.

When this switch is enabled the node is not supposed to act as a standard node being part of network, that is, the node will never acknowledge any received frame, etc. Instead the node will be able to receive any proper IEEE 802.15.4 frame on its channel within range and present it to the application. The application can reside on top of the MAC layer (using MAC-API callback function *usr_mcps_data_ind()*) or on top of the TAL layer (using TAL callbacks).

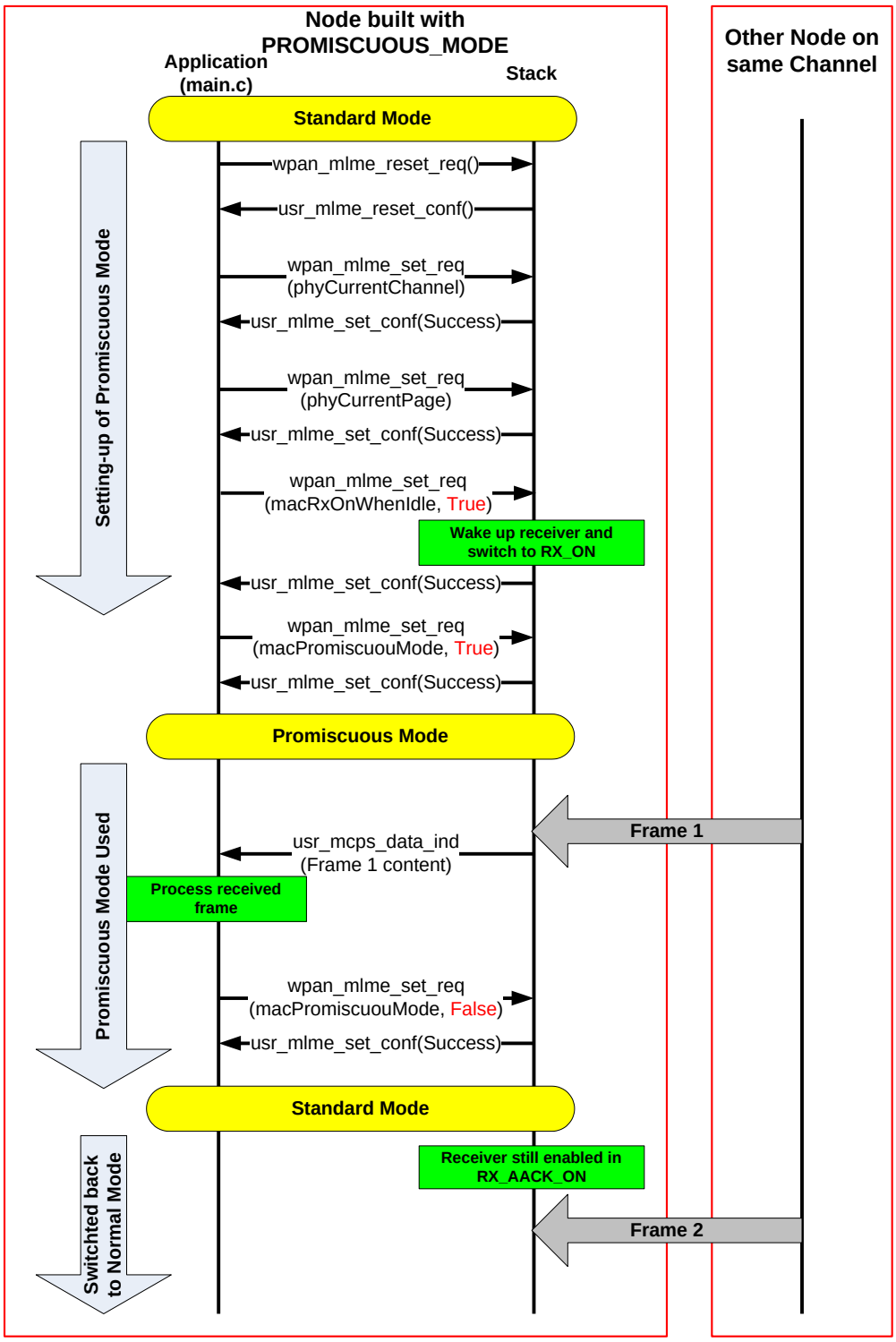
Switching promiscuous mode on or off is controlled by the standard MAC PIB macPromiscuousMode (see IEEE 802.15.4-2006, Section 7.5.6.5 Promiscuous Mode). The payload contained in the usr_mcps_data_ind() callback function is the MAC Header (MHR) of the received frame concatenated with the original payload of the received frame.

Promiscuous mode can be switched on or off by setting or resetting the PIB attribute macPromiscuousMode. If the radio is awake and in receive mode on the current channel, the node will present all received frames to the upper layer. In order to work properly, the receiver needs to be enabled. This can be done, for example, by setting MAC PIB attribute macRxOnWhenIdle to 1 before turning promiscuous mode on. Generally the transceiver state can be controlled similar to normal operation (see Chapter 6).

Once the PIB attribute macPromiscuousMode is reset, the node switches back to normal operation. It will be in the same state as before switching to promiscuous mode, for example, a node that was associated will still be connected to the same network.

The following picture indicates the proper handling of promiscuous mode.

Figure 7-2. Handling of promiscuous mode.



7.1.1.4 ENABLE_TSTAMP

This build switch allows creation of timestamping information throughout the entire MAC stack. This includes two different angles:

- Generation of timestamping information within the TAL

- Inclusion of timestamp information in the MAC-API primitives (see function usr_mcps_data_conf() and usr_mcps_data_ind() in file *mac/inc/mac_api.h*)

In case timestamping information shall be generated and included in the MAC-API for further utilization within the application, the compile switch needs to be set.

Usage in Makefiles:

```
CFLAGS += -DENABLE_TSTAMP
```

enables the timestamping.

Usage in IAR ewp-files:

```
ENABLE_TSTAMP
```

enables the timestamping.

7.1.2 Standard and user build configuration switches

The standard and user build configuration switches are described in detail in Sections 7.2.1 and 7.2.2.

7.1.3 Platform switches

Since the PAL layer is part of ASF, please refer [9] for more details on Platform switches.

7.1.4 Transceiver specific switches

7.1.4.1 TAL_TYPE

The TAL (Transceiver Abstraction Layer) contains all transceiver based functionality and provides an API to the MAC which is independent from the underlying transceiver. Certain functionality that for instance the MAC or an application may require is dependent from the actual used transceiver chip. Examples are the utilization of antenna diversity, time stamping mechanisms, or the automatic CRC calculation in hardware.

Examples of currently supported transceivers are:

- Atmel Atmega256RFR2
- Atmel AT86RF230B (AT86RF230 Revision B)
- Atmel AT86RF231
- Atmel AT86RF212
- Atmel AT86RF212B
- Atmel AT86RF233

For more information please check the tal\inc*tal_types.h* file.

Usage in Makefiles:

```
CFLAGS += -DTAL_TYPE=AT86RF212
```

Selects the AT86RF212 transceiver.

Usage in IAR ewp-files:

```
TAL_TYPE=AT86RF231
```

selects the AT86RF231 transceiver.

7.1.4.2 ENABLE_TFA

This build switch enables the usage of non-standard compliant features of the transceiver based on the block Transceiver Feature Access (TFA).

Currently the following features are implemented in the TFA:

- Changing of Receiver Sensitivity
- Perform CCA
- Perform ED (Energy Detect) measurement
- Reading the current transceiver supply voltage (transceiver battery monitor). Reading the current transceiver supply voltage (transceiver battery monitor). The reading of the supply voltage can also be enabled separately by setting the build switch TFA_BAT_MON
- Reading of current temperature (only Atmel ATmega256RFR2 only)

CCA and ED measurement are an inherent part of the MAC/TAL, so there is no need for a standard application to use this. On the other hand there could be special test applications which might use such functionality.

If only standard defined behavior is required or code size is important this switch should not be set.

Usage in Makefiles:

```
CFLAGS += -DENABLE_TFA
```

enables the usage of the TFA.

Usage in IAR ew-files:

```
ENABLE_TFA
```

enables the usage of the TFA.

7.1.4.3 HIGH_DATA_RATE_SUPPORT

All 802.15.4 Atmel transceivers supported with this software package beyond AT86RF230 provide high data modes not defined within the IEEE 802.15.4 standard. In order to enable these high speed transmission modes, the build switch HIGH_DATA_RATE_SUPPORT needs to be set. An example application where this switch is used to gain a significantly higher data throughput is the Performance Analyzer application (see avr2025_mac\apps\tal\performance_analyzer). If only standard rates are used or code size is important this switch should not be set.

Usage in Makefiles:

```
CFLAGS += -DHIGH_DATA_RATE_SUPPORT
```

enables support of high speed data rates.

Usage in IAR ewp-files:

```
HIGH_DATA_RATE_SUPPORT
```

enables support of high speed data rates.

7.1.4.4 CHINESE_BAND

This build switch is used in conjunction with a Sub-GHz transceiver chip that is capable of working properly within the Chinese 780MHz radio band (for example, the Atmel AT86RF212). Within the stack this switch is solely used to set the proper

default value for the channel page. So any application that per default is required to operate within this particular band (and uses the proper TAL type) may use this build switch to set the channel page to a proper default value. This switch currently not used in any example application.

Usage in Makefiles:

```
CFLAGS += -DCHINESE_BAND
```

enables the channel page for the Chinese band as default.

Usage in IAR ewp-files:

```
CHINESE_BAND
```

enables the channel page for the Chinese band as default.

7.1.4.5 RSSI_TO_LQI_MAPPING

This build switch is used to control the mechanism for calculation of the normalized LQI value of received frames. The normalized LQI value (that is provided to the higher layer of the MAC as parameter *ppduLinkQuality*) is

- based on the RSSI/ED value (switch RSSI_TO_LQI_MAPPING is used) where only the ED value (signal strength) is mapped to a LQI value, or is
- based on the ED (signal strength) and the measured LQI value (quality of received packet) (switch RSSI_TO_LQI_MAPPING is not used)

For further information about the LQI value, see IEEE 802.15.4-2006 Section 6.9.8. The build switch RSSI_TO_LQI_MAPPING reflects the “and/or” relation described in the first paragraph of the mentioned section. If RSSI_TO_LQI_MAPPING is set, signal strength is only used for LQI measurement.

Usage in Makefiles:

```
CFLAGS += -DRSSI_TO_LQI_MAPPING
```

enables the calculation of the normalized LQI value based on RSSI/ED value.

Usage in IAR ewp-files:

```
RSSI_TO_LQI_MAPPING
```

enables the calculation of the normalized LQI value based on RSSI/ED value.

For more information please check the implementation in the files tal/*tal_type*/Src/*tal_rx.c.*

7.1.4.6 ENABLE_FTN_PLL_CALIBRATION

This build switch is used to enable the filter tuning and PLL calibration for the transceiver. Once this feature is enabled in an application, a period timer is started, which ensures that the proper filter tuning and PLL calibration is executed after a certain amount of time.

This feature might be required in case the environment conditions (that is, temperature) vary over time.

The filter tuning and PLL calibration is done automatically when a node periodically enters sleep state, or when other specific state changes are performed within the transceiver periodically. For more information please check the corresponding transceiver data sheets.

The current timer interval is five minutes, that is, whenever the node does not enter sleep state within this timer interval, the filter tuning and PLL calibration mechanism will be invoked.

This switch is currently not enabled in any example application.

7.1.4.7 DISABLE_IEEE_ADDR_CHECK

This build switch is used to disable the check whether the node has a valid IEEE address (that is, the IEEE address is different from 0 or 0xFFFFFFFFFFFFFFFF).

Currently all TAL and MAC based applications require a valid and unique IEEE address being present on each node. Since some board may not necessarily have a valid IEEE address stored in (either internal or external) EEPROM (for example Atmel AT91SAM7X-EK boards), the applications cannot run properly. For the purpose of proper example demonstration a check for this IEEE address is implemented. If the IEEE address is not correct (that is, it is 0 or 0xFFFFFFFFFFFFFFFF), a random IEEE address is assigned to this node.

Some applications do not require this specific IEEE check. In this case the code can be significantly smaller by setting the build switch DISABLE_IEEE_ADDR_CHECK.

Also this build switch gives a very good indication which portions or the TAL need to be changed or removed for smaller code size by simply searching for build switch DISABLE_IEEE_ADDR_CHECK.

This switch is currently not enabled in any example application, since all example applications require a valid IEEE address.

Boards providing a valid IEEE address always use their original unique IEEE address during operation.

7.1.4.8 DISABLE_TSTAMP_IRQ

This build switch is used to disable the Timestamp interrupt (that is, the second transceiver interrupt for Atmel AT86RF231, Atmel AT86RF233 and Atmel AT86RF212) on systems which do not utilize this transceiver interrupt. This is for example valid for boards based on Atmel Xmega MCUs in conjunction with AT86RF231.

The TAL for AT86RF231, AT86RF233 and AT86RF212 is designed to utilize the Timestamp interrupt as default for generating timestamp information. If this build switch is explicitly set in the project files or Makefiles for an application, the timestamp information is generated similar as for Atmel AT86RF230 based systems.

7.1.4.9 TRX_REG_RAW_VALUE

This build switch is used to disable the scaling of ED value.By default ED value read from the RG_PHY_ED_LEVEL register shall be scaled using CLIP_VALUE REG value.By using this buid switch, this can be avoided and raw value of ED sall be used.

7.1.4.10 SW_CONTROLLED_CSMA

This build switch SW_CONTROLLED_CSMA includes functions to the build process that control the CSMA-CA algorithm by software. CCA, backoffs and re-transmissions are controlled by software while the transceiver handles frame validation (e.g. CRC and frame filtering) and ACK transmission using its automatic (extended) modes.

7.1.4.11 TX_OCTET_COUNTER

This build switch TX_OCTET_COUNTER includes functions to the build process that count the number of actually sent bytes over-the-air. The numbers of bytes are accumulated in the variable tal_tx_octet_cnt. This includes data frames that are sent “actively” and ACK frames that are sent due to received frames including preamble, SFD and length field.
The application can reset and read the tal_tx_octect_cnt variable. It can be used to calculate an actual duty-cycle. In order to use the TX_OCTET_COUNTER switch the build switch SW_CONTROLLED_CSMA needs to be set as well

7.1.4.12 TX RX_WHILE_BACKOFF

The build switch RX_WHILE_BACKOFF enables switching the receiver on during backoff periods of the CSMA algorithm. If the build switch RX_WHILE_BACKOFF is not defined for the build, the receiver remains off during backoff periods. In order to use the RX_WHILE_BACKOFF switch the build switch SW_CONTROLLED_CSMA needs to be set as well.

If the receiver is switched on during backoff periods and a frame is received (including automatic ACK transmission), the received frames get queue into the receive frame queue. After frame reception the CSMA algorithm is continued with the next CSMA attempt.

Impacts with RX_WHILE_BACKOFF

- In a “normal” scenario (low traffic) the maximum data throughput is reduced by about 2 percent (at 250kb/s and about 1 percent at 100kb/s) due to the software-controlled CSMA using a SPI-based transceiver.
- In high density networks the outgoing data throughput highly varies based on the number of received frames during the backoff periods.
- The implementation of the software-controlled CSMA algorithm requires about 0.5 kB (0.5kB for Xmega; 0.8kB for SAM4L) program code.
- An additional (software) timer is required and the MCU utilization is higher during the actual CSMA procedure.

7.1.5 Test and debug switches

7.1.5.1 DEBUG

This build switch enables further debug functionality and additional sanity checks within the software package, but also increased code size or changes run time behavior since the optimization level is changed.

If the GCC compiler is used, this switch also enables printout functionality for debug purposes (ASSERT).

7.1.5.2 TEST_HARNESS

This build switch is solely present for Atmel internal regression testing and not to be used by an application.

NOTE Although the code in file mac_pib.c may lead to the conclusion, that this switch needs to be set in order to be allowed to set the IEEE address of this node via software, this is not supposed to be used this way.

Each device has its own IEEE address that is fixed for this device and usually it is located in any type of persistent storage. This is actually not a PIB attribute but rather a MAC constant (see IEEE 802.15.4-2006 Section 7.4.1, Table 85 (aExtendedAddress)). So the value is only READ_ONLY and in normal mode not supposed to be written by the application or stack. Therefore this attribute can be read using API functions (wpan_mlme_get_req()) but not written (wpan_mlme_set_req()).

7.2 Build configurations

7.2.1 Standard build configurations

Based on the IEEE 802.15.4 standard there are four basic standard configurations available which provide different functionality to the user application. This implies different APIs for each of these configurations and also different footprints (codes size, SRAM utilization).

These standard configurations can be classified in two categories:

- Support of beacon enabled networks
- Reduced Functional Devices - RFD (for example, End Devices) vs. Full Functional Devices – FFD (PAN Coordinators, Coordinators)

The Atmel MAC provides two build switches that can be used in Makefiles or IAR Embedded Workbench project files to enable or disable these configurations. Depending on the usage of these switches in the project, the MAC provides certain functionality to the user application.

I. FFD
   - Omission of this build switch only enables functionality required for a simple RFD node
   - Setting of this build switch additionally enables functionality required for a FFD node

II. BEACON_SUPPORT
   - Omission of this build switch only enables functionality required for a networks without using a superframe structure, that is, non-beacon enabled networks
   - Setting of this build switch additionally enables functionality required for a network using a superframe structure, that is, beacon-enabled networks

Example 1: A node that shall start its own network always has to be an FFD, because starting of networks is only supported for an FFD configuration.

Example 2: A node that shall be able to join both non-beacon and beacon-enabled networks has to use an application built with BEACON_SUPPORT.

Please refer to file “/include/mac_build_config.h” for more information about the supported functionality.

7.2.1.1 FFD feature set

The following features are enabled if FFD is set during the build process:

- MAC_ASSOCIATION_INDICATION_RESPONSE
  The node is able to accept and process association attempts from other nodes. Also this node can provide Short Addresses to other nodes if desired

- MAC_ASSOCIATION_REQUEST_CONFIRM
  The node is able accept and process a request from its upper layer (for example, the network layer) to associate itself to another node (that is, its parent)
- MAC_BEACON_NOTIFY_INDICATION
  The node is able to present received beacon frame to its upper layer in case the beacon frame contains a beacon payload or the MAC PIB attribute macAutoRequest is set to false
- MAC_DISASSOCIATION_BASIC_SUPPORT
  The node is able to accept and process a request from its upper layer to disassociate itself from its network or disassociate one of its children, or to process a received disassociation frame from another node
- MAC_DISASSOCIATION_FFD_SUPPORT
  The node is able to transmit an indirect disassociation notification frame. This requires that the switch MAC_DISASSOCIATION_BASIC_SUPPORT is also set
- MAC_INDIRECT_DATA_BASIC
  The node is able to poll its own parent for indirect data
- MAC_INDIRECT_DATA_FFD
  The node is able to handle requests to transmit data frames to its children indirectly once being polled by those nodes. This requires that the switch MAC_INDIRECT_DATA_BASIC is also set
- MAC_ORPHAN_INDICATION_RESPONSE
  The node is able to accept and process a received orphan indication frame by one of its children and respond appropriately
- MAC_PAN_ID_CONFLICT_AS_PC
  The node is able to detect a PAN-Id conflict situation while acting as a PAN Coordinator by checking received beacon frames from other PAN Coordinators and being able to act upon the reception of PAN-Id Conflict Notification Command frames from its children
- MAC_PAN_ID_CONFLICT_NON_PC
  The node is able to detect a PAN-Id conflict situation while NOT acting as a PAN Coordinator by checking received beacon frames from other PAN Coordinators and being able to initiate the transmission of PAN-Id Conflict Notification Command frames from its parents if required
- MAC_PURGE_REQUEST_CONFIRM
  The node is able to purge indirect data frames from its Indirect-Data-Queue upon request by its upper layer
- MAC_RX_ENABLE_SUPPORT
  The node is able to switch on or off its receiver for a certain amount of time upon request by its upper layer. This is required in order to allow for the upper layer to receive frames in case the node is generally in a power safe state
- MAC_SCAN_ACTIVE_REQUEST_CONFIRM
  The node is able to perform an active scan to search for existing networks
- MAC_SCAN_ED_REQUEST_CONFIRM
  The node is able to perform an energy detect scan
- MAC_SCAN_ORPHAN_REQUEST_CONFIRM
  The node is able to perform an orphan scan in case it has lost its parent
- MAC_SCAN_PASSIVE_REQUEST_CONFIRM
  The node is able to perform a passive scan to search for existing networks. This feature is only enabled if also BEACON_SUPPORT is enabled

- MAC_START_REQUEST_CONFIRM
  The node is able to start its own network. Depending on the setting of BEACON_SUPPORT this can be either only a non-beacon enabled network or also a beacon-enabled network
- MAC_SYNC_LOSS_INDICATION
  The node is able to report a sync loss condition to its upper layer. This can be either the reception of a coordinator realignment frame from its parent, or (if BEACON_SUPPORT is enabled and the node is synchronized with its parent) caused by the fact that the node has not received beacon frames from its parent for a certain amount of time

Please check IEEE 802.15.4-2006 for further information about the MAC primitives and the implementation of their corresponding features in the MAC.

7.2.1.2 RFD feature set

The following features are enabled if FFD is NOT set during the build process:

- MAC_ASSOCIATION_REQUEST_CONFIRM
  The node is able accept and process a request from its upper layer (for example, the network layer) to associate itself to another node (that is, its parent)
- MAC_BEACON_NOTIFY_INDICATION
  The node is able to present received beacon frame to its upper layer in case the beacon frame contains a beacon payload or the MAC PIB attribute macAutoRequest is set to false
- MAC_DISASSOCIATION_BASIC_SUPPORT
  The node is able to accept and process a request from its upper layer to disassociate itself from its network, or to process a received disassociation frame from its parent
- MAC_INDIRECT_DATA_BASIC
  The node is able to poll its own parent for indirect data
- MAC_PAN_ID_CONFLICT_NON_PC
  The node is able to detect a PAN-Id conflict situation while NOT acting as a PAN Coordinator by checking received beacon frames from other PAN Coordinators and being able to initiate the transmission of PAN-Id Conflict Notification Command frames from its parents if required
- MAC_RX_ENABLE_SUPPORT
  The node is able to switch on or off its receiver for a certain amount of time upon request by its upper layer. This is required in order to allow for the upper layer to receive frames in case the node is generally in a power safe state
- MAC_SCAN_ACTIVE_REQUEST_CONFIRM
  The node is able to perform an active scan to search for existing networks
- MAC_SCAN_ORPHAN_REQUEST_CONFIRM
  The node is able to perform an orphan scan in case it has lost its parent
- MAC_SYNC_LOSS_INDICATION
  The node is able to report a sync loss condition to its upper layer. This can be either the reception of a coordinator realignment frame from its parent, or (if BEACON_SUPPORT is enabled and the node is synchronized with its parent) caused by the fact that the node has not received beacon frames from its parent for a certain amount of time

The following features are disabled if FFD is NOT set during the build process:

- MAC_ASSOCIATION_INDICATION_RESPONSE
  The node is not able to handle association attempts from other nodes
- MAC_DISASSOCIATION_FFD_SUPPORT
  The node is not able to transmit an indirect disassociation notification frame
- MAC_INDIRECT_DATA_FFD
  The node is not able to handle requests to transmit data frames indirectly
- MAC_ORPHAN_INDICATION_RESPONSE
  The node is not able to handle orphan indication frames by other nodes
- MAC_PAN_ID_CONFLICT_AS_PC
  The node is not able to act upon the reception of PAN-Id Conflict Notification Command frames
- MAC_PURGE_REQUEST_CONFIRM
  The node is not able to purge indirect data
- MAC_SCAN_ED_REQUEST_CONFIRM
  The node is not able to perform an energy detect scan
- MAC_SCAN_PASSIVE_REQUEST_CONFIRM
  The node is not able to perform a passive scan
- MAC_START_REQUEST_CONFIRM
  The node is not able to start its own network

Please check IEEE 802.15.4-2006 for further information about the MAC primitives and the implementation of their corresponding features in the MAC.

7.2.1.3 BEACON_SUPPORT feature set

If BEACON_SUPPORT is set during the build process all functionality required to support beacon-enabled networks are enabled. The actually enabled functionality differs depending on the internal requirements for FFDs or RFDs.

Additionally the following feature is enabled:

- MAC_SYNC_REQUEST
  The node is able to sync itself with its parent by tracking the corresponding beacon frames

Please check IEEE 802.15.4-2006 for further information about the MAC primitives and the implementation of their corresponding features in the MAC.

Please check IEEE 802.15.4-2006 for further information about the MAC primitives and the implementation of their corresponding features in the MAC.

7.2.2 User build configurations – MAC_USER_BUILD_CONFIG

7.2.2.1 Introduction

Since a number of applications do not necessarily need the functionality provided by any of the standard build configurations, or may even have more rigid requirements concerning FLASH or RAM utilization, the concept of user build configuration has been introduced. The usage of the build MAC_USER_BUILD_CONFIG in the Makefiles or IAR Embedded Workbench project files allows the end user to tailor its MAC completely according to its own needs.

The following MAC features can be separately selected or removed from the build:

- Association

- Disassociation
- Support of scanning (energy detect, active, passive, or/and orphan scanning)
- Starting of networks
- Support of transmitting or receiving indirect data including polling of data
- Purging of indirect data
- Enabling of the receiver
- Synchronization in beacon-enabled networks
- Presentation of loss of synchronization
- Handling of orphan notifications
- Handling of beacon notifications

Each feature can be used independently from each other. It is also possible to deselect all of the above features, which leads to a minimum application in terms of resource utilization. In this case only the following basic MAC features are available:

- Direct data transmission and reception
- Initiation of a MAC reset
- Reading and writing of PIB attributes

An example application implementing the proper utilization of this feature can be found in avr2025_mac/examples/mac/no_beacon. In this example only RX-ENABLE is used in addition to the standard features. For more information about this example application please refer to Section 11.2.1.1.

7.2.2.2 File mac_user_build_config.h

If the switch MAC_USER_BUILD_CONFIG is activated, the C-pre-processor looks for a file mac_user_build_config.h in the current include path (usually in the Inc directory of the application). See file /Include/mac_build_config.h:

```
#ifdef MAC_USER_BUILD_CONFIG
#include "mac_user_build_config.h"
#else
...
```

The standard feature definitions for an FFD or RFD configuration are by-passed, and instead the user defined feature set from mac_user_build_config.h is used. This features set needs to be defined entirely, that is, each feature needs to be either enabled or disabled.

7.2.2.2.1 Examples

Example 1: An end device that does neither use association nor disassociation functionality, but still wants to poll indirect data from its parent, may set the following the build switches in file mac_user_build_config.h:

```
#define MAC_ASSOCIATION_INDICATION_RESPONSE     (0)
#define MAC_ASSOCIATION_REQUEST_CONFIRM         (0)
#define MAC_DISASSOCIATION_BASIC_SUPPORT        (0)
#define MAC_DISASSOCIATION_FFD_SUPPORT          (0)
#define MAC_INDIRECT_DATA_BASIC                 (1)
#define MAC_INDIRECT_DATA_FFD                   (0)
```

Example 2: A network whose nodes read their fixed network parameters from a persistent store, and thus never perform scanning or start a network, may set the following build switches in file mac_user_build_config.h:

```
#define MAC_SCAN_ACTIVE_REQUEST_CONFIRM          (0)
#define MAC_SCAN_ED_REQUEST_CONFIRM              (0)
#define MAC_SCAN_ORPHAN_REQUEST_CONFIRM          (0)
#define MAC_SCAN_PASSIVE_REQUEST_CONFIRM         (0)
#define MAC_START_REQUEST_CONFIRM                (0)
```

The other features are omitted in the examples, but have to be set according to the application need.

7.2.2.3 Implications and Internal Checks

There are a number of dependencies between several of the features mentioned above. In order to keep the burden for the end user low, certain required internal checks or further implicit settings are done while configuring the build. These checks and implications can be seen in file /Include/mac_build_config.h.

7.2.2.3.1 MAC_COMM_STATUS_INDICATION

Communication systems usually follow the approach to implement primitives in pairs. This is (1) Request / Confirm (for example, MLME_ASSOCIATE-request and MLME_ASSOCIATE.confirm, or (2) Indication / Response (for example, MLME_ASSOCIATE.indication and MLME_ASSOCIATE.response). Whenever such an Indication / Response scheme is applied, the corresponding node needs a confirmation that its last transaction has finished successfully (for example, the last transmitted frame has been acknowledged by its receiver). This confirmation is done within IEEE 802.15.4 by creating an MLME_COMMUNICATION_STATUS.indication message to the upper layer.

This implies that whenever MAC_ASSOCIATION_INDICATION_RESPONSE or MAC_ORPHAN_INDICATION_RESPONSE is used (both is valid for an FFD only), the feature MAC_COMM_STATUS_INDICATION is enabled automatically.

7.2.2.3.2 MAC_SYNC_REQUEST vs. MAC_SYNC_LOSS_INDICATION

Whenever the feature MAC_SYNC_REQUEST is used, also the feature MAC_SYNC_LOSS_INDICATION is required to be included in the build. If the requirement is not met, the C-pre-processor will indicate an error.

7.2.2.3.3 Dependency from MAC_INDIRECT_DATA_BASIC

Whenever one of the subsequently listed features is used, also the feature MAC_INDIRECT_DATA_BASIC is required to be included in the build:

- MAC_ASSOCIATION_INDICATION_RESPONSE
- MAC_ASSOCIATION_REQUEST_CONFIRM
- MAC_DISASSOCIATION_BASIC_SUPPORT
- MAC_DISASSOCIATION_FFD_SUPPORT
- MAC_INDIRECT_DATA_FFD
- MAC_PURGE_REQUEST_CONFIRM

If this requirement is not met, the C-pre-processor will indicate an error.

7.2.2.3.4 Dependency from MAC_INDIRECT_DATA_FFD

Whenever one of the subsequently listed features is used, also the switch MAC_INDIRECT_DATA_FFD is required to be included in the build:

- MAC_ASSOCIATION_INDICATION_RESPONSE
- MAC_DISASSOCIATION_FFD_SUPPORT
- MAC_PURGE_REQUEST_CONFIRM

7.2.2.3.5 MAC_PAN_ID_CONFLICT_AS_PC

Whenever the feature MAC_PAN_ID_CONFLICT_AS_PC is used, also the following features are required to be included in the build:

- MAC_START_REQUEST_CONFIRM
- MAC_SYNC_LOSS_INDICATION

7.2.2.3.6 MAC_PAN_ID_CONFLICT_NON_PC

Whenever the feature MAC_PAN_ID_CONFLICT_ NON _PC is used, also the following features are required to be included in the build:

- MAC_SYNC_LOSS_INDICATION
- MAC_ASSOCIATION_REQUEST_CONFIRM or MAC_SYNC_REQUEST

7.2.2.3.7 Dependency from BEACON_SUPPORT

Whenever the feature MAC_SYNC_REQUEST is used, also the switch BEACON_SUPPRT is required to be included in the build. If the requirement is not met, the C-pre-processor will indicate an error.

8 Migration History

8.1 Guide from version 2.8.x to 3.0.x

With the release of Atmel AVR2025 version 3.0.x a number of significant improvements have been achieved by introducing design changes, enhancements, changes in mac package structure and release approach.

The complete MAC software is integrated into the Atmel Software Framework (ASF) in this release version.

The design changes do not affect the MAC and TAL layers. This release version contains significant changes in the PAL layer; the PAL drivers no more exists in the AVR2025 MAC software package, only the PAL wrapper functions will be available. The Stack will be using the ASF-PAL drivers with the help of PAL wrapper functions.

Atmel Studio 6® support (with Wireless composer integrated) is added for all the Atmel AVR 8bit and 32bit families supported in this version. MAC examples are provided for Beacon and No-beacon applications. Performance Analyzer application is added in TAL.

8.2 Guide from version 2.7.x to 2.8.x

Atmel AVR2025 version 2.8.x contains Mac application Power Management enhancement, Performance Test Evk application redesign, Watchdog support for SAM3S and a significant number of new hardware platforms and boards are added. New transceiver's AT86RF233, ATMEGARFR2 support added to this release. AVR Studio 5.1® support got added for all the Atmel AVR 8bit and 32bit families. Mesh bean support added for ATmega1281 MCU's.

8.3 Guide from version 2.6.x to 2.7.x

Atmel AVR2025 version 2.7.x contains mac security and example applications to demonstrate the mac security feature. AVR Studio 5® support got added for all the Atmel AVR 8bit and 32bit families. Two new MAC applications got added which covers most of the features of Beacon and No-beacon MAC.

8.4 Guide from version 2.5.x to 2.6.x

With the release of Atmel AVR2025 version 2.6.x a number of significant hardware platforms added. Different families of MCUs are added to the current release and naming few of them are Atmel AVR32, Atmel SAM3S, and CBB kit. The kits supported on AVR32 Family are Atmel RZ600 (Atmel AT32UC3A3256), Atmel AT32UC3L-EK (Atmel AT32UC3L04) and Atmel STK®600 (Atmel AT32UC3B1128). The kits supported on SAM3S family are Atmel SAM3S-EK (Atmel AT91SAM3S4C) and DERF_USB_13E00, DERF_USB_23E00 USB Kits (Atmel AT91SAM3S4B).

8.5 Guide from version 2.4.x to 2.5.x

With the release of Atmel AVR2025 version 2.5.x a number of significant improvements have been achieved by introducing design changes throughout the PAL, TAL and MCL layer and by introducing the Tiny_TAL layer.

Although these design changes do not significantly change the MAC-API, both the TAL-API and partially the PAL-API have been changed. This is based on the fact that large portions of the code formerly residing inside the TAL layer have been shifted up

to the MCL layer. This leads to an overall code size reduction, as well as to the much simpler TAL-API, which can be used more easily for simple applications.

The PAL-API was changed mostly for handling transceiver related interrupts with the focus of reducing the overall code size and excluding functionality not used per default.

In order to better understand the impact of these design improvements on the end user applications or stack layers the following sections will describe the most important changes in detail.

8.5.1 MAC-API changes

The API changes within the MAC-API can be classified into the following groups:

- Handling of Timestamp parameter in MCPS-DATA primitives
- Type of AddrList parameter in MLME-BEACON-NOTIFY.indication primitive

8.5.1.1 Handling of timestamp parameter in MCPS-DATA primitives

According to 4 both the MCPS-DATA.confirm and the MCPS-DATA.indication primitive contain a Timestamp parameter. This is implemented in the MAC callback functions usr_mcps_data_conf() and usr_mcps_data_ind().

In many cases the timestamping functionality is not used within the entire application. In order to save code size, and simply the application (if the Timestamp parameter is not required), the Timestamp parameter as well as the complete handling to timestamping in the entire stack can be omitted. This can be controlled via the build switch ENABLE_TSTAMP. For more information about this build switch see Section 7.1.

Starting with release 2.5.x timestamping is excluded as default, i.e. the Timestamp parameter is not included into the mentioned callback functions. If an application utilizes the build switch in its Makefile or project files, timestamping is performing within the stack, and thus the Timestamp parameter is included in the callback functions. An example of a MAC application using timestamping can be found in App3_Beacon_Payload.

The updated API for the corresponding callback functions is defined as:

```
#if defined(ENABLE_TSTAMP)
void usr_mcps_data_conf(uint8_t msduHandle,
                        uint8_t status,
                        uint32_t Timestamp);
#else
void usr_mcps_data_conf(uint8_t msduHandle,
                        uint8_t status);
#endif  /* ENABLE_TSTAMP */
```

and

```
void usr_mcps_data_ind(wpan_addr_spec_t *SrcAddrSpec,
                       wpan_addr_spec_t *DstAddrSpec,
                       uint8_t msduLength,
                       uint8_t *msdu,
                       uint8_t mpduLinkQuality,
    #ifdef ENABLE_TSTAMP
                       uint8_t DSN,
```

```
                                uint32_t Timestamp);
#else
                                uint8_t DSN);
#endif  /* ENABLE_TSTAMP */
```

The function prototypes can be found in MAC/Inc/mac_api.h file.

If the build switch ENABLE_TSTAMP is not used within an application, the corresponding implementation in the application needs to be adjusted accordingly. An example of this can be found in file *main.c* in the directory *Applications\MAC_Examples\Star_Nobeacon\Src*.

8.5.1.2 AddrList parameter in MLME-BEACON-NOTIFY.indication primitive

The MLME-BEACON-NOTIFY.indication primitive implemented in the MAC callback function usr_mlme_beacon_notify_ind() function has an updated type of the included parameter AddrList:

The callback function definition has been changed from

```
void usr_mlme_beacon_notify_ind(uint8_t BSN,
                                wpan_pandescriptor_t
*PANDescriptor,
                                uint8_t PendAddrSpec,
                                void *AddrList,
                                uint8_t sduLength,
                                uint8_t *sdu);
```

to

```
void usr_mlme_beacon_notify_ind(uint8_t BSN,
                                wpan_pandescriptor_t
*PANDescriptor,
                                uint8_t PendAddrSpec,
                                uint8_t *AddrList,
                                uint8_t sduLength,
                                uint8_t *sdu);
```

The function prototype can be found at /MAC/Inc/mac_api.h.

8.5.2 TAL-API Changes

In order to reduce code size and complexity of the entire stack, and to allow the development of TAL based applications, the design and the API of the TAL have been simplified significantly. The API changes within the TAL-API can be classified into the following groups:

- Simplification of structure frame_info_t used within the TAL frame handling functions
- Simplification of frame indication callback function tal_rx_frame_cb()
- Simplification of Beacon handling API

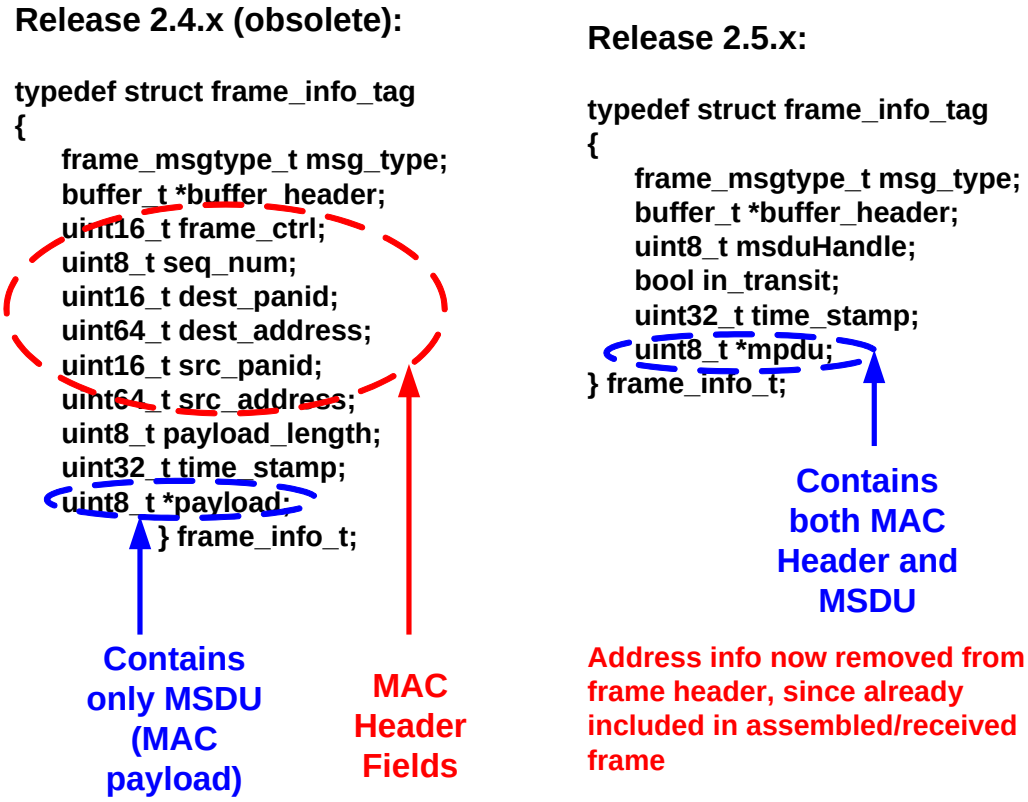
8.5.2.1 Simplification of structure frame_info_t

Frames being exchanged between the MCL and the TAL layer (that is, frames to be transmitted and frames being received), are handled at the TAL-API by means of a specific frame structure containing all relevant frame information. This structure has

the type frame_info_t and is defined in file *tal.h* in directory *TAL\Inc*. It is used in the following functions:

- tal_rx_frame_cb()
- tal_tx_frame()
- tal_tx_beacon() (new since 2.5.x)

Figure 8-1. Content of frame_info_t structure



The MAC header information previously included by means of several structure elements has been migrated into the new element “mdpu”, which contains the complete MPDU (both the MAC Header and the MSDU = Data Payload). This implies that starting with release 2.5.x the MAC header information is only parsed and formatted inside the MAC layer and is fully transparent for the TAL layer.

8.5.2.2 Simplification of Function tal_rx_frame_cb()

The TAL callback function for a frame indication (once a valid has been received and needs to be forwared to the MCL) has been simplified. The function tal_rx_frame_cb() has changed from

```
void tal_rx_frame_cb(frame_info_t *mac_frame_info, uint8_t lqi)
```

to

```
void tal_rx_frame_cb(frame_info_t *rx_frame)
```

The parameter LQI has been removed, since the LQI value of the current frame is now part of the element “mpdu” of the frame_info_t structure variable “rx_frame”. For more information check function tal_rx_frame_cb() in file *TAL/Src/tal_rx_frame_cb.c*.

For more information how to extract and use the LQI value of the received frame see Section 5.2.1.2.

8.5.2.3 Simplification of Beacon Handling API

The API handling (periodic) Beacon frames (within a beacon-enabeld network) has been simplified by removing function tal_prepare_beacon() and updating function tal_tx_beacon().

Originally (release 2.4.x) the periodic Beacon transmission was performed by obeying the following steps:

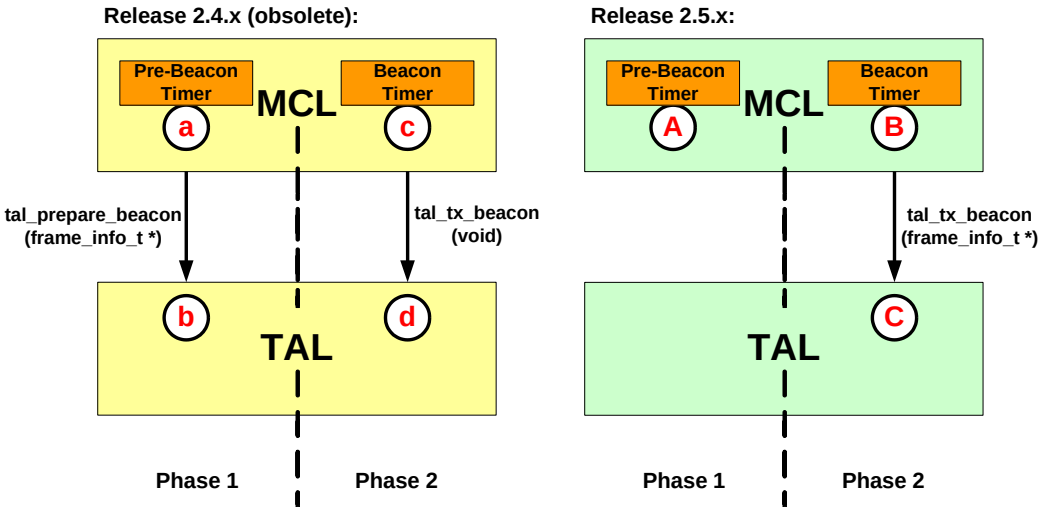
(**a**) Expiration of Pre-Beacon timer in MCL - Preparation of next Beacon frame within MCL

(**b**) Calling of TAL-API function tal_prepare_beacon() including the frame information structure to initiate formatting of frame in TAL; Beacon frame is stored inside TAL until transmission time

(**c**) Expiration of Beacon time in MCL – Calling of TAL-API function tal_tx_beacon to trigger immediate Beacon frame transmission

(**d**) TAL used stored frame and initiates Beacon frame transmission

Figure 8-2. Transmission of periodic Beacon Frames



Starting with release 2.5.x this has been simplified:

(**A**) Expiration of Pre-Beacon timer in MCL – Complete formatting of next Beacon frame within MCL; not further interaction with TAL; function tal_prepare_beacon() does not exist anymore

(**B**) Expiration of Beacon time in MCL – Calling of TAL-API function tal_tx_beacon() to trigger immediate Beacon frame transmission; function tal_tx_beacon contains complete Beacon frame already formatted

(**C**) TAL uses parameter of type frame_info_t in function tal_tx_frame() to initate Beacon frame transmission

8.5.3 PAL-API Changes

In order to reduce code size and to taylor the actually included functionality of the PAL according to the user application needs, the PAL-API has been updated. The

API changes within the PAL-API mainly affect the functions handling transceiver interrupts.

8.5.3.1 TRX IRQ Initialization

8.5.3.1.1 Releaes 2.4.x

The initialization of transceiver interrupts has been changed from release 2.4.x:

```
void pal_trx_irq_init(trx_irq_hdlr_idx_t trx_irq_num,
                      FUNC_PTR trx_irq_cb)
```

The implementation comprised of exactly one function for interrupt initialization, which required an ID for the actual interrupt number.

8.5.3.1.2 Releaes 2.5.x

Since most applications do not require all transceiver interrupts, this approach has been changed starting from release 2.5.x. The focus here is clearly on reduced footprint. Each single transceiver interrupt is initialized with a dedicated initializiation function. The original parameter specifying the dedicated transceiver interrupt is not used anymore. Furthermore only required transceiver interrupts are included into the code based on additional build switches that can be used.

The following functions are provided starting from release 2.5.x:

Initialization of main transceiver interrupt:

```
void pal_trx_irq_init(FUNC_PTR trx_irq_cb)
```

Initialization of transceiver timestamp interrupt (only available if ENABLE_TSTAMP is used):

```
void pal_trx_irq_init_tstamp(FUNC_PTR trx_irq_cb)
```

Initialization of additionally required transceiver interrupts for MEGA-RF single chips:

```
void pal_trx_irq_init_rx_end(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_tx_end(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_cca_ed(FUNC_PTR trx_irq_cb)
```

Initialization of further optional transceiver interrupts for MEGA-RF single chips (only available if ENABLE_ALL_TRX_IRQS is used):

```
void pal_trx_irq_init_ami(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_batmon(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_awake(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_pll_lock(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_pll_unlock(FUNC_PTR trx_irq_cb)
void pal_trx_irq_init_aes_ready(FUNC_PTR trx_irq_cb)
```

For more information see file *PAL/Inc/pal.h* and the corresponding *pal_irq.c* file for each platform implementation.

8.5.3.2 TRX IRQ Enabling and Disabling

8.5.3.2.1 Releaes 2.4.x

Enabling and disabling of transceiver interrupts has been changed from release 2.4.x:

```
inline void pal_trx_irq_enable(trx_irq_hdlr_idx_t trx_irq_num)
```

```
inline void pal_trx_irq_disable(trx_irq_hdlr_idx_t trx_irq_num)
```

The implementation comprised of exactly one inline function for enabling or disabling transceiver interrupts, which required an ID for the actual interrupt number.

8.5.3.2.2 Releaes 2.5.x

Since most applications do not require all transceiver interrupts, this approach has been changed starting from release 2.5.x. Both the main transceiver interrupt and the timestamp interrupt are now enabled or disable by using a macro. The original parameter specifying the dedicated transceiver interrupt is not used anymore.

The following macros are provided starting from release 2.5.x:

Enabling and disabling of the main transceiver interrupt:

```
#define pal_trx_irq_en()            (ENABLE_TRX_IRQ())
#define pal_trx_irq_dis()           (DISABLE_TRX_IRQ())
```

Enabling and disabling of the transceiver timestamp interrupt (only available if ENABLE_TSTAMP is used):

```
#define pal_trx_irq_en_tstamp()     (ENABLE_TRX_IRQ_TSTAMP())
#define pal_trx_irq_dis_tstamp()    (DISABLE_TRX_IRQ_TSTAMP())
```

Please note that the macro above are only available for non-single chip transceivers, since in single chip transceivers (MEGA_RF platforms) there is no separation between enabling/disabling transceiver interrupts at the transceiver, and setting/clearing the IRQ mask at the MCU. Therefore the transceiver interrupts in single chips are enabled/disabled by setting the MCU IRQ mask.

For more information see file *PAL/Inc/pal.h* and the corresponding *pal_config.h* file for each platform implementation.

8.5.3.3 TRX IRQ Flag Clearing

8.5.3.3.1 Releaes 2.4.x

Clearing the interrupt flag of transceiver interrupts has been changed from release 2.4.x:

```
inline void pal_trx_irq_flag_clr(trx_irq_hdlr_idx_t trx_irq_num)
```

The implementation comprised of exactly one inline function for clearing the transceiver interrupt flag, which required an ID for the actual interrupt number.

8.5.3.3.2 Releaes 2.5.x

Since most applications do not require all transceiver interrupts, this approach has been changed starting from release 2.5.x. Each single transceiver interrupt flag is cleared using a macro. The original parameter specifying the dedicated transceiver interrupt is not used anymore. Furthermore only required transceiver interrupts are included into the code based on additional build switches that can be used.

The following macros are provided starting from release 2.5.x:

Clearing the main transceiver interrupt flag:

```
#define pal_trx_irq_flag_clr()          (CLEAR_TRX_IRQ())
```

Clearing the transceiver timestamp interrupt flag (only available if ENABLE_TSTAMP is used):

```
#define pal_trx_irq_flag_clr_tstamp()    (CLEAR_TRX_IRQ_TSTAMP())
```

Clearing of additionally required transceiver interrupt flags for MEGA-RF single chips:

```
#define pal_trx_irq_flag_clr_rx_end()    (CLEAR_TRX_IRQ_RX_END())
#define pal_trx_irq_flag_clr_tx_end()    (CLEAR_TRX_IRQ_TX_END())
#define pal_trx_irq_flag_clr_cca_ed()    (CLEAR_TRX_IRQ_CCA_ED())
```

Clearing of further optional transceiver interrupt flags for MEGA-RF single chips (only available if ENABLE_ALL_TRX_IRQS is used):

```
#define pal_trx_irq_flag_clr_ami()       (CLEAR_TRX_IRQ_AMI())
#define pal_trx_irq_flag_clr_batmon()    (CLEAR_TRX_IRQ_BATMON())
#define pal_trx_irq_flag_clr_awake()     (CLEAR_TRX_IRQ_AWAKE()
#define pal_trx_irq_flag_clr_pll_lock()  (CLEAR_TRX_IRQ_PLL_LOCK())
#define pal_trx_irq_flag_clr_pll_unlock()
CLEAR_TRX_IRQ_PLL_UNLOCK())
```

For more information see *PAL/Inc/pal.h* file and the corresponding *pal_config.h* file for each platform implementation

9 Toolchain

The following sections describe the required tools and toolchain for the development and build process and how the provided example applications can be built.

9.1 General prerequisites

The following tools and tool-chains are used for building the applications from this MAC package:

- Atmel Studio 6
  (see http://www.atmel.com/tools/ATMELSTUDIO.aspx)
- Atmel AVR and AVR32 GNU Toolchain for Windows
  (see http://www.atmel.com/tools/atmelavrtoolchainforwindows.aspx)
- ARM Code Sorcery GCC Toolchain for Windows(IA32 Windows Installer)
  (see http://www.codesourcery.com/sgpp/lite/arm/portal/release642)
- IAR Embedded Workbench for Atmel AVR V6.11
  (see http://www.iar.com/)
- IAR Embedded Workbench for Atmel AVR32 V4.10
  (see http://www.iar.com/)
- IAR Embedded Workbench for Atmel ARM V6.40
  (see http://www.iar.com/)

9.2 Building the applications

9.2.1 Using GCC makefiles

Each application should be built using the provided Makefiles. Please follow the procedure as described:

- Change to the directory where the Makefile for the desired platform of the corresponding application is located, for example:

```
cd D:\ASF\thirdparty\wireless\avr2025_mac\apps\mac\beacon\coord\ncp
cd xmega_a3bu_xplained_rz600rf212\gcc
```

- Run the desired Makefile, for example:

```
make –f Makefile
```

NOTE

Makefile builds a binary optimized for code size without Serial I/O support, whereas Makefile_Debug builds a version for better debug support without optimization but with additional Serial I/O support

- After running one of the Makefiles the same directory contains both a hex-file and an elf-file which can be downloaded onto the hardware (see Section 10)
- The above procedure for building the Makefiles is common for Atmel AVR8, Atmel AVR32, and ARM Platforms

9.2.2 Using IAR Embedded Workbench

Each application can be rebuilt using the IAR Embedded Workbench directly. Please follow the procedure as described:

- Change to the directory where the IAR Embedded Workbench workspace file (eww-file) for the desired platform of the corresponding application is located, for example:

```
cd D:\ASF\thirdparty\wireless\avr2025_mac\apps\mac\beacon\coord\ncp
cd xmega_a3bu_xplained_rz600rf231\iar
```

- Double click on the corresponding IAR Embedded Workbench file (eww-file), for example *beacon_coord.eww*
- Select the desired workspace (Release or Debug) and Rebuild the entire application in IAR Embedded Workbench
- After building the application the subdirectory IAR/Exe contains either an a90-file (in case the Release configuration was selected) or a d90-file (in case a Debug configuration was selected). Both binaries can be downloaded onto the hardware (see Section 10)
- The Release configuration binary (a90-file) can be downloaded using IAR Embedded Workbench directly or AVR Studio
- The Debug configuration binary (d90-file) can only be downloaded using IAR Workbench and can be debugged using IAR C-Spy®
- In case is it desired to create a binary with IAR Embedded Workbench, which contains AVR Studio Debug information and can thus directly be downloaded and debugged using AVR Studio, the following changes need to be done with IAR Embedded Workbench:
    - Select the Debug configuration
    - Open the “Options” dialog
    - Select “Category” “Linker”
    - Select tab “Output”
    - Change “Format” from “Debug information for C-Spy” to “Other”
    - Select “ubrof 8 (forced)” as “Output format”
    - Select “None” as “Format variant”
    - Rebuild the application
    - The generated binary can now contains debug information that can be used directly within AVR Studio

9.2.3 Using IAR AVR32 Embedded Workbench

Each application can be rebuilt using the IAR AVR32 Embedded Workbench directly. Please follow the procedure as described:

- Change to the directory where the IAR Embedded Workbench workspace file (eww-file) for the desired platform of the corresponding application is located, for example:

```
cd D:\ASF\thirdparty\wireless\avr2025_mac\apps\mac\beacon\coord\ncp
cd at32uc3a3256s_rz600_at86rf212\iar
```

- Double click on the corresponding IAR Embedded Workbench file (eww-file), for example *beacon_coord.eww*
- Select the desired workspace (Release or Debug) and Rebuild the entire application in IAR Embedded Workbench
- After building the application the subdirectory IAR/Exe contains either an elf-file (in case the Release or Debug configuration was selected).The binaries can be downloaded onto the hardware (see Section 10)

- The Release configuration binary (elf-file) can both be downloaded using IAR AVR32 Embedded Workbench directly or AVR Studio
- The Debug configuration binary (elf-file) can only be downloaded using IAR AVR32 Workbench and can be debugged using IAR AVR32 C-Spy®

9.2.4 Using IAR ARM Embedded Workbench

Each application can be rebuilt using the IAR ARM Embedded Workbench directly. Please follow the procedure as describe:

- Change to the directory where the IAR Embedded Workbench workspace file (eww-file) for the desired platform of the corresponding application is located, for example:

```
cd
D:\ASF\thirdparty\wireless\avr2025_mac\apps\mac\beacon\coord\host
cd sam4lc4c_sam4l_xplained_pro\iar
```

- Double click on the corresponding IAR Embedded Workbench file (eww-file), for example *beacon_coord.eww*
- Select the desired workspace (Release or Debug) and Rebuild the entire application in IAR Embedded Workbench
- After building the application the subdirectory IAR/Exe contains either a binary file or an elf-file (in case the Release or Debug configuration was selected). The binaries can be downloaded onto the hardware (see Section 10)
- The Release and Debug configuration binary or elf-file can both be downloaded using IAR ARM Embedded Workbench directly

10 Downloading an application

This section describes how the binaries of the applications can be downloaded onto the hardware.

In the subsequent section this is indicated both for Atmel Studio and IAR Embedded Workbench.

10.1 Using Atmel Studio 6

Please refer to Atmel Studio-help [10]

10.2 Using IAR Embedded Workbench

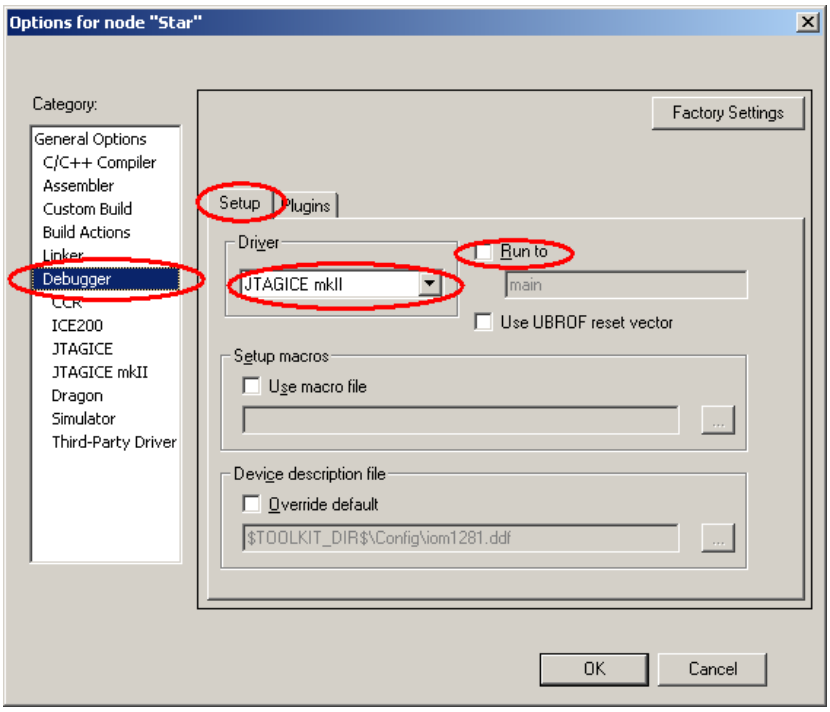
When using IAR Embedded Workbench directly by double clicking the corresponding eww-file, the application can be downloaded onto the desired hardware platform as described below.

10.2.1 Starting the release build

The release build can simply be started as follows:

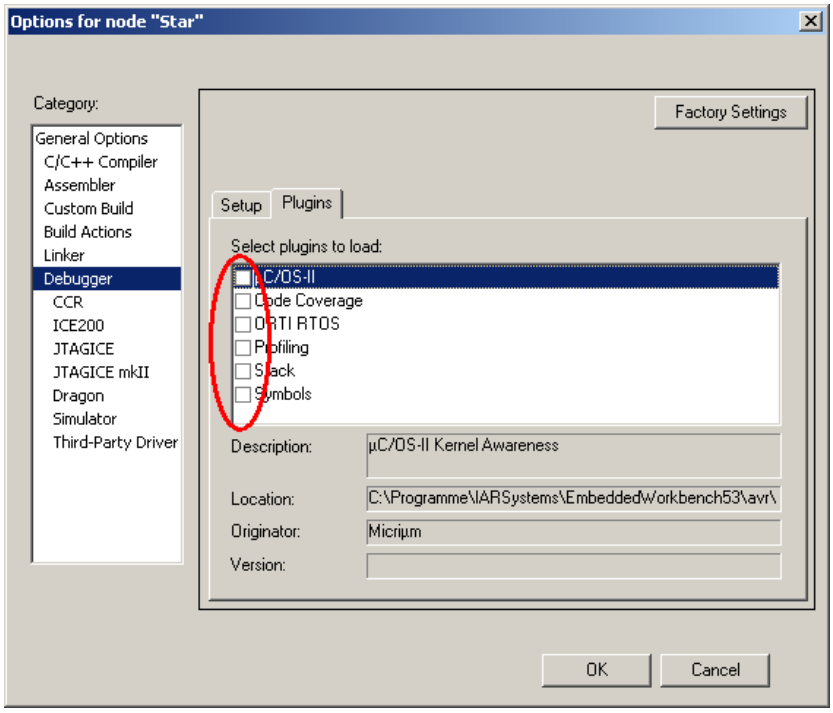
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Embedded Workbench by double clicking the desired eww-file
- Select the “Release” Workspace
- Open the “Options” window for the “Release” Workspace. Select the “Debugger” window and within this window select the “Setup” tab

Figure 10-1. IAR Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



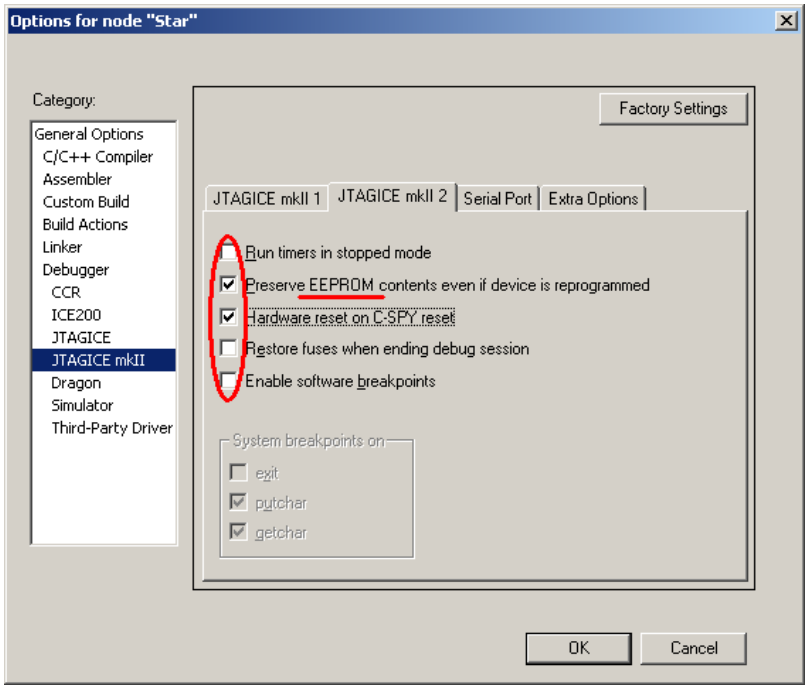
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and deselect “Run to (main)”
- Change to the “Plugins” tab and deselect all entries

Figure 10-2. IAR Embedded Workbench – “Options” -> “Debugger” -> “Plugins”.



- Within the “Options” window for the “Release” Workspace select now the “JTAGICE mkII” window and within this window select the “JTAGICE mkII 2” tab

Figure 10-3. IAR Embedded Workbench – “Options” -> “JTAGICE mkII” -> “JTAGICE mkII 2”.



- Within the “JTAGICE mkII 2” tab check the items as shown above. Especially make sure that the EEPROM will be preserved if the device is reprogrammed
- Press “OK”

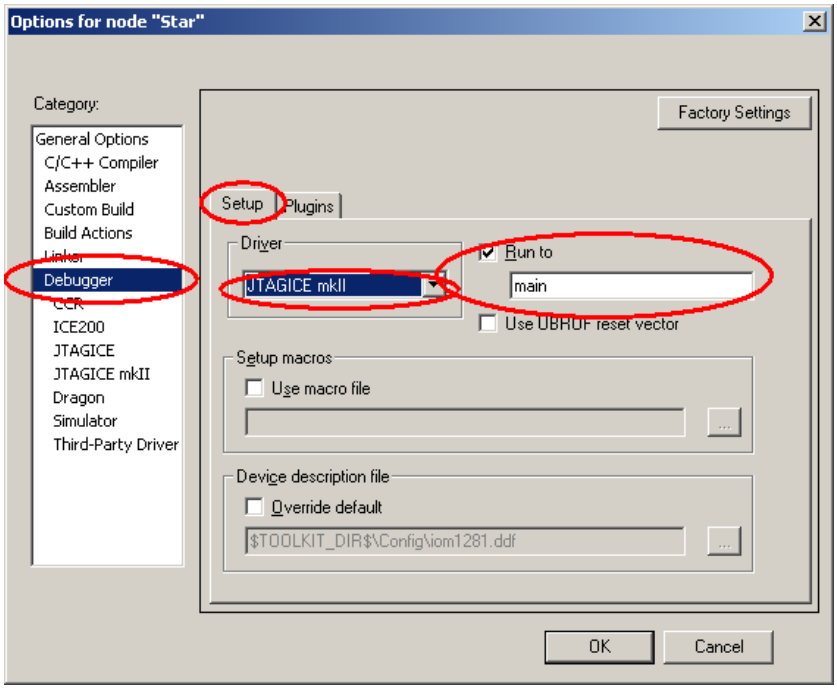
- Press the button “Download and Debug”
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and deselect “Run to (main)”
- Change to the “Plugins” tab and deselect all entries

10.2.2 Starting the debug build

The debug build can simply be started as follows:

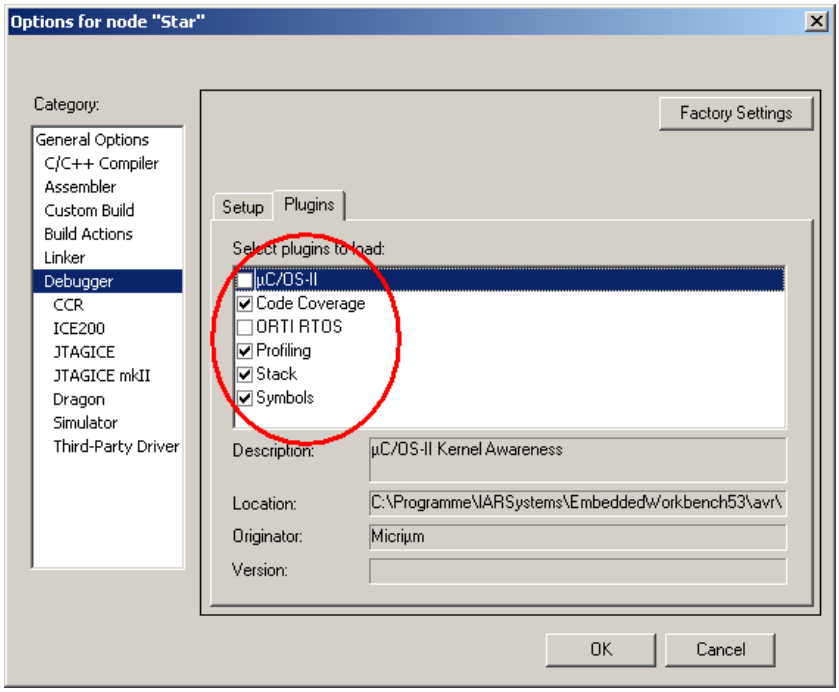
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Workbench by double clicking the desired eww-file
- Select the “Debug” Workspace
- Open the “Options” window for the “Debug” Workspace. Select the “Debugger” window and within this window select the “Setup” tab

Figure 10-4. IAR Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



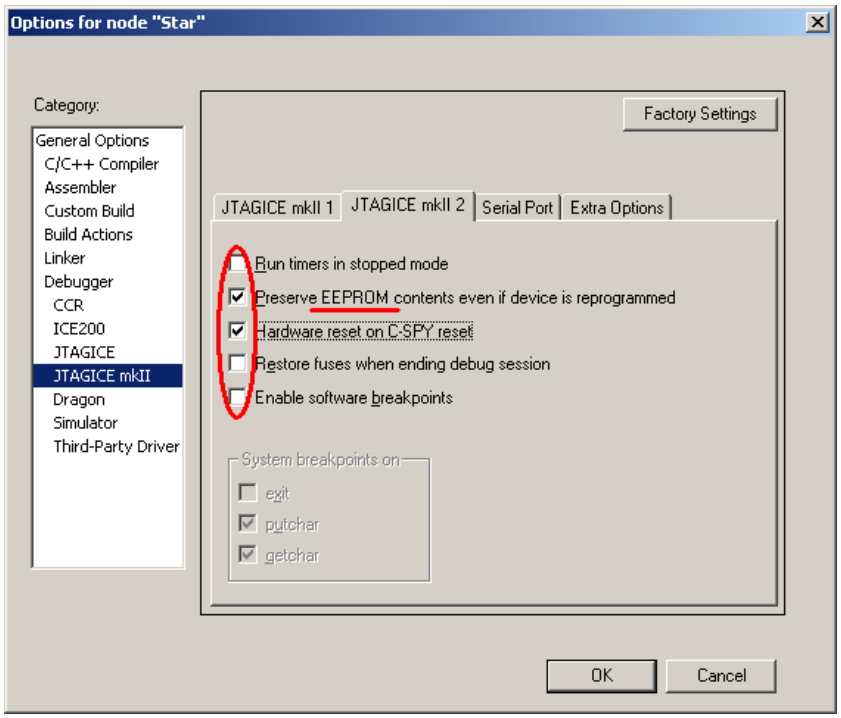
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and select “Run to (main)” (this is different to “Release” build)
- Change to the “Plugins” tab and select the entries as shown in Figure 10-5

Figure 10-5. IAR Embedded Workbench – “Options” -> “Debugger” -> “Plugins”.



- Within the “Options” window for the “Release” Workspace select now the “JTAGICE mkII” window and within this window select the “JTAGICE mkII 2” tab

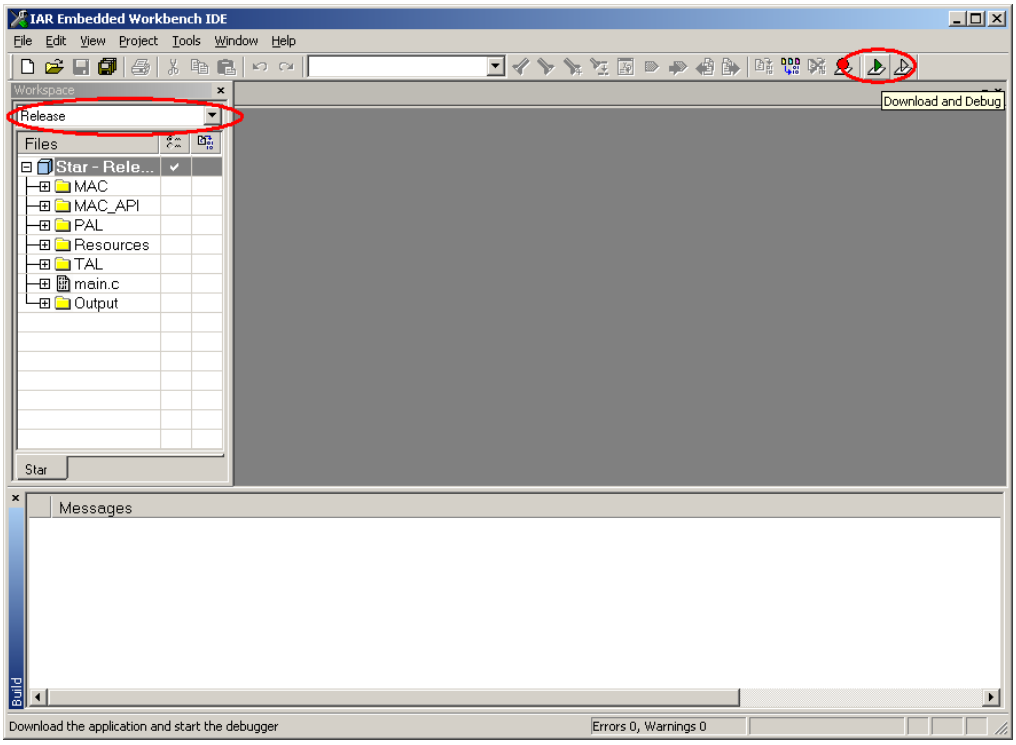
Figure 10-6. IAR Embedded Workbench – “Options” -> “JTAGICE mkII” -> “JTAGICE mkII 2”.



- Within the “JTAGICE mkII 2” tab check the items as shown in Figure 10-6. Especially make sure that the EEPROM will be preserved if the device is reprogrammed

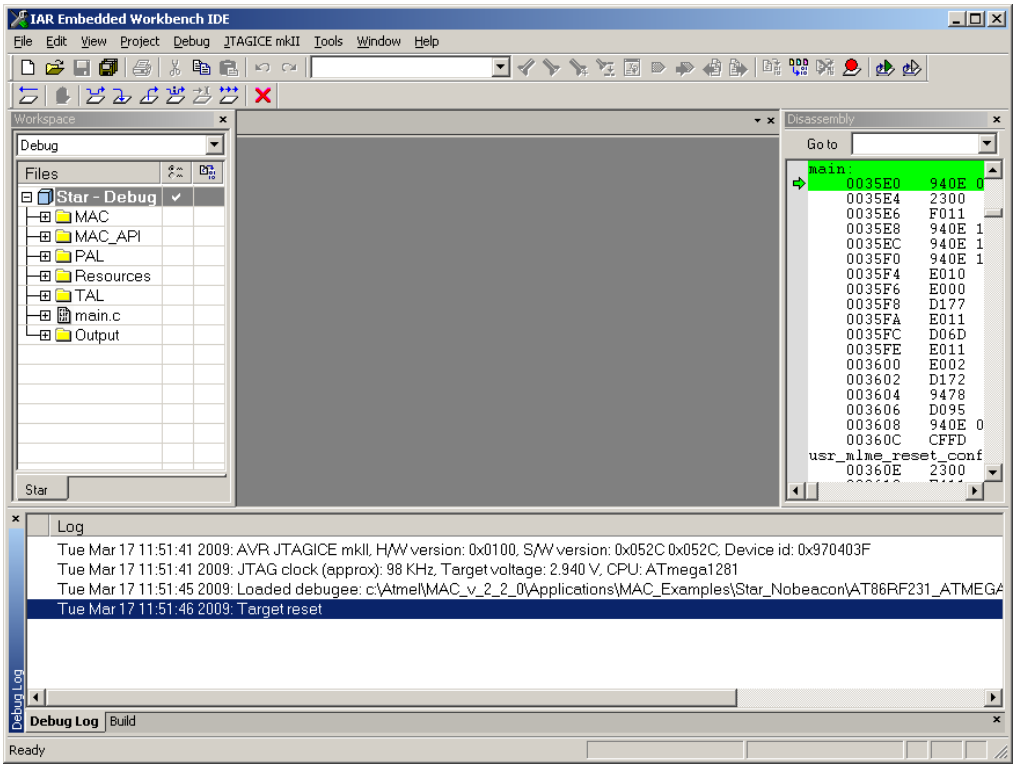
- Press “OK”
- Press the button “Download and Debug” as shown below

Figure 10-7. IAR Embedded Workbench – start “Download and Debug”.



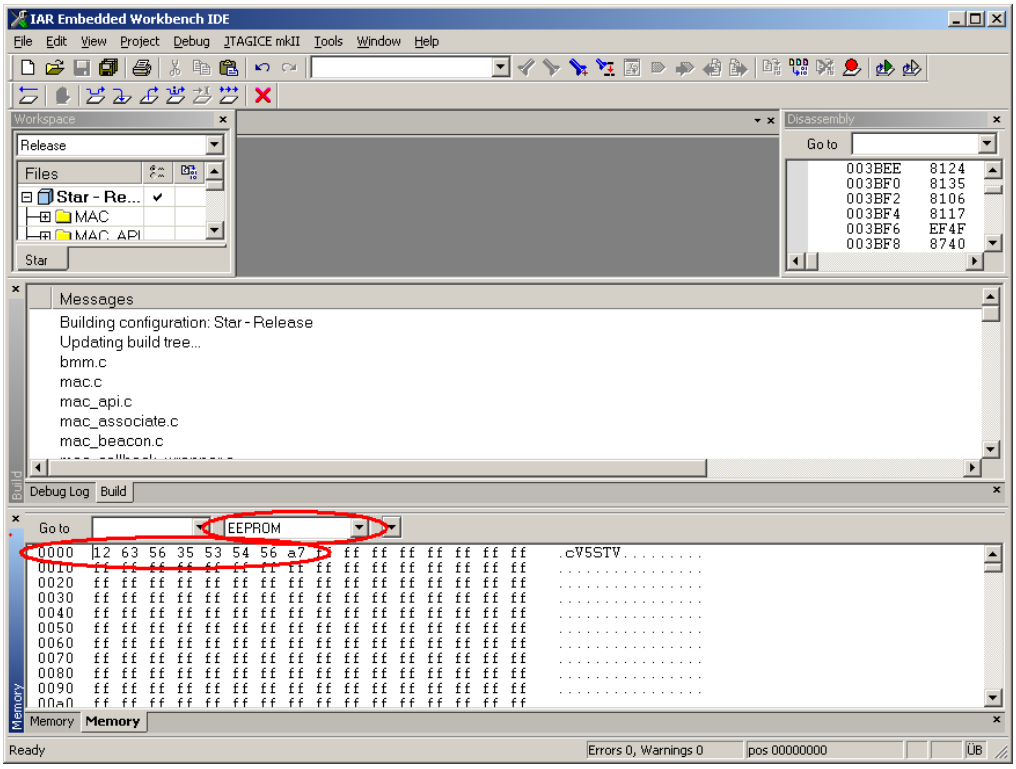
- A Window will pop-up indicating the status of the download of the binary
- After successful download the IAR Workbench will looks as shown in Figure 10-8.
- After the download of the debug build the main function is displayed since all debug and source code information is included in the build

Figure 10-8. IAR Embedded Workbench – successful download of debug build.



- In case the IEEE address of the node is stored in the internal EEPROM of the microcontroller perform as follows (only for 8-bit IAR):
    - Check whether a valid IEEE address (different from 0xFFFFFFFFFFFFFFFF and 0x0000000000000000) is stored in the internal EEPROM. Select menu “View” item “Memory”. In this window select “EEPROM” as shown in Figure 10-9.
    - If the IEEE address is not set properly, add the correct IEEE to the first eight octets of the EEPROM

Figure 10-9. IAR Embedded Workbench – verifying and setting of IEEE address.



- Start the application by pressing the “F5” or the “Go” button

10.3 Using IAR AVR 32 Embedded Workbench

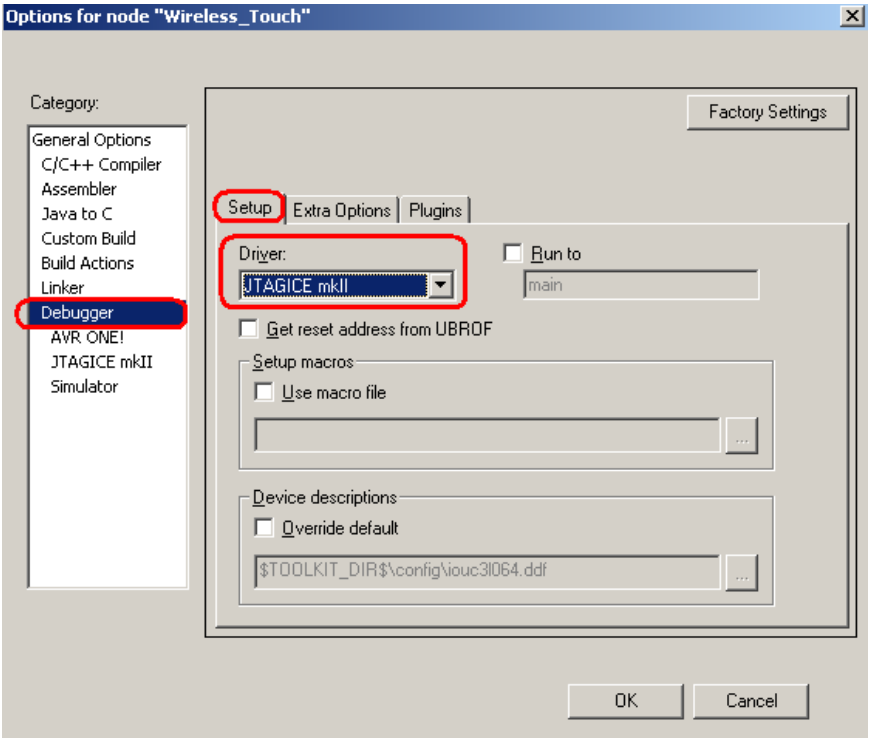
When using IAR AVR32 Embedded Workbench directly by double clicking the corresponding eww-file, the application can be downloaded onto the desired hardware platform as described below.

10.3.1 Starting the release build

The release build can simply be started as follows:

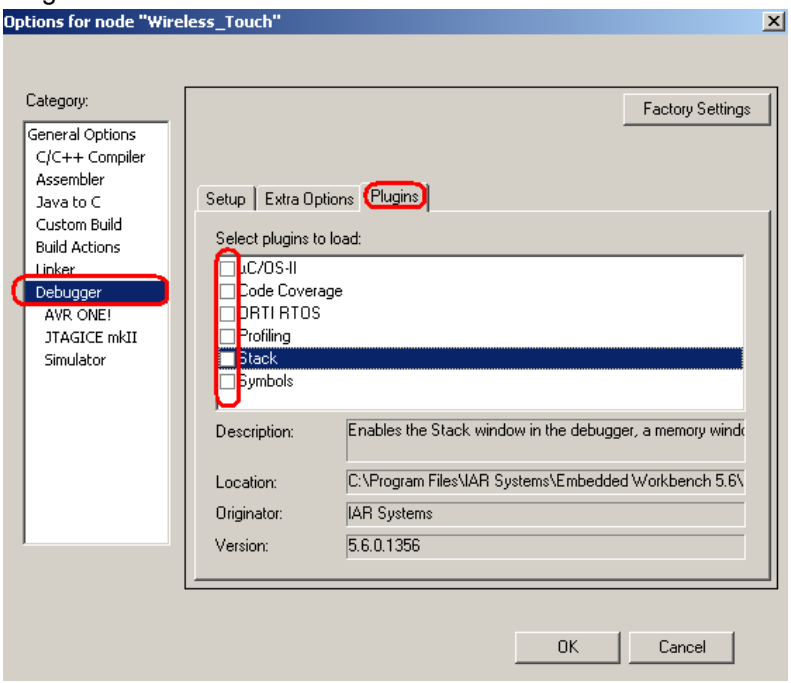
- Make sure that only the Atmel AVR JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Embedded Workbench by double clicking the desired eww-file
- Select the “Release” Workspace
- Open the “Options” window for the “Release” Workspace. Select the “Debugger” window and within this window select the “Setup” tab

Figure 10-10. IAR AVR32 Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and deselect “Run to (main)”
- Change to the “Plugins” tab and deselect all entries as shown in Figure 10-11

Figure 10-11. IAR AVR32 Embedded Workbench – “Options” -> “Debugger” -> Plugins”.



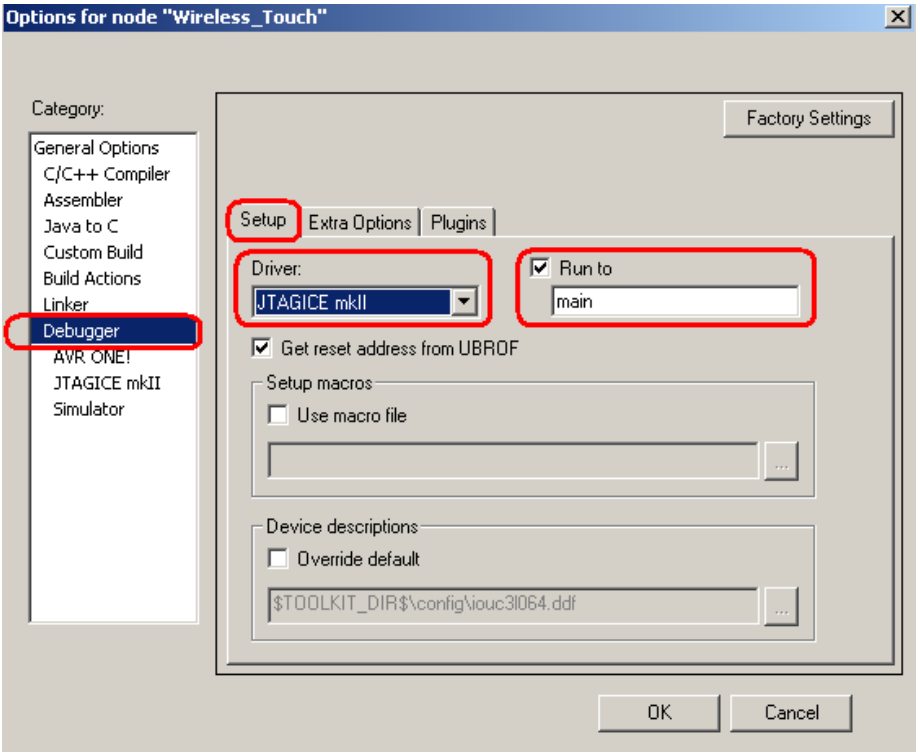
- Press “OK”
- Press the button “Download and Debug”

10.3.2 Starting the debug build

The debug build can simply be started as follows:

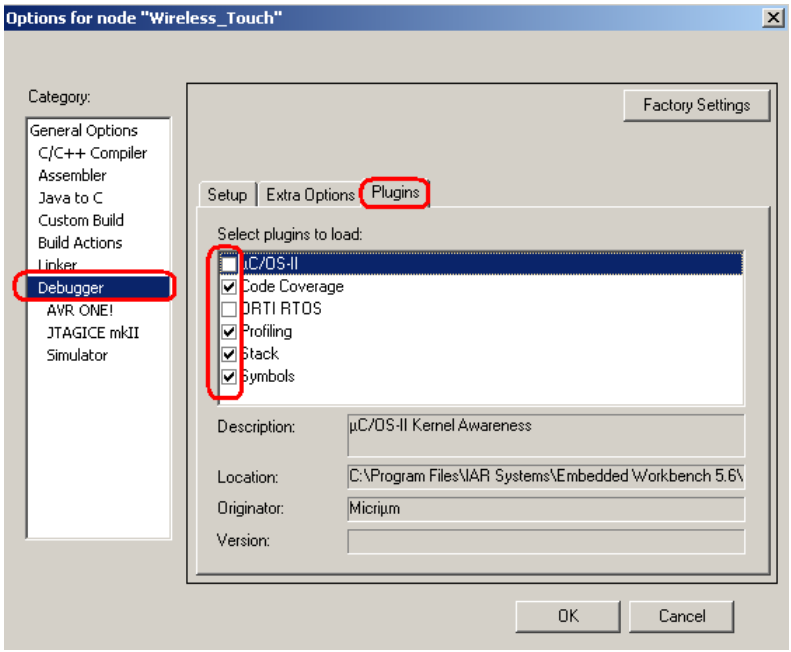
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Embedded Workbench by double clicking the desired eww-file
- Select the “Debug” Workspace
- Open the “Options” window for the “Debug” Workspace. Select the “Debugger” window and within this window select the “Setup” tab

Figure 10-12. IAR AVR32 Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



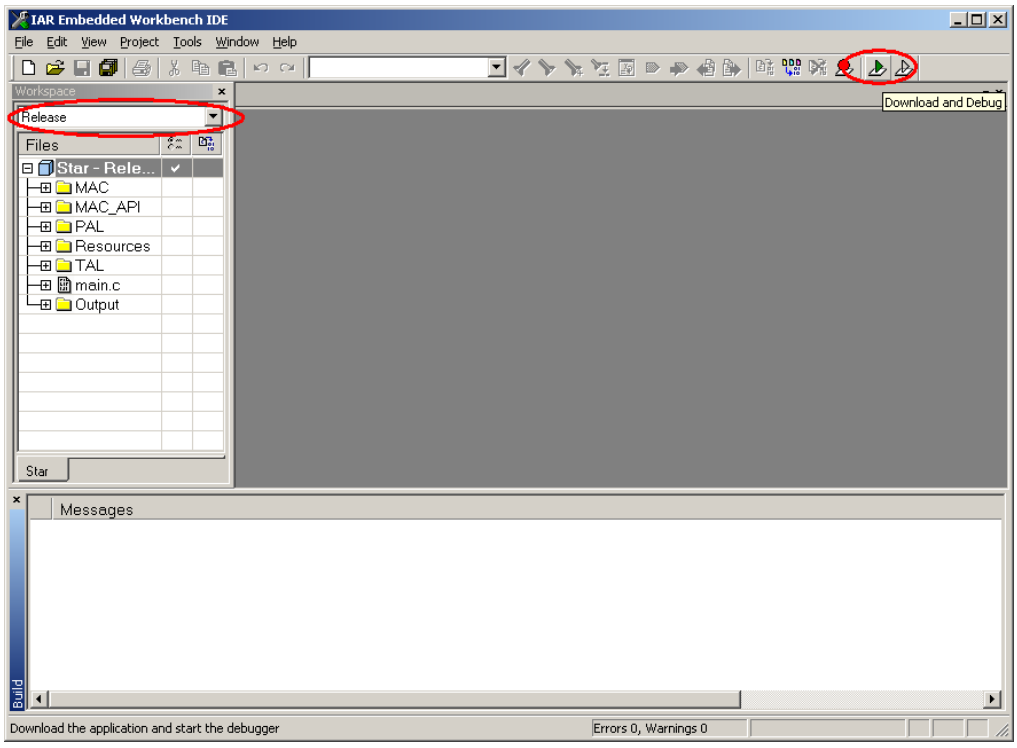
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and select “Run to (main)” (this is different to “Release” build)
- Change to the “Plugins” tab and select the entries as shown in Figure 10-13

Figure 10-13. IAR AVR32 Embedded Workbench – “Options” -> “Debugger” -> “Plugins”.



- Press “OK”
- Press the button “Download and Debug”

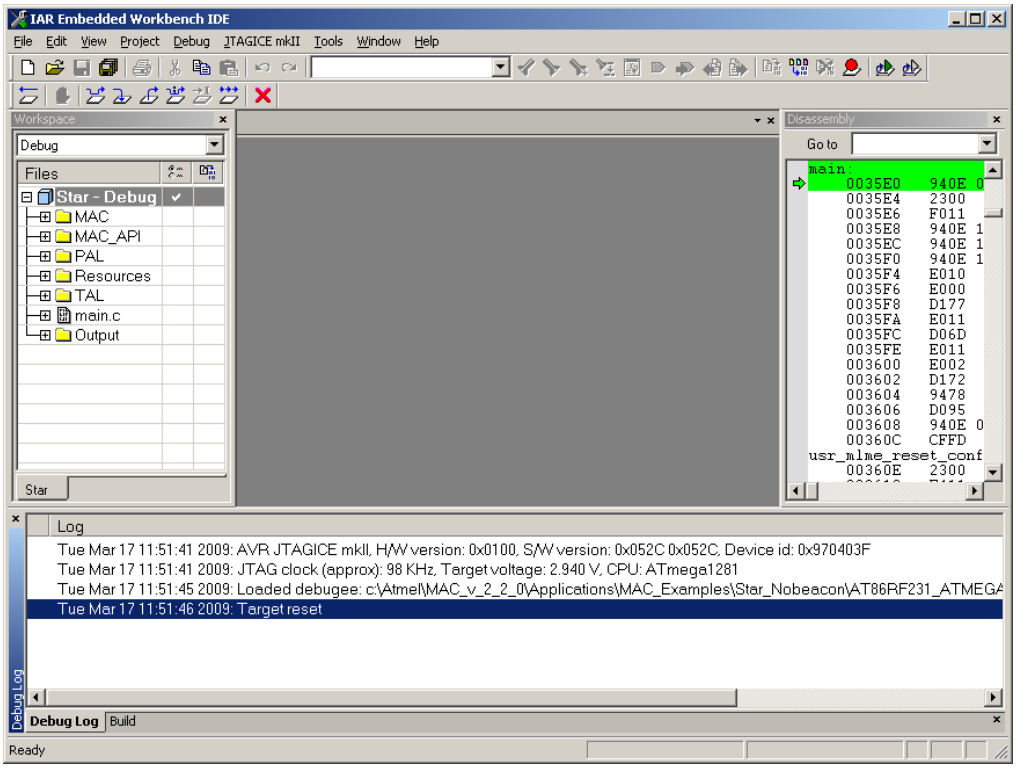
Figure 10-14. IAR AVR32 Embedded Workbench – start “Download and Debug”.



- A Window will pop up indicating the status of the download of the binary

- After successful download the IAR AVR32 Workbench will looks as shown in Figure 10-15. After the download of the debug build the main function is displayed since all debug and source code information is included in the build

Figure 10-15. IAR AVR32 Embedded Workbench – successful download of debug build.



- Start the application by pressing the “F5” or the “Go” button

10.4 Using AVR32 GCC commandline programming

The application can be downloaded using GNU utility avr32program.Change to the directory where the Makefile for the application is located, for example:

cd ASF\thirdparty\wireless\avr2025_mac\apps\mac\beacon\coord\ncp

cd at32uc3a3256s_rz600_at86rf231

cd gcc

Download the project using the following command

- For AT32UC3A3256S:

avr32program.exe -p jtagicemkii --part UC3A3256S program -finternal@0x80000000 -e -r -v -R *.elf

- For AT32UC3B1128:

avr32program.exe -p jtagicemkii --part UC3B1128 program -finternal@0x80000000 -eu --run -R -cint *.elf

- For AT32UC3L064:

avr32program.exe -p jtagicemkii --part UC3L064 program -finternal@0x80000000 -eu --run -R -cint *.elf

10.5 Using IAR ARM Embedded Workbench

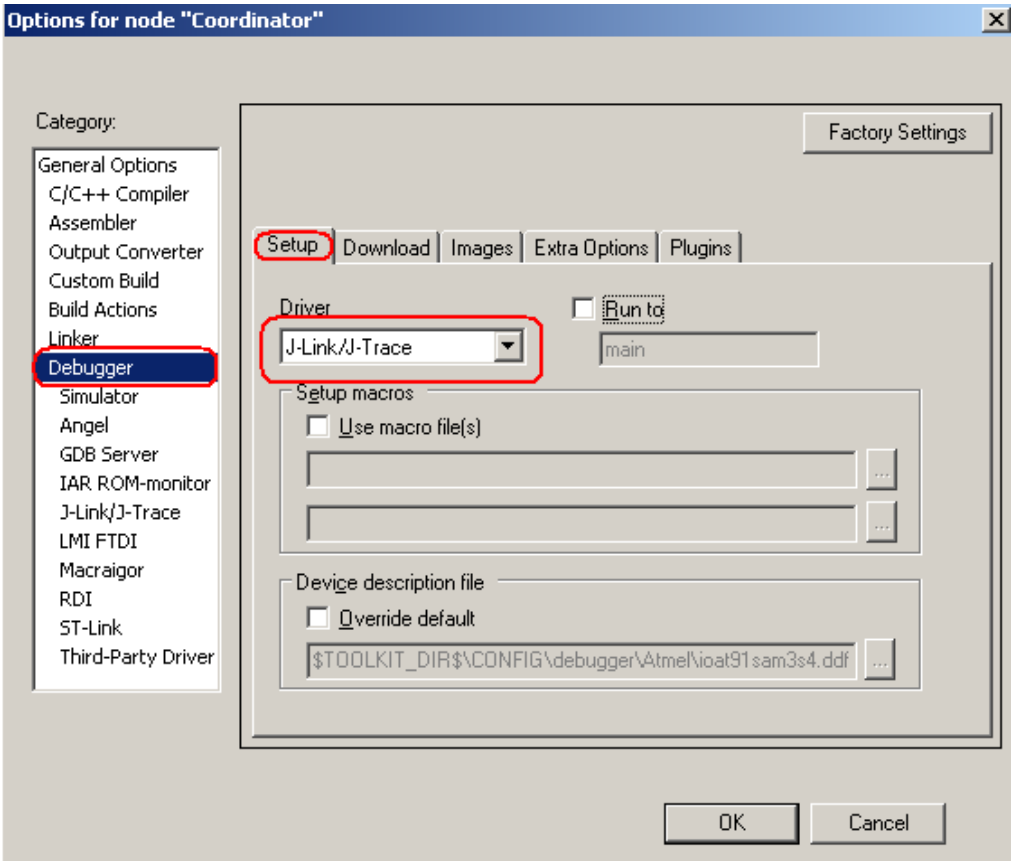
When using IAR ARM Embedded Workbench directly by double clicking the corresponding eww-file, the application can be downloaded onto the desired hardware platform as described below.

10.5.1 Starting the release build

The release build can simply be started as follows:

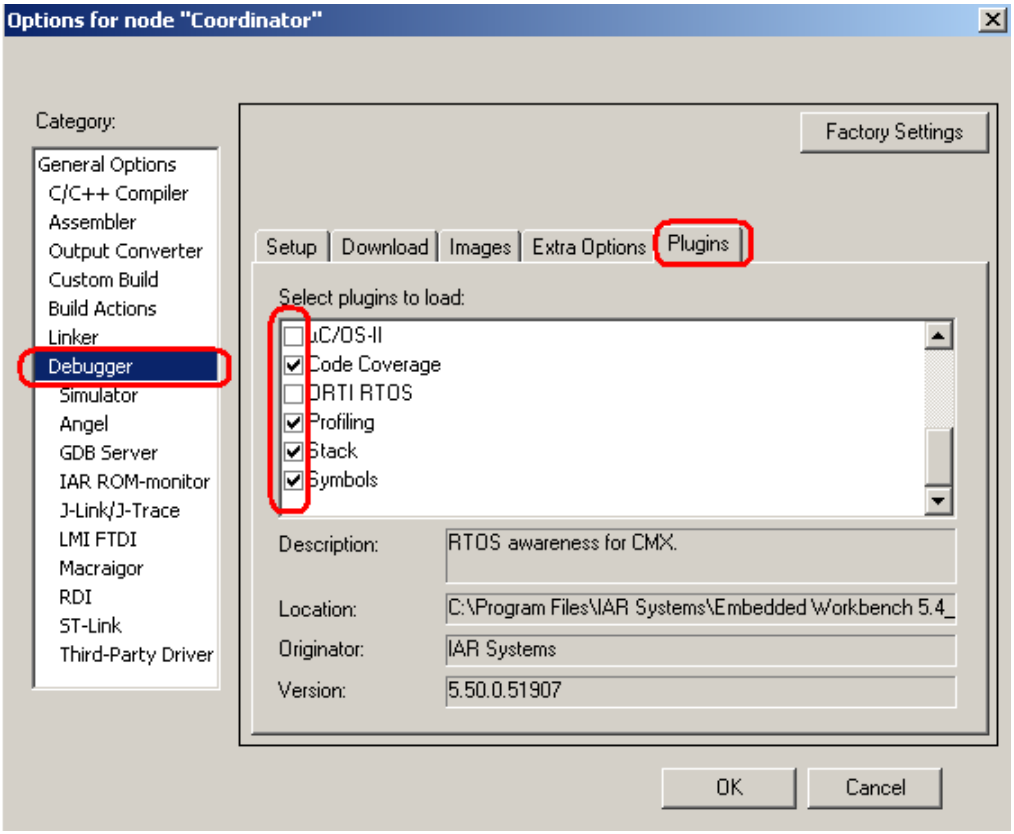
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Embedded Workbench by double clicking the desired eww-file
- Select the “Release” Workspace
- Open the “Options” window for the “Release” Workspace. Select the “Debugger” window and within this window select the “Setup” tab

Figure 10-16. IAR ARM Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



- Within the “Setup” tab select the “Driver” “J-Link/J-Trace” and deselect “Run to (main)”
- Change to the “Plugins” tab and deselect all entries

Figure 10-17. IAR ARM Embedded Workbench – “Options” -> “Debugger” -> “Plugins”.



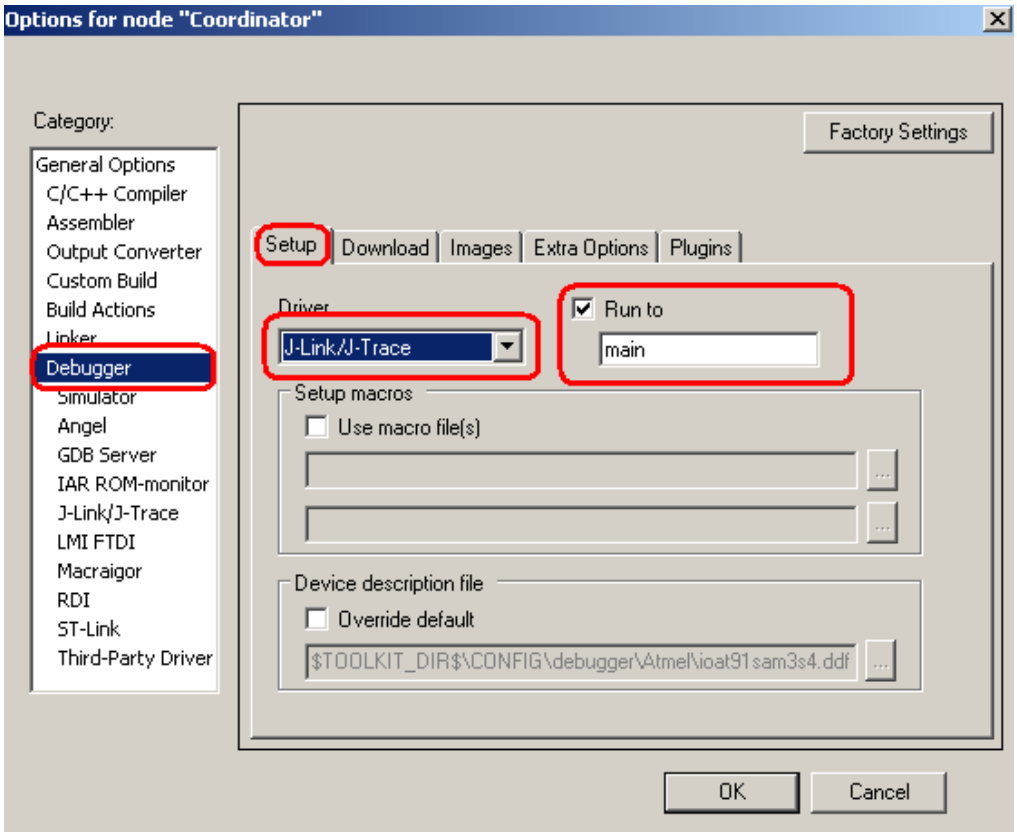
- Press “OK”
- Press the button “Download and Debug”

10.5.2 Starting the debug build

The debug build can simply be started as follows:

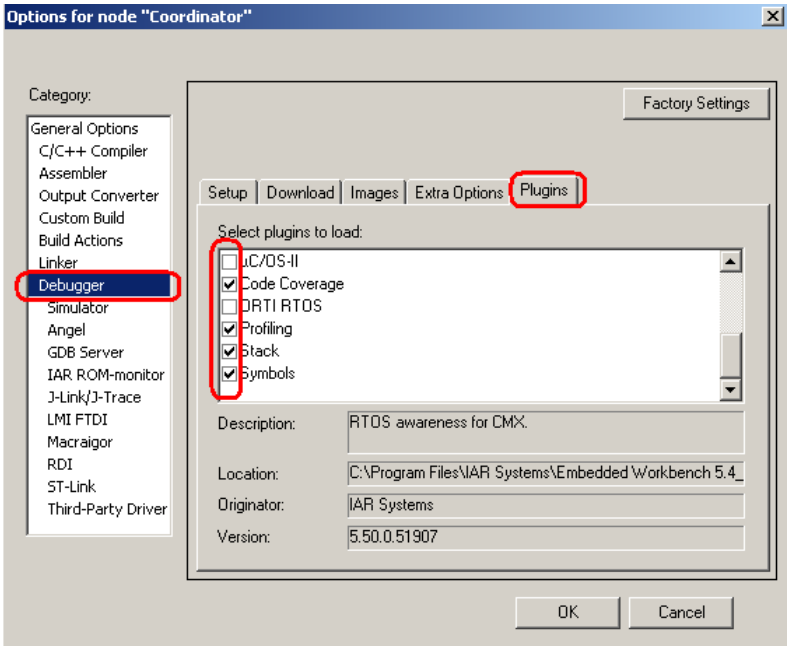
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE
- Open IAR Embedded Workbench by double clicking the desired eww-file
- Select the “Debug” Workspace
- Open the “Options” window for the “Debug” Workspace. Select the “Debugger” window and within this window select the “Setup” tab as shown in Figure 10-18

Figure 10-18. IAR ARM Embedded Workbench – “Options” -> “Debugger” -> “Setup”.



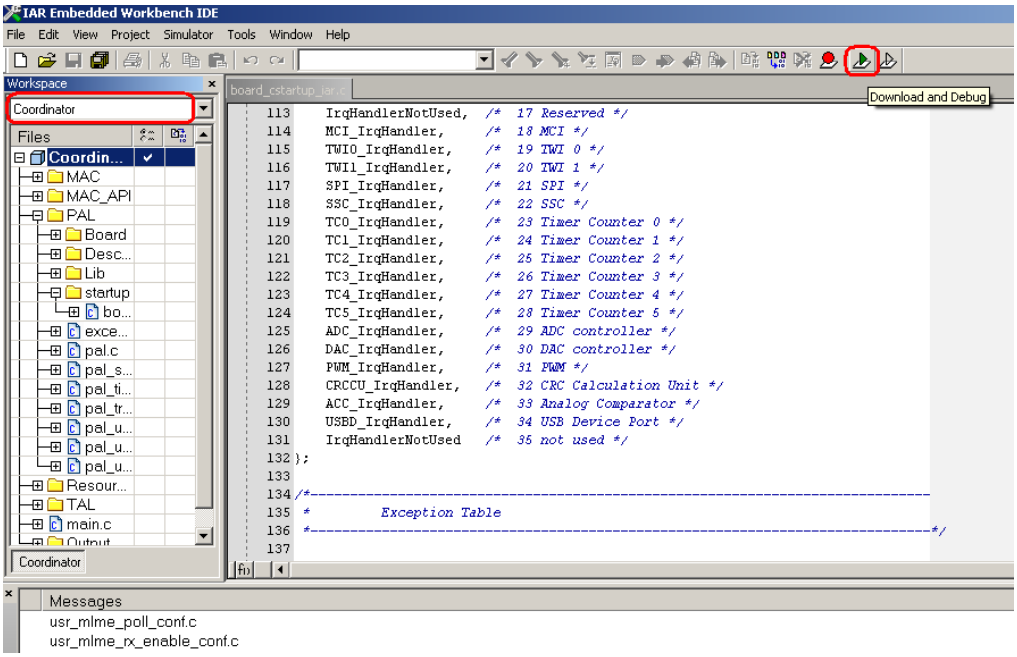
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and select “Run to (main)” (this is different to “Release” build)
- Change to the “Plugins” tab and select the entries as shown below

Figure 10-19. IAR ARM Embedded Workbench – “Options” -> “Debugger” -> “Plugins”.



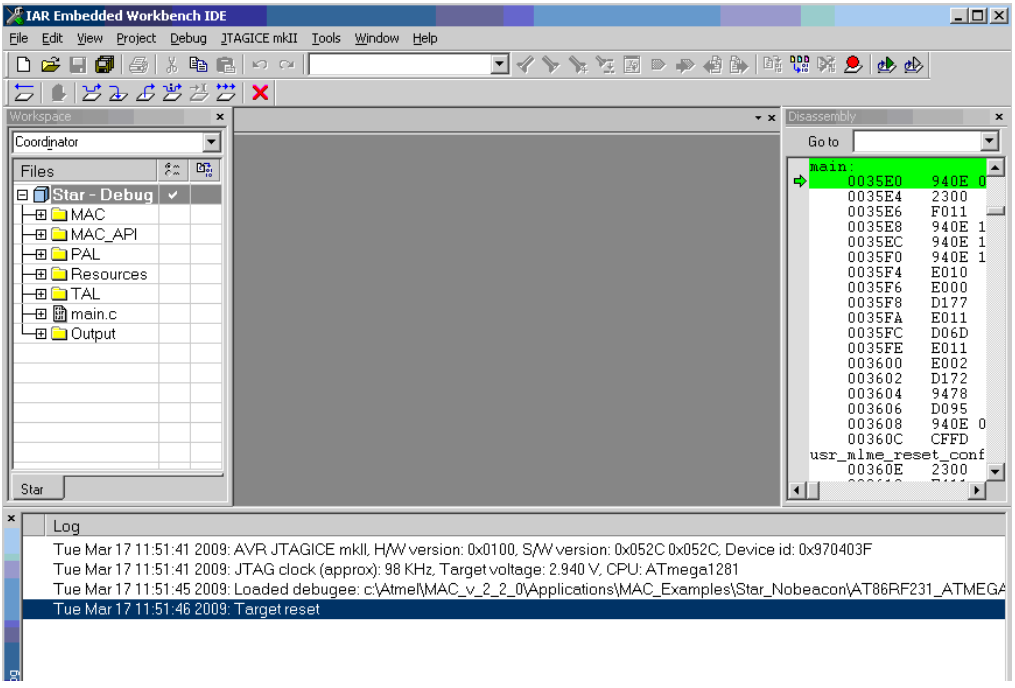
- Press “OK”
- Press the button “Download and Debug”

Figure 10-20. IAR ARM Embedded Workbench – start “Download and Debug”.



- A Window will pop-up indicating the status of the download of the binary
- After successful download the IAR ARM Workbench will looks as shown in Figure 10-21. After the download of the debug build the main function is displayed since all debug and source code information is included in the build

Figure 10-21. IAR ARM Embedded Workbench – successful download of debug build.



- Start the application by pressing the “F5” or the “Go” button

11 Example applications

The MAC package includes a variety of example applications which can be flashed on the supported hardware platforms and be executed immediately. On the other hand the complete source code is provided to help the application developer to more easily understand the proper utilization of the stack and to be able to build its own applications as fast as possible.

The provided example applications are categorized into two groups as MAC and TAL. These applications are located under the apps subdirectory of avr2025_mac directory.

These applications will be explained in more detail in the subsequent sections. If the example application makes use of the UART interface, the UART is set to 9600 (8,N,1).

11.1 Walking through a basic application

This section describes a basic example application provided with this MAC release (see Section 11.2.1.1) more thoroughly to allow better understanding of all other examples. The example serves as a first introduction on how to control the MAC-API and how to start an 802.15.4 compliant network. It can be used by a developer as a starting point for further designs. The example implements a network with star topology.

There are two types of nodes in the network: PAN Coordinator and device. A PAN Coordinator can be understood as the central hub of a network. It handles association requests from devices and assigns a short address if appropriate.

In this example, the PAN Coordinator does not initiate any data transmissions; it receives data from the associated devices. The usr_mcps_data_ind() callback function is provided only as stub and can be extended by user.

The Devices scan all channels for the PAN Coordinator. Once the coordinator is found, the Device will initiate the association procedure. If the association is also successful, the Device periodically (that is, every two seconds) sends out a data packet to the Coordinator. The data packets contain a random payload. As already mentioned earlier, this example can be extended by the user.

The following sections describe the application code in more detail.

11.1.1 Implementation of the coordinator

The source code of the coordinator is located in:

avr2025_mac\apps\mac\beacon\coord\main.c

and the header file in:

avr2025_mac\apps\mac\beacon\coord\ncp\app_config.h

Platform related project / Makefiles for GCC (AVR, AVR32, and ARM), AVR Studio, and IAR Workbench are located in the corresponding subdirectories

avr2025_mac/apps/mac/beacon/coord/ncp/<platform>

The example application can be opened using the Atmel Studio, the IAR EWW or any other editor. To open the example application project from the Atmel Studio select the file "beacon_coord.atsln / beacon_coord.cproj" or from the IAR EWW select the file " beacon_coord.atsln.eww". If the Atmel Studio is used, the source code can be compiled from the menu "Build" -> "Rebuild All". If the IAR EWW is used, the source code can be compiled from the menu "Project" -> "Rebuild All".

The main function of the coordinator performs the following steps:

Initialize the MAC layer and its underlying layers, like PAL, TAL, BMM:

```
if(MAC_SUCCESS != wpan_init())
{
 /* Alert to indicate something has gone wrong in the
application.*/
        app_alert();
}
```

Initialize LEDs:

```
LED_On(LED_START);          // indicating application is started
LED_Off(LED_NWK_SETUP);     // indicating network is started
LED_Off(LED_DATA);          // indicating data reception
```

Enable the global interrupts:

```
cpu_irq_enable();
```

Initiate a reset of the MAC layer:

```
wpan_mlme_reset_req(true);
```

Run the main loop:

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

Once the main loop is running ,the MAC layer will execute the previously requested reset and call the implementation of usr_mlme_reset_conf() callback function. Depending on the returned status information the program continues either with the request get the current page or with a new reset request.

```
void usr_mlme_reset_conf(uint8_t status)
{
    if (status == MAC_SUCCESS)
{
       wpan_mlme_get_req(phyCurrentPage);
}
else
{
    /* Something went wrong; restart. */
    wpan_mlme_reset_req(true);
}}
```

The request to get the current page will be processed once the control flow of the application enters the main loop again. The MAC layer will call the implementation of usr_mlme_get_conf () ,which will once again call get_req for the supported channels

which will call back this function again to set the short address using wpan_mlme_set_req.

```
void usr_mlme_get_conf(uint8_t status,
                       uint8_t PIBAttribute,
                       void *PIBAttributeValue)
{

    if((status == MAC_SUCCESS) && (PIBAttribute == phyCurrentPage))
    {
           current_channel_page = *(uint8_t *)PIBAttributeValue;
           wpan_mlme_get_req(phyChannelsSupported);
    }
    else if((status == MAC_SUCCESS) && (PIBAttribute ==
phyChannelsSupported))
    {
           uint8_t index;

           channels_supported = *(uint32_t *)PIBAttributeValue;

           for(index = 0; index < 32; index++)
           {
                  if(channels_supported & (1 << index))
                  {
                         current_channel = index + CHANNEL_OFFSET;
                         break;
                  }
           }
        /*
         * Set the short address of this node.
         * Use: bool wpan_mlme_set_req(uint8_t PIBAttribute,
         *                             void *PIBAttributeValue);
         *
         * This request leads to a set confirm message ->
usr_mlme_set_conf
         */
        uint8_t short_addr[2];

        short_addr[0] = (uint8_t)COORD_SHORT_ADDR;          // low
byte
        short_addr[1] = (uint8_t)(COORD_SHORT_ADDR >> 8);   // high
byte
        wpan_mlme_set_req(macShortAddress, short_addr);
    }
}
```

The request to set the shortaddress will be processed once the control flow of the application enters the main loop again. The MAC layer will call the implementation of usr_mlme_set_conf ()

```
void usr_mlme_set_conf(uint8_t status, uint8_t PIBAttribute)
{
    if ((status == MAC_SUCCESS) && (PIBAttribute ==
macShortAddress))
    {

            uint8_t association_permit = true;
    wpan_mlme_set_req(macAssociationPermit,&association_permit);
    }
    else if ((status == MAC_SUCCESS) && (PIBAttribute ==
macAssociationPermit))
    {

        bool rx_on_when_idle = true;

        wpan_mlme_set_req(macRxOnWhenIdle, &rx_on_when_idle);
    }
    else if ((status == MAC_SUCCESS) && (PIBAttribute ==
macRxOnWhenIdle))
    {
        /* Set the beacon payload length. */
        uint8_t beacon_payload_len = BEACON_PAYLOAD_LEN;
        wpan_mlme_set_req(macBeaconPayloadLength,
&beacon_payload_len);

    }
    else if ((status == MAC_SUCCESS) && (PIBAttribute ==
macBeaconPayloadLength))
    {
        /*
         * Once the length of the beacon payload has been defined,
         * set the actual beacon payload.
         */
        wpan_mlme_set_req(macBeaconPayload, &beacon_payload);
    }
    else if ((status == MAC_SUCCESS) && (PIBAttribute ==
macBeaconPayload))
    {
        if (COORD_STARTING == coord_state)
        {


            wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,
                               SCAN_CHANNEL,
```

```
                                    SCAN_DURATION_COORDINATOR,
                                    current_channel_page);
            }
            else
            {
                /* Do nothing once the node is properly running. */
            }
        }
        else
        {
            /* Something went wrong; restart. */
            wpan_mlme_reset_req(true);
        }
```

Depending on the status information, the application will proceed either with the request to set the association permit PIB attribute (see macAssociationPermit for further details).

The MAC layer will process the request and executes the function usr_mlme_set_conf().

Now the PIBAttribute parameter is equal to macAssociationPermit and the scan procedure will be initiated with wpan_mlme_scan_req(). Next time the main loop is running this request is processed by the MAC layer and the usr_mlme_scan_conf() callback function will be called with the result of the scan.

After the scan procedure has finished, a new network is started by invoking the function wpan_mlme_start_req().

```
void usr_mlme_scan_conf(uint8_t status,
                        uint8_t ScanType,
                        uint8_t ChannelPage,
                        uint32_t UnscannedChannels,
                        uint8_t ResultListSize,
                        void *ResultList)
{
    wpan_mlme_start_req(DEFAULT_PAN_ID,
                        current_channel,
                        current_channel_page,
                        DEFAULT_BO,
                        DEFAULT_SO,
                        true, false, false);

    /* Keep compiler happy. */
    status = status;
    ScanType = ScanType;
    ChannelPage = ChannelPage;
    UnscannedChannels = UnscannedChannels;
    ResultListSize = ResultListSize;
    ResultList = ResultList;
}
```

The PAN Coordinator is waiting for devices to associate. If a device initiates the association procedure, the Coordinator's MAC layer indicates this with the callback function usr_mlme_associate_ind(). The coordinator either responds with a short address for this device passed to wpan_mlme_associate_resp() or denies the request with the error code PAN_AT_CAPACITY. The function get_next_short_addr() is an application specific implementation and checks if an association request is accepted or not.

```
void usr_mlme_associate_ind(uint64_t DeviceAddress,
                            uint8_t CapabilityInformation)
{
    uint16_t associate_short_addr = macShortAddress_def;

    if (assign_new_short_addr(DeviceAddress, &associate_short_addr)
== true)
    {
        wpan_mlme_associate_resp(DeviceAddress,
                                 associate_short_addr,
                                 ASSOCIATION_SUCCESSFUL);
    }
    else
    {
        wpan_mlme_associate_resp(DeviceAddress,
associate_short_addr,
                                 PAN_AT_CAPACITY);
    }

    /* Keep compiler happy. */
    CapabilityInformation = CapabilityInformation;
}}
```

As soon as the usr_mlme_comm_status_ind() callback function is called by the coordinator's MAC layer with status MAC_SUCCESS, the device is associated successfully with the coordinator and will periodically (that is, about every two seconds) send data to the coordinator. Received data packets are indicated by the MAC layer to the application by calling the usr_mcps_data_ind() callback function. Further handling of the received (dummy) data can be implemented by the user as desired.

11.1.2 Implementation of the device

The source code of the device is located in:

avr2025_mac\apps\mac\beacon\dev\ main.c

and the header file in:

avr2025_mac\ apps \mac\beacon\dev\ncp\app_config.h

avr2025_mac/apps/mac/beacon/dev/ncp/<platform>

The main function of the device performs the following steps:

Initialize the MAC layer and its underlying layers, like PAL, TAL, BMM:

```
    if(MAC_SUCCESS != wpan_init())
```

```
    {
     /* Alert to indicate something has gone wrong in the
application.*/
           app_alert();
    }
```

Initialize LEDs:

```
    LED_On(LED_START);          // indicating application is started
    LED_Off(LED_NWK_SETUP);     // indicating network is started
    LED_Off(LED_DATA);          // indicating data reception
```

Enable the global interrupts:

```
    cpu_irq_enable();
```

Initiate a reset of the MAC layer:

```
    wpan_mlme_reset_req(true);
```

Run the main loop:

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

Once the main loop is running the MAC layer will execute the previously requested reset and call the implementation of usr_mlme_reset_conf() callback function. Depending on the returned status information the program continues either with the request to get the current page and channels supported as mentioned above or with a new reset request.

```
void usr_mlme_reset_conf(uint8_t status)
{
    if (status == MAC_SUCCESS)
    {
           wpan_mlme_get_req(phyCurrentPage);
    }
    else
    {
        /* Set proper state of application. */
        app_state = APP_IDLE;

        /* Something went wrong; restart. */
        wpan_mlme_reset_req(true);
    }
}
```

Once the main loop is running this request is processed by the MAC layer and the usr_mlme_scan_conf() callback function is called with the result of the scan req. The usr_mlme_scan_conf() function handles following cases:

- A coordinator was found
- No coordinator was found

```
{
   if (status == MAC_SUCCESS)
```

```
    {
        wpan_pandescriptor_t *coordinator;
        uint8_t i;
        coordinator = (wpan_pandescriptor_t *)ResultList;
        for (i = 0; i < ResultListSize; i++)
        {

            if ((coordinator->LogicalChannel == current_channel) &&
                (coordinator->ChannelPage == current_channel_page) &&
                (coordinator->CoordAddrSpec.PANId  ==  DEFAULT_PAN_ID)
&&
                ((coordinator->SuperframeSpec   &   ((uint16_t)1   <<
ASSOC_PERMIT_BIT_POS)) == ((uint16_t)1 << ASSOC_PERMIT_BIT_POS))
               )
            {
                /* Store the coordinator's address information. */
                coord_addr_spec.AddrMode = WPAN_ADDRMODE_SHORT;
                coord_addr_spec.PANId = DEFAULT_PAN_ID;

ADDR_COPY_DST_SRC_16(coord_addr_spec.Addr.short_address,  coordinator-
>CoordAddrSpec.Addr.short_address);
#ifdef SIO_HUB
                printf("Found network\r\n");
#endif

                /* Set proper state of application. */
                app_state = APP_SCAN_DONE;

                uint16_t pan_id;
                pan_id = DEFAULT_PAN_ID;
                wpan_mlme_set_req(macPANId, &pan_id);

                return;
            }

            /* Get the next PAN descriptor. */
            coordinator++;
        }
```

```
            wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,
                               SCAN_CHANNEL,
                               SCAN_DURATION_SHORT,
                               current_channel_page);
        }
        else if (status == MAC_NO_BEACON)
        {
            wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,
                               SCAN_CHANNEL,
                               SCAN_DURATION_LONG,
                               current_channel_page);
        }
        else
        {
            /* Set proper state of application. */
            app_state = APP_IDLE;

            /* Something went wrong; restart. */
            wpan_mlme_reset_req(true);
        }

        /* Keep compiler happy. */
        ScanType = ScanType;
        ChannelPage = ChannelPage;
        UnscannedChannels = UnscannedChannels;
    }
```

If the pre-configured coordinator is part of the scan result list, the device's application issues an association request to the coordinator. The association procedure is finished once the callback usr_mlme_associate_conf() is invoked and the corresponding status information is checked.

```
void usr_mlme_associate_conf(uint16_t AssocShortAddress,
                             uint8_t status)
{
if (status == MAC_SUCCESS)
    {
#ifdef SIO_HUB
        printf("Connected to beacon-enabled network\r\n");
#endif
```

```
            /* Set proper state of application. */
            app_state = APP_DEVICE_RUNNING;

            /* Stop timer used for search indication */
            sw_timer_stop(APP_TIMER);

            LED_On(LED_NWK_SETUP);
        }
        else
        {
            LED_Off(LED_NWK_SETUP);

            /* Set proper state of application. */
            app_state = APP_IDLE;

            /* Something went wrong; restart. */
            wpan_mlme_reset_req(true);
        }

        /* Keep compiler happy. */
        AssocShortAddress = AssocShortAddress;
    }
```

11.2 Provided examples applications

11.2.1 MAC examples

11.2.1.1 Nobeacon_Application

11.2.1.1.1 Introduction

The basic MAC Example Nobeacon Application deploys a non-beacon enabled network consisting of PAN Coordinator and Device utilizing the mechanism of indirect data transfer between Coordinator and Device.

In this example the Coordinator wants to send data to the Device and since a Device in a non-beacon enabled network is in sleep mode as default, direct transmission to the Device is not possible.

In order to enable communication with the Device, indirect data transmission using polling by the device is applied. For further explanation of indirect transmission see section 5.4.1/5.4.2. For power management and indirect transmission see Section 6.2.4.

After the Device receives the data from the Indirect_Data_Queue from the Coordinator, the Device sends back the data received from the Coordinator to the Coordinator itself by direct data transmission

This example application uses MAC-API as interface to the stack.

The application and all required build files are located in directory avr2025_mac\apps\mac\no_beacon\. The source code of the application can be

found in the subdirectories coord or dev. The common source code for handling Serial I/O can be found in the subdirectory wireless\addons\sio2ncp.

11.2.1.1.2 Requirements

The application requires (up to three) LEDs on the board in order to indicate the proper working status. A sniffer is suggested in order to check frame transmission between the nodes.

For further status information this application requires a serial connection. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to start the application and to see the output of the application please start a terminal application on your host system and press any key for the application to begin.

11.2.1.1.3 Implementation

The PAN Coordinator starts a PAN at first channel(Channel 11 for 2.4GHz) with the PAN ID DEFAULT_PAN_ID. The Device scans for this network and associates to the PAN Coordinator.

Once the device is associated, it uses a timer that fires every 5 seconds to poll for pending data at the coordinator by means of transmitting a data request frame to the coordinator. On the other hand the coordinator every 5 seconds queues a dummy data frame for each associated device into its Indirect-Data-Queue. If the coordinator receives a data request frame from a particular device, it transmits the pending data frame to the device. Device after receiving the data from the Coordinator sends back the data to the Coordinator itself by direct data transmission. While the device is idle (when the timer is running) the transceiver enters sleep in order to save as much power as possible.

11.2.1.1.4 Limitations

- The current channel is coded within the application. In order to run the application on another channel, change the default channel in file *main.c* and re-built the application.
- Currently only six devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.

11.2.1.2 Beacon_Application

11.2.1.2.1 Introduction

The basic MAC Example Beacon Application deploys a beacon enabled network consisting of PAN Coordinator and (up to 100 associated) Devices. The application shows how basic MAC features can be utilized within an application using beacon-enabled devices, such as announcement of pending broadcast data at the coordinator within beacon frames (that is, whenever the coordinator has pending broadcast data to be delivered in a beacon-enabled network it sets the Frame Pending Bit in the transmitted beacon frame) and synchronization with the coordinator and utilization of beacon payload by the coordinator. Also the Coordinator in this example wants to send data to the Device using indirect transmission. In order to enable communication with the Device, indirect data transmission using polling by the

device is applied. For further explanation of indirect transmission see Section 5.4.1 /5.4.2. For power management and indirect transmission see Section 6.2.4.

This example application uses MAC-API as interface to the stack.

The application and all required build files are located in directory avr2025_mac\apps\mac\beacon. The source code of the application can be found in the subdirectories Coord or Dev.

11.2.1.2.2 Requirements

The application requires (up to three) LEDs on the board in order to indicate the proper working status. A sniffer is suggested in order to check frame transmission between the nodes.

For further status information this application requires a serial connection. Depending on the available serial I/O interface for each board this can be either UART or USB. In order to start the application and to see the output of the application please start a terminal application on your host system and press any key for the application to begin.

11.2.1.2.3 Implementation

The coordinator in this application creates a beacon-enabled network and periodically transmits beacon frames with a specific beacon payload. The beacon payload changes after a certain time period.

Each device of this application joins the beacon-enabled network by first attempting to synchronize with the coordinator to be able to receive each beacon frame. Once it has successfully synchronized with the coordinator, the device associates with the coordinator.

The connected devices wake-up whenever a new beacon frame is expected, extract the received payload of each beacon frame from its coordinator. This received payload is printed on the terminal and sent back to the coordinator by means of a direct data frame transmission to the coordinator. After successful beacon reception and data transmission, the devices enter sleep mode until the next beacon is expected.

The coordinator indicates each received data frame from each device on its terminal.

Whenever a device looses synchronization with its parent, it initiates a new synchronization attempt.

Also in this application the coordinator periodically tries to transmit broadcast data frames to all children nodes in its network. When ever broadcast frames are pending at the coordinator, it sets the Frame Pending Bit of the next beacon frame.

The connected devices wake-up whenever a new beacon frame is expected. Once it receives a beacon frame that has the Frame Pending Bit set, it remains awake until a broadcast data frame is received. After successful reception of the expected broadcast data frame the devices enter sleep mode until the next beacon is expected.

Once the device is associated, when it receives a beacon frame that has the data Frame Pending Bit set, the device sends a data request frame to the coordinator. On the other hand the coordinator every 5 seconds queues a dummy data frame for each associated device into its Indirect-Data-Queue. If the coordinator receives a data request frame from a particular device, it transmits the pending data frame to the

device. While the device is idle (when the timer is running) the transceiver enters sleep in order to save as much power as possible.

11.2.1.2.4 Limitations

- The current channel is coded within the application. In order to run the application on another channel, change the default channel in file *main.c* and re-built the application.
- Currently 100 devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.

11.2.1.3 No_beacon_sleep

11.2.1.3.1 Introduction

The application No_beacon_sleep provides a simple start network application based on IEEE 802.15.4-2006.The application uses two nodes: a PAN Coordinator (1) and an End Device (2). The firmware is implemented as such that a node can either act as a PAN Coordinator or an End Device.

This application demonstrates how MCU sleep modes can be utilized in the wireless networks inorder to save more power.By default, End Device MCU is full sleep mode wakes for every 2 seconds and sends a data to the coordinator before going to sleep again.

This example application uses MAC-API as interface to the stack.

The application is located in directory avr2025_mac\apps\mac\no_beacon_sleep.

11.2.1.3.2 Requirements

The application requires (up to three) LEDs on the board in order to indicate the proper working status.A sniffer is suggested in order to check the proper association and the data transfer between the devices.

11.2.1.3.3 Implementation

The application works as described subsequently.

Node one:

- Switch on node one.
- LED 0 indicates that the node has started properly.
- Flashing of LED 1 indicates that the node is scanning its environment. Scanning is done three times on each available channel depending on the radio type.
- If no other network with the pre-defined channel and PAN Id is found, the node establishes a new network at the pre-defined channel (channel 20 for 2.4GHz radio). This node now becomes the PAN Coordinator of this network. The successful start of a new network is indicated by switching LED 1 on.

Node two:

- Switch on the other node.

- LED 0 indicates that the node has started properly. Flashing of LED 1 indicates that the node is scanning its environment. Scanning is again done three times on each available channel depending on the radio type.
- If a proper network is discovered, the node joins the existing network, indicates a successful association by switching on LED 1 and goes both MCU and transceiver to sleep.
- For every 2 seconds the end device wakes up and sends a data to the coordinator before going to sleep again.

11.2.1.3.4 Limitations

- The current channel is coded within the application. In order to run the application on another channel, change the default channel in file main.c and re-built the application.
- Currently only 2 devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.

11.2.2 TAL examples

11.2.2.1 Performance_Analyzer

11.2.2.1.1 Introduction

The TAL example Performance_Analyzer is a GUI-based application used to demonstrate various features and capabilities of Atmel 802.15.4 Transceivers such as

- Range of the Transceiver for peer-to-peer communication (Range Measurement)
- Robust Link Quality
- Antenna Diversity
- TX Power of Radio
- Rx Sensitivity
- CSMA-CA Transmission
- Read / Write Transceiver Registers
- Continuous transmit test modes
- Reduced Power Consumption mode
- Energy Detection
- Cyclic Redundancy Check
- Battery Monitor

The different states of the Performance Analyzer application are explained and also the state diagram is shown in Figure 11-1.Each state is represented by a number in the state diagram

1 INIT

- Initializes all underlying layers like TAL, PAL and Resource Management (BMM/QMM).
- Initializes all board utilities like LEDs and buttons etc.

2 WAIT_FOR_EVENT

- Initializes the TAL PIB attributes PAN Id with 0xCAFE, physical channel with 0x0B on both the nodes, and their radios are kept in receive state.
- Continuously search for the user events like Initiating peer search from GUI or key press on the board (Peer Request) received on air.

3 PEER_SEARCH_RANGE_TX

- Enters after key press event is detected from user. Peer Search process in Range Measurement mode as Transmitter node starts here.
- Nodes shall go into different sub states like PEER_INIT, PEER_REQ_SENT, PEER _RSP_RCVD, PEER_SEARCH_SUCCESS.

4 PEER_SEARCH_PER_TX

- Enters after initiating peer search from Wireless Composer. Peer Search process in PER mode as Transmitter node starts here.
- Nodes shall go into different sub states like PEER_INIT, PEER_REQ_SENT, PEER _RSP_RCVD, PEER_SEARCH_SUCCESS.

5 PEER_SEARCH_RANGE_RX

- Enters after receiving a valid frame (Peer Request) from Transmitter. Peer Search process in Range Measurement mode as Reflector node starts here.
- Nodes shall go into different sub states like PEER_INIT, PEER _RSP_SENT, WAIT_FOR_ PEER_CONF, PEER_SEARCH_SUCCESS.

6 PEER_SEARCH_PER_RX

- Enters after receiving a valid frame (Peer Request) from Transmitter. Peer Search process in PER Measurement mode as Reflector node starts here.
- Nodes shall go into different sub states like PEER_INIT, PEER _RSP_SENT, WAIT_FOR_ PEER_CONF, PEER_SEARCH_SUCCESS.

8 RANGE_TEST_TX_ON

- Starts Range Measurement mode as Transmitter.
- Enters after successful Peer Search on key press event.
- Continuous packet transmission with a period of 200 ms time interval.
- Enters from RANGE_TEST_TX_OFF(7) state on button press

7 RANGE_TEST_TX_OFF

- Starts Range Measurement mode as Reflector.
- Enters after successful Peer Search on a valid frame (Peer Request in Range Measurement mode) received from Transmitter.
- Receives the packets from other node and acknowledges (by an 802.15.4 protocol ACK) each packet received
- Enters from RANGE_TEST_TX_ON(8) state on button press

10 PER_TEST_INITIATOR

- PER Measurement mode as Transmitter
- Enters after successful Peer Search on initiating from Wireless Composer.
- Node shall go into sub states like TX_PER, TEST_FRAMES_SENT, WAIT_FOR_TEST_RES, SET_PARAMETER, IDENTIFY_PEER, DIVERSITY_SET_REQ, DIVERSITY_STATUS_REQ, CRC_SET_REQ_WAIT, CRC_STATUS_REQ_WAIT, CONTINOUS_TX_MODE, RESULT_RSP, DIV_STAT_RSP etc.

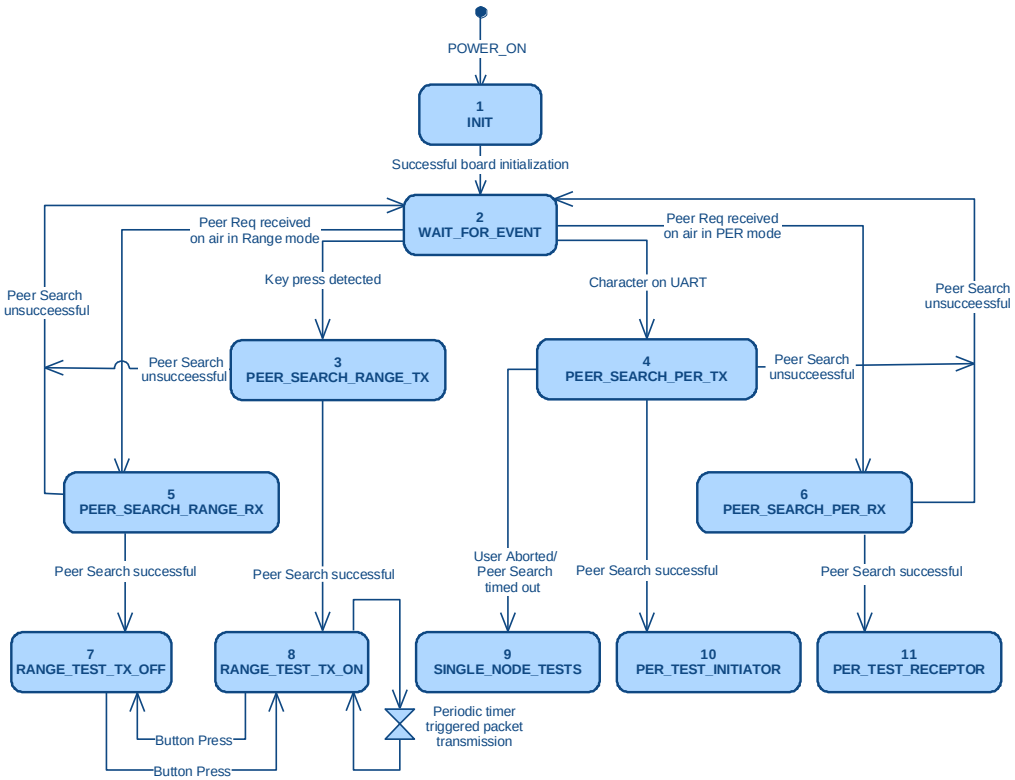
11 PER_TEST_RECEPTOR

- PER Measurement mode as Reflector
- Enters after successful Peer Search on a valid frame (Peer Request) received from Transmitter.
- Node shall respond back for the cmds sent from different states in Transmitter node.

9 SINGLE_NODE_TESTS

- Transmitter node in single node operation.
- Enters on user abort or Peer Search time out, i.e. No Peer Response is received during the Peer Search process.
- Node shall go into sub states similar (but applicable) states in PER_TEST_INITIATOR (10) by considering the Peer Search status as failed.

Figure 11-1. Performance Analyzer Application State Diagram



11.2.2.1.2 Requirements

Some of the pre-requisites for using this application are the Wireless Composer GUI for PER Mode of Operation which can be found from Atmel Studio Extensions,Atmel Studio 6.0 and above.

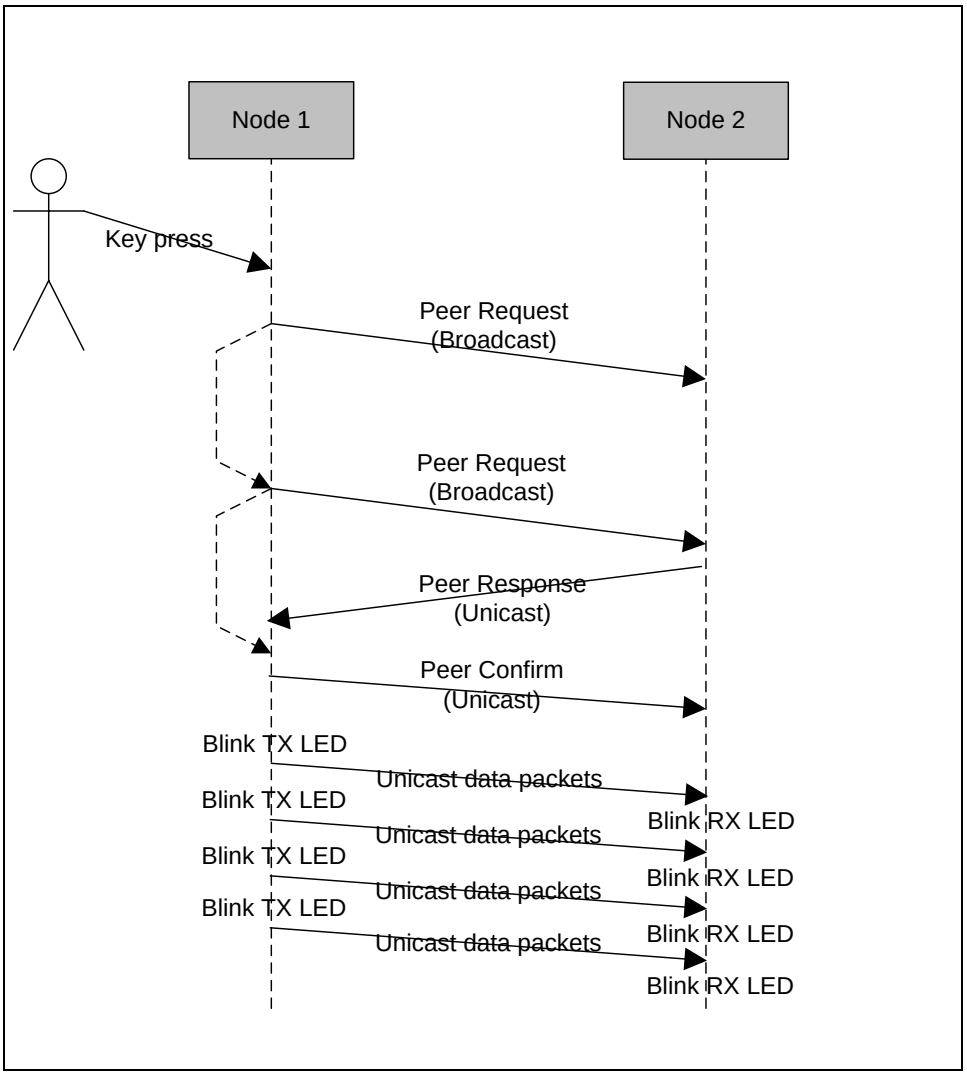
11.2.2.1.3 Implementation

11.2.2.1.3.1 Range Measurement mode

During Range measurement, the transmitter node will initiate a sequence to find a peer node. Once peer node is found, packet transmission is initiated by the

transmitter to the receiver. The Receiver node acknowledges each packet received. The procedure used for finding the peer node is explained in detail in the Section 10.2.2.1.3.3

Figure 11-2. Sequence diagram of Range measurement



The LED on the receiver will blink sequentially and repeat at the rate at which the packets are received. The LED on the transmitter will blink sequentially & repeat at the rate at which the packets are transmitted. The LED will blink at a constant rate on the transmitter as the packets are transmitted at a constant duration. The packet format is described in the section Packet Format

11.2.2.1.3.1.1 Packet Format

The transmitted packet format and content for the operation mode is customized to suite only the requirements of this application. The format is shown in Table 11-5:

Table 11-5. Packet payload format for Range measurement

Octets	**1**	**1**	**8**
Payload	Command ID	Sequence Number	Packet Count

Field Description is as follows:

- Command-ID:

  (0x00) the value of command ID (DATA FRAME)

- Sequence Number:

  To have a sequence of packets transmitted from the transceiver to the receiver. The range of sequence number is 0x00 ~ 0xFF, will roll-over respectively. This is to track the packet loss for a continuous transmission of packets.

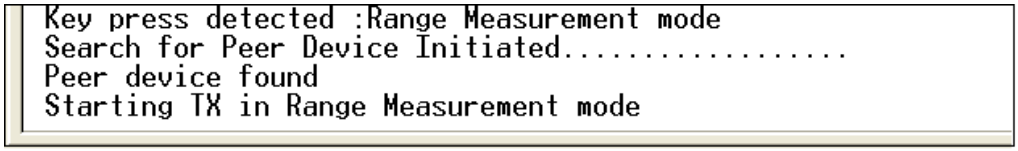
- Packet Count:

  The packet format maintains a 32 bit packet counter to count the number of packets at any instant, by using an external sniffer tool. Once the limit is reached (4294967295) then the counter resets itself to start again from 0x00000000.

11.2.2.1.3.1.2 Debug message support for – Range measurement
Debug prints can be viewed if the node is connected to a UART terminal

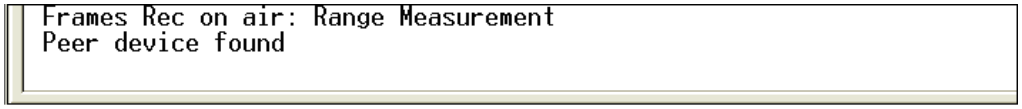
The node on which the key was pressed will display a print as shown in Figure 11-3. This node initiates the transmission and calls itself the TX node

Figure 11-3. Initializing Range measurement - transmitter (TX)



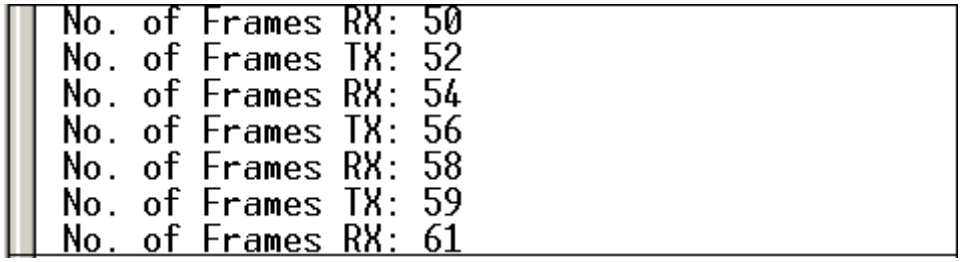
The node connected to the TX node will display a print as shown in Figure 11-4. This node receives the packets and calls itself the RX node.

Figure 11-4. Initializing Range measurement - receiver (RX)



On input of any character in the UART Terminal it prints the statistics of the messages received and messages sent as shown in Figure 11-5. Two way communications can be enabled if the button is pressed on both the nodes.

Figure 11-5. Statistics of Range measurement



11.2.2.1.3.2 PER Measurement mode

The primary intent of this application is PER measurement. One of the nodes should be connected to the Wireless Composer GUI in studio and other node can be connected to the Terminal Window or can be left alone. The node connected to the Wireless Composer is referred as transmitter and other node is referred as receiver. For Comprehensive information of using this application along with the Wireless Composer please refer http://www.atmel.no/webdoc/wirelesscomposer/wirelesscomposer.html

11.2.2.1.3.2.1 Sensitivity testing

In the IEEE 802.15.4 standard, the receiver sensitivity is defined as the lowest received signal power that yields a packet error rate loss of less than 1%. IEEE 802.15.4 requires only -85 dBm of sensitivity for operations in the 2.4 GHz ISM band.

Using the PER test, sensitivity can be tested by configuring one of the nodes as a transmitter and another as a receiver. The number of packets to be transferred is configured using Wireless composer and all the packets received by the receiver are acknowledged. The receiver keeps a count of the packets received. At end of the test the transmitter asks the receiver for the test results. The test results are displayed on the Wireless Composer GUI.

For this test, unicast with ACK is used. Since the boards are not factory connected they are field connected by the method described in the Section 11.2.2.1.3.3. Please refer the Atmel Transceiver datasheet for expected sensitivity.

11.2.2.1.3.2.2 TX Power handling

The Atmel AT86RF231 provides the programmable TX output power from -17dBm to 3dBm. The output power of the transmitter can be controlled over a range of 20 dB. Default TX power is set to 3dBm. The PER measurement mode gives an option to configure the TX out power in the form of absolute power in dBm or TX PWR register value in the composer GUI. If the AT86RF231 is connected with front end module (e.g.REB231FE2 –EK kit), TX power can be extended till 21dBm.

The control of an external RF front-end is done via digital control pins DIG3/DIG4. The function of this pin pair is enabled with register bit PA_EXT_EN (register 0x04, TRX_CTRL_1). While the transmitter is turned off pin 1 (DIG3) is set to low level and pin 2 (DIG4) to high level. If the radio transceiver starts to transmit, the two pins change the polarity. This differential pin pair can be used to control PA, LNA, and RF switches.

If the AT86RF231 is not in a receive or transmit state, register bit PA_EXT_EN (register 0x04, TRX_CTRL_1) is disabled to reduce the power consumption or avoid leakage current of external RF switches and other building blocks, especially during SLEEP state. If register bits PA_EXT_EN = 0, output pins DIG3/DIG4 are pulled-down to analog ground.

If AT86RF231 is connected with RF front end module (i.e.REB231FE2 –EK kit), default TX power is 20dBm.To ensure FCC compliance TX Power of CH26 has to be limited to 13dBm (TX_PWR = 0x0d).So if user changes the channel to 26 and the default TX power is more than 13dBm, it shall be automatically changed to 13dBm.For CH26 the allowed range of TX power is 4dbm to 13dBm, for other channels it is 4dBm to 21dBm.

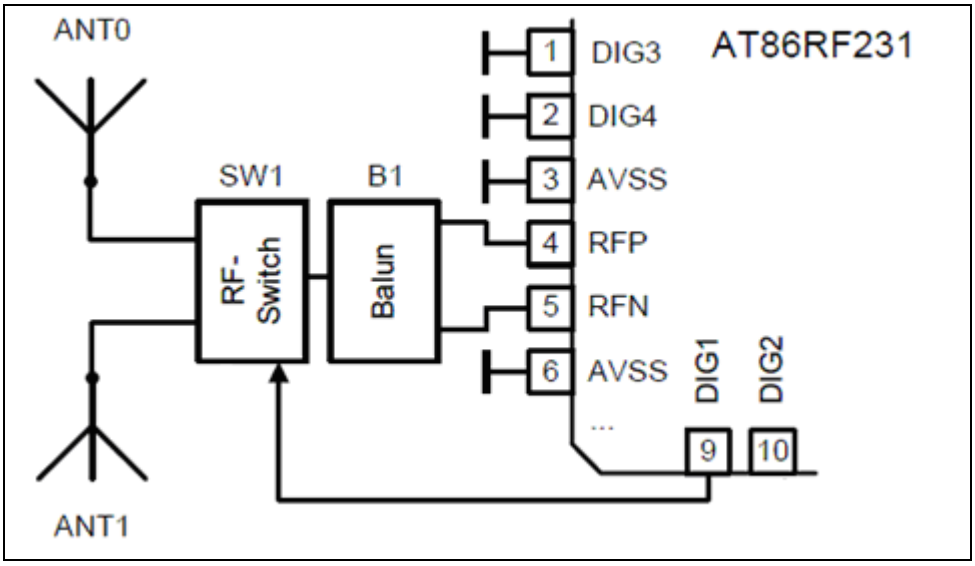
11.2.2.1.3.2.3 Diversity feature testing

In a multi-path environment, several versions of the same signal with different phases, delays, and attenuations will be added together at the receiver location, so there is always the possibility that at some locations, the signals could cancel each other out almost entirely. One way to overcome the multi-path issue is to use the *receiver antenna diversity* technique. In this method, two antennas are used instead of one in the receiver. This way if one antenna is in a multi-path null (also known as *deep-fading region)*, the other antenna has a good chance of being outside the deep-fading region. The receiver can switch between these two antennas to escape from a multi-path null.

Enable Diversity in the radio by Selecting the Antenna Diversity tab in the GUI. Diversity can also be configured in the reflector node by using Antenna Diversity on Peer. (By default diversity is enabled). AT86RF231 has a built-in antenna diversity feature. Upon reception of a frame the AT86RF231 selects one antenna during preamble field detection. The REB Rx path is shown in the Figure 11-6.

The antenna diversity feature can be tested by doing the PER measurement on conductive medium.

Figure 11-6. REB Rx path



11.2.2.1.3.2.4 Read Write Radio Registers

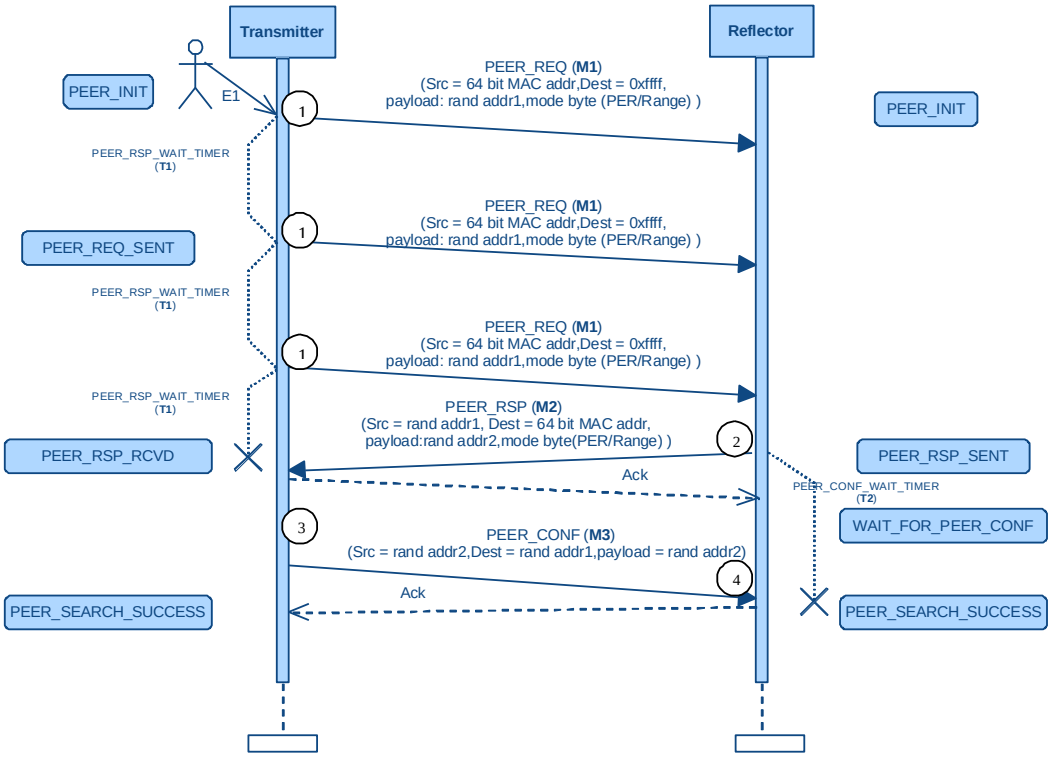
The Atmel AT86RF231 provides a register space of 64, 8-bit registers, used to configure, control and monitor the radio transceiver. The PER measurement mode gives an option to write / read the content of range of these registers through Wireless Composer. Please note that when writing to a register, any reserved bits shall be overwritten only with their reset value.

NOTE If the nodes are connected to each other and registers related to channel selection (0x08-PHY_CC_CCA) or channel page selection (0x0C-TRX_CTRL_2) or transmission power setting (0x05-PHY_TX_PWR) are changed, the changes will be reverted to the old setting to prevent loss of connection with remote node. To test these registers use the PER measurement mode with Peer Search aborted.

11.2.2.1.3.3 Peer Search Process

The Peer Search process is described in detail below and is illustrated by a sequence diagram. Initially the nodes uses their 64-bit MAC address as source address during the Peer Search Process to get connected each other. During the Peer search process 16-bit random address shall be assigned to both the devices.

Figure 11-7. Sequence Diagram for Peer Search process



1. On pressing the button T1 or Initiating peer search through Wireless Composer, which is shown as an event E1, the node becomes Transmitter and sends a **Peer Request (M1)** as broadcast at a constant period. Then the node enters into PEER_REQ_SENT state. A timer PEER_RSP_WAIT_TIMER (T1) is started to send the Peer Requests with 50ms time interval. A counter is started to count the no. of Peer Requests sent. On every expiry of the timer T1, count value is incremented by one. The Transmitter stops sending the Peer Request if the count reaches to 0xff and the node goes to the WAIT_FOR_EVENT state again. The Peer Request packet payload consists of a 16 bit random number and used as source address (rand addr1) of the receiving node and the mode byte to indicate in which mode (PER/Range) this Peer Request has been sent.

2. If any other node is in power on state and it receives this Peer Request packet, it becomes Reflector, assigns itself this address (rand addr1) as its source address (IEEE 802.15.4 protocol) and sends a **Peer Response (M2)** which is a unicast to Transmitter as destination address using extended MAC address. The Reflector node then enters into PEER_RSP_SENT state. After receiving the acknowledgement for Peer Response from Transmitter node, Reflector node

enters into WAIT_FOR_PEER_CONF state, and a timer called PEER_CONF_WAIT_TIMER (T2) is started with a timeout value of 200ms. If the Peer confirm is not received within this time, the node goes to the WAIT_FOR_EVENT state again. Peer Response payload consists of a random generated 16 bit number used as source address (rand addr2) of the receiving node i.e. Transmitter and the mode byte to indicate in which mode (PER/Range) this Peer Response has been sent.

3. On receipt of the Peer Response packet, the Transmitter node assigns itself the address received in the frame(rand addr2) as its source address (IEEE 802.15.4 protocol) and the node sends a **Peer Confirm (M3)** which is a unicast to Reflector (source address = rand addr2 and destination address = rand addr1). Peer Confirm frame consists of the address (rand addr2) sent to the Transmitter node in the payload of Peer Response. After receiving the acknowledgement for Peer confirm from Reflector, Transmitter node enters into PEER_SEARCH_SUCCESS state.

4. On receipt of the Peer Confirm, Reflector node stops the timer T2, checks the packet and verifies the address is the same as the address it sent to the Transmitter node in the Peer Response (rand addr2). If it is the same, Reflector node enters into PEER_SEARCH_SUCCESS, and the nodes are connected each other. If Reflector node does not receive any Peer Confirm within timeout the node goes into WAIT_FOR_EVENT state. This process is followed to connect only a pair of nodes if two nodes are in the powered on state.

The boards are assigned random addresses and if the Peer Search process is successful then the test commences and the nodes operate in the respective operation modes.

11.2.2.1.3.4 Configuration mode

Configuration mode is the startup mode in which two nodes (i.e. Transmitter and Reflector) can connect each other if they are only within in the vicinity of one meter approximately. This is to restrict the distance range for connecting devices. This is used generally in seminars where there are no. of participants may start the Performance test at the same time. With configuration mode provided, each individual participant can make sure that his/her two devices are only getting connected without disturbing the other devices. Once the Peer Search is done successfully, the nodes shall come to the normal mode where the nodes can be kept afar.

User can enter into configuration mode by pressing the button T1 while Power on /reset. Then user can initiate Peer Search by Initiating Peer search in composer for PER measurement or button press for Range measurement. Then the device (transmitter) shall go to the low TX level (TX_PWR = 0x0F) and sends the peer request with the config_mode bit set to true. On the other device (reflector), if the peer request received with config_mode bit true, it checks the ED level and if it is above defined threshold, it will connect with the transmitter device. This ED threshold is defined to allow connecting with the devices which are approximately below one meter range. In configuration mode Transmitter node works with lowest TX power and the Reflector sends the received packets to the application layer if it is above ED threshold.

After successful Peer Search, devices shall come to the normal mode, which means Transmitter node work with default TX power and the receiver node shall not do any filtering based on the ED threshold value

11.2.2.1.3.5 Application Contents

Using the PAL and TAL package for the corresponding controller and radio transceiver, the entire implementation of the Performance Analyzer Application requires an additional set of following files.

Table 11-10. File list for Performance Test application

File name	Short description of contents
main.c	Main application task ,TX-done handler & Rx callback functions (represents entire state machine)
user_interface.c	User interface related functions like LED, Serial prints, buttons
init_state.c	Board and startup application initialization functions (represents INIT State)
wait_for_event.c	Functions for waiting for events like character on UART, Key press etc (represents WAIT_FOR_EVENT state)
peer_search_initiator.c	Proprietary Peer Search related functions as Initiator (represents PEER_SEARCH_ PER_TX & PEER_SEARCH_RANGE_TX states)
peer_search_receptor.c	Proprietary Peer Search related functions as Receptor (represents PEER_SEARCH_ PER_RX & PEER_SEARCH_RANGE_RX states)
range_measure.c	Range measurement related functions (represents RANGE_TEST_TX_OFF & RANGE_TET_TX_OFF states)
per_mode_initiator.c	PER measurement related functions as Initiator (represents PER_TEST_INITIATOR & SINGLE_NODE_TESTS states)
per_mode_receptor.c	PER measurement related functions as Receptor (represents PER_TEST_RECEPTOR state)
per_mode_common_utils.c	Common utility function related to all PER mode states
perf_api_serial_handler	Performs serial i/o communication with the composer GUI

11.2.2.1.4 Limitations

Only two devices are allowed connecting each other and communicating.

A switch on the board is required to enter into Range Measurement mode and configuration mode

In case of the ATREB231FE2-EK, two nodes should be kept at least 50cm apart to avoid the distortion due to high TX Power levels.

11.3 Common SIO handler

Applications that require Serial input or output Communication (via UART or USB) can use the sio2host and sio2ncp addons in

avr2025_mac\addons

11.3.1 SIO2HOST

The sio2host addon is used for serial i/o communication between the host(PC) and the device.The sio2host can be either USB or UART.

11.3.2 SIO2NCP

The sio2ncp addon is used for serial i/o communication between the host(device 1) and the NCP(Network-Co Processor)(device2) for 2p approach.The sio2ncp will be UART.

11.4 Handling of callback stubs

The MAC stack must support asynchronous operation by all layers, for instance to allow for callbacks from lower layers back to higher layers without blocking the control flow. This is required to implement the request/confirm or indication/response primitive handling. A common way of implementing asynchrony operation by lower layers is the installation of callback functions, which are called a lower layer, but actually implemented in the higher layer. Callbacks are required by both the TAL and the MAC layer.

11.4.1 MAC callbacks

The MAC Core layer (MCL) requires the following callback functions:

- usr_mcps_data_conf
- usr_mcps_data_ind
- usr_mcps_purge_conf
- usr_mlme_associate_conf
- usr_mlme_associate_ind
- usr_mlme_beacon_notify_ind
- usr_mlme_comm_status_ind
- usr_mlme_disassociate_conf
- usr_mlme_disassociate_ind
- usr_mlme_get_conf
- usr_mlme_orphan_ind
- usr_mlme_poll_conf
- usr_mlme_reset_conf
- usr_mlme_rx_enable_conf
- usr_mlme_scan_conf
- usr_mlme_set_conf
- usr_mlme_start_conf
- usr_mlme_sync_loss_ind

These callback functions are declared in file *mac/inc/mac_api.h*. Each MAC based application (HIGHEST_STACK_LAYER = MAC) needs to implement these usr_...() callback functions.

For example an application that uses data transmission mechanisms, will call a function wpan_mcps_data_request, which in return requires the implementation of the corresponding asynchronous callback function usr_mcps_data_conf() to indicate the status of the requested data transmission.

But the same application might, for example, not want to use the MAC primitive MLME-SYNC-LOSS.indication. Nevertheless the callback function usr_mlme_sync_loss_ind() needs to be available or the linker generates a build error. This can be solved by either implementing an empty stub function in the application, or, more conveniently, use an already existing stub function in files eg. usr_mcps_data_conf.c All required MAC stub functions are already implemented in the application files.

11.4.2 TAL callbacks

The TAL requires the following callback functions:

- tal_ed_end_cb
- tal_rx_frame_cb
- tal_tx_frame_done_cb

These callback functions are declared in file *tal/inc/tal.h*. Each TAL based application (HIGHEST_STACK_LAYER = TAL) needs to implement these tal_..._cb() callback functions. The MAC layer (residing on top of the TAL) has also implemented these callback functions.

In case these callbacks are not used within the TAL based application, the existing stub functions can easily be used. All required TAL stub functions are already implemented in the files *tal_*_cb.c* in directory tal/src.

11.4.3 Example for MAC callbacks

All mac callbacks are included directly inside the main.c file and necessary callbacks are used and others are left blank.These callbacks can also be included by adding the file in the name of the call back eg.usr_mlme_reset_conf.c where the callback is added but left unsed.

11.5 Bootloader

All mac and tal application images can be uploaded without the use of a JTAG .The BootloaderPcTool available in avr2025_mac/addons/bootloader can be used for this purpose when using two-processor approach as well.The Bootloder hex files for atxmega256A3U_zigbit_ext and atmega256rfr2_Zigbit available in the same folder .Refer Figure 10-16 for the general approach for flashing a application image to the NCP and Figure 10-17 for Bootloade PC Tool. Following are the steps to flash a MAC or TAL application image to the device(NCP).

Flash the bootloader hex file to the device(NCP)

Flash the serial bridge hex file to the Host .

Connect the NCP to the Host and connect the host to the PC.

Open the BootloaderPcTool and select the desired COM port and baud rate as 9600.

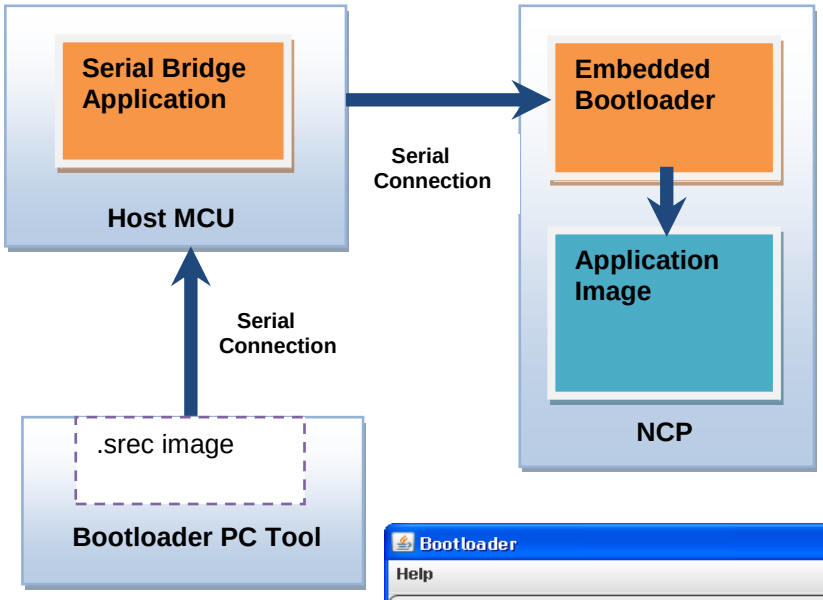
Select the srec file generated from either AtmelStudio or IAR.

Press the reset button on the NCP and release it once upload is pressed in the tool.The process is done once the tool shows ‘Upload Complete’

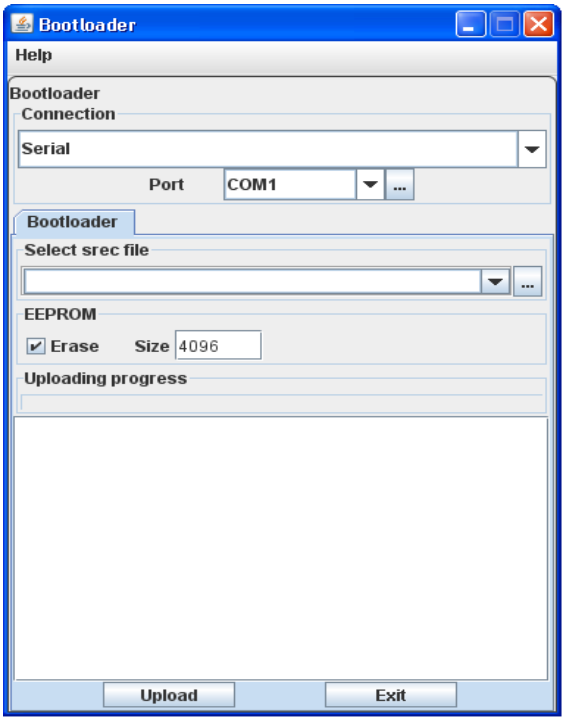
For more details refer readme.txt in the same folder.

Note: This bootloader is supported only for 2P boards.

Figure 11-8. General approach for using bootloader to program a NCP in 2p approach



Bootloader PC Tool Window



12 Supported platforms

This chapter describes which hardware platforms are currently supported with the Atmel AVR2025 MAC software package. A platform usually comprises of three major components:

- An MCU,
- A transceiver chip (this may be integrated into the MCU for Single Chips)
- A specific Board or even several boards that contain the MCU or the transceiver chip

12.1 Supported MCU families

Currently the following generic MCU families are supported:

- AVR32: Atmel AVR32 platforms
- SAM4: Atmel SAM4L platforms
- MEGA_RF: Atmel AVR 8-bit ATmega RF Single Chip platforms
- XMEGA: Atmel AVR 8-bit ATxmega platforms

The dedicated code for each platform family can be found in the corresponding subdirectories.

12.2 Supported transceivers

For a complete list of all supported transceivers please refer to the AVR2025 release notes.

12.3 Supported boards

Few of the currently supported boards and combinations are given below.

- RF Xplained Pro – AtmegaRFR2
- ZigBit-ATmega256RFR2 (extender board)
- ZigBit-ATRF233 Xmega (extender board)
- ZigBit-ATRF212B Xmega (extender board)
- USB stick with ZigBit Xmega-AT86RF233
- USB stick with ZigBit Xmega-AT86RF212B
- ARM-4L XPLD PRO with ZigBit ATmegaRFR2
- ARM-4L XPLD PRO with ZigBit ATRF233 Xmega
- ARM-4L XPLD PRO with ZigBit ATRF212B Xmega
- RZ600 kits
- Xmega-a3bu Xplained with RZ600 radio modules

The following sections describe the currently supported boards and platforms in more detail. All described hardware boards are available from Atmel (see 1) or third party vendors (for example, see 2).

12.3.1 RF Xplained Pro – AtmegaRFR2

The RF Xplained Pro-AtmegaRFR2 is a radio module, with an Atmel Single Chip - SOC (for example, Atmel ATmega256RFR2). These boards come with an EDBG (Embedded Debugging) interface and supports only Atmel Studio 6.1 and next versions.

The following figure depicts an RF Xplained Pro board with ATmega256RFR2 chip.

Figure 12-1. RF_Xplained_Pro with ATmega256RFR2

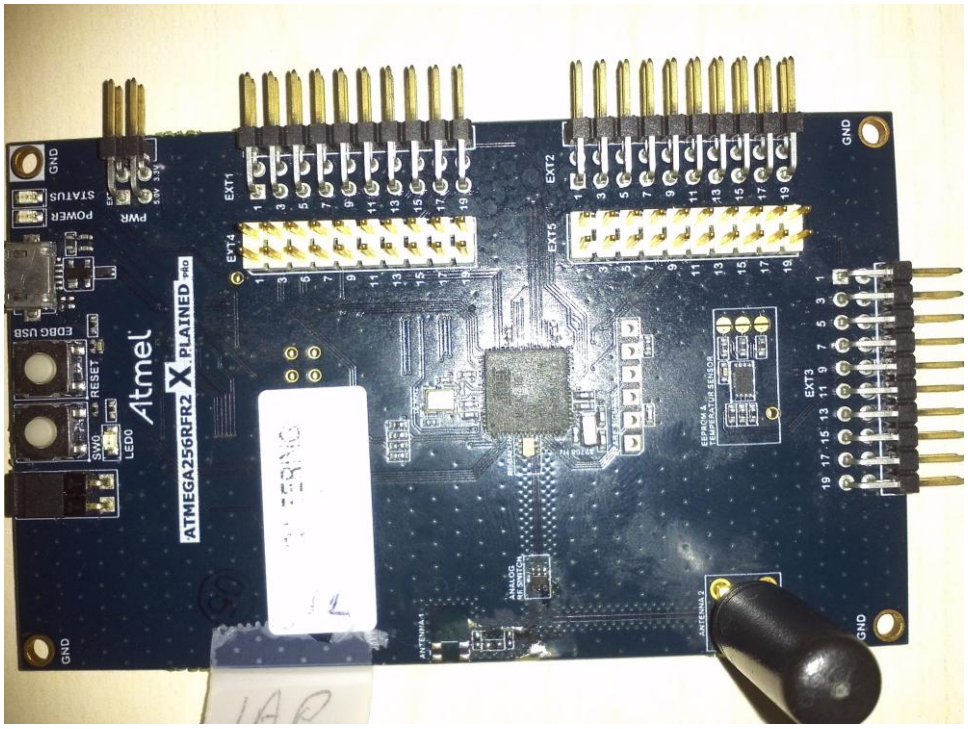
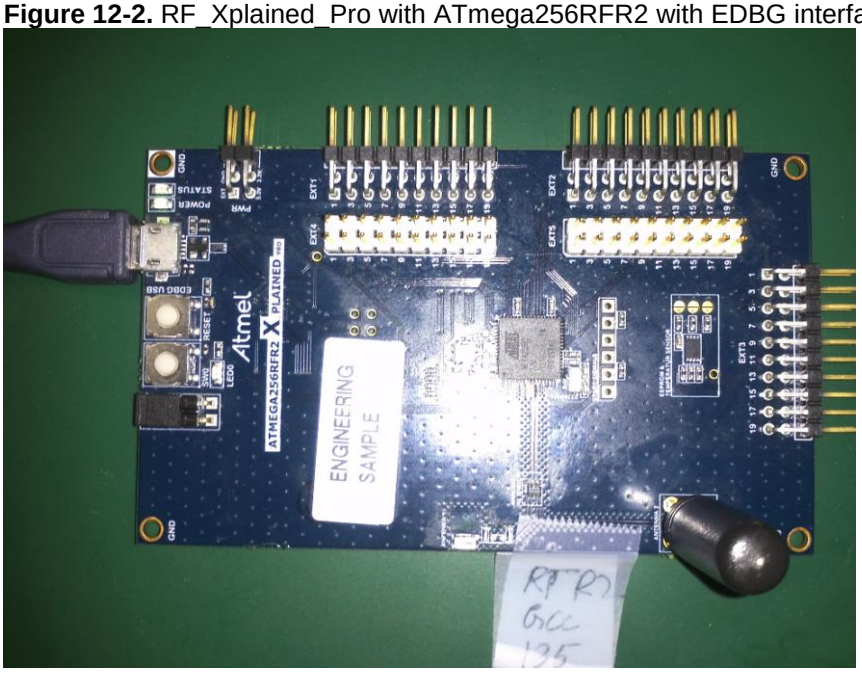


Figure 12-2. RF_Xplained_Pro with ATmega256RFR2 with EDBG interface



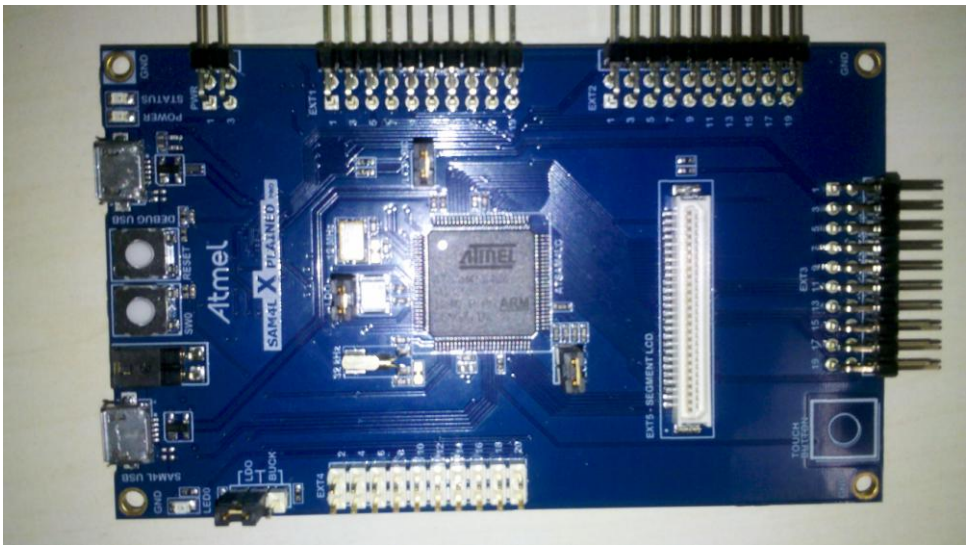
12.3.2 SAM4L Xplained Pro boards

The SAM4L Xplained Pro is a plain board, with an Atmel MCU of SAM4L series. These boards come with an EDBG (Embedded Debugging) interface and supports only Atmel Studio 6.1 and next versions.

These SAM4L Xplained boards are used in combination with Xmega-ATRFxx Zigbit extender modules and AtmegaRFR2 Zigbit extender modules.

The Figure 12-3 shows a plain SAM4L Xplained Pro board.

Figure 12-3. SAM4L Xplained Pro board



12.3.3 Zigbit Extender boards

The Zigbit Extender Board is a radio module containing an Atmel MCU and a transceiver or Single Chip solution (SOC like AtmegaRFR2). These boards cannot be used stand alone and so requires an additional baseboard for the application to run, such as SAM4L-Xplained Pro board.

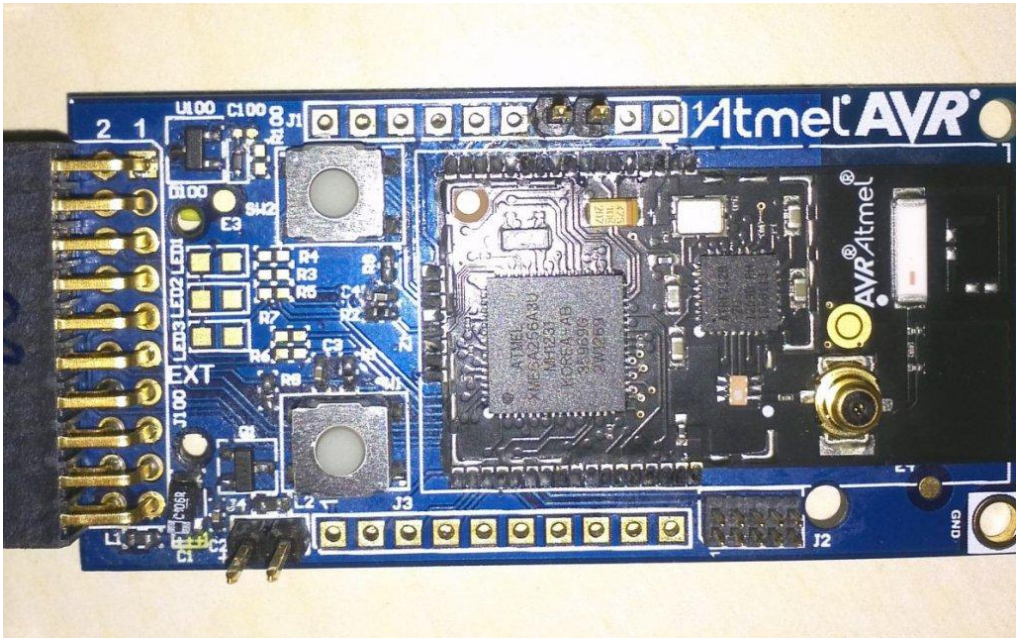
The following sections will detail more about different types of Zigbit extenders and boards that can be supported.

12.3.3.1 Xmega- RF212B Zigbit extender module

- MCU – Xmega256a3bu
- Transceiver- AT86RF212B

The Figure 12-4 shows an Xmega-RF212B Zigbit extender (stand alone).

Figure 12-4. Xmega-RF212B Zigbit extender module

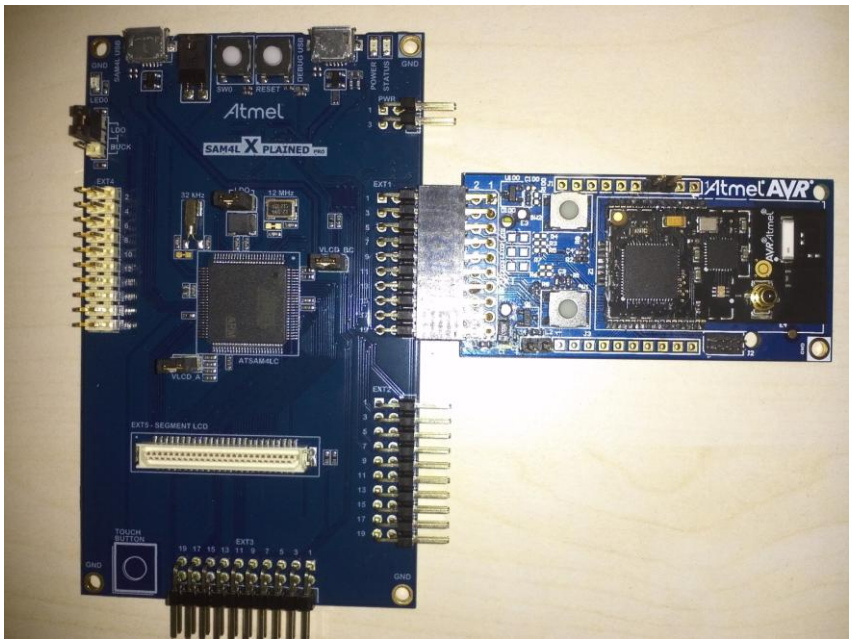


12.3.3.2 SAM4L Xplained Pro with Xmega-RF Zigbit modules

As we can't use the Zigbit extender modules alone, we need a base board to work. So we can use a SAM4L Xplained Pro board with these Zigbit extender modules to run an application.

The Figure 12-5 shows a SAM4L Xplained Pro board connected to Xmega-RF212B Zigbit extender module.

Figure 12-5. SAM4L Xplained Pro with Xmega-212b Zigbit extender

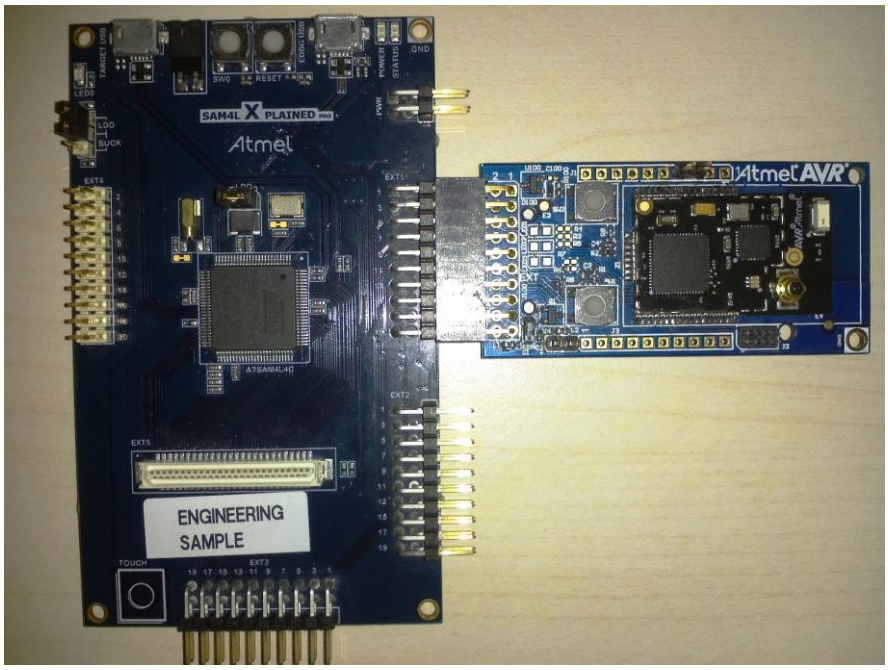


Similary we can have Xmega-RF233 Zigbit extender module with a SAM4L Xplained Pro board. The following figures show a stand alone Zigbit Xmega-RF233 module and a SAM4L Xplained Pro board connected to Zigbit Xmega-RF233 module as shown in Figure 12-7.

Figure 12-6. Xmega-RF233 Zigbit extender module

- MCU – Xmega256a3bu
- Transceiver- AT86RF233

Figure 12-7. SAM4L Xplained Pro with Zigbit Xmega-RF233 module



12.3.3.3 SAM4L Xplained Pro with AtmegaRFR2 Zigbit modules

Similar to the SAM4L Xplained Pro combination with Xmega-RF212/233 zigbit modules, we can connect an Atmega256RFR2 Zigbit module to an SAM4L Xplained Pro board.The following figures shows a stand alone AtmegaRFR2 Zigbit module and a SAM4L Xplained pro board connected to a AtmegaRFR2-Zigbit module as shown in Figure 12-9

Figure 12-8. Atmega256RFR2 Zigbit extender module

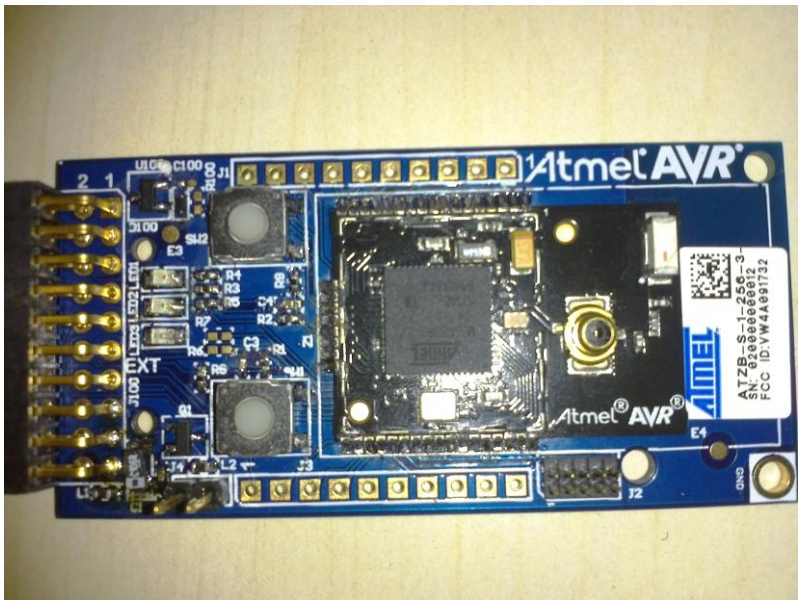
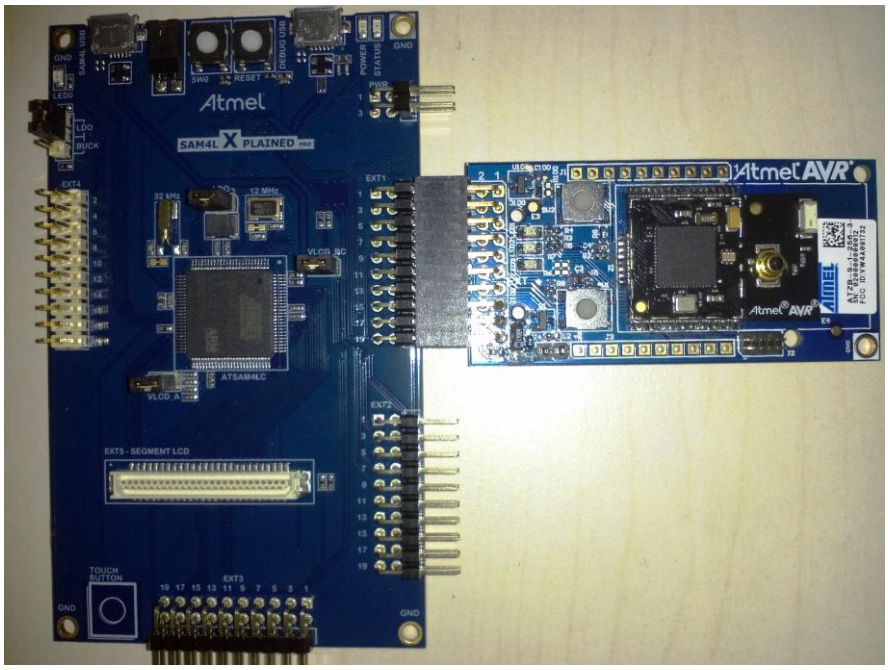


Figure 12-9. SAM4L Xplained Pro with Atmega256RFR2 Zigbit module



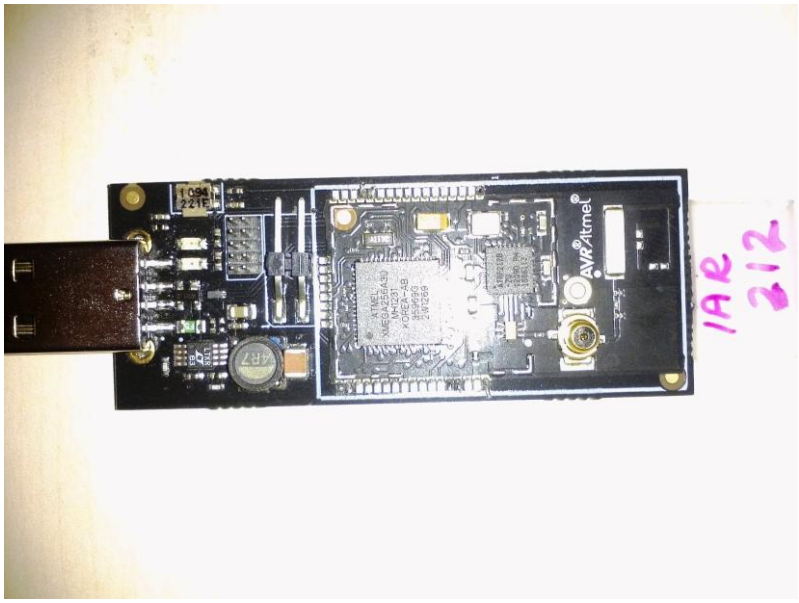
12.3.4 USB Zigbit Modules

These radio modules come with an Xmega MCU and an AT86RF Transceiver on a single USB stick. In the below sections more details are provided.

12.3.4.1 USB stick with Zigbit Xmega-AT86RF212B

- MCU – Atxmega256a3u
- Transceiver – AT86RF212B

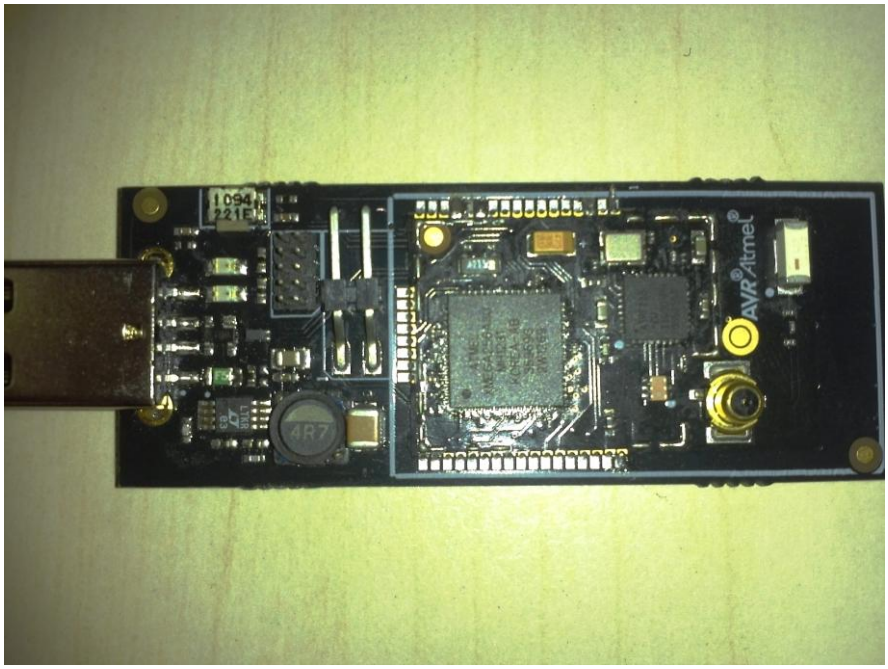
Figure 12-10. USB stick with Zigbit Xmega-AT86RF212B



12.3.4.2 USB stick with Zigbit Xmega-AT86RF233

- MCU – Atxmega256a3u
- Transceiver – AT86RF233

Figure 12-11. USB stick with Zigbit Xmega-AT86RF233



These USB sticks can be directly connected to a USB port and no need of any additional power supply.

12.3.5 RZ600 kits

This RZ600 family contains two radio modules of two 2.4GHz device AT86RF230, AT86RF231 and a sub gigahertz radio AT86RF212.The radio modules are used in combination with UC3A and XMEGA_A3BU based processor boards.

For complete details about the usage, schematics and connection diagrams etc, please refer to the AVR2064:RZ600 HW Manual and related documents (refer [6])

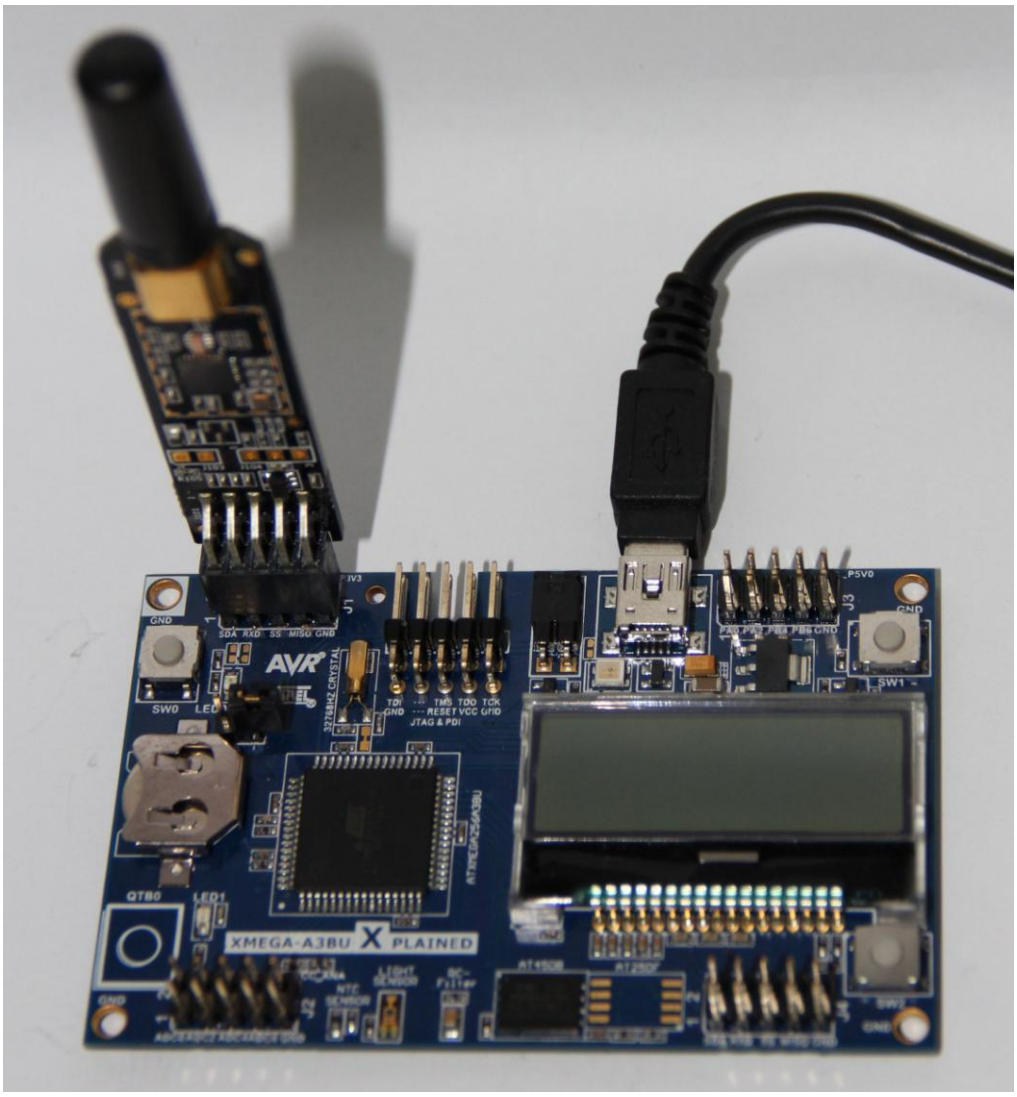
12.3.6 Xmega-a3bu Xplained with RZ600 radio modules

The Atmel AVR XMEGA-A3BU Xplained evaluation kit is a hardware platform based one ATxmega256A3BU microcontroller.

In this AVR2025_MAC package there are few applications demonstrated with Xmegaa3bu Xplained board and RZ600 radio modules combination.

The following Figure 12-12 depicts an example of Xmega-a3bu Xplained board connected to a RZ600-RF212 module.

Figure 12-12. Xmega-a3bu Xplained with RZ600-RF212



Similary we can have an Xmega-a3bu Xplained board connected with RZ600-RF230b and RZ600-RF231 radio modules.

For more details about the usage, schematics and programming features of Xmega-a3bu Xplained boards, please refer to the AVR1923: Xmega-a3bu Xplained HW User guide and other related documents (refer [5]).

13 Platform porting

For details and description about platform porting please refer to Atmel Software Framework documentation [9] and Atmel Studio-help [10]

14 Protocol implementation conformance statement (PICS)

This chapter lists the conformance of the Atmel AVR2025 MAC implementation with the requirements and optional features as defined by the standard specified in 4 in Section D.7.

14.1 Major roles for devices compliant with IEEE Std 802.15.4-2006

Table 14-1. Functional device types.

Item number	Item description	Status	Support		
			N/A	Yes	No
FD1	Is this a full function device (FFD)	O.1		X	
FD2	Is this a reduced function device (RFD)	O.1		X	
FD3	Support of 64 bit IEEE address	M		X	
FD4	Assignment of short network address (16 bit)	FD1:M		X (FFD only)	
FD5	Support of short network address (16 bit)	M		X	
O1: At least one of these features shall be supported.					

14.2 Major capabilities for the PHY

Table 14-2. PHY functions.

Item number	Item description	Status	Support		
			N/A	Yes	No
PLF1	Transmission of packets	M		X	
PLF2	Reception of packets	M		X	
PLF3	Activation of radio transceiver	M		X	
PLF4	Deactivation of radio transceiver	M		X	
PLF5	Energy detection (ED)	FD1:MO		X (FFD only)	
PLF6	Link quality indication (LQI)	M		X	

Item number	Item description	Status	Support		
			N/A	Yes	No
PLF7	Channel selection	M		X	
PLF8	Clear channel assessment (CCA)	M		X	
PLF8.1	Mode 1	O.2		X	
PLF8.2	Mode 2	O.2		X	
PLF8.3	Mode 3	O.2		X	
O2: At least one of these features shall be supported.					

14.3 Major capabilities for the MAC sub-layer

14.3.1 MAC sub-layer functions

Table 14-3. MAC sub-layer functions.

Item number	Item description	Status	Support		
			N/A	Yes	No
MLF1	Transmission of data	M		X	
MLF1.1	Purge data	FD1: M FD2: O		X (FFD only)	
MLF2	Reception of data	M		X	
MLF2.1	Promiscuous mode	FD1: M FD2: O		X (FFD only)	
MLF2.2	Control of PHY receiver	O		X	
MLF2.3	Timestamp of incoming data	O		X	
MLF3	Beacon management	M		X	
MLF3.1	Transmit beacons	FD1: M FD2: O		X (FFD only)	
MLF3.2	Receive beacons	M		X	
MLF4	Channel access mechanism	M		X	
MLF5	Guaranteed time slot (GTS) management	O			X
MLF5.1	GTS management (allocation)	O			X
MLF5.2	GTS management (request)	O			X
MLF6	Frame validation	M		X	
MLF7	Acknowledged frame delivery	M		X	
MLF8	Association and disassociation	M		X	
MLF9	Security	M		X (data frames)	
MLF9.1	Unsecured mode	M		X	
MLF9.2	Secured mode	O			X
MLF9.2.1	Data encryption	O.4			X
MLF9.2.2	Frame integrity	O.4			X

Item number	Item description	Status	Support		
			N/A	Yes	No
MLF10.1	ED	FD1: M FD2: O		X (FFD only)	
MLF10.2	Active scanning	FD1: M FD2: O		X	
MLF10.3	Passive scanning	M		X	
MLF10.4	Orphan scanning	M		X	
MLF11	Control/define/determine/declare superframe structure	FD1: O		X (FFD only)	
MLF12	Follow/use superframe structure	O		X	
MLF13	Store one transaction	FD1: M		X (FFD only)	
O4: At least one of these features shall be supported.					

14.3.2 MAC frames

Table 14-4. MAC frames.

Item number	Item description	Transmitter		Receiver	
		Status	Support N/A Yes/No	Status	Support N/A Yes/No
MF1	Beacon	FD1: M	Yes (FFD only)	M	Yes
MF2	Data	M	Yes	M	Yes
MF3	Acknowledgment	M	Yes	M	Yes
MF4	Command	M	Yes	M	Yes
MF4.1	Association request	M	Yes	FD1: M	Yes (FFD only)
MF4.2	Association response	FD1: M	Yes (FFD only)	M	Yes
MF4.3	Disassociation notification	M	Yes	M	Yes
MF4.4	Data request	M	Yes	FD1: M	Yes
MF4.5	PAN identifier conflict notification	M	Yes	FD1: M	Yes
MF4.6	Orphaned device notification	M	Yes	FD1: M	Yes (FFD only)
MF4.7	Beacon request	FD1: M	Yes	FD1: M	Yes
MF4.8	Coordinator realignment	FD1: M	Yes (FFD only)	M	Yes
MF4.9	GTS request	MLF5: O	No	MLF5: O	No

15 Abbreviations

API	Application Programming Interface
ASF	Atmel Software Framework
BMM	Buffer Management Module
GPIO	General Purpose Input/Output
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MAC	Medium Access Control
MCL	MAC Core Layer
MCPS	MAC Common Part Sub-layer
MCU	Microcontroller Unit
MHR	MAC Header
MIC	Message Integrity Code
MLME	MAC Sub-layer Management Entity
MPDU	MAC Protocol Data Unit
MSDU	MAC Service Data Unit
NHLE	Next Higher Layer Entity
NWK	Network Layer
PAL	Platform Abstraction Layer
PAN	Personal Area Network
PIB	PAN Information Base
QMM	Queue Management
RCB	Radio Controller Board
REB	Radio Extender board
SAL	Security Abstraction Layer
SIO	Serial I/O
SPI	Serial Peripheral Interface
STB	Security Toolbox
TAL	Transceiver Abstraction Layer
TFA	Transceiver Feature Access
TPS	Transceiver Programming Suite
TRX	Transceiver
WPAN	Wireless Personal Area Network

16 References

1. Atmel Wireless MCU Software Website
   http://www.atmel.com/products/microcontrollers/wireless/default.aspx?tab=tools
2. Dresden Elektronik Wireless data transmission 802.15.4 Website
   http://www.dresden-elektronik.de/shop/cat4.html?language=en
3. Atmel Wireless Support avr@atmel.com
4. IEEE Std 802.15.4™-2006 Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)
5. Atmel XMEGA-A3BU Xplained Kits
   http://www.atmel.com/tools/xmega-a3buxplained.aspx
   http://www.atmel.com/tools/xmega-a3buxplained.aspx?tab=documents
6. RZ600 Evaluation Kit Website
   http://www.atmel.com/tools/rz600.aspx?tab=overview
   http://www.atmel.com/tools/rz600.aspx?tab=documents
7. ATAVRXPLAIN Evaluation and Demonstration Kit Website
   http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4506&source=xplain_page
8. ATZB ZigBit Module Website
   http://www.atmel.com/products/zigbee/zigbit_modules.asp?family_id=676
9. Atmel Software Framework
   http://www.atmel.com/tools/avrsoftwareframework.aspx
   http://asf.atmel.com/docs/latest/
10. Atmel Studio - http://www.atmel.in/microsite/atmel_studio6/

17 User guide revision history

Please note that the referring page numbers in this section are referring to this document. The referring revisions in this section are referring to the document revision.

17.1 Rev. 2025M-MCU Wireless-03/13

Released with AVR2025 MAC Version 3.0.0

1. This release version is a re-architecture of MAC and PAL layers.
2. The complete stack is ported into Atmel Software Framework [9]

17.2 Rev. 2025M-MCU Wireless-06/12

Released with AVR2025 MAC Version 2.8.0

3. AT86RF233 TAL support and demonstrated with TAL and MAC Examples
4. ATMEGARFR2 TAL support and demonstrated with TAL and MAC Examples
5. Platform support added for ATmega1281_REB_4_1_STK600 and ATZB_24_MN2
6. MAC examples and TAL examples added for ATZB_24_MN2
7. Pal_nvm_multi_write functionality added for AVR32

8. Watchdog support added for SAM3S
9. MAC Application Power Management Support – Star Push Button, a MAC Example demonstrates the feature.
10. Accelerometer_Display_App added for KEY_RC platform as a part of TAL Example to demonstrate the LCD and Motion Sensor on it

17.3 Rev. 2025M-MCU Wireless-10/11

Released with AVR2025 MAC Version 2.7.1

11. Performance_Test_EVK and Performance Test Application merged as a single application
12. Serial_AT_Interface added as a MAC application
13. Atmel AT86RF232 Radio support added for Atmel ATXmega256A3 and Atmel AT32UC3B1128 as a part of MAC & TAL applications

17.4 Rev. 2025L-MCU Wireless-07/11

Released with AVR2025 MAC Version 2.7.0

14. New applications added along with this release namely,
    i. Beacon_Application
    ii. Nobeacon_Applications
15. Added Security Example Application to demonstrate MAC security
16. Atmel AVRStudio 5 Support for all AVR 8bit & AVR 32bit
17. Filtered and removed some of the platforms & MCU families
18. Added Performance_Test_EVK application description

17.5 Rev. 2025K-MCU Wireless-08/11

Released with AVR2025 MAC Version 2.6.1

19. mac_security.c file contents are completely removed for the web release
20. UART software driver issue working on SAM3S-EK Platform got fixed

17.6 Rev. 2025J-MCU Wireless-03/11

Released with AVR2025 MAC Version 2.6.0

21. Platform description for AVR32, SAM3S, CBB boards added
22. Tool Chain section is updated for AVR32, SAM.
23. Build Switches WATCHDOG,SLEEPING_TIMER for AVR32, XMEGA Platforms added

17.7 Rev. 2025I-MCU Wireless-10/10

Released with AVR2025 MAC Version 2.5.3

24. Build switch ENABLE_RC_OSC_CALIBRATION for Mega-RF platforms added

17.8 Rev. 2025H-MCU Wireless-08/10

Released with AVR2025 MAC Version 2.5.2

25. MAC Example Star_High_Rate added
26. High Data Rate support added
27. Platform description for RZ600 on top of Xplain board added
28. Platform description for ATZB ZigBit Modules on top of MeshBean2 board added

17.9 Rev. 2025G-MCU Wireless-08/10

Released with AVR2025 MAC Version 2.5.1

29. Description of new design of TAL and MCL added
30. Description of Tiny-TAL added
31. Support for ZigBit 212 added
32. New compiler switches added
33. Support for ATxmega256A3 added
34. Migration Guide from 2.4.x to 2.5.x added
35. High-density Network Configuration added
36. Frame transmission and reception procedure added
37. Buffer handling description added

17.10 Rev. 2025F-MCU Wireless-02/10

Released with AVR2025 MAC Version 2.4.2

38. Support of program code larger than 128KByte added
39. PAN-Id conflict detection handling added
40. Support for AT91SAM7XC added
41. Description of application security added
42. Description of security build switches updated

17.11 Rev. 2025E-MCU Wireless-01/10

Released with AVR2025 MAC Version 2.4.0

43. Support for AT86RF231 and AT86RF212 with AT91SAM7X256 added
44. Support for ATmega128RFA1-EK1 added
45. Build switch DISABLE_TSTAMP_IRQ added
46. Support for ATmega1284P removed
47. Support for ATxmega256A3
48. MAC Porting Guide using ATxmega256A3 as example added
49. MAC Examples App 3 (Beacon Payload) and App 4 (Beacon Broadcast Data) added
50. Build switch SYSTEM_CLOCK_MHZ renamed to F_CPU
51. Updated handling of MAC PIB attribute macRxOnWhenIdle and MAC power management
52. New build switch BAUD_RATE added
53. Support for AT86RF230A and related hardware platforms discontinued
54. Handling of callback stub functions added Handling of callback stub functions added
55. PICS Table added

17.12 Rev. 2025B-MCU Wireless-09/09

Released with AVR2025 MAC Version 2.3.1

56. Name of Radio Controller Board (RCB) changed: transceiver number suffix is replaced by board suffix
57. Support for ATmega128RFA1 added
58. Handling of Promiscuous Mode updated.
59. Migration Guide for previous MAC versions removed
60. Filter tuning section added
61. Description of Performance test application extended
62. Support for ATxmega MCU based AES within SAL
63. Handling of MAC components updated
64. Initial Support for AT91SAM7X256 added

65. DISABLE_IEEE_ADDR_CHECK added
66. Chapter Supported Platforms added
67. Chapter Topics on Platforms Porting added

17.13 Rev. 2025-AVR-04/09

Released with AVR2025 MAC Version 2.2.0

68. Initial Version

Table of contents

18 Table of figures

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium 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
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581

8412A-AVR-02/13