

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
- Support of all Atmel IEEE 802.15.4 transceivers and single chips, i.e. ATmega128RFA1, AT86RF212, AT86RF231, and AT86RF230
- Example application 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 the MAC software package AVR2025 release 2.4.2.

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 802.15.4 based transceivers and single chips
- Allows easy and quick platform porting
- Provides project files for two supported IDEs (IAR Embedded Workbench, AVR Studio / WinAVR)
- Supports star networks and peer-to-peer communication
- Supports nonbeacon and beacon-enabled networks

The MAC software package is a reference implementation demonstration the use of Atmel's 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.



Application Note

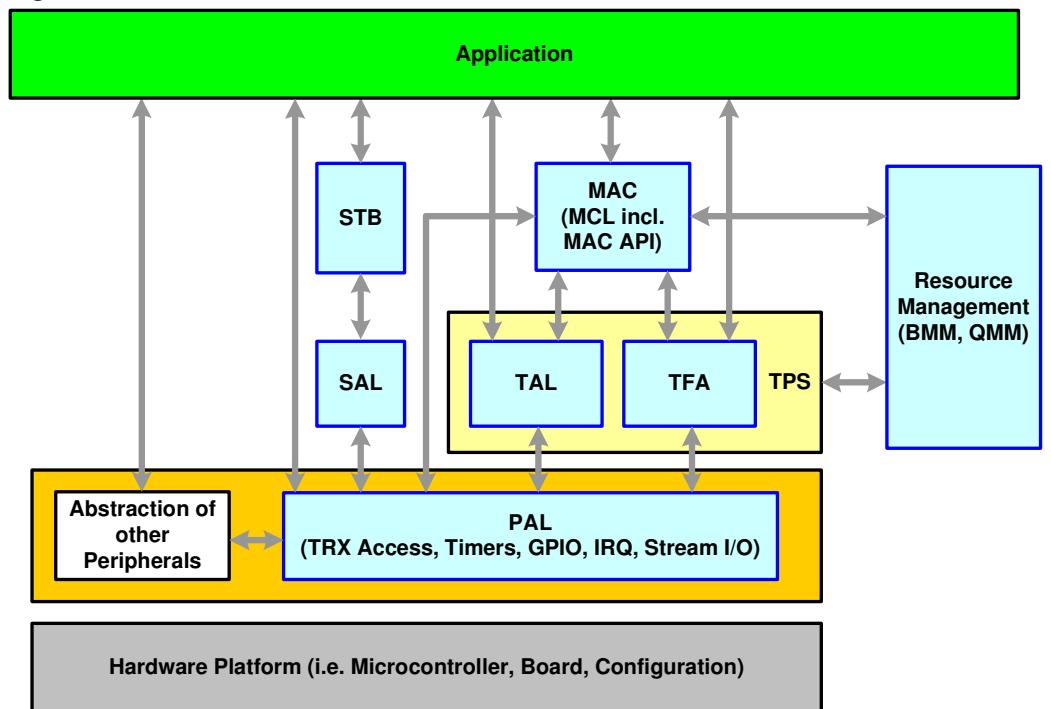
Preliminary

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)
- Transceiver Abstraction Layer (TAL) and Transceiver Feature Access (TFA)
- MAC including MAC Core Layer and MAC API
- Security Abstraction Layer (SAL) and Security Toolbox (STB)
- Resource Management including Buffer and Queue Management (BMM and QMM)

Figure 2-1. MAC Architecture



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

2.1.1 Platform Abstraction Layer (PAL)

The Platform Abstraction Layer (PAL) contains all platform (i.e. MCU and board) specific functionality (required by the MAC software packages) and provides interfaces to the upper modules. Therefore, all upper modules are independent from the underlying platform. Since some components of the PAL maybe dependent on the actual platform/board, certain functionality within the PAL has to be implemented for each setup, like LEDs and buttons.

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

The PAL provides interfaces to the following components:

- Transceiver access functionality, i.e. via SPI or direct memory access
- GPIO control (access from microcontroller to the GPIO pins connected to the transceiver)
- Low-level interrupt handling
- Timers (high-priority and software timers)
- Serial stream I/O support (via USB or UART)
- Access to persistent storage (e.g. EEPROM)
- LED control or button access support

These components are implemented as software blocks and are ported based on the target microcontroller. The transceiver access module provides interface to the registers and frame buffer of the transceiver. The timer module implements software timer functionality used by the MAC, TAL, and application layer. The serial stream I/O module provides communication services for transmission and reception of serial data, e.g. UART/USB communication. The GPIO module controls the general purpose I/O pins of the microcontroller. The interrupt module handles the transceiver interrupt(s). Hardware timer interrupts and other interrupts are handled internally by the PAL.

The function prototypes for all PAL API functions 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, extracts the frame content 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 nonbeacon-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.

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 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 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 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

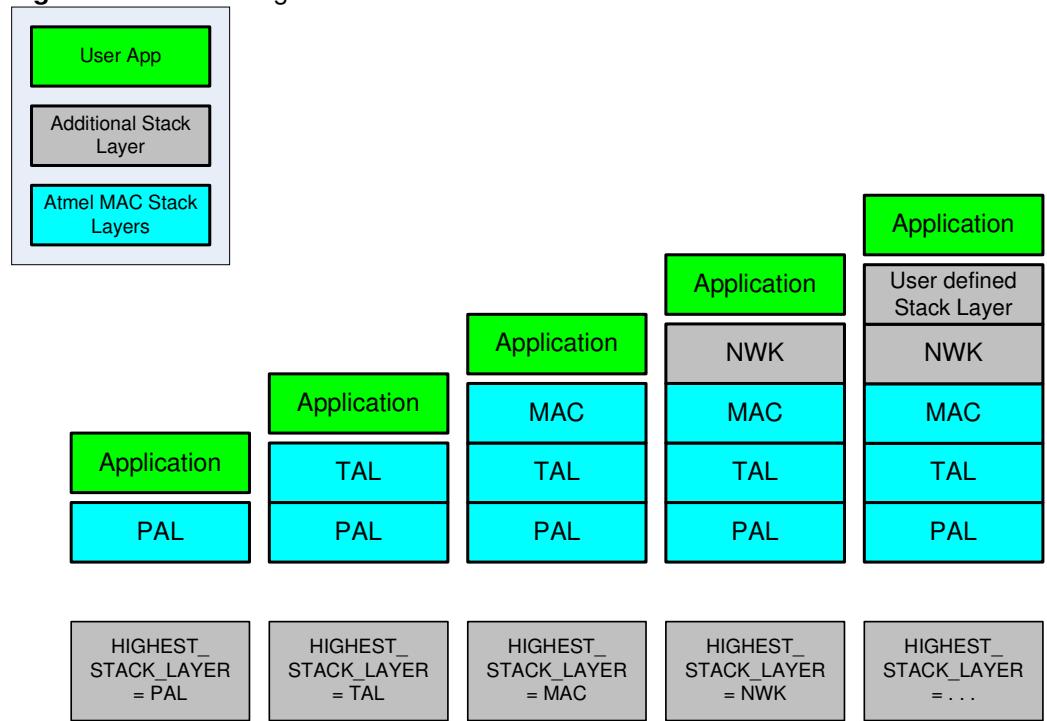
An application can use any layer as desired depending on the required functionality. An application that is based on a standard IEEE 802.15.4 MAC uses the MAC API based on the stack built by PAL, TAL, and MCL. Another application (e.g. a simple data pump) may want to use only the basic channel access 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. A very simple application may even only use the PAL API based on the PAL layer. 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 (i.e. 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 6.1.1.1`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:

1. Queue used by MAC Core Layer
 - a. MAC queue
 - b. Internal event queue
 - c. Indirect data queue
 - d. Broadcast queue
2. Queue used by TAL
 - a. Incoming frame queue
3. Additional queues and buffers can be used by higher layers, like application.

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, e.g. 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).

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.

2.2.4 Transceiver Feature Access

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).

2.3 Application

The application uses the services provided by the MAC API. Besides that, the application can use the resource hooks, like timer functionality of the PAL, buffer and queue management.

It is in the responsibility of the application to implement the callback functions as confirmation and indication primitives for respective MAC data/management services.





3 Understanding the Software Package

The following chapter describes the content of the MAC software package and explains some general guidelines how the various software layers are structured.

3.1 MAC Package Directory Structure

Once the MAC package has been extracted into the proper place the directory structure looks as follows:

```
MAC_v_2_3_1
├── Applications
│   ├── Helper_Files
│   │   └── SIO_Support
│   ├── MAC_Examples
│   │   ├── App_1_Nobeacon
│   │   │   ├── Coordinator
│   │   │   │   ├── AT86RF212_ATMEGA1281_RCB_5_3_PLAIN
│   │   │   │   ├── AT86RF212_ATXMEGA128A1_REB_5_0_STK600
│   │   │   │   └── AT86RF230B_AT90USB1287_USBSTICK_C
│   │   │   ├── ...
│   │   │   └── Inc
│   │   └── Src
│   └── Device
│       ├── AT86RF212_ATMEGA1281_RCB_5_3_PLAIN
│       ├── ...
│       └── Src
│   └── App_2_Nobeacon_Indirect_Traffic
│       ├── Coordinator
│       │   ├── ...
│       │   └── ATMEGA128RFA1_RCB_6_3_SENS_TERM_BOARD
│       ├── Inc
│       └── Src
│   └── Device
│       ├── ...
│       └── Src
│   └── Inc
│   └── Src
└── Basic_Sensor_Network
    ├── ...
    ├── Inc
    └── Src
    └── Promiscuous_Mode_Demo
        ├── ...
        ├── Inc
        └── Src
```

```
└── Star_Nobeacon
    ├── ...
    ├── Inc
    └── Src

└── PAL_Examples
    └── Simple_Remote_LED_Control
        ├── ATMEGA128RFA1_RCB_6_3_PLAIN
        ├── Inc
        └── Src

└── STB_Examples
    └── Secure_Remote_Control
        ├── ...
        ├── ATMEGA128RFA1_RCB_6_3_PLAIN
        ├── ATMEGA128RFA1_RCB_6_3_SENS_TERM_BOARD
        ├── Inc
        └── Src

    └── Secure_Sensor
        ├── Data_Sink
        ├── ...
        ├── ATMEGA128RFA1_RCB_6_3_SENS_TERM_BOARD
        ├── Inc
        └── Src

        ├── Inc
        ├── Sensor
        ├── ...
        ├── Inc
        └── Src

    └── Src

└── TAL_Examples
    └── Performance_Test
        ├── ...
        ├── Inc
        └── Src

    └── Wireless_UART
        ├── ...
        ├── Inc
        └── Src

└── Build
    └── Build_MAC
        ├── GCC
        └── AVR

        └── IAR
            └── AVR

└── Doc
    └── User_Guide
        └── Reference_Manual
            └── MAC
```

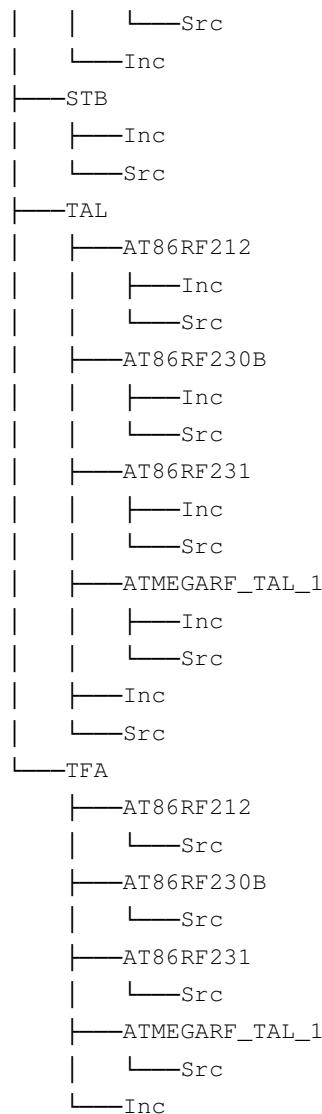




```
|-Include  
|-MAC  
| |-Inc  
| |-Src  
|-PAL  
| |-ARM7  
| | |-AT91SAMX256  
| | | |-Boards  
| | | | |-REB_5_0_REX_ARM_REV_3  
| | | | |-REB_2_3_REX_ARM_REV_2  
| | | | |-REB_4_0_2_REX_ARM_REV_3  
| | | |-Inc  
| | | |-Src  
| | | |-Startup  
| | | |-Generic  
| | | | |-Inc  
| | | | |-Src  
| |-AVR  
| | |-AT90USB1287  
| | | |-Boards  
| | | | |-USBSTICK_C  
| | | |-Inc  
| | | |-Src  
| | |-ATMEGA1281  
| | | |-Boards  
| | | | |-RCB_3_2_BREAKOUT_BOARD  
| | | | |-RCB_3_2_PLAIN  
| | | | |-RCB_3_2_SENS_TERM_BOARD  
| | | | |-RCB_4_0_BREAKOUT_BOARD  
| | | | |-RCB_4_0_PLAIN  
| | | | |-RCB_4_0_SENS_TERM_BOARD  
| | | | |-RCB_4_1_BREAKOUT_BOARD  
| | | | |-RCB_4_1_PLAIN  
| | | | |-RCB_4_1_SENS_TERM_BOARD  
| | | | |-RCB_5_3_BREAKOUT_BOARD  
| | | | |-RCB_5_3_PLAIN  
| | | | |-RCB_5_3_SENS_TERM_BOARD  
| | | | |-REB_2_3_STK500_STK501  
| | | | |-REB_4_0_STK500_STK501  
| | | | |-REB_4_1_STK500_STK501  
| | | | |-REB_5_0_STK500_STK501  
| | | |-Inc  
| | | |-Src  
| | |-ATMEGA2561  
| | | |-Boards  
| | | | |-REB_2_3_STK500_STK501  
| | | |-Inc
```

```
    |   |   |   └── Src
    |   |   └── ATMEGA644P
    |   |   └── Boards
    |   |       └── REB_2_3_STK500
    |   |   └── Inc
    |   |   └── Src
    |   └── Generic
        └── Inc
        └── Src
    └── Inc
    └── MEGA_RF
        └── ATMEGA128RFA1
            └── Boards
                └── RCB_6_3_BREAKOUT_BOARD
                └── RCB_6_3_PLAIN
                └── RCB_6_3_SENS_TERM_BOARD
            └── Inc
            └── Src
        └── Generic
            └── Inc
            └── Src
    └── XMEGA
        └── ATXMEGA128A1
            └── Boards
                └── REB_2_3_STK600
                └── REB_4_0_STK600
                └── REB_4_1_STK600
                └── REB_5_0_STK600
            └── Inc
            └── Src
        └── ATXMEGA256A3
            └── Generic
                └── Inc
                └── Src
        └── Resources
            └── Buffer_Management
                └── Inc
                └── Src
            └── Queue_Management
                └── Inc
                └── Src
        └── SAL
            └── AT86RF2xx
                └── Src
            └── ATMEGARF_SAL
                └── Src
            └── ATXMEGA_SAL
```





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

- Applications:
 - The MAC package comes with a variety of applications which comprise MAC applications (using the MAC API on top of the MAC Core Layer), TAL applications (using the TAL API), and STB Applications (Secure Remote Control using the TAL API, Secure Sensor using the MAC API on top of the MAC Core Layer).
 - Makefiles and AVR Studio and IAR Embedded Workbench project files are available for every supported microcontroller / transceiver / board configuration and can be used as quick start.
 - Hex files to be downloaded onto available hardware are provided
- Build: This directory contain Windows batch files, that can be easily used to rebuilt any desired (MAC, TAL, or STB) application or all applications at once.
- Doc:

- This directory contains the MAC reference manual in html format, which can be started by double clicking file *MAC_readme.html* in the root directory of the MAC package.
- Also this MAC User Guide is located here.
- Include: This directory contains header files that are of general interest both for applications and for all layers of the stack, such as IEEE constants, data types, return values, etc.
- MAC: This directory contains the MAC Core Layer (MCL) and the MAC API.
- PAL: This directory contains the Platform Abstraction Layer with subdirectories for each microcontroller family. It provides all required source and header files for each microcontroller and the supported board configurations.
- 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 subdirectories 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 (e.g. TAL/AT86RF231/Inc) or each microcontroller family and each single microcontroller have their own Inc directories (e.g. PAL/XMEGA/Generic/Inc or PAL/AVR/ATMEGA1281/Inc).

Generally the following header file naming conventions are followed:

1. 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

2. 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, PAL/AVR/Generic/Inc/pal_internal.h

3. 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, pal_types.h, sal_types.h, pal_boardtypes.h

4. layer_config.h:

- This file contains definitions of layer specific stack resources such as timers or buffers.
- For further information see section 4.1.
- Examples: mac_config.h, tal_config.h, pal_config.h

4 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

- Only the PAL, or
- 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 6.

4.1 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:

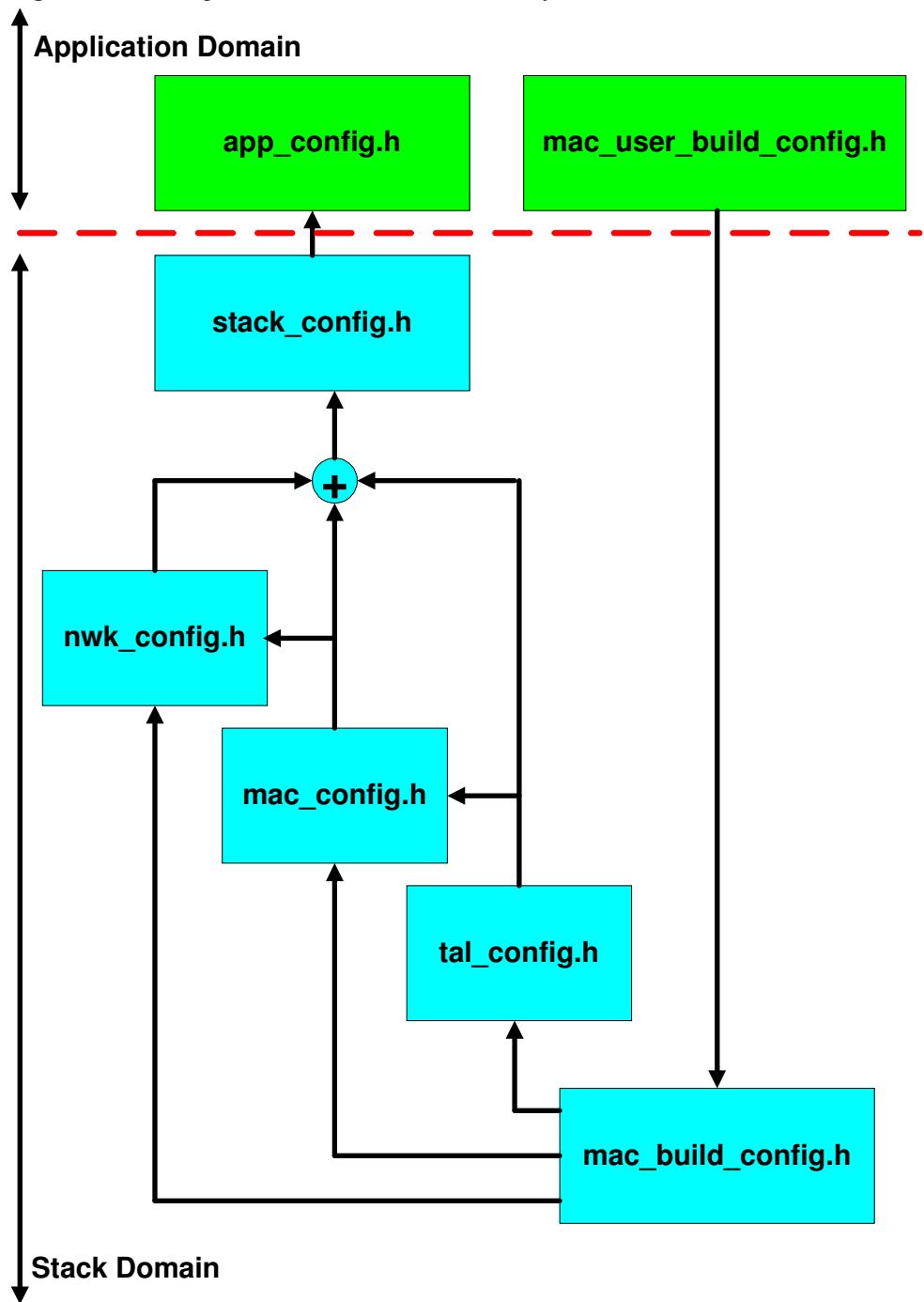
- app_config.h
- stack_config.h
- pal_config.h
- tal_config.h
- mac_config.h
- mac_build_config.h
- mac_user_build_config.h

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 4-1. Configuration File #include-Hierarchy



4.1.1 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 (e.g. 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.

4.1.2 Stack Resource 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.

This file must not be changed by the end user.

4.1.3 PAL Resource Configuration – `pal_config.h`

The PAL layer does not require own resources such as timer or buffers, so this does not need to be taken into consideration in the resource calculation for the application. Nevertheless there exists a unique `pal_config.h` file for each platform type (e.g. for each board) which can be found in the directories `PAL/pal_generic_type_name/pal_type_name/Boards/board_type_name`, for example `PAL/AVR/ATMEGA1281/Boards/REB_4_0_231`. These `pal_config.h` files define all platform specific hardware components.

4.1.4 TAL Resource Configuration – `tal_config.h`

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

- `TAL/ATMEGARF_TAL_1/Inc` (for ATmega128RFA1)
- `TAL/AT86RF212/Inc`
- `TAL/AT86RF230B/Inc`
- `TAL/AT86RF231/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.

This file must not be changed by the end user.

4.1.5 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 the 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.

This file must not be changed by the end user.

4.1.6 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.

This file must not be changed by the end user.

4.1.7 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 6.2.1 for more information about *mac_build_config.h*.

This file must not be changed by the end user.

4.1.8 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 6.2.2 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 4.2 and section 6.2.2.

An example for an application using the configuration file *mac_user_build_config.h* to create an application defined stack can be found in section 8.2.1.3.

4.2 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.

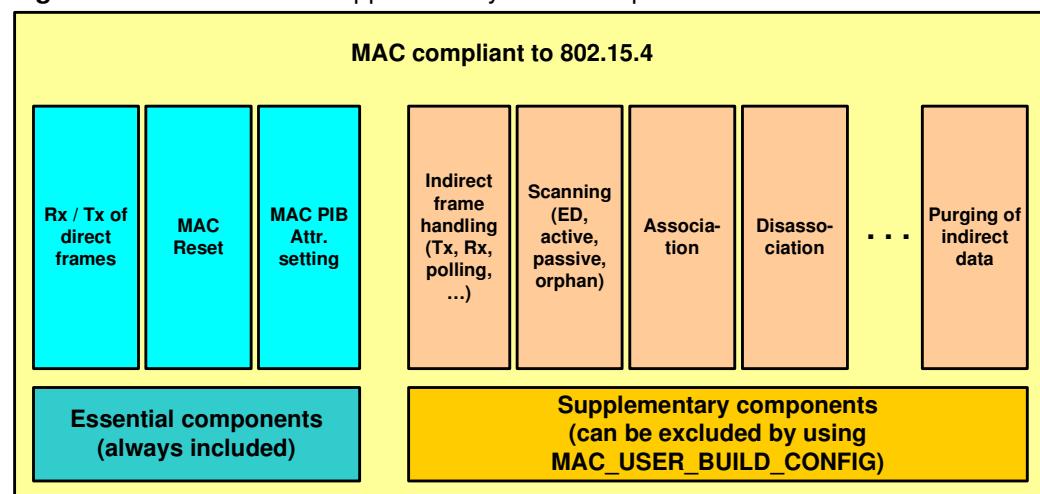
Essential components are required for a minimum reasonable application based on the MAC and are thus always included in a build. These components are:

- 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 6.2 (using build switch `MAC_USER_BUILD_CONFIG`).

Figure 4-2. Essential and Supplementary MAC Components



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

4.2.1 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 (i.e. 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

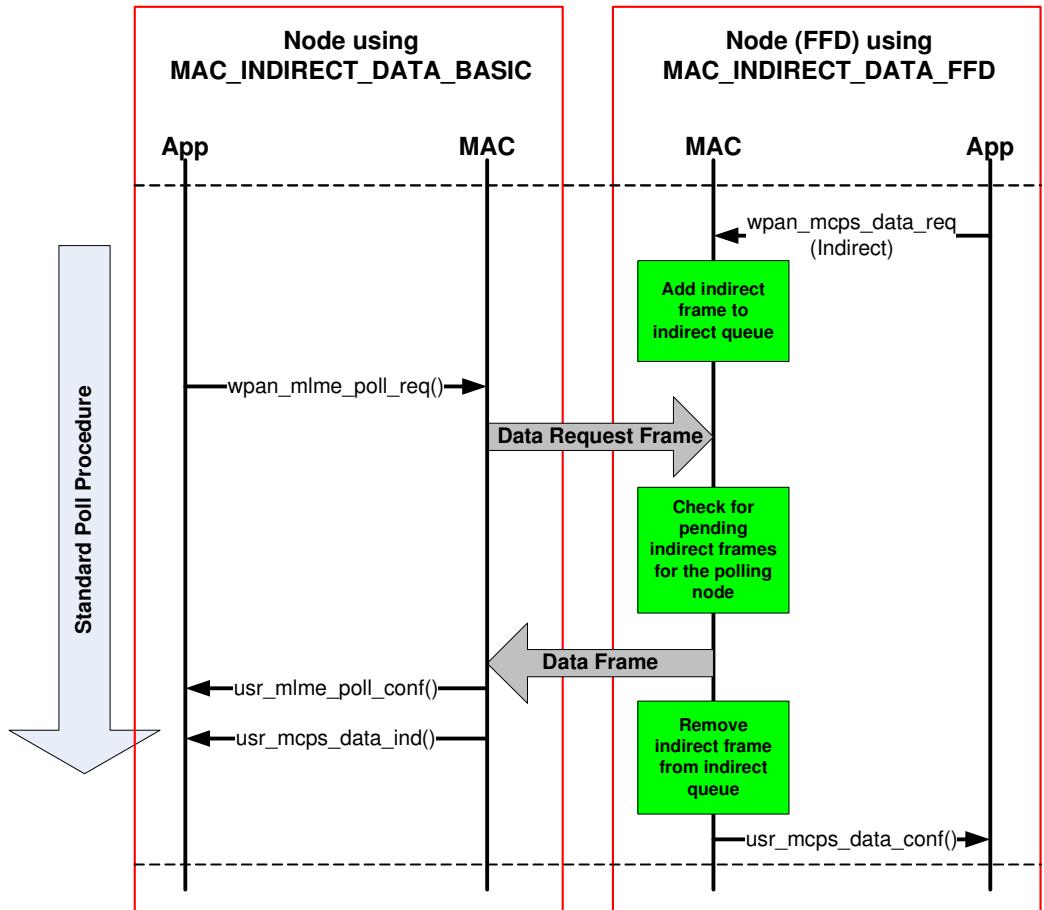
4.2.2 MAC_INDIRECT_DATA_FFD

This feature is a further extension of the feature MAC_INDIRECT_DATA_BASIC (i.e. 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 frame 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 4-3. Example of provided Functionality for MAC_INDIRECT_DATA_BASIC and MAC_INDIRECT_DATA_FFD



4.2.3 MAC_PURGE_REQUEST_CONFIRM

This feature is a typical FFD feature that allows a node to purge pending indirect frames from its indirect 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.

4.2.4 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 performed 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.

4.2.5 MAC_ASSOCIATION_REQUEST_CONFIRM

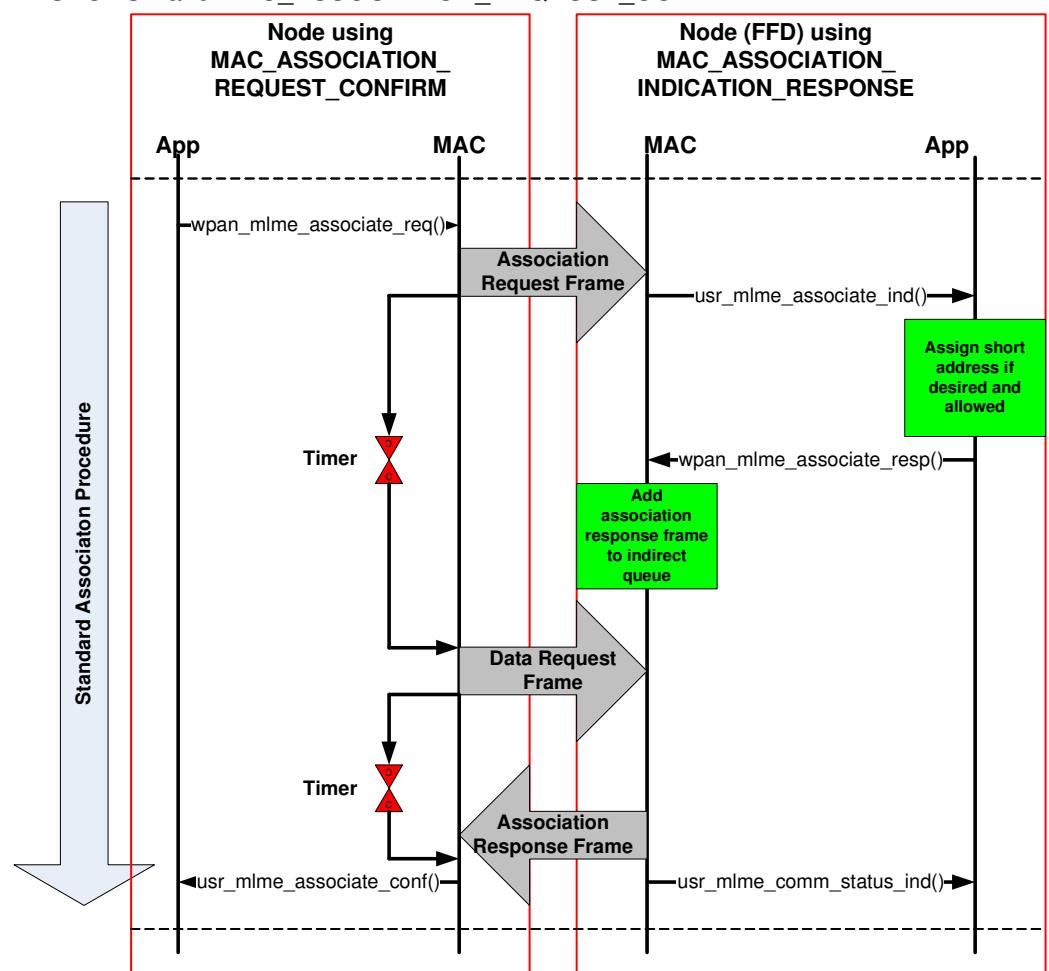
This feature allows a node (both RFD and FFD) to associate to a parent (PAN Coordinator or Coordinator) to initiate an association procedure (by transmitting an association request frame) and handle the reception of an association response frame.

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 (e.g. the network layer) to associate itself to another node (i.e. its parent).

Figure 4-4. Provided Functionality for MAC_ASSOCIATION_INDICATION_RESPONSE and MAC_ASSOCIATION_REQUEST_CONFIRM



4.2.6 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.

4.2.7 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`

4.2.8 MAC Scan Components

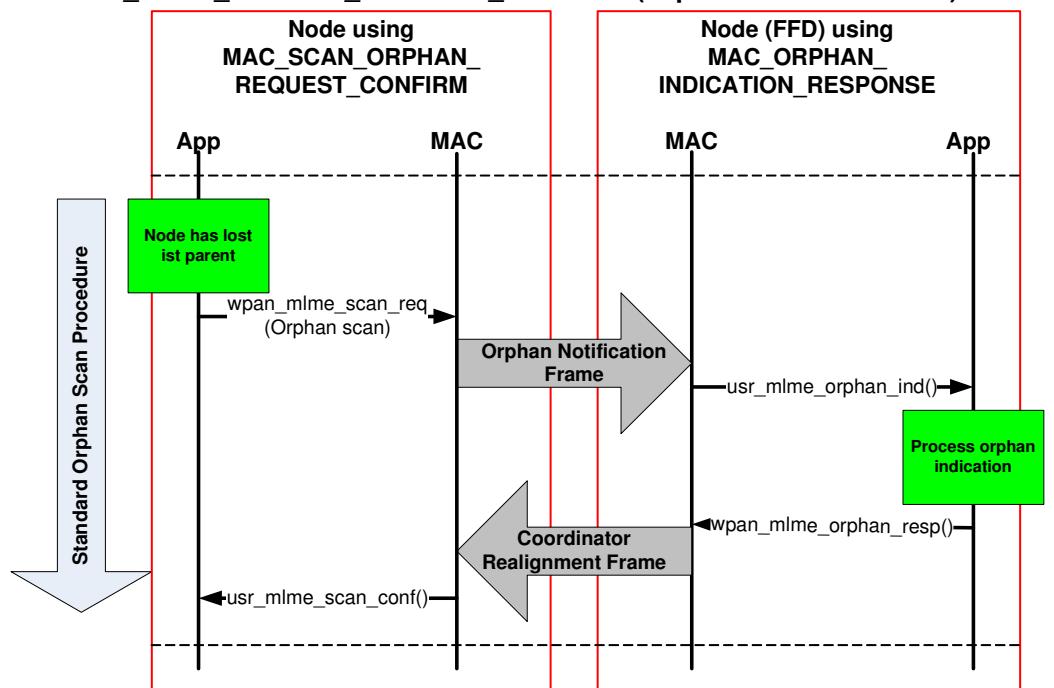
These components allow a node to perform a specific type of scanning.

- `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.

4.2.9 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 4-5. Provided Functionality for MAC_ORPHAN_INDICATION_RESPONSE and MAC_SCAN_ORPHAN_REQUEST_CONFIRM (Orphan Scan Procedure)



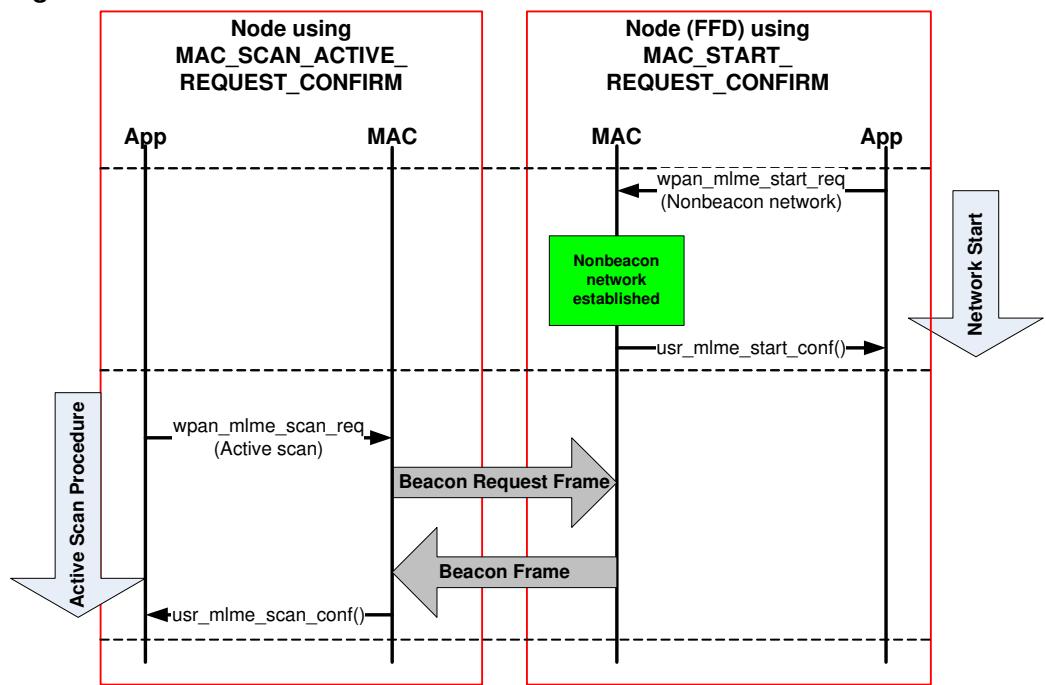
4.2.10 MAC_START_REQUEST_CONFIRM

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 nonbeacon-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`)

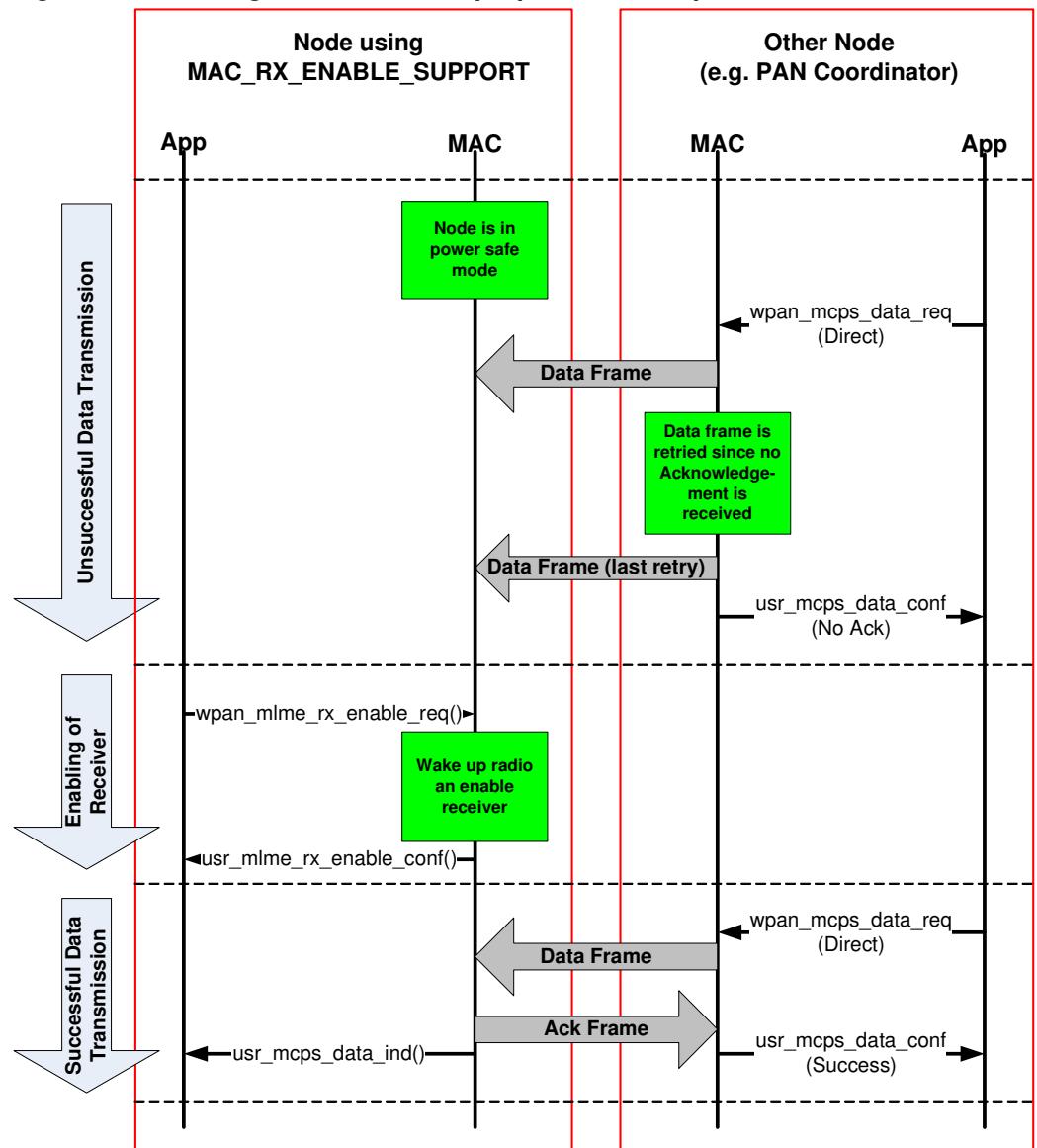
Figure 4-6. Start of Nonbeacon Network and Active Scan



4.2.11 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 8.2.1.3).

Figure 4-7. Enabling of Receiver and proper Data Reception



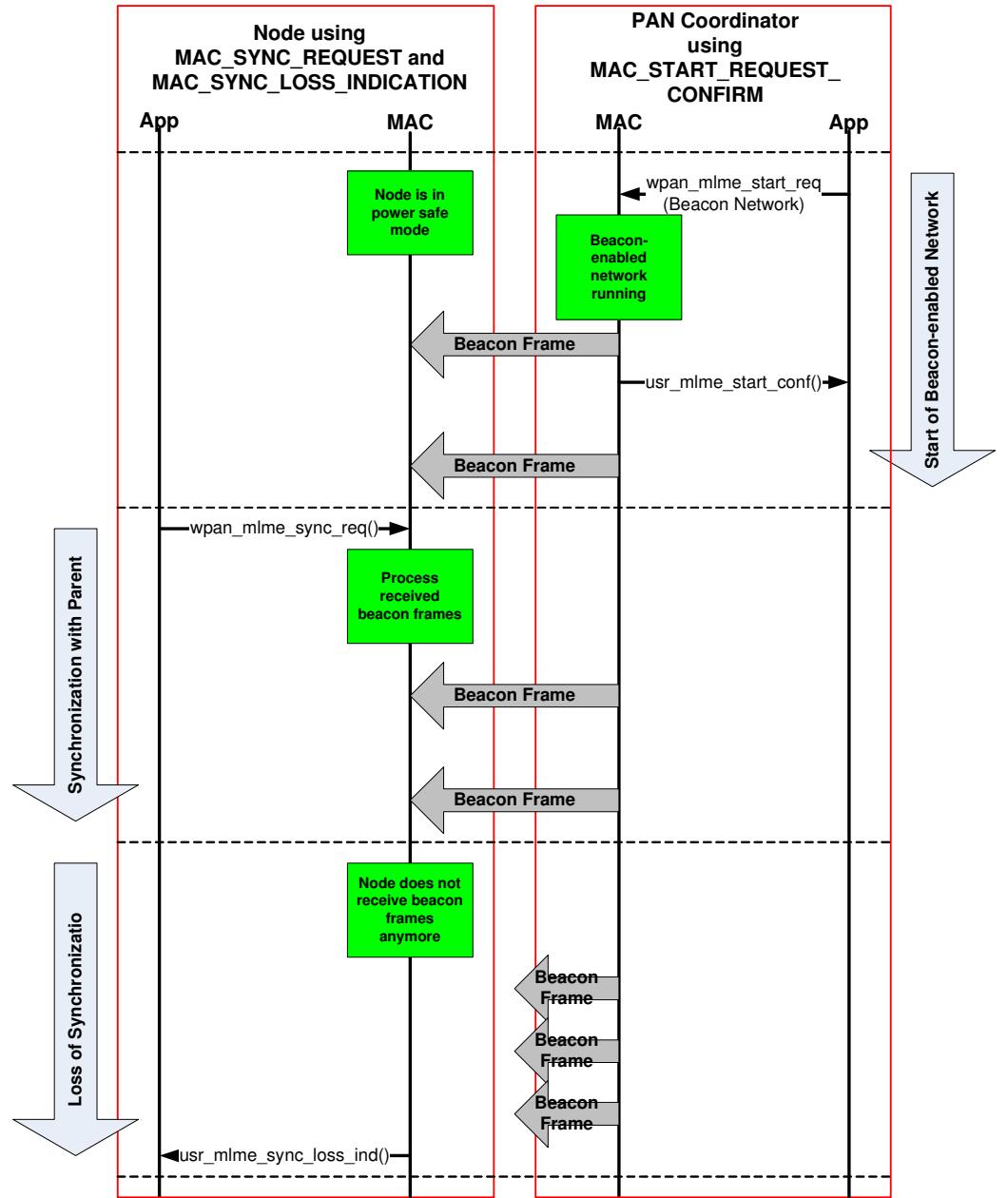
4.2.12 MAC_SYNC_REQUEST

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()`.

4.2.13 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 4-8. Synchronization and Loss of Synchronization



4.2.14 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.

4.2.15 MAC_GET_SUPPORT

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



4.2.16 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

4.2.17 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

4.3 Support of AVR Platforms larger than 128 KByte Program Memory

4.3.1 General

In order to support applications larger than 128 KByte program memory that are using function pointers (such as callbacks in the stack), special implementation support both by the stack and the application is required.

Function pointers are usually 16 Bit pointers, which can address only 64 KByte of program memory. Functions located in the memory between 64 KByte and 128 KByte can still be addressed by 16 Bit function pointers, since all addresses start on an even address in program memory, which increases the addressable space of a 16 Bit pointer to 127 KByte. So no special care is required for code up to 128 KByte.

In case function pointers shall be used for builds which are larger than 128 KByte (when using for instance the ATxmega256A3 MCU), the function pointers need to be declared as large pointers, i.e. pointers which are 24 Bit. The following section describes the current implementation used for IAR compilers.

Note: Currently AVR-GCC does not provide a clean method to support such constructs. Therefore program code larger than 128 KByte is not supported by this software package. Nevertheless there are a number of workarounds to overcome this issue by forcing all functions which shall be accessed via function pointers into the lower 128 KByte of memory. Such a workaround implementation for AVR-GCC is not provided by this software package.

4.3.2 Stack Implementation

The stack has been modified in order to support function pointers which can address functions residing in the program memory larger than 128 KByte.

4.3.2.1 File avrtypes.h

The header file *avrtypes.h* residing in directory *PAL/Inc* contains the following construct in the section dedicated to IAR based builds:

```
#if (FLASHEND > 0xFFFF)
    // Required for program code larger than 128K
    #define FUNC_PTR void __farflash *
#else
    #define FUNC_PTR void *
#endif /* ENABLE_FAR_FLASH */
```

In case the utilized AVR MCU provides more than 128 KByte of flash memory, all function pointers defined by FUNC_FAR are defined as 24 bit pointers.

4.3.2.2 Function Pointers as Function Parameters

Function parameters which are actually function pointers need to be defined as FUNC_PTR in order to allow for 24 Bit support. This is implemented in the following two PAL API functions:

- pal_trx_irq_init :

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

```
retval_t pal_timer_start(uint8_t timer_id,
                         uint32_t timer_count,
                         timeout_type_t timeout_type,
                         FUNC_PTR timer_cb,
                         void *param_cb)
```

4.3.2.3 Usage of Function Pointers as Function Parameters

These functions containing function pointers need to be called with the corresponding FUNC_PTR type. Examples are:

- File *mac_mcps_data.c* in directory *MAC/Src*

```
/* Start the indirect data persistence timer now. */
status = pal_timer_start(T_Data_Persistence,
                         persistence_int_us,
                         TIMEOUT_RELATIVE,
                         (FUNC_PTR)mac_t_persistence_cb,
                         NULL);
```
- File *tal_init.c* in directory *TAL/AT86RF231/Src*

```
/*
 * Configure interrupt handling.
 * Install a handler for the transceiver interrupt.
 */
pal_trx_irq_init(TRX_MAIN IRQ_HDLR_IDX,
                  (FUNC_PTR)trx_irq_handler_cb);
```



4.3.3 Application Support

Applications that are built for AVR MCUs with more than 128 KByte flash (and thus potentially need large function pointers) shall use the same way of calling these specific PAL API functions containing functions pointers. Therefore all applications are updated within this software package as described in section 4.3.2.3.

As an example see file main.c of the MAC example application Star_Nobeacon (in directory Applications/MAC_Examples/Star_Nobeacon/Src):

```
pal_timer_start(TIMER_TX_DATA,
                DATA_TX_PERIOD,
                TIMEOUT_RELATIVE,
                (FUNC_PTR)app_timer_cb,
                NULL);
```

It is strongly recommended to follow this scheme in all components of the application.

4.4 Application Security Support

The MAC stack supports application security features (such as the CCM* algorithm or plain AES encryption support) by means of two different layers:

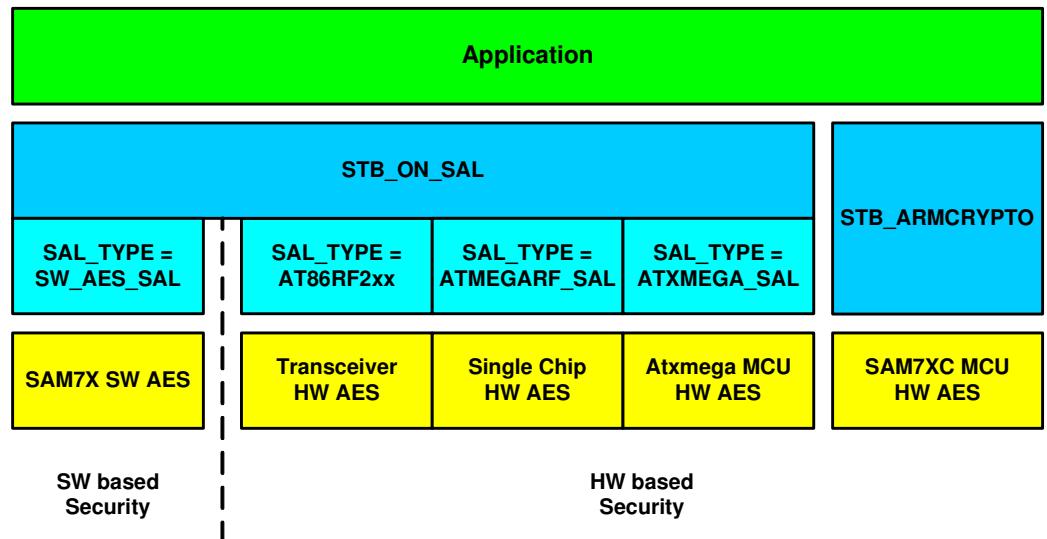
- SAL: Security Abstraction Layer (see section 2.2.2)
- STB: Security Toolbox (see section 2.2.3)

The SAL as low level AES functionality API is supported for various hardware AES engines (in MCUs and transceivers) and software AES (see section 6.1.5.1).

The STB as high level security abstraction API is

- Supported for SAL based systems (the STB requires either one of the SAL implementations, see section 6.1.5.2 and 6.1.5.1), or
- Used stand-alone (without using any SAL implementation) for ARM systems with incorporated hardware crypto engine (see section 6.1.5.3) such as the AT91SAM7C256.

Figure 4-9. Usage of the Security Layers by an Application



The figure above gives an overview which combinations of the two security layers are meaningful. It can be seen that the following security combinations are supported:

1. Utilization of hardware AES in transceivers, such as AT86RF212 and AT86RF231, by using the STB on top of the SAL (SAL_TYPE = AT86RF2xx);
This combinations can be used in conjunction with any MCU
2. Utilization of hardware AES in single chip transceivers, such as ATmega128RFA1, by using the STB on top of the SAL (SAL_TYPE = ATMEGARF_SAL)
3. Utilization of hardware AES in ATxmega MCUs, such as ATxmega128A1, by using the STB on top of the SAL (SAL_TYPE = ATXMEGA_SAL);
This combination can be used in conjunction with any transceiver, even with AT86RF230
4. Utilization of software AES, by using the STB on top of the SAL (SAL_TYPE = SW_AES_SAL);
This combination is recommended only for powerful MCUs, such SAM7X based systems, although theoretically any MCU is supported, and can be used with any transceiver, even with AT86RF230; In order to get the SW AES implementation please contact avr@atmel.com
5. Utilization of the ARM hardware crypto engine by using STB_ARMCRYPTO without any SAL;
This combination is supported for SAM7XC MCUs and can be used in conjunction with any transceiver, even with AT86RF230

5 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.

5.1 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 5.4.

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, i.e. 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 safe 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 6.2.1.1).

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, i.e. 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 in more 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, 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 (e.g. 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 use 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 idle periods in SLEEP is not able to receive any frame during these time periods, i.e. a PAN

Coordinator would not be able to send a direct frame to its sleeping child being and End Device. This can be reproduced using the provided MAC example applications App_1_Nobeacon (see 8.2.1.1) and Star_Nobeacon (see 8.2.1.7). 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 Basic_Sensor_Network (see 8.2.1.3).

5.2 Reception of Data at Nodes applying Power Management

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

5.2.1 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 use is 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.

5.2.2 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, e.g. one symbols is 16us for 2.4 GHz 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 (i.e. in case macRxOnWhenIdle is False).





5.2.3 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 5.2.2), 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 by 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 (i.e. in case macRxOnWhenIdle is False).

5.2.4 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 section 4.2.1 and 4.2.2.

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

5.3 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.

5.3.1 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 (e.g. 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 8.2.1.3):

```
/* 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 transceiver will be immediately enabled or disabled.

5.3.2 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()`.

For an example implementation of this features see MAC example application Basic_Sensor_Network in section 8.2.1.3.

5.4 TAL Power Management API

The MAC power management mechanisms as described in the within 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 only residing on top 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.

6 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 IAR project files
- Several Makefile or IAR project file templates are provided for getting started (see directory Applications).
- 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.

6.1 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:

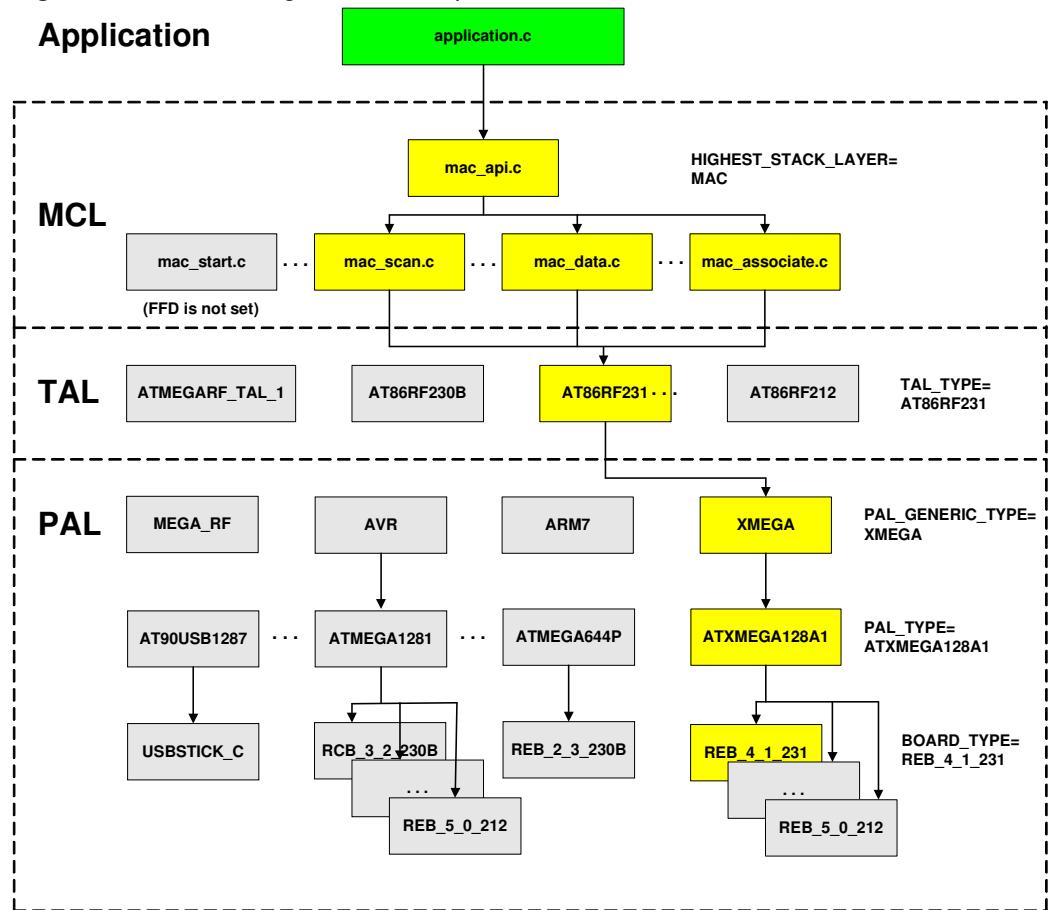
1. Global stack switches
 - HIGHEST_STACK_LAYER
 - REDUCED_PARAM_CHECK
 - PROMISCUOUS_MODE
2. Standard or user build configuration switches
 - BEACON_SUPPORT
 - FFD
 - MAC_USER_BUILD_CONFIG
3. Platform switches
 - PAL_TYPE
 - PAL_GENERIC_TYPE
 - BOARD_TYPE
 - SIO_HUB;UART0,UART1,USB0
 - ENABLE_TRX_SRAM
 - ENABLE_HIGH_PRIO_TMR





- EXTERN_EEPROM_AVAILABLE
 - NON_BLOCKING_SPI
 - F_CPU (formerly SYSTEM_CLOCK_MHZ)
 - BAUD_RATE
4. Transceiver specific switches
- TAL_TYPE
 - ANTENNA_DIVERSITY
 - ENABLE_TFA
 - HIGH_DATA_RATE_SUPPORT
 - CHINESE_BAND
 - RSSI_TO_LQI_MAPPING
 - ENABLE_FTN_PLL_CALIBRATION
 - DISABLE_IEEE_ADDR_CHECK
5. Security switches
- SAL_TYPE
 - STB_ON_SAL (formerly ENABLE_STB)
 - STB_ARMCRYPTO
6. 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 (e.g. TFA or STB have been omitted). The used build switches are explained in the subsequent sections.

Figure 6-1. Build Configuration Example

6.1.1 Global Stack Switches

6.1.1.1 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 are forming 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 (i.e. which API it is using). If an application for instance only uses the PAL and TAL layers (i.e. 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“.

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

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 timer resources are used and included in the end application depending upon the highest stack used.

6.1.1.2 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.

As an example what this build switch causes, please check file MAC\Src\MAC\mac_mcps_data.c. In function mcps_data_request() a number of additional checks are performed if this switch is not set.

It is strongly recommended to disable this 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.

6.1.1.3 PROMISCUOUS_MODE

This build switch allows for the creation of a special node that can be put into promiscuous mode thus allowing it acting as a very simple frame sniffer on a specific sniffer. It can be used 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, i.e. the node will never acknowledge any received frame, etc. Instead the node will be able 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.

An example application presenting all received frames on a serial terminal can be found in the directory Applications/MAC_Examples/Promiscuous_Mode_Demo (see section 8.2.1.6).

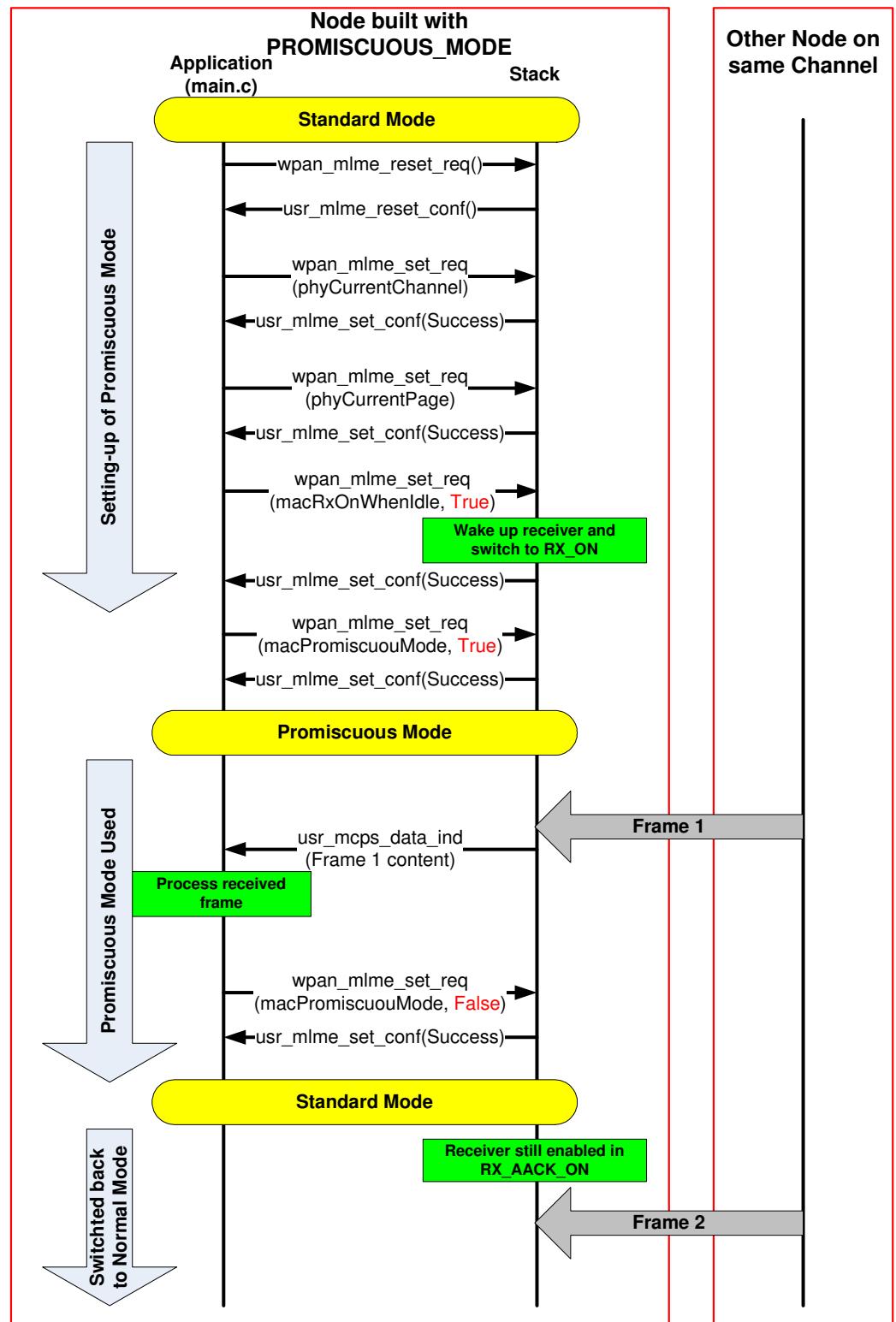
Promiscuous mode can be switched on or off by setting or resetting the PIB attribute *macPromiscuousMode*. If the radio is awake and in receive mode on the current channel, the node will present all received frames to the upper layer. In order to work properly, the receiver needs to be enabled. This 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 5).

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, e.g. a node that was associated will still be connected to the same network.

The following picture indicates the proper handling of promiscuous mode.



Figure 6-2. Handling of Promiscuous Mode



6.1.2 Standard and User Build Configuration Switches

The standard and user build configuration switches are described in detail in sections 6.2.1 and 6.2.2.

6.1.3 Platform Switches

The PAL (Platform Abstraction Layer) contains all platform (i.e. MCU and board) based functionality and provides an API to the TAL which is independent from the underlying platform.

6.1.3.1 PAL_GENERIC_TYPE

Certain portions of the PAL are generic to a specific microcontroller family. Examples for such generic code are transceiver access via SPI for AVR 8-bit controller (see file AVR\Generic\Src\pal_trx_access.c), or using Event System of AVR XMEGA controller. This code is separated from the non-generic code in special directories.

In order to make sure that the correct code belonging to the proper microcontroller family is used for the application, the build switch PAL_GENERIC_TYPE needs to be set accordingly.

The currently defined microcontroller families are:

- AVR
- XMEGA
- AVR32
- ARM7

For more information check file PAL\Inc\pal_types.h.

Usage in Makefiles:

```
CFLAGS += -DPAL_GENERIC_TYPE=XMEGA
```

selects the AVR XMEGA microcontroller family.

Usage in IAR ewp files:

```
PAL_GENERIC_TYPE=AVR
```

selects the AVR 8-bit ATmega microcontroller family.

6.1.3.2 PAL_TYPE

To build the proper code the compiler needs to know which microcontroller is used. Also depending on the microcontroller specific code portions needs to be used within the build (since for example some microcontrollers do not support USB while others do). The type of microcontroller is specified by the build switch PAL_TYPE.

Examples of currently supported microcontrollers are:

- AT90USB1287
- ATmega1281
- ATmega2561
- ATmega644P
- ATxmega128A1

For more information check file PAL\Inc\ pal_types.h.

Usage in Makefiles:

```
CFLAGS += -DPAL_TYPE=ATXMEGA128A1
```

selects the ATxmega128A1 microcontroller.

Usage in IAR ewp files:





PAL_TYPE=ATMEGA644P
selects the ATmega644p microcontroller.

6.1.3.3 BOARD_TYPE

Since the platform may vary between different boards even for one specific microcontroller, certain functionality within a specific PAL has to be implemented depending on the type of board used.

The supported board types for one microcontroller can be found in file pal_boardtypes.h for each MCU, e.g. for ATmega2561 this file is located in directory PAL\AVR\ATMEGA2561\Boards.

If a new board is designed which requires specific software changes, a new board has to be added to the PAL and the board type has to be added to file pal_boardtypes.h.

Usage in Makefiles:

```
CFLAGS += -DBOARD_TYPE=USBSTICK_C  
selects the Raven USB stick revision C.
```

Usage in IAR ewp files:

```
BOARD_TYPE=RCB_4_1_SENS_TERM_BOARD  
selects the Radio Controller Board Revision 4.1 using antenna diversity for AT86RF231  
microcontroller.
```

6.1.3.4 SIO_HUB, UART0, UART1, USBO

Certain applications (e.g. TAL example Wireless_UART) require Serial I/O support (SIO) in order to print characters on a serial terminal or get input from the user. For easy SIO configuration the concept of a SIO-Hub was introduced. This implements a simple software multiplexer / demultiplexer which allows the application or stack to access SIO hardware resources without actually knowing which kind of SIO hardware support is provided. The actually supported hardware is currently UART or USB.

In order to enable SIO functionality the build switch SIO_HUB has to be set. Also the proper USB or UART channel needs to be set.

Usage in Makefiles:

```
CFLAGS += -DSIO_HUB -DUSBO  
enables the SIO-Hub and uses USB channel 0.
```

Usage in IAR ewp files:

```
SIO_HUB  
UART1  
enables the SIO-Hub and uses UART channel 1.
```

6.1.3.5 ENABLE_TRX_SRAM

In general there are two different ways of accessing the memory of the transceiver, either by frame buffer access (1) or by direct SRAM access (2). Please check the data sheet for more information. When a frame needs to be uploaded or downloaded into or from the transceiver, this can be done by using the corresponding frame buffer functions or transceiver SRAM access functions. It is always recommended to use the frame buffer functions.

For more information please see file PAL\Inc\pal.h or file PAL\AVR\Generic\Src\pal_trx_access.c.

Functions for frame buffer access (always enabled):

```
void pal_trx_frame_read(uint8_t* data, uint8_t length);
void pal_trx_frame_write(uint8_t* data, uint8_t length);
```

Functions for direct transceiver SRAM access (only enabled if ENABLE_TRX_SRAM is set):

```
void pal_trx_sram_read(uint8_t addr, uint8_t *data, uint8_t length);
void pal_trx_sram_write(uint8_t addr, uint8_t *data, uint8_t length);
```

For transceivers providing AES security the SRAM functions are required if transceiver security is used.

For AT86RF230 it is recommended not using this switch in order to save code space.

6.1.3.6 ENABLE_HIGH_PRIO_TMR

AT86RF230 transceivers need to have an ED sampling timer running in case Energy Detect scanning is used. This timer is implemented by means of a unique and very accurate hardware timer (High priority timer). In other transceivers (AT86RF231 or AT86RF212) this hardware timer is not required for Energy Detect scanning, since additional hardware support is provided.

On the other hand the provided standard software timers have limited accuracy, since several software timers share one hardware resource. An application may want to use this timer if desired for their own purpose. In this case this build switch has to be set. But the application has to make sure that it is not using this hardware timer concurrently with the stack.

All provided example applications within the MAC package do not use this feature, so in order to save code space this timer is not enabled.

Usage in Makefiles:

```
CFLAGS += -DENABLE_HIGH_PRIO_TMR
enables the code for the high priority timer.
```

Usage in IAR ewp files:

```
ENABLE_HIGH_PRIO_TMR
enables the code for the high priority timer.
```

6.1.3.7 EXTERN_EEPROM_AVAILABLE

Each node needs to have a unique 64-bit IEEE address (Extended address) to identify itself within the network. All example applications delivered within the MAC software package are only working if the node has such an IEEE address.

This address is usually stored in an EEPROM, either in the internal EEPROM of the MCU or in an external chip on the board. If the hardware board has such an external EEPROM where the IEEE address is stored, this build switch has to be enabled. The actual implementation of the code for reading the address from an external EEPROM is board dependent and might be adapted by the end user depending on their hardware.





If the IEEE address is stored within the internal EEPROM of the MCU (located in the first 8 octets of the internal EEPROM), this build switch should to be omitted.

Usage in Makefiles:

```
CFLAGS += -DEXTERN_EEPROM_AVAILABLE
```

enables reading the IEEE address from the external EEPROM.

Usage in IAR ewp files:

```
EXTERN_EEPROM_AVAILABLE
```

enables reading the IEEE address from the external EEPROM.

6.1.3.8 NON_BLOCKING_SPI

Currently, for all transceivers connected to the microcontroller via SPI, the SPI frame download or upload is done blocking, i.e. while the controller is access the transceiver no other action can be on the microcontroller.

Since more powerful microcontroller may require performing other application or stack tasks while accessing the transceiver via SPI, an option to perform non-blocking SPI access has been implemented. This access makes use of interrupt driven SPI access for downloading frames. In between the microcontroller is able to do other tasks as required.

The non-blocking SPI can be controlled via the build switch NON_BLOCKING_SPI.

Usage in Makefiles:

```
CFLAGS += -DNON_BLOCKING_SPI
```

enables non-blocking SPI frame download.

Usage in IAR ewp files:

```
NON_BLOCKING_SPI
```

enables non-blocking SPI frame download.

6.1.3.9 F_CPU (formerly SYSTEM_CLOCK_MHZ)

Currently each supported platform (i.e. each supported board) has a specific clock speed set for the system clock. This clock speed might need to be changed for certain applications.

For the platform ATxmega128A1 with AT86RF231 on REB_4_1_600 the clock speed can be changed at compile time (4...32MHz).

The current system clock speed can be controlled via the build switch F_CPU. If the build switch is omitted, the standard system clock speed is used.

Usage in Makefiles:

```
CFLAGS += -DF_CPU=4/8/32
```

configures the system clock to 4/8/32MHz.

Usage in IAR ewp files:

```
F_CPU=4/8/32I
```

configures the system clock to 4/8/32MHz.

Example Makefiles and IAR project files are provided for the MAC example application Star_Nobeacon.

For more information please see section 10.1.

6.1.3.10 BAUD_RATE

This build switch allows for changing the used UART baud rate. The default UART baud rate is 9600 bps. This may be changed by setting this build switch to a different meaningful value.

Usage in Makefiles:

```
CFLAGS += -DBAUD_RATE=38400
```

configures the UART baud rate to 38400 bps.

Usage in IAR ewp files:

```
BAUD_RATE=38400
```

configures the UART baud rate to 38400 bps.

Omitting this build switch sets the default baud rate of 9600 bps. For USB this build switch has no meaning.

Please note that the proper working of the UART with certain baud rates is depending on the current system clock speed (see build switch F_CPU) and the error in the used oscillator frequency. As a rule of thumb the error of the oscillator frequency shall not be more than 2%.

For example, in case the ATmega1281 is used with F_CPU=8MHz, the error for 38400 bps is 0.2%, which is perfectly fine, but for 115200 bps the error is -3.5% which may lead to improper functioning of the UART. For more information about the UART oscillator frequency error please check the data sheet of the utilized MCU type.

6.1.4 Transceiver specific Switches

6.1.4.1 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:

- ATmega128RFA1 (TAL directory ATMEGARF_TAL_1)
- AT86RF230B (AT86RF230 Revision B)
- AT86RF231
- AT86RF212

For more information please check file TAL\Inc\tal_types.h.

Usage in Makefiles:

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

selects the AT86RF212 transceiver.





Usage in IAR ewp files:

`TAL_TYPE=AT86RF231`

selects the AT86RF231 transceiver.

6.1.4.2 ANTENNA_DIVERSITY

Some IEEE 802.15.4 Atmel transceivers allow for the utilization of antenna diversity as hardware feature (e.g. AT86RF231) if the board also does support this feature. The current MAC release supports antenna diversity where applicable. In order to use antenna diversity this build switch has to be set.

For AT86RF230 transceivers this switch is not used.

Usage in Makefiles:

`CFLAGS += -DANTENNA_DIVERSITY`

enables the usage of antenna diversity.

Usage in IAR ewp files:

`ANTENNA_DIVERSITY`

enables the usage of antenna diversity.

6.1.4.3 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 of current temperature (only ATmega128RFA1 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 ewp files:

`ENABLE_TFA`

enables the usage of the TFA.

6.1.4.4 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 Test Application (see Applications\TAL_Examples\Performance_Test). 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.
```

6.1.4.5 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 (e.g. 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.

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.
```

For more information please check file *tal_constants.h* in directory TAL/AT86RF212/Inc. Example applications using this build switch in Makefiles or IAR project files can be found in directory

Applications/TAL_Examples/Wireless_UART/AT86RF212_780_MHZ_ATMEGA1281_R
CB_5_3_SENS_TERM_BOARD.

6.1.4.6 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.

6.1.4.7 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 (i.e. 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 5 minutes, i.e. 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.

6.1.4.8 DISABLE_IEEE_ADDR_CHECK

This build switch is used to disable the check whether the node has a valid IEEE address (i.e. the IEEE address is different from 0 or 0xFFFFFFFFFFFFFF).

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 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 (i.e. it is 0 or 0xFFFFFFFFFFFFFF), 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 of 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.

6.1.4.9 DISABLE_TSTAMP_IRQ

This build switch is used to disable the Timestamp interrupt (i.e. the second transceiver interrupt for AT86RF231 and AT86RF212) on systems which do not utilize this transceiver interrupt. This is for example valid for boards based on ARM AT91SAM7X256 MCUs in conjunction with AT86RF231.

The TAL for AT86RF231 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 AT86RF230 based systems.

6.1.5 Security Switches

In order to provide security support to the application a number of switches controlling the incorporation of code for security is defined.

For examples on how to use security within an application based on MAC or based on the TAL please check the applications in directory Applications\STB_Examples.

6.1.5.1 SAL_TYPE

Security support is implemented in two different layers. The lower layer (SAL – Security Abstraction Layer) implements all portions of security that are dependent of the used platform (e.g. AES engine implemented in transceiver or microcontroller, etc.). The upper layer (STB – Security Toolbox) implements functionality that is independent from the used platform and provides a generic API to the application (for instance securing a frame with CCM* security).

In order to provide the proper code or hardware support for basic security functions the type of the used Security Abstraction Layer needs to be specified as build switch.

The currently supported SAL types are:

- AT86RF2xx – SAL with transceiver based AES via SPI
- ATMEGARF_SAL – SAL with single chip transceiver based AES
- ATXMEGA_SAL – SAL with ATxmega family based AES
- SW_AES_SAL - AES software implementation

For more information please check file SAL\Inc\sal_types.h and see section 4.4.

Usage in Makefiles:

```
CFLAGS += -DSAL_TYPE=AT86RF2xx
```

enables transceiver based AES security via SPI.

Usage in IAR ewp files:

```
SAL_TYPE=AT86RF2xx
```

enables transceiver based AES security via SPI.

6.1.5.2 STB_ON_SAL (formerly ENABLE_STB)

While the SAL only provides a basic security API, most applications do require more advance security features, e.g. securing a complete frame with a specific key. Such functionality is implemented in the Security Toolbox.

To incorporate the Security Toolbox features for system requiring the STB layer on top of an SAL implementation (see section 4.4) during the build process the switch STB_ON_SAL needs to be set.





Usage in Makefiles:

```
CFLAGS += -DSTB_ON_SAL
```

enables the Security Toolbox features.

Usage in IAR ewp files:

```
STB_ON_SAL
```

enables the Security Toolbox features.

6.1.5.3 STB_ARMCRYPTO

The ARM hardware crypto engine residing in the SAM7XC is very powerful and thus does not require the implementation of an additional SAL layer for such systems. Therefore the utilization of security for SAM7XC systems allows for the easy handling of security features by only relying on a special STB implementation called STB_ARMCRYPTO.

To incorporate the Security Toolbox features using the ARM crypto engine during the build process the switch STB_ARMCRYPTO needs to be set (see section 4.4).

Usage in IAR ewp files:

```
STB_ARMCRYPTO
```

enables the Security Toolbox features for SAM7XC systems.

6.1.6 Test and Debug Switches

6.1.6.1 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).

Using IAR compiler the printout functionality is currently not provided. In order to have printout capabilities included for DEBUG=1, one has to implement a system specific function __write(). See TAL application Performance_Test as an example how this can be implemented.

6.1.6.2 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()).

6.2 Build Configurations

6.2.1 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 (e.g. End Devices) vs. Full Functional Devices – FFD (PAN Coordinators, Coordinators)

The Atmel MAC provides 2 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

- Omitting 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

- Omitting of this build switch only enables functionality required for a networks without using a superframe structure, i.e. nonbeacon-enabled networks
- Setting of this build switch additionally enables functionality required for a network using a superframe structure, i.e. 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 nonbeacon- 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.

6.2.1.1 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 (e.g. the network layer) to associate itself to another node (i.e. 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 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 save 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 nonbeacon-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.

6.2.1.2 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 (e.g. the network layer) to associate itself to another node (i.e. 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 save 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.

6.2.1.3 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.

6.2.2 User Build Configurations – MAC_USER_BUILD_CONFIG

6.2.2.1 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 Applications/MAC_Examples/Basic_Sensor_Network. In this example only RX-

ENBALE is used in addition to the standard features. For more information about this example application please refer to section 8.2.1.3.

6.2.2.2 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, i.e. each feature needs to be either enabled or disabled.

An example for a basic MAC application that only needs minimum features, and thus only requires minimum resources, can be found at Applications/MAC_Examples/Basic_Sensor_Network.

6.2.2.2.1 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 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.





6.2.2.3 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.

6.2.2.3.1 MAC_COMM_STATUS_INDICATION

Communication systems usually follow the approach to implement primitives in pairs. This is (1) Request / Confirm (e.g. MLME_ASSOCIATE-request and MLME_ASSOCIATE.confirm, or (2) Indication / Response (e.g. 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 (e.g. 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.

6.2.2.3.2 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.

6.2.2.3.3 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.

6.2.2.3.4 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

6.2.2.3.5 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

6.2.2.3.6 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

6.2.2.3.7 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.



7 Tool Chain

The following chapter describes the used tool chain for the development and build process and how the provided example applications can be built.

7.1 General Prerequisites

The following tool chain is used for building the applications from this MAC package:

- AVR Studio 4.18
(see http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725)
- WinAVR 20100110 including AVR-GCC for Windows
(see <http://sourceforge.net/projects/winavr>)
- IAR Embedded Workbench for Atmel AVR V5.40.1
(see <http://www.iar.com/>)
- IAR Embedded Workbench for Atmel ARM V5.41
(see <http://www.iar.com/>)

7.2 Building the Applications

All example applications contain precompiled hex-files that can be downloaded and run out-of-the-box.

In order to rebuild any of the applications for any desired hardware platform one of the following three ways described in the subsequent sections can be chosen generally.

7.2.1 Using GCC Makefiles

Each application can be rebuilt 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, e.g.

```
cd Applications\MAC_Examples\Promiscuous_Mode_Demo  
cd AT86RF212_ATMEGA1281_RCB_5_3_SENS_TERM_BOARD  
cd GCC  
• Run the desired Makefile, e.g.  
make -f Makefile  
or  
make -f Makefile_Debug
```

Note: Makefile builds a binary optimized for code size without Serial I/O support, whereas Makefile_Debug build 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 7.3).

7.2.2 Using AVR Studio

Each application can be rebuilt using the AVR Studio directly. Please follow the procedure as described:

- Change to the directory where the AVR Studio project file (aps-file) for the desired platform of the corresponding application is located, e.g.

```
cd Applications\MAC_Examples\Promiscuous_Mode_Demo
cd AT86RF212_ATMEGA1281_RCB_5_3_SENS_TERM_BOARD
```
- Double click on the corresponding AVR Studio Project file (aps-file), e.g. *Promiscuous_Mode_Demo.aps*.
- Select the desired configuration (Release or Debug). Depending on the selected configuration the corresponding external Makefile is chosen during the build process. These Makefiles are exactly those Makefiles (located in subdirectory GCC) that are used to build the application from command line (see section 7.2.1).
- Rebuild the entire application in AVR Studio.
- After building the application the subdirectory GCC contains both a hex-file and an elf-file which can be downloaded onto the hardware (see section 7.3).

7.2.3 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, e.g.

```
cd Applications\MAC_Examples\Promiscuous_Mode_Demo
cd AT86RF212_ATMEGA1281_RCB_5_3_SENS_TERM_BOARD
```
- Double click on the corresponding IAR Embedded Workbench file (eww-file), e.g. *Promiscuous_Mode_Demo.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 7.3).
- The Release configuration binary (a90-file) can both 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 it is desired to create a binary with IAR 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 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 “ubprof 8 (forced)” as “Output format”
 - Select “None” as “Format variant”
 - Rebuild the application
 - The generated binary can now contain debug information that can be used directly within AVR Studio





7.2.4 Batch Build

If several applications shall be built or all applications need to be re-built, bat-files are provided to initiate an automatic batch build. Please check directory Build and run the corresponding bat-file as desired.

7.3 Downloading an Application

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

Please note that in case the board stores its IEEE address in the internal EEPROM of the microcontroller, it is important to ensure, that this IEEE address is not overwritten by the tool, i.e. the content of the internal EEPROM needs to be preserved. If this rule is not followed properly, the EEPROM content might be overwritten, and the example applications will not run, since all example application are required to have a unique IEEE available for each node being detected during run-time.

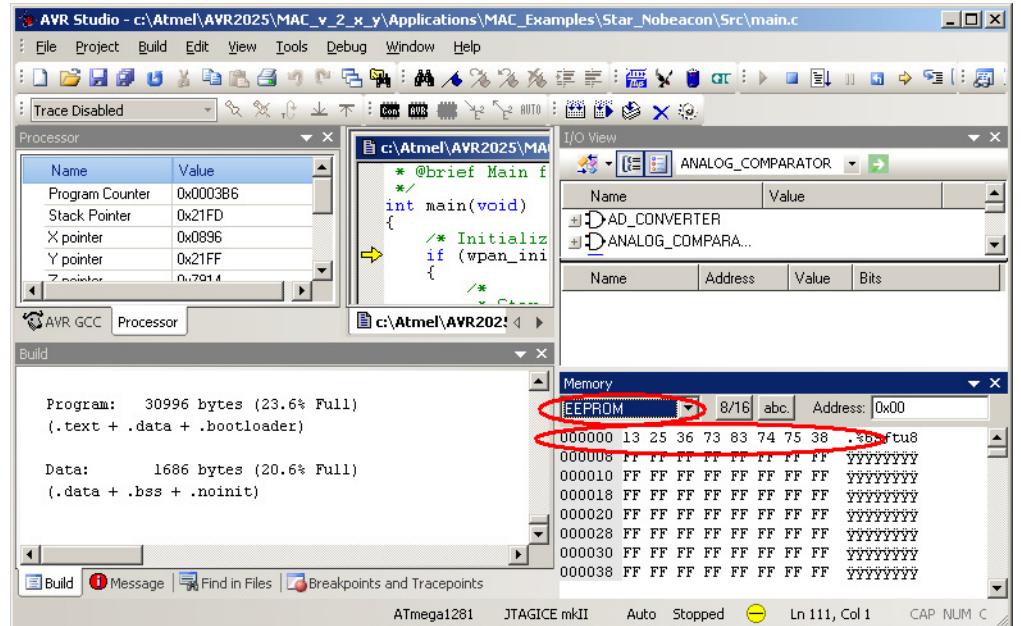
In the subsequent section this is indicated both for AVR Studio and IAR Embedded Workbench.

7.3.1 Using AVR Studio directly

When the application has been built using AVR Studio directly by double clicking the corresponding aps-file, the EEPROM settings are handled properly without any user interaction. The provided aps-files contain an entry that requires the system to preserve the current EEPROM settings (<PRESERVER_EEPROM>1</PRESERVE_EEPROM>).

In order to re-build the application and download it onto the desired hardware platform, follow the steps below:

- Select menu “Build” item “Build and Run”.
- Once the application has been built successfully, (and in case there has been more than one JTAGICE mkII detected) a window pops-up asking to select the proper JTAGICE mkII to be used.
- In case the IEEE address of the node is stored in the internal EEPROM of the microcontroller perform as follows:
 - After successful download of the application check whether a valid IEEE address (different from 0xFFFFFFFF) is stored in the internal EEPROM. Select menu “View” item “Memory”.
 - If the IEEE address is not set properly, write the correct IEEE to the first 8 octets of the EEPROM (see picture below).
- Start the application by pressing “F5” or clicking the “Run” button.

Figure 7-1. AVR Studio - Verifying and Setting of IEEE Address

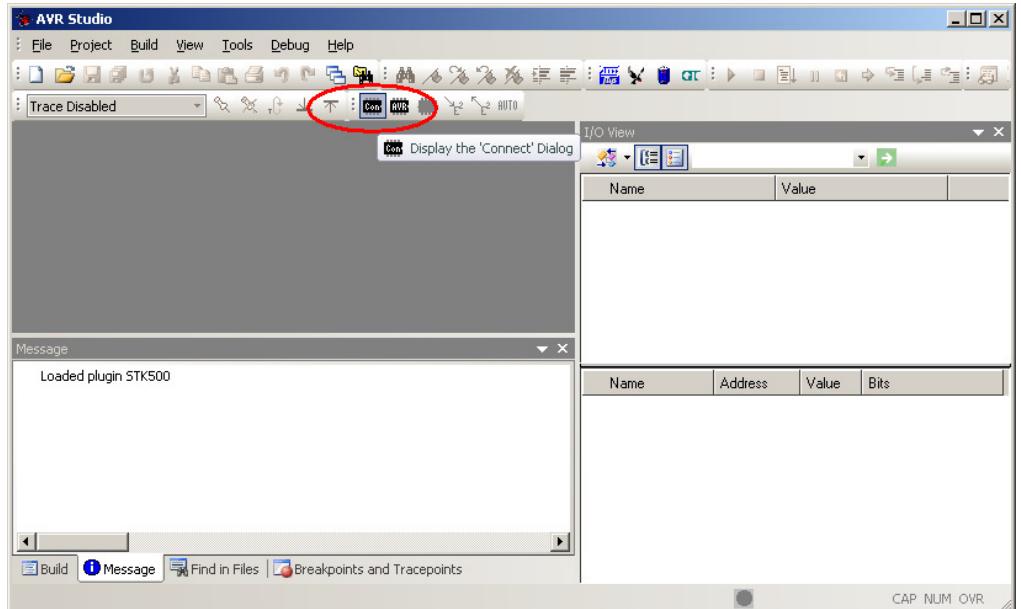
7.3.2 Using AVR Studio after Command Line Build of Application

When the application has been built using external Makefiles from the command line, the application can be started following the steps described below.

7.3.2.1 Starting the Release Build

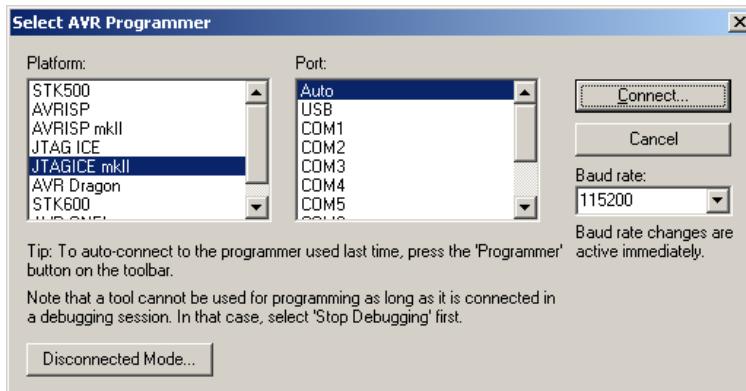
The release build can simply be started as follows:

- Start AVR Studio.
- Display the “Connect” dialog.

Figure 7-2. AVR Studio “Connect” Dialog

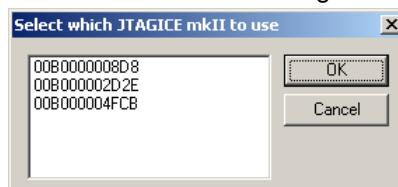
- Select the proper AVR Programmer (e.g. JTAGICE mkII) and press “Connect”.

Figure 7-3. AVR Studio “Select AVR Programmer” Dialog

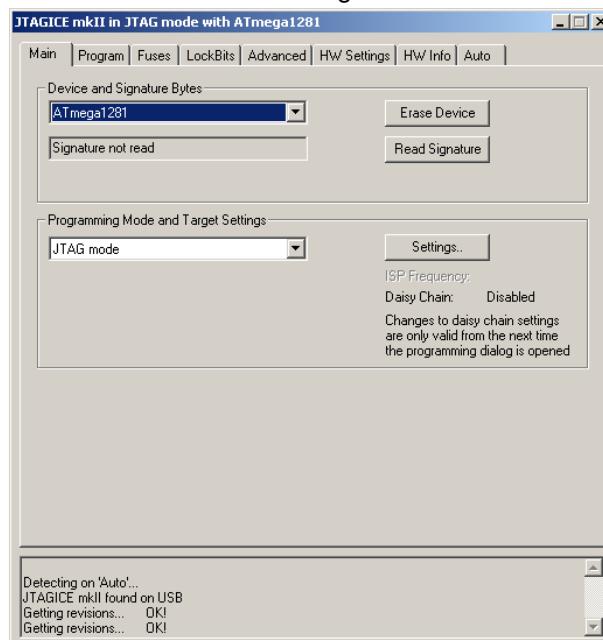


- In case more than one JTAGICE mkII are detected by AVR Studio select the proper Id.

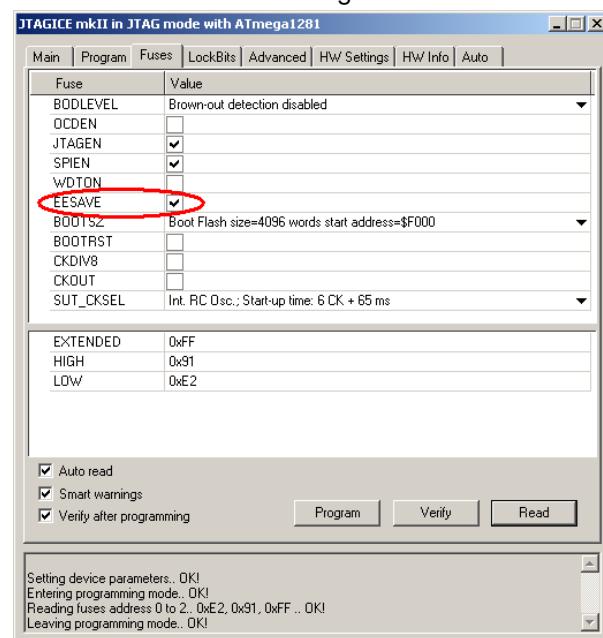
Figure 7-4. AVR Studio “Select JTAGICE mkII” Dialog



- After the JTAGICE selection the “JTAGICE mkII” dialog opens. Select the “Main” tab. Within this tab select the proper “Device and Signature Bytes” (e.g. “ATmega1281”) and the proper “Programming Mode and Target Settings” (e.g. “JTAG mode”).

Figure 7-5. AVR Studio “JTAGICE mkII” Dialog with “Main” Tab

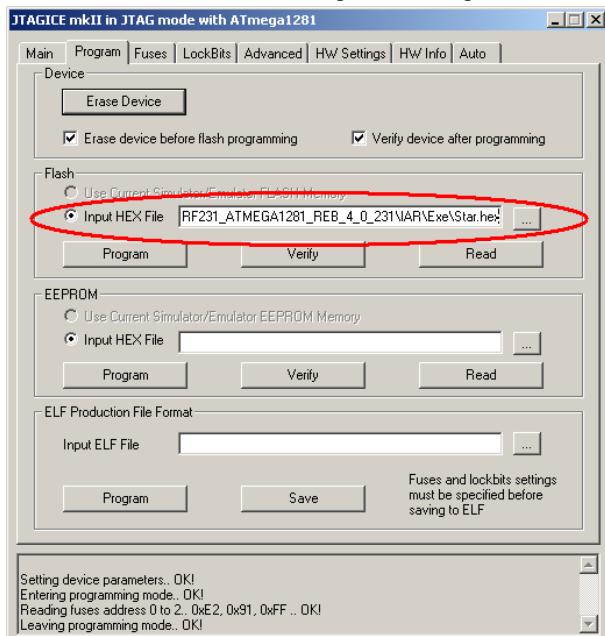
- Select the “Fuses” tab. Within this tab make sure that the EESAVE fuse is selected. This preserves the EEPROM through the Chip Erase cycle.

Figure 7-6. AVR Studio “JTAGICE mkII” Dialog with “Fuses” Tab

- Select the “Program” tab. Within this tab select the “Flash” section selection the proper Intel Hex File in either hex- or a90-format. Afterwards press button “Program”.

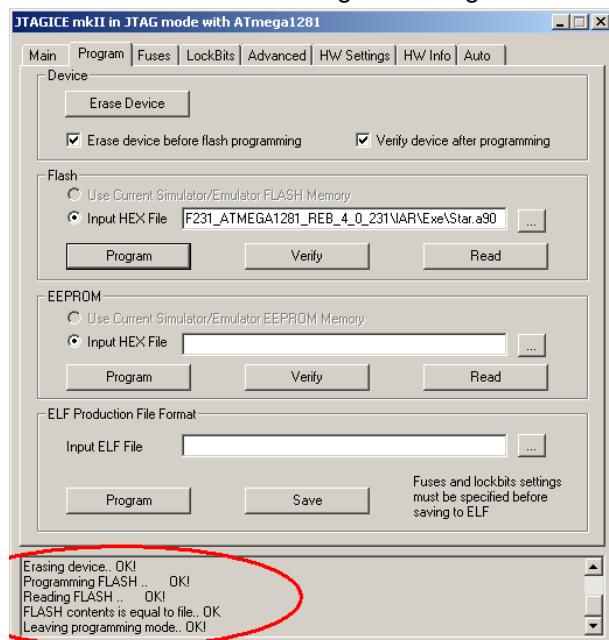


Figure 7-7. AVR Studio “JTAGICE mkII” Dialog with “Program” Tab



- The image will be downloaded onto the target and the status will be indicated.

Figure 7-8. AVR Studio “JTAGICE mkII” Dialog with “Program” Tab Download Status



- Close the “JTAGICE mkII” dialog. This starts the application. No further feedback within AVR Studio can be seen. In order to verify the proper functioning of the application check the corresponding outputs (LED status, Sniffer output, Terminal Window output, etc.).
- In case the application does not work as desired (and especially if the three standard LEDs are blinking fast) check first the proper IEEE address setting

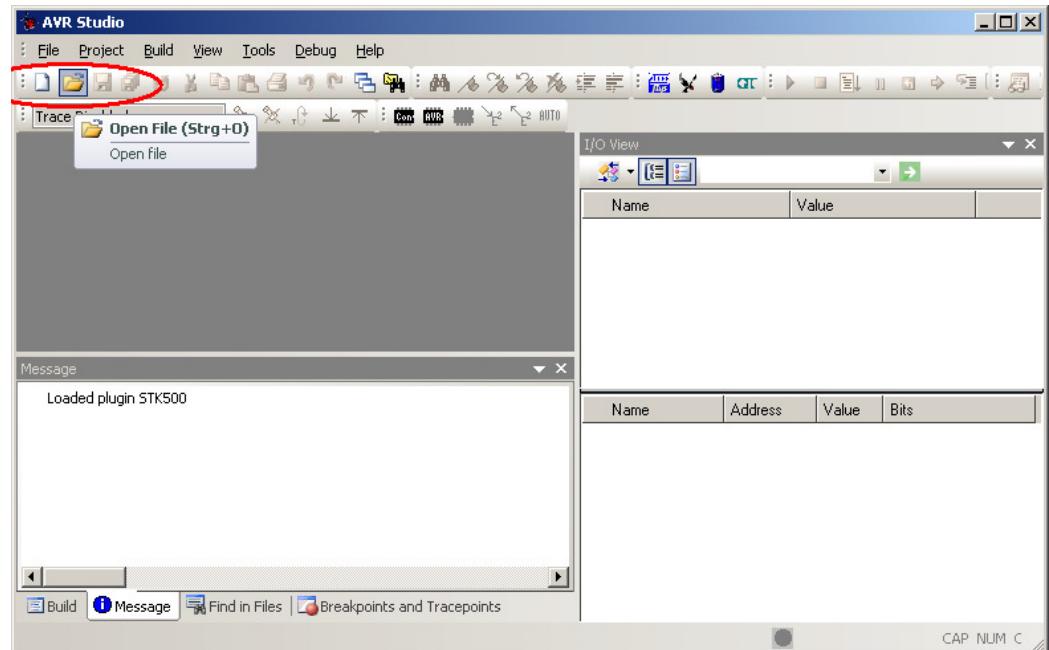
(see section 7.3.1 or section 7.3.2.2), or whether the proper binary has been selected (selection of transceiver type, microcontroller type, and board type).

7.3.2.2 Starting the Debug Build

The debug build can simply be started as follows:

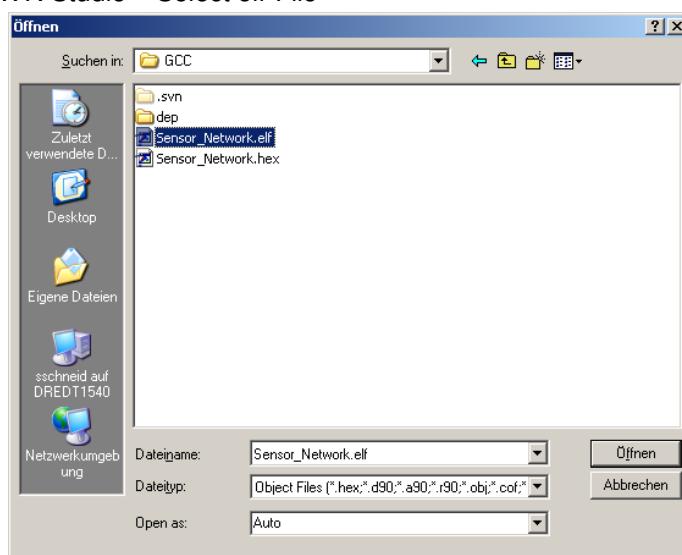
- Start AVR Studio.
- Display the “Open File” dialog.

Figure 7-9. AVR Studio “Open File” Dialog



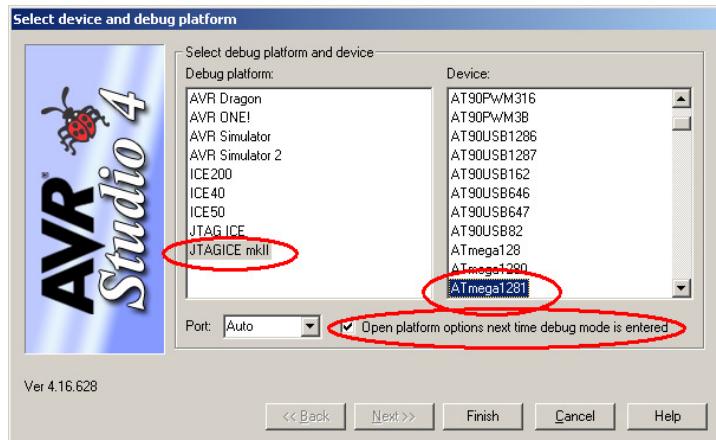
- Select the proper elf-file.

Figure 7-10. AVR Studio – Select elf-File



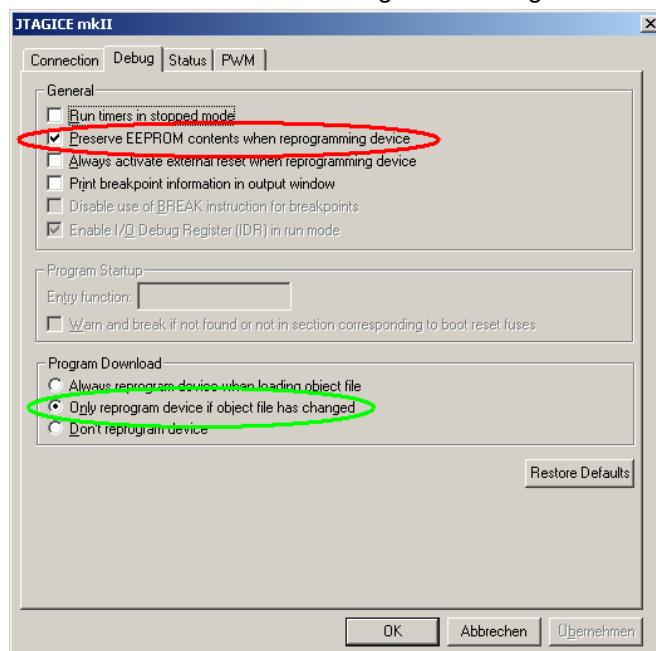
- In case a windows pops-up asking whether to save changes to a specific project press “Yes” and select the proper place to store the existing project file.
- The “Select device and debug platform” window opens. Select the proper “Debug platform” (e.g. JTAGICE mkII) and “Device” (e.g. ATmega1281). Make sure that “Open platform options next time debug mode is entered” is selected.

Figure 7-11. AVR Studio “Select debug platform and device” Window



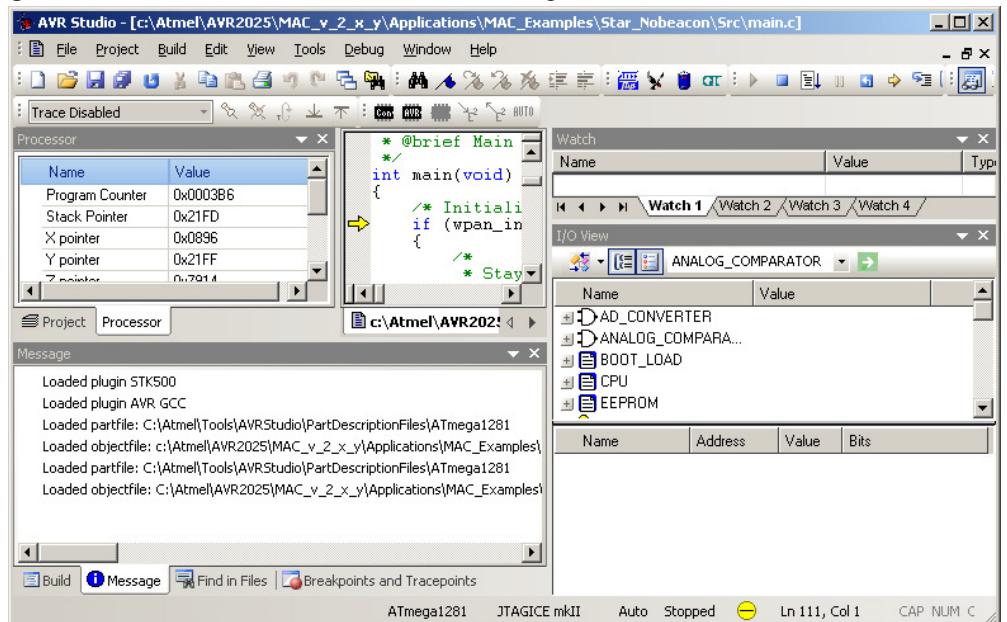
- Press “Finish”.
- In case there has been more than one JTAGICE mkII detected a window pops up asking to select the proper JTAGICE mkII to be used.
- The JTAGICE mkII Dialog opens. Select the “Debug” tab. Within the “Debug” tab make sure that “Preserve EEPROM contents when reprogramming device” is selected.

Figure 7-12. AVR Studio “JTAGICE mkII” Dialog with “Debug” Tab



- Press “Ok”. The image will be downloaded and the download status is indicated within AVR Studio.
- Now AVR Studio looks like below.

Figure 7-13. AVR Studio after successful Debug Build Download



- In case the IEEE address of the node is stored in the internal EEPROM of the microcontroller perform as follows:
 - Check whether a valid IEEE address (different from 0xFFFFFFFFFFFFFFF and 0x0000000000000000) is stored in the internal EEPROM. Select menu “View” item “Memory”.
 - If the IEEE address is not set properly, add the correct IEEE to the first 8 octets of the EEPROM (see section 7.3.1).
- Start the application by pressing “F5” or clicking the “Run” button.

7.3.3 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.

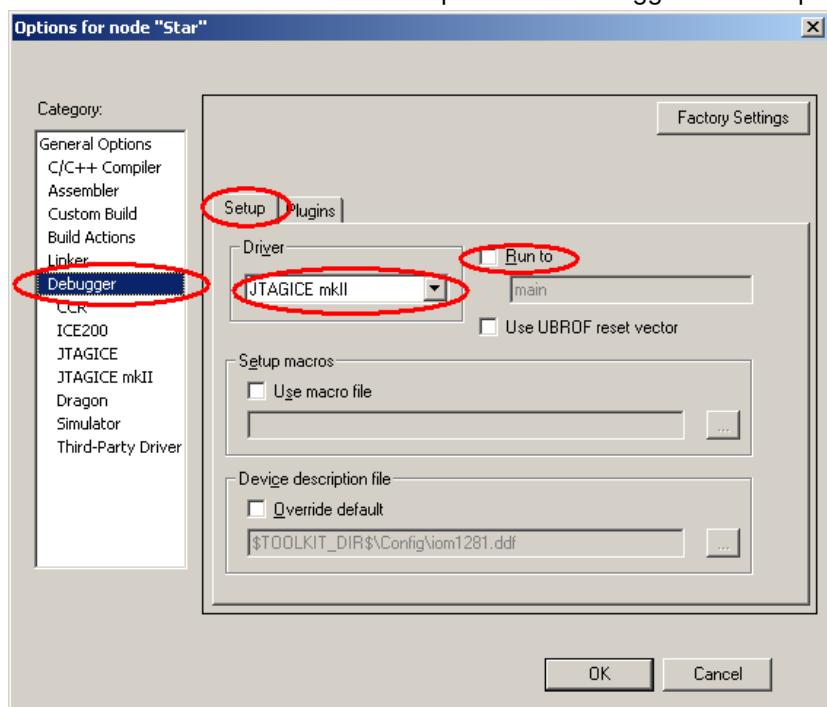
7.3.3.1 Starting the Release Build

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 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.

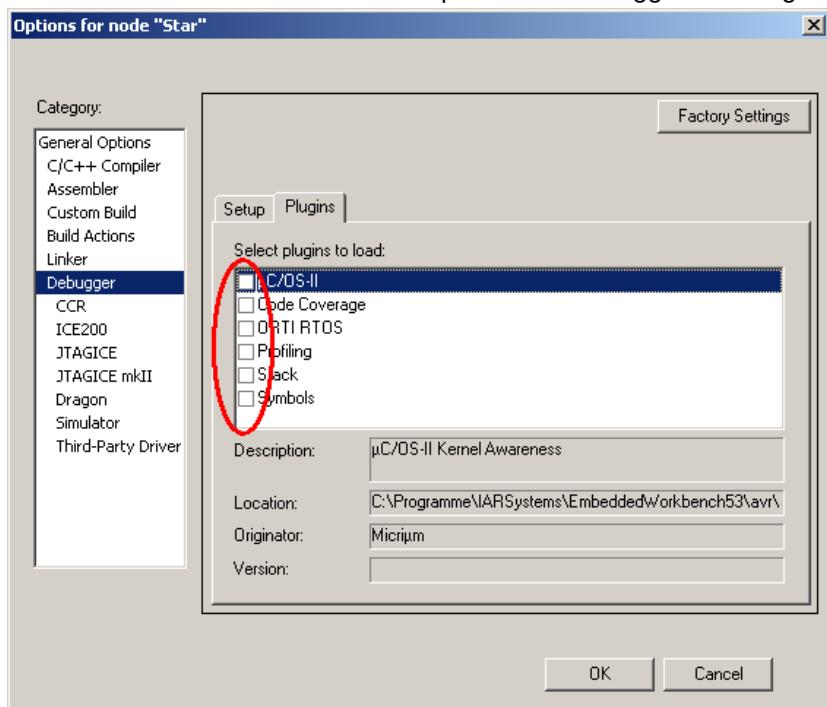


Figure 7-14. IAR Embedded Workbench – “Options” -> “Debugger” -> “Setup”



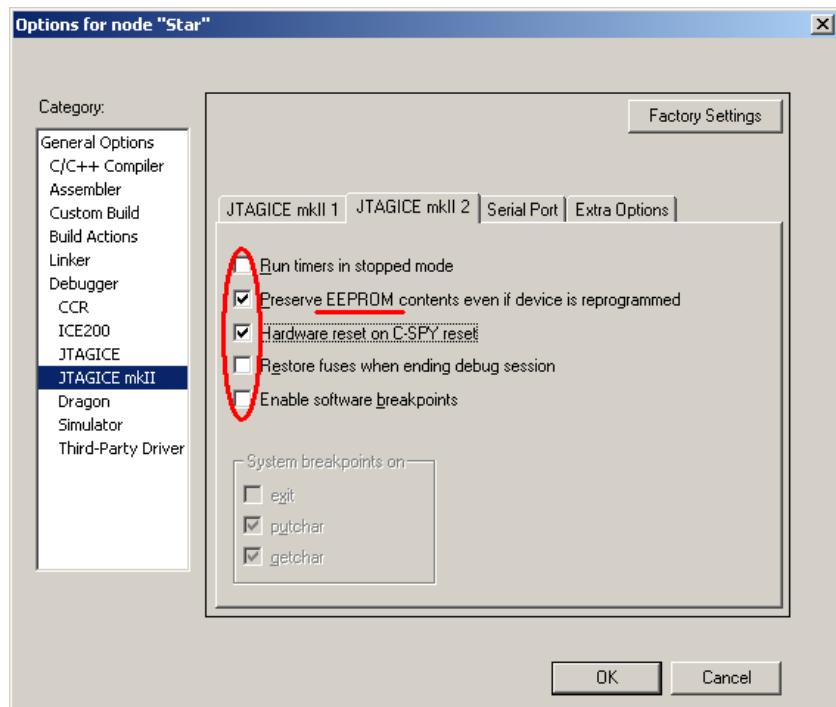
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and deselect “Run to (main)”.
- Change to the “Plugins” tab and deselect all entries.

Figure 7-15. 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 7-16. IAR Embedded Workbench – “Options” -> “JTAGICE mkII” -> “JTAGICE mkII 2”



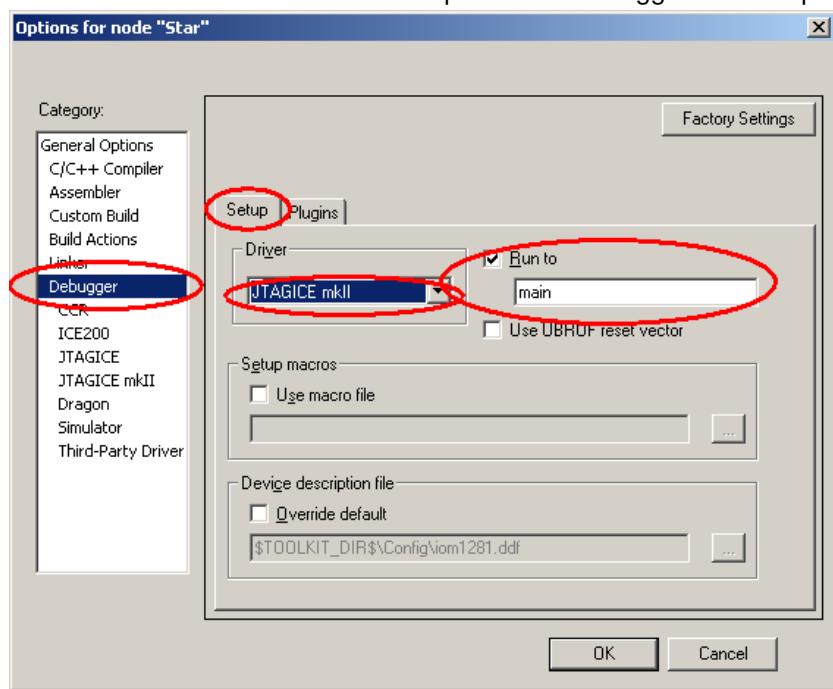
- Within the “JTAGICE mkII 2” tab check the items as shown above. Especially make sure that the EEPROM will be preserved if the device is reprogrammed.
- Press “Ok”.
- Press the button “Download and Debug”.

7.3.3.2 Starting the Debug Build

The debug build can simply be started as follows:

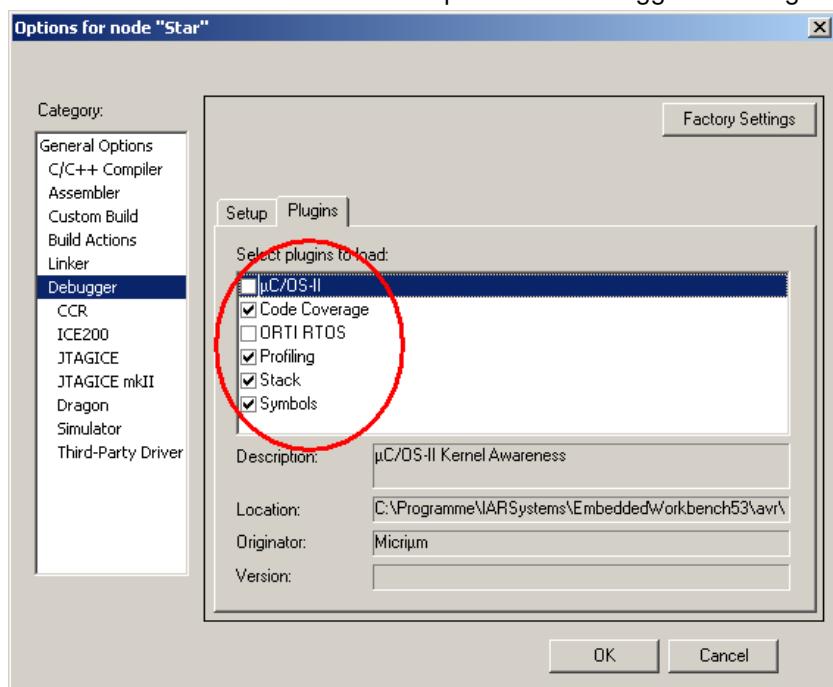
- Make sure that only the JTAGICE of the node where the current build shall be downloaded to is switched on. Switch off all other JTAGICE.
- Open IAR Workbench by double clicking the desired eww-file.
- Select the “Debug” Workspace.
- Open the “Options” window for the “Debug” Workspace. Select the “Debugger”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

Figure 7-17. IAR Embedded Workbench – “Options” -> “Debugger” -> “Setup”



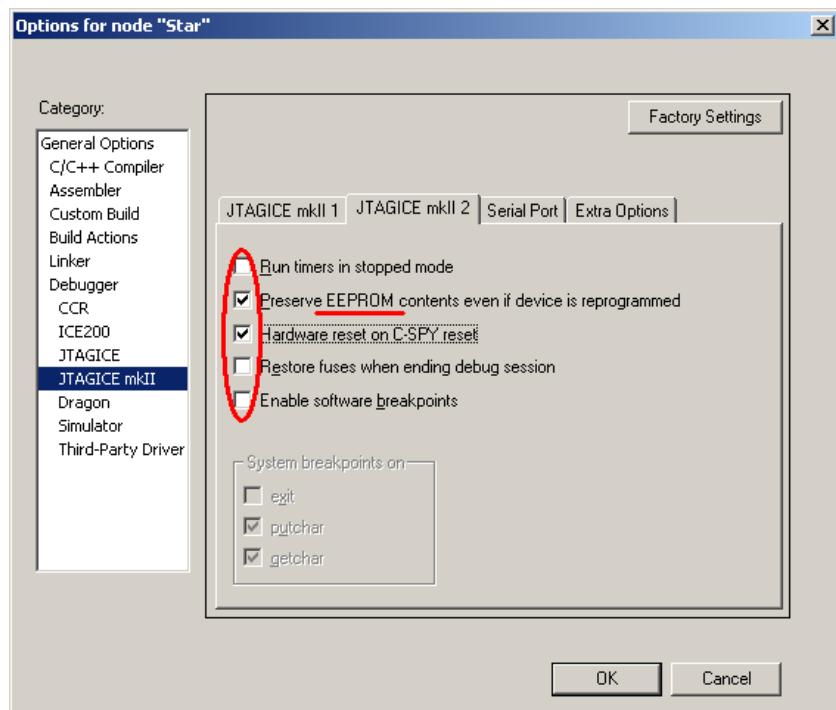
- Within the “Setup” tab select the “Driver” “JTAGICE mkII” and select “Run to (main)” (this is different to “Release” build).
- Change to the “Plugins” tab and select the entries as below.

Figure 7-18. IAR Embedded Workbench – “Options” -> “Debugger” -> “Plugins”



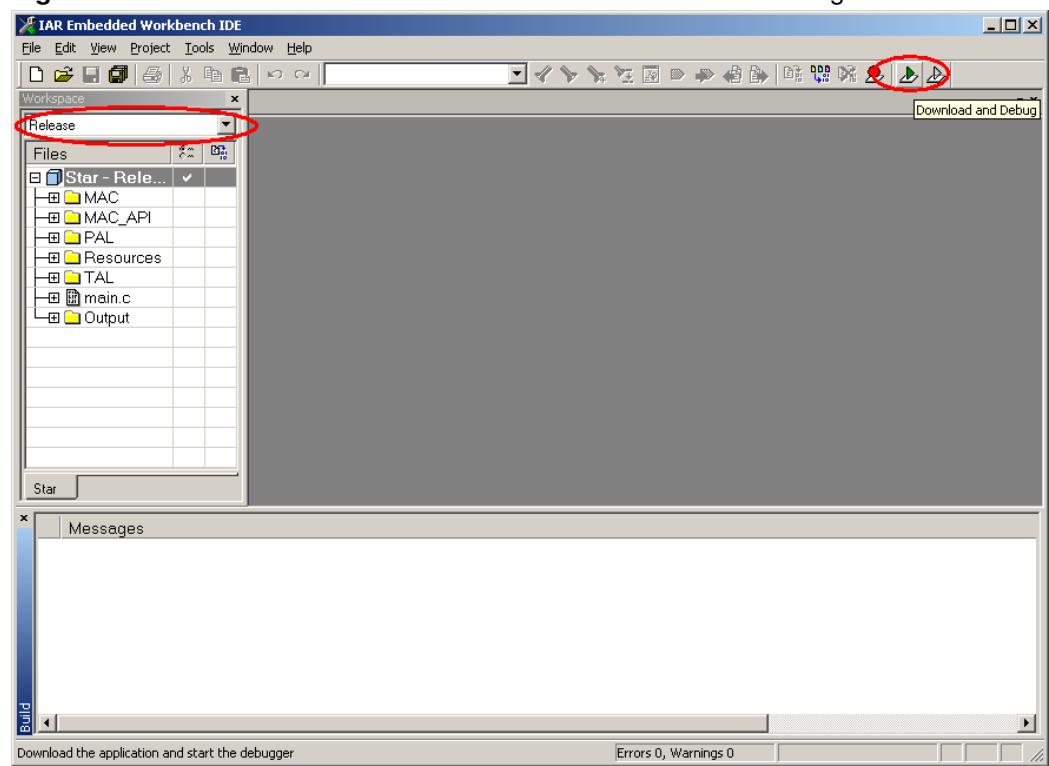
- Within the “Options” window for the “Release” Workspace select now the “JTAGICE mkII” window and within this window select the “JTAGICE mkII 2” tab.

Figure 7-19. IAR Embedded Workbench – “Options” -> “JTAGICE mkII” -> “JTAGICE mkII 2”

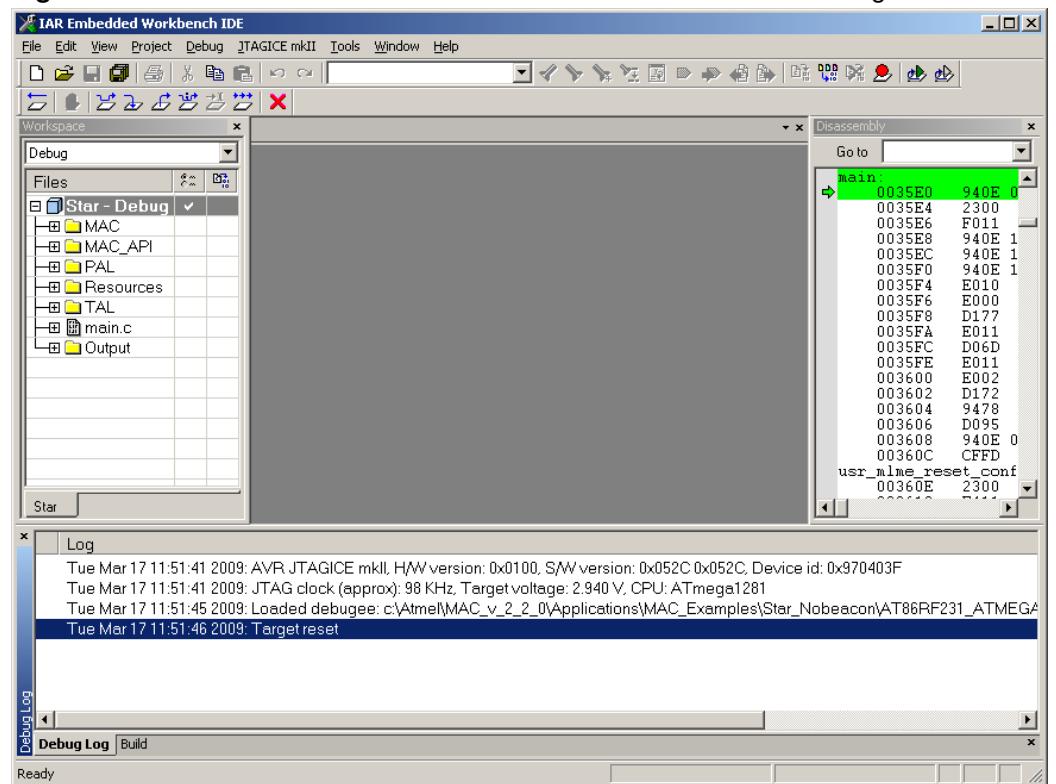


- Within the “JTAGICE mkII 2” tab check the items as shown above. Especially make sure that the EEPROM will be preserved if the device is reprogrammed.
- Press “Ok”.
- Press the button “Download and Debug”.

Figure 7-20. IAR Embedded Workbench - Start “Download and Debug”



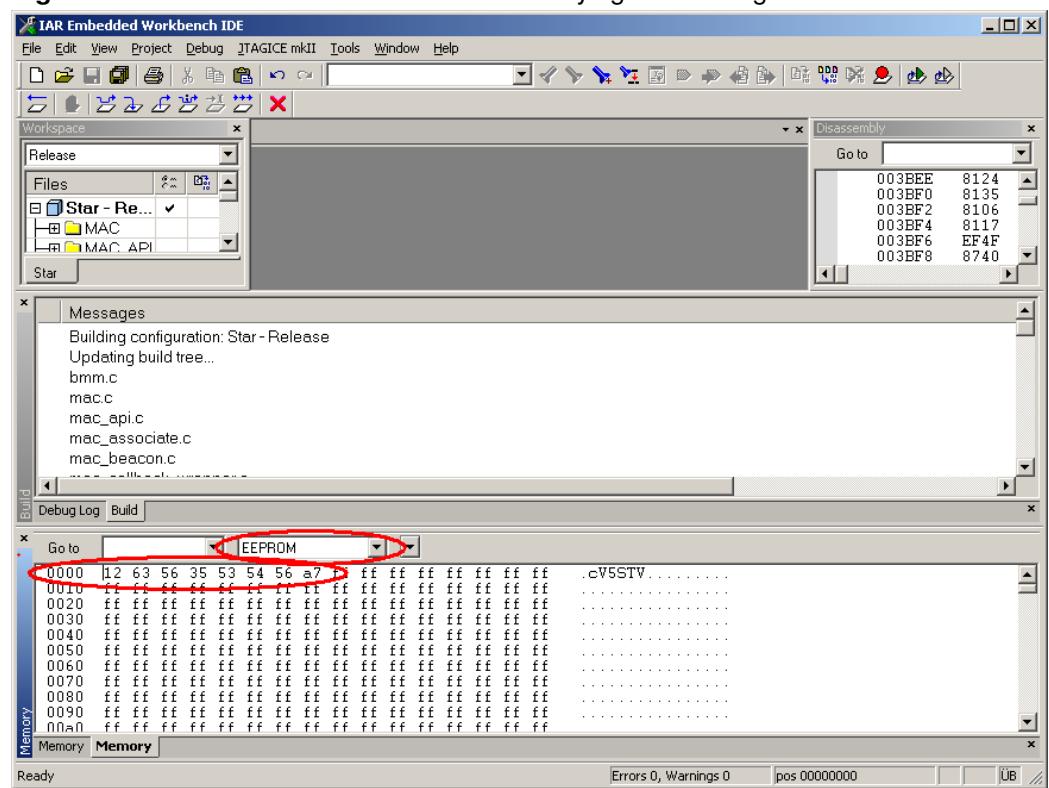
- A Window will pop-up indicating the status of the download of the binary.
- After successful download the IAR Workbench will looks as shown below. After the download of the debug build the main function is displayed since all debug and source code information is included in the build.

Figure 7-21. 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:
 - Check whether a valid IEEE address (different from 0xFFFFFFFFFFFFFF and 0x0000000000000000) is stored in the internal EEPROM. Select menu “View” item “Memory”. In this window select “EEPROM”.
 - If the IEEE address is not set properly, add the correct IEEE to the first 8 octets of the EEPROM.



Figure 7-22. IAR Embedded Workbench – Verifying and Setting of IEEE Address



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

8 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 can categorized in three main groups (being located in the corresponding subdirectories of directory Applications):

- MAC Examples
- TAL Examples
- STB Examples

These applications will be explained in more detail in the subsequent sections (see section 8.2). If the example application makes use of the UART interface, the UART is set to 9600 (8,N,1).

8.1 Walking through a Basic Application

This section describes one basic example application provided with this MAC release (see section 8.2.1.1) 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. If the Coordinator with the default PAN ID is located on the default channel (i.e. channel 20), the Device will initiate the association procedure. If the association is also successful, the Device periodically (i.e. every 2 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.

8.1.1 Implementation of the Coordinator

The source code of the coordinator is located in

`Applications/MAC_Examples/App_1_Nobeacon/Coordinator/Src/main.c`

and the header file in

`Applications/MAC_Examples/App_1_Nobeacon/Coordinator/Inc/app_config.h`

Platform related project / Makefiles files for AVR GCC, AVR Studio, and IAR Workbench are located in the corresponding subdirectories

`Applications/MAC_Examples/App_1_Nobeacon/Coordinator/<platform>`

The example application can be opened using the AVR Studio, the IAR EWW or any other editor. To open the example application project from the AVR Studio select the file





"Coordinator.aps" or from the IAR EWW select the file "Coordinator.eww". If the AVR 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 (wpan_init() != SUCCESS)
{
    /*
     * Stay here; we need a valid IEEE address.
     * Check kit documentation how to create an IEEE address
     * and store it to the EEPROM
     */
    pal_alert();
}
```

Calibrate MCU's RC oscillator:

```
pal_calibrate_rc_osc();
```

Initialize LEDs:

```
pal_led_init();
pal_led(LED_0, LED_ON);      // indicating application is started
pal_led(LED_1, LED_OFF);    // indicating network is started
pal_led(LED_2, LED_OFF);    // indicating data reception
```

Enable the global interrupts:

```
pal_global_irq_enable();
```

Initiate a reset of the MAC layer:

```
wpn_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 set the short address of the coordinator or with a new reset request.

```
void usr_mlme_reset_conf(uint8_t status)
{
    if (status == MAC_SUCCESS)
    {
        /*
         * Set the short address of this node.
         */
        uint8_t short_addr[2];

        short_addr[0] = (uint8_t)COORD_SHORT_ADDR;
```

```

        short_addr[1] = (uint8_t)(COORD_SHORT_ADDR >> 8);
        wpan_mlme_set_req(macShortAddress, short_addr);
    }
    else
    {
        // something went wrong; restart
        wpan_mlme_reset_req(true);
    }
}

```

The request to set the short address of the coordinator 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))
    {
        /**
        /* Allow other devices to associate to this coordinator.
        */
        uint8_t association_permit = true;

        wpan_mlme_set_req(macAssociationPermit,
                           &association_permit);
    }
    else if ((status == MAC_SUCCESS) &&
              (PIBAttribute == macAssociationPermit))
    {
        /**
        /* Initialize an active scan over all channels to determine
        /* which channel to use.
        */
        wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,
                           SCAN_ALL_CHANNELS,
                           SCAN_DURATION_COORDINATOR);
    }
    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)
{
    /*
     * We are not interested in the actual scan result,
     * because we start our network on the pre-defined channel
     * anyway.
     * Start a nonbeacon-enabled network
     */
    wpan_mlme_start_req(DEFAULT_PAN_ID,
                        DEFAULT_CHANNEL,
                        DEFAULT_CHANNEL_PAGE,
                        15, 15,
                        true, false, false);
}
```

The wpan_mlme_start_req() will be confirmed with usr_mlme_start_conf().

```
void usr_mlme_start_conf(uint8_t status)
{
    if (status == MAC_SUCCESS)
    {
        /*
         * Network is established.
         * Waiting for association indication from a device.
         */
        pal_led(LED_1, LED_ON);
    }
    else
    {
        // something went wrong; restart
        wpan_mlme_reset_req(true);
    }
}
```

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)
{
    /*
     * Any device is allowed to join the network.
     *
     * Get the next available short address for this device.
     */
    uint16_t associate_short_addr = macShortAddress_def;

    if (get_next_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);
    }
}

```

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 (i.e. about every 2 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.

8.1.2 Implementation of the Device

The source code of the device is located in `Applications/MAC_Examples/App_1_Nobeacon/Device/Src/main.c` and the header file in

`Applications/MAC_Examples/App_1_Nobeacon/Device/Inc/app_config.h`

Platform related project/Makefiles files for AVR GCC, AVR Studio, and IAR Workbench are located in the corresponding subdirectories

`Applications/MAC_Examples/App_1_Nobeacon/Device/<platform>`

The example application can be opened using the AVR Studio, the IAR EWW or any other editor. To open the example application project from the AVR Studio select the file "Device.aps" or from the IAR EWW select the file "Device.eww". If the AVR 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 device performs the following steps:

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

```
if (wpan_init() != SUCCESS)
{
    /*
     * Stay here; we need a valid IEEE address.
     * Check kit documentation how to create an IEEE address
     * and store it to the EEPROM
     */
    pal_alert();
}
```

Calibrate MCU's RC oscillator:

```
pal_calibrate_rc_osc();
```

Initialize LEDs:

```
pal_led_init();
pal_led(LED_0, LED_ON);           // indicating application is started
pal_led(LED_1, LED_OFF);         // indicating network is started
pal_led(LED_2, LED_OFF);         // indicating data reception
```

Enable the global interrupts:

```
pal_global_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 scan all channels for the coordinator or with a new reset request.

```
void usr_mlme_reset_conf(uint8_t status)
{
    if (status == MAC_SUCCESS)
    {
        /*
         * Initiate an active scan over all channels to determine
         * which channel is used by the coordinator.
         */
        wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,
                           SCAN_ALL_CHANNELS,
                           SCAN_DURATION_SHORT,
                           DEFAULT_CHANNEL_PAGE);

        // Indicate network scanning by a LED flashing
    }
}
```

```

        pal_timer_start(TIMER_LED_OFF,
                        500000,
                        TIMEOUT_RELATIVE,
                        (void *)network_search_indication_cb,
                        NULL);
    }
else
{
    // 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. The `usr_mlme_scan_conf()` function handles three cases:

- A coordinator was found.
- No coordinator was found.

```

coordinator = (wpan_pandescriptor_t *)ResultList;
for (i = 0; i < ResultListSize; i++)
{
    /*
     * Check if the PAN descriptor belongs to our coordinator.
     * Check if coordinator allows association.
     */
    if ((coordinator->LogicalChannel == DEFAULT_CHANNEL) &&
        (coordinator->ChannelPage == DEFAULT_CHANNEL_PAGE) &&
        (coordinator->CoordAddrSpec.PANId == DEFAULT_PAN_ID) &&
        ((coordinator->SuperframeSpec & 0x8000) == 0x8000))
        // association permit
    {
        // Store the coordinator's address
        if (coordinator->CoordAddrSpec.AddrMode ==
            WPAN_ADDRMODE_SHORT)
        {
            coord_addr.short_addr =
                (uint16_t)(coordinator->CoordAddrSpec.Addr);
        }
        else if (coordinator->CoordAddrSpec.AddrMode ==
                  WPAN_ADDRMODE_LONG)
        {
            coord_addr.ieee_addr = coordinator->CoordAddrSpec.Addr;
        }
        else
        {
            // Something went wrong; restart
            wpan_mlme_reset_req(true);
            return;
        }
    }
}

```





```
    }

    /*
     * Associate to our coordinator
     */
    wpan_mlme_associate_req(coordinator->LogicalChannel,
                            coordinator->ChannelPage,
                            &(coordinator->CoordAddrSpec),
                            WPAN_CAP_ALLOCADDRESS);

    return;
}
```

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

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)
    {
        // Store own short address
        own_short_addr = AssocShortAddress;

        // Stop timer used for search indication
        // (same as used for data transmission)
        pal_timer_stop(TIMER_LED_OFF);
        pal_led(LED_1, LED_ON);

        // Start a timer that sends some data to the coordinator
        // every 2 seconds.
        pal_timer_start(TIMER_TX_DATA,
                        DATA_TX_PERIOD,
                        TIMEOUT_RELATIVE,
                        (void *)app_timer_cb,
                        NULL);
    }
    else
    {
        // Something went wrong; restart
        wpan_mlme_reset_req(true);
    }
}
```

If MAC_SUCCESS is returned, the coordinator has assigned a short address to this device and the application is ready for data transmissions. An application timer is started with 2 seconds timeout. If the timer triggers, the following callback function is executed. It initiates the data transmission and restarts the timer again.

```
static void app_timer_cb(void *parameter)
{
    /*
     * Send some data and restart timer.
     */
    uint8_t src_addr_mode;
    wpan_addr_spec_t dst_addr;
    uint8_t payload;
    static uint8_t msduHandle = 0;

    src_addr_mode = WPAN_ADDRMODE_SHORT;

    dst_addr.AddrMode = WPAN_ADDRMODE_SHORT;
    dst_addr.PANId = DEFAULT_PAN_ID;
    dst_addr.Addr = coord_addr.short_addr;

    payload = (uint8_t)rand(); // any dummy data
    msduHandle++;           // increment handle
    wpan_mcps_data_req(src_addr_mode,
                        &dst_addr,
                        1,
                        &payload,
                        msduHandle,
                        WPAN_TXOPT_ACK);

    pal_timer_start(TIMER_TX_DATA,
                    DATA_TX_PERIOD,
                    TIMEOUT_RELATIVE,
                    (void *)app_timer_cb,
                    NULL);
}
```

The `usr_mcps_data_conf()` callback function is a stub indicating the status of the data transmission. It can be adapted to the user's needs.

```
void usr_mcps_data_conf(uint8_t msduHandle,
                        uint8_t status,
                        uint32_t Timestamp)
{
    if (status == MAC_SUCCESS)
    {
        /*
         * Dummy data has been transmitted successfully.
         * Application code could be added here ...
         */
    }
}
```





```
    pal_led(LED_2, LED_ON);
    // Start a timer switching off the LED
    pal_timer_start(TIMER_LED_OFF,
                    500000,
                    TIMEOUT_RELATIVE,
                    (void *)data_exchange_led_off_cb,
                    NULL);
}
}
```

8.2 Provided Examples Applications

8.2.1 MAC Examples

8.2.1.1 App_1_Nobeacon

8.2.1.1.1 Introduction

The basic MAC Example 1 deploys a nonbeacon-enabled network consisting of PAN Coordinator and Device. The application shows how basic MAC features can be utilized within an application (Scanning for network, starting a network, association procedure, data transmission from Device to PAN Coordinator).

This application works very similar to the MAC Example Star_Nobeacon (see section 8.2.1.7) but different source code implementations are provided for both types of devices. A node is either a PAN Coordinator or a Device.

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

The application and all required build files are located in directory Applications/MAC_Examples/App_1_Nobeacon. The source code of the application can be found in the subdirectories Coordinator/Src or Device/Src.

8.2.1.1.2 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.

8.2.1.1.3 Implementation

The PAN Coordinator starts a PAN at channel DEFAULT_CHANNEL 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 2 seconds to transmit a random payload to the PAN Coordinator. While the device is idle (when the timer is running) the transceiver enters sleep in order to save as much power as possible.

8.2.1.1.4 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 2 devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.
- The current implementation only provides direct data transmission from device to coordinator. In order to save as much power as possible, the device periodically enters sleep mode between its data transmissions. During these sleeping periods the receiver of the device is not enabled. It is therefore not possible to simply extend the application so that direct data transmission is performed in the other direction (from coordinator to device). In case the data transmission from coordinator to device is required more changes within the application are required. For more information please see section 4.3. An example using indirect data transmission to the Device can be found in 8.2.1.2. Another example implementing the feature MAC_RX_ENABLE_SUPPORT can be found in 8.2.1.3 (MAC Example Basic_Sensor_Network).

8.2.1.2 App_2_Nobeacon_Indirect_Traffic

8.2.1.2.1 Introduction

The basic MAC Example 2 (Indirect Traffic) deploys a nonbeacon-enabled network consisting of PAN Coordinator and Device utilizing the mechanism of indirect data transfer between Coordinator and Device. In terms of setting up the basic network (network start, scanning, association) the application operates similar to the basic MAC Example 1 (see 8.2.1.1).

While in MAC Example 1 always the Device is transmitting data frames to the Coordinator directly, in this example the Coordinator wants to send data to the Device. Since a Device in a nonbeacon-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 4.2.1/4.2.2. For power management and indirect transmission see section 5.2.4.

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

The application and all required build files are located in directory Applications/MAC_Examples/App_2_Nobeacon_Indirect_Traffic. The source code of the application can be found in the subdirectories Coordinator/Src or Device/Src. The common source code for handling Serial I/O can be found in the subdirectory Src.

8.2.1.2.2 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 see the output of the application please start a terminal application on your host system.

8.2.1.2.3 Implementation

The PAN Coordinator starts a PAN at channel DEFAULT_CHANNEL 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 frame 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.

8.2.1.2.4 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 only 2 devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.

8.2.1.3 App_3_App_3_Beacon_Payload

8.2.1.3.1 Introduction

The basic MAC Example 3 Beacon Payload 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 synchronization with the coordinator and utilization of beacon payload by the coordinator.

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

The application and all required build files are located in directory Applications/MAC_Examples/App_3_App_3_Beacon_Payload. The source code of the application can be found in the subdirectories Coordinator/Src or Device/Src.

8.2.1.3.2 Requirements

The application requires (up to three) LEDs on the board in order to indicate the proper working status. Also 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 see the output of the application please start a terminal application on your host system.

8.2.1.3.3 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 mean 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 loses synchronization with its parent, it initiates a new synchronization attempt.

8.2.1.3.4 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 100 devices are allowed to associate to the PAN Coordinator. This can be easily extended by increasing the define MAX_NUMBER_OF_DEVICES.

8.2.1.4 App_4_Beacon_Broadcast_Data

8.2.1.4.1 Introduction

The basic MAC Example 3 Beacon Broadcast 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 (i.e. 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).

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

The application and all required build files are located in directory Applications/MAC_Examples/App_4_Beacon_Broadcast_Data. The source code of the application can be found in the subdirectories Coordinator/Src or Device/Src.

8.2.1.4.2 Requirements

The application requires (up to three) LEDs on the board in order to indicate the proper working status. Also 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 see the output of the application please start a terminal application on your host system.

8.2.1.4.3 Implementation

The coordinator in this application creates a beacon-enabled network. Periodically the application of the coordinator try 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.



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. 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.

8.2.1.4.4 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.

8.2.1.5 Basic_Sensor_Network

8.2.1.5.1 Introduction

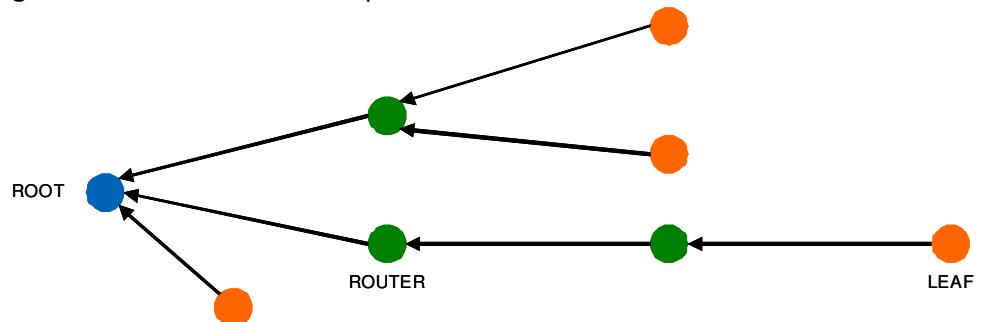
The application *Basic_Sensor_Network* implements a basic sensor network. The network consists of devices being sensor nodes (called LEAF), nodes (called ROUTER) that provide router functionality and a central node collecting all data (called ROOT).

The network route is a static route. The network and its routing path is configured during network setup. A tree topology is created from ROOT to LEAF nodes during the network setup. The network uses a pre-defined PAN Id (0xBEEF) and channel (1 or 20).

The sensor nodes gather their battery status and another sensor value, like temperature value. Every 10s the node transmits the gather data to its parent. All data is routed/forwarded to the ROOT node where it is printed via UART/USB to a terminal program.

An example network topology is shown by Figure 8-1.

Figure 8-1. Tree Network Example



8.2.1.5.2 Implementation

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

This application uses the user build configuration feature described in section 6.2.2. In order to achieve the proper functionality in conjunction with minimal footprint, the actually supported MAC features (i.e. MAC primitives) are defined in a corresponding header file *mac_user_build_config.h* in subdirectory Inc of this application.

In order to use this header file, the build switch "MAC_USER_BUILD_CONFIG" needs to be set in the corresponding Makefiles or IAR project files.

The application and all required build files are located in directory Applications/MAC_Examples/Basic_Sensor_Network. The source code of the application can be found in the subdirectory Src.

8.2.1.5.3 Network Setup

The static tree network topology is established by first starting the ROOT node and after that connecting one or more other nodes (ROUTER or LEAF).

The node type (ROOT, ROUTER, or LEAF) is defined during power-up using the push button. If the push button is pressed during power-up, the node is operated as a ROOT node or as a ROUTER node.

To determine that a node acts as a ROOT or ROUTER node, hold the push button pressed during power-up. If the node does not receive any broadcast messages from another node. It configures itself to a ROOT node after 10s and switches LED_0 on. If the node receives a broadcast message within the first 10s after power-up, the node stores its parent address and operates as ROUTER node.

Once the ROOT node has started the network, ROUTER nodes or LEAF nodes can be connected to the ROOT node.

Connecting a node to a parent, e.g. a ROUTER node to the ROOT node, during application start broadcast messages need to be received by the child node containing the parent's address. Broadcast messages can be sent by a ROOT or ROUTER node by pressing the button.

The next node that might be added to the network could be a LEAF node that gets connected to the ROUTER.

A node becomes a LEAF node, if the push button is not pressed during power-up. To connect a LEAF node to a parent, the parent node needs to send broadcast messages (by pressing the push button at the parent node) within the first 10s after start-up of the LEAF node.

Once a LEAF node is connected to a parent, every TX_INTERVAL_S seconds (by default every 10s) the node transmits the sensor data to its parent. An LED is switched on shortly indicating the active period.

Also ROUTER nodes can be connected to already started ROUTER nodes.

A ROUTER node forwards all received data to its parent. The ROOT node prints the received data to the UART/USB output. The ROOT and ROUTER nodes should be mains-powered devices.

8.2.1.5.4 Configuration

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.





8.2.1.6 Promiscuous_Mode_Demo

8.2.1.6.1 Introduction

The application Promiscuous_Mode_Demo provides a simple network diagnostic tool based on the promiscuous mode as described in IEEE 802.15.4-2006 (section 7.5.6.5 Promiscuous Mode). During the build process the switch PROMISCUOUS_MODE is enabled.

This tool uses MAC-API as interface to the stack.

The application and all required build files are located in directory Applications/MAC_Examples/Promiscuous_Mode_Demo. The source code of the application can be found in the subdirectory Src.

8.2.1.6.2 Requirements

This application requires a serial connection for proper demonstration of the promiscuous mode. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to see the output of the application please start a terminal application on your host system.

8.2.1.6.3 Implementation

The application works as described subsequently.

- The node performs a reset of the stack (wpan_mlme_reset_req())
- After successful processing of the reset, the default channel and channel page are set within the application source code depending on the transceivers.
 - 2.4 GHz: Channel 20
 - 900MHz: Channel 1
- Afterwards the promiscuous mode is switched on immediately.
- In the serial terminal on the host system printouts indicate the detected hardware (radio and microcontroller), set channel and channel page, and the status of promiscuous mode.
- Now the application will print each received frame (that has a valid CRC) on the terminal. In order to limit the load on the serial connections, currently only the following items are displayed:
 - Number of received frame
 - Type of frame
 - Content of frame (received octets in hexadecimal format), this includes the MAC Header (MHR) of the original frame and the payload within this frame.
- In the application itself the received frame is already parsed so that the variable app_parse_data contains all information for the MAC header (i.e. addressing information) of the received frame already in a structure of type prom_mode_payload_t.

Figure 8-2. Promiscuous_Mode_Demo Terminal Program Snapshot

```

*****
Promiscuous mode demo application (AT86RF230B / ATxmega128A1)

Current channel: 24
Current channel page: 0
Promiscuous mode is on

No. 1 Cmd: 03 08 22 FF FF FF FF 07
No. 2 Cmd: 03 08 14 FF FF FF FF 07
No. 3 Beacon: 00 80 7C 78 56 00 00 FF CF 00 00
No. 4 Cmd: 23 C8 15 78 56 00 00 FF FF 12 13 14 15 16 17 18 19 01 82
No. 5 Ack: 02 00 15
No. 6 Cmd: 63 C8 16 78 56 00 00 12 13 14 15 16 17 18 19 04
No. 7 Ack: 12 00 16
No. 8 Cmd: 63 CC 23 78 56 12 13 14 15 16 17 18 19 A1 A2 A3 A4 A5 A6 A7 A8 02 FE
CA 00
No. 9 Ack: 02 00 23
No. 10 Cmd: 63 88 17 78 56 00 00 FE CA 04
No. 11 Ack: 12 00 17
No. 12 Data: 61 88 24 78 56 FE CA 00 00 AA BB CC DD
No. 13 Ack: 02 00 24

```

Verbunden 00:03:02 Auto-Erkenn. 9600 8-N-1 RF GROSS NUM Aufzeichnen Druckerecho

8.2.1.6.4 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.
- The processing of each received frame within the application and the corresponding serial output is the limiting factor of this application in terms of throughput.
- The actual processing and output implemented within the application for each received frame should be adapted to the end user's requirements.

8.2.1.7 Star_Nobeacon

8.2.1.7.1 Introduction

The application *Star_Nobeacon* 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 works very similar to the MAC Example *App_1_Nobeacon* (see section 8.2.1.1) but the same source code is used for both types of devices. The type of device is detected by the firmware during run-time.

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

The application and all required build files are located in directory Applications/MAC_Examples/Star_Nobeacon. The source code of the application can be found in the subdirectory Src.



8.2.1.7.2 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.

8.2.1.7.3 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 and indicates a successful association by switching on LED 1.
- Every two seconds this node sends out a dummy data packet. If the packet is acknowledged by the other node the LED 2 is flashing.

8.2.1.7.4 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.
- The current implementation only provides direct data transmission from device to coordinator. In order to save as much power as possible, the device periodically enters sleep mode between its data transmissions. During these sleeping periods the receiver of the device is not enabled. It is therefore not possible to simply extend the application so that direct data transmission is performed in the other direction (from coordinator to device). In case the data transmission from coordinator to device is required more changes within the application are required. For more information please see section 4.3. An example where this scenario has been implemented by means of using the feature MAC_RX_ENABLE_SUPPORT can be found in 8.2.1.3 (MAC Example Basic_Sensor_Network).

8.2.2 TAL Examples

8.2.2.1 Performance_Test

8.2.2.1.1 Introduction

The TAL example `Performance_Test` is a terminal-based application. It demonstrates transceiver performance including high data rate modes. Configuration parameters are set using a terminal program, like

- Channel
- Channel page
- Transmit power
- Number of frames sent during a test
- Frame length (PSDU)
- ACK request enable/disable
- CSMA enable/disable
- Frame retry enable/disable

This allows to measure throughput performance and PER testing with different radios, data rates and transmits powers. In addition to that the following features are supported:

- ED scan over all channels
- Continuous wave (CW) transmission
- Promiscuous mode

This example application uses TAL API as interface to the stack.

The application and all required build files are located in directory `Applications/TAL_Examples/ Performance_Test`. The source code of the application can be found in the subdirectory `Src`.

8.2.2.1.2 Requirements

This application requires a serial connection for controlling the application and displaying the results. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to see the output of the application please start a terminal application on your host system.

8.2.2.1.3 Implementation

Once the application is started hit any key to display the initial screen. A menu indicates which options are currently available (such as channel, Tx power, CSMA usage, etc.).

After the desired parameters have been set, chose the proper operating mode. The performance application requires a node acting as receiver ("R") and another node acting as transmitter ("T"). An additional node can operate in promiscuous mode ("I") as diagnostic tool.



Figure 8-3. Terminal Program Snapshot of Performance_Test Application

```

*****
Performance test application (AT86RF231 / ATmega1281)
Settings:
(C) : Channel = 20
(P) : Channel page = 0
(W) : Tx power = 3 dBm
(N) : Number of test frames = 100
(L) : Frame length (PSDU) = 20
(A) : ACK request = ACK requested
(F) : Frame retry enabled = true
(M) : CSMA enabled = true
(T/R/O/I) : Operating mode Tx/Rx/Off/Promiscuous = Tx
(S) : Start test
>
Transmitting... Wait until test is completed. Done.

Test result:
Test duration = 0.285221 s
Transmitted frames = 100
Frames w/o ACK = 0
Channel access failures = 0
Net data rate = 56.10 kbit/s
Press any key to continue

```

Verbunden 00:01:50 Auto-Erkenn. 9600 8-N-1 RF GROSS NUM Aufzeichnen Druckerecho

8.2.2.2 Wireless_UART

8.2.2.2.1 Introduction

The TAL example Wireless_UART is a terminal-based application. It acts as a simple communication program between nodes (i.e. a wireless UART connection).

This example application uses TAL API as interface to the stack.

The application and all required build files are located in directory Applications/TAL_Examples/ Wireless_UART. The source code of the application can be found in the subdirectory Src.

8.2.2.2.2 Requirements

This application requires a serial connection for typing messages and displaying the received data from the other device. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to type input or see the output of the application, please start a terminal application on your host system.

8.2.2.2.3 Implementation

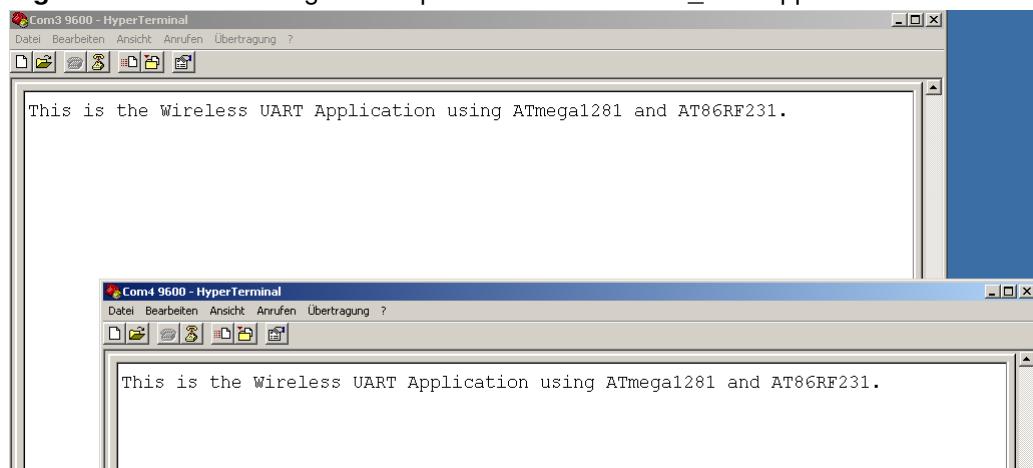
The basic TAL Example Wireless_UART deploys a nonbeacon-enabled network. The example application is based on the TAL.

Both nodes operate at channel DEFAULT_CHANNEL with the PAN ID DEFAULT_PAN_ID and use the same short address settings. More than two nodes should not be operated with the same settings.

A terminal program (Terminal program settings: No flow control and local echo should be disabled) can be used to enter text on both nodes.

If a byte is received via the serial connection from one node, the node transmits the byte to the other node. If the transmission was successful, the entered bytes are forwarded to the terminal indicating successful transmission.

Figure 8-4. Terminal Program Snapshots of Performance_Test Application



8.2.2.2.4 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.

8.2.3 STB Examples

8.2.3.1 Secure_Remote_Control

8.2.3.1.1 Introduction

The STB example *Secure_Remote_Control* is an application demonstrating application security for a remote control acting as a light switch. It deploys a nonbeacon-enabled network using ZigBee/CCM* security.

This example application uses both STB and TAL API as interface to the stack. The STB is used to secure and unsecure application data, and the TAL is used to transmit or receive (secured or unsecured) frames.

The application and all required build files are located in directory Applications/STB_Examples/ Secure_Remote_Control. The source code of the application can be found in the subdirectory Src.

8.2.3.1.2 Requirements

This 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.
- One button on the board in order to determine whether the node starts in secure or unsecure mode, and in order to initiate an action by the remote control.





- A serial connection for typing messages and displaying the received data from the other device. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to see the output of the application, please start a terminal application on your host system.

8.2.3.1.3 Implementation

The application involves exactly two nodes. Both nodes operate at channel DEFAULT_CHANNEL with the PAN ID DEFAULT_PAN_ID and use the same short address settings. More than two nodes should not be operated with the same settings.

The application uses a button to determine the mode of operation (secure or unsecure mode) or to initiate frame transmissions, and (up to three) LEDs to indicate the current status of the node.

Once the node is powered up the LED_0 switched on permanently. If the node is powered while the button is pressed (until the LED_0 is switched on), the application runs in unsecured mode (LED_1 is off). Otherwise (i.e. button is not pressed during system start-up) the application runs in secured mode (LED_1 is on).

If the button is pressed during operation, the node sends a frame which contains the message "Toggle light!". This message is either to be seen in plaintext or it is encrypted.

A node in secured mode only transmits frames being encrypted using ZigBee/CCM* security, and also expects frames being encrypted. A node in unsecured mode only transmits frames with plaintext, and also expects frames with plaintext.

If the transmitter has received an Acknowledgment frame as response to its frame transmission, it indicates the successful transmission by toggling the LED_2. If the transmission is not successfully, the LED_2 is flashing for a short time.

Receiver is in secured mode: If the node has received a secured frame, that could be decrypted properly, the LED_2 is toggled. If the received frame is either unsecured or has a security error (e.g. incorrect key), the LED_2 is flashing for a short time.

Receiver is in unsecured mode: If the node has received a plaintext frame, the LED_2 is toggled. If the received frame is secured, the LED_2 is flashing for a short time.

If a serial terminal is used, the used mode (security on or off) and the status of the last frame transmission or reception are printed.

The following pictures show the printouts on the serial terminal depending whether the nodes are started in secure or unsecure mode.

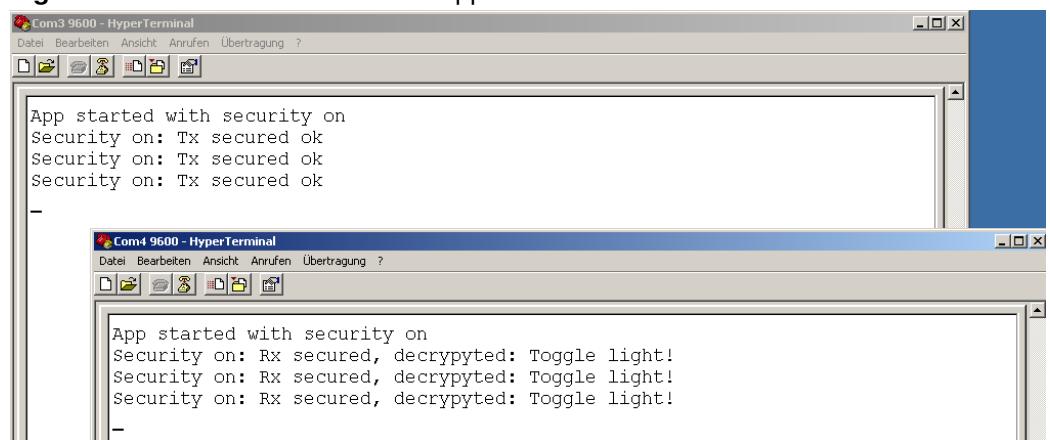
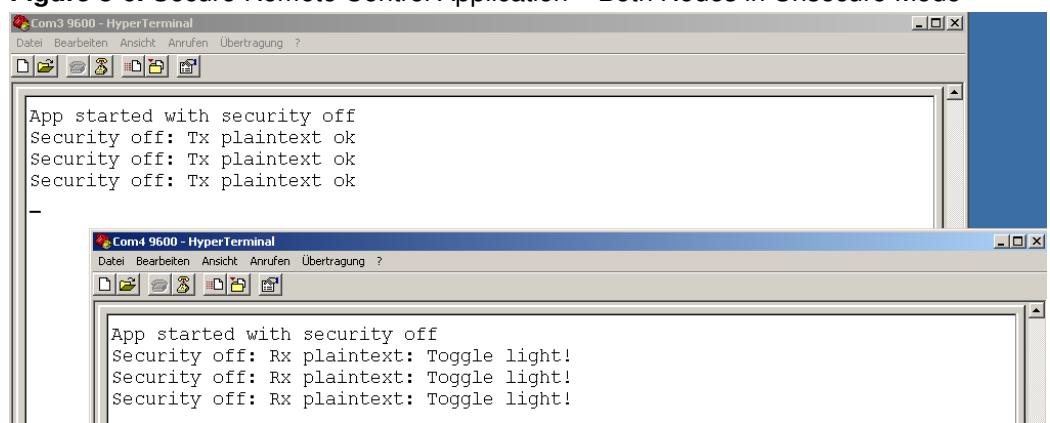
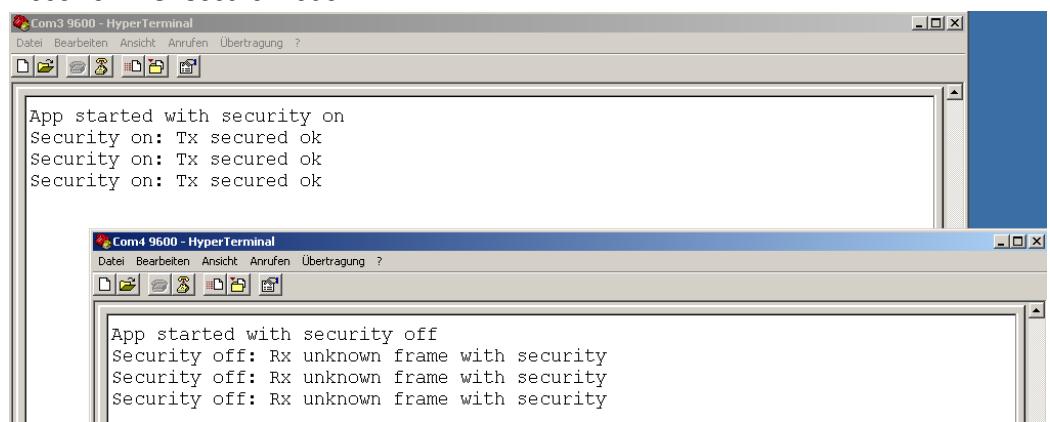
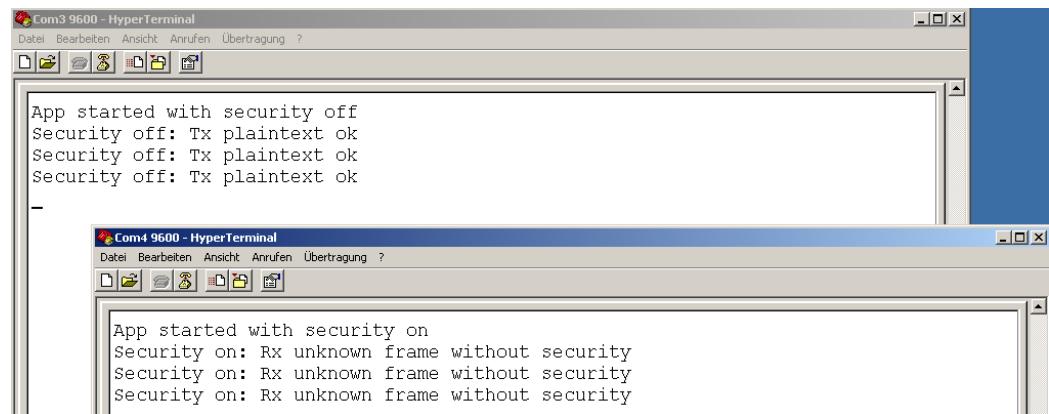
Figure 8-5. Secure Remote Control Application – Both Nodes in Secure Mode**Figure 8-6.** Secure Remote Control Application – Both Nodes in Unsecure Mode**Figure 8-7.** Secure Remote Control Application – Transmitter in Secure Mode, Receiver in Unsecure Mode

Figure 8-8. Secure Remote Control Application – Transmitter in Unsecure Mode, Receiver in Secure Mode



8.2.3.1.4 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.
- It is not recommended operating more than two nodes with the application simultaneously.

8.2.3.2 Secure_Sensor

8.2.3.2.1 Introduction

The STB example *Secure_Sensor* deploys a nonbeacon-enabled network with encrypted and authenticated frames. It is using ZigBee/CCM* security. The application consists of Sensor (i.e. End Devices) and a Data Sink (i.e. a PAN Coordinator) collecting data from the sensors.

This example application uses both STB and MAC API as interface to the stack. The STB is used to secure and unsecure application data, and the MAC is used to set-up a nonbeacon-enabled network, perform scanning and network association, and to transmit or receive (secured or unsecured) frames.

The application and all required build files are located in directory Applications/STB_Examples/ *Secure_Sensor*. The source code of the application can be found in the subdirectory *Data_Sink/Src* and *Sensor/Src*.

8.2.3.2.2 Requirements

This 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.
- A serial connection for typing messages and displaying the received data from the other device. Depending on the available Serial I/O interface for each board this can be either UART or USB. In order to see the output of the application, please start a terminal application on your host system.

8.2.3.2.3 Implementation

The node acting as data sink (PAN Coordinator) starts a nonbeacon-enabled PAN at channel DEFAULT_CHANNEL with the PAN ID DEFAULT_PAN_ID. The sensor (Device) scans for this network and associates to the data sink (PAN Coordinator). Once the sensor (Device) is associated, it uses a timer that fires every 2 seconds to transmit a random payload (sensor measurement data) to the data sink.

The frames are secured according to ZigBee/CCM* network layer security:

- The network header is omitted, only the auxiliary security header is constructed (14 byte long).
- The applied security level is 0x06, i.e. encrypted payload, authentication applied, MIC 8 byte long.
- The random payload has 13 byte length.

The LEDs signal the following status:

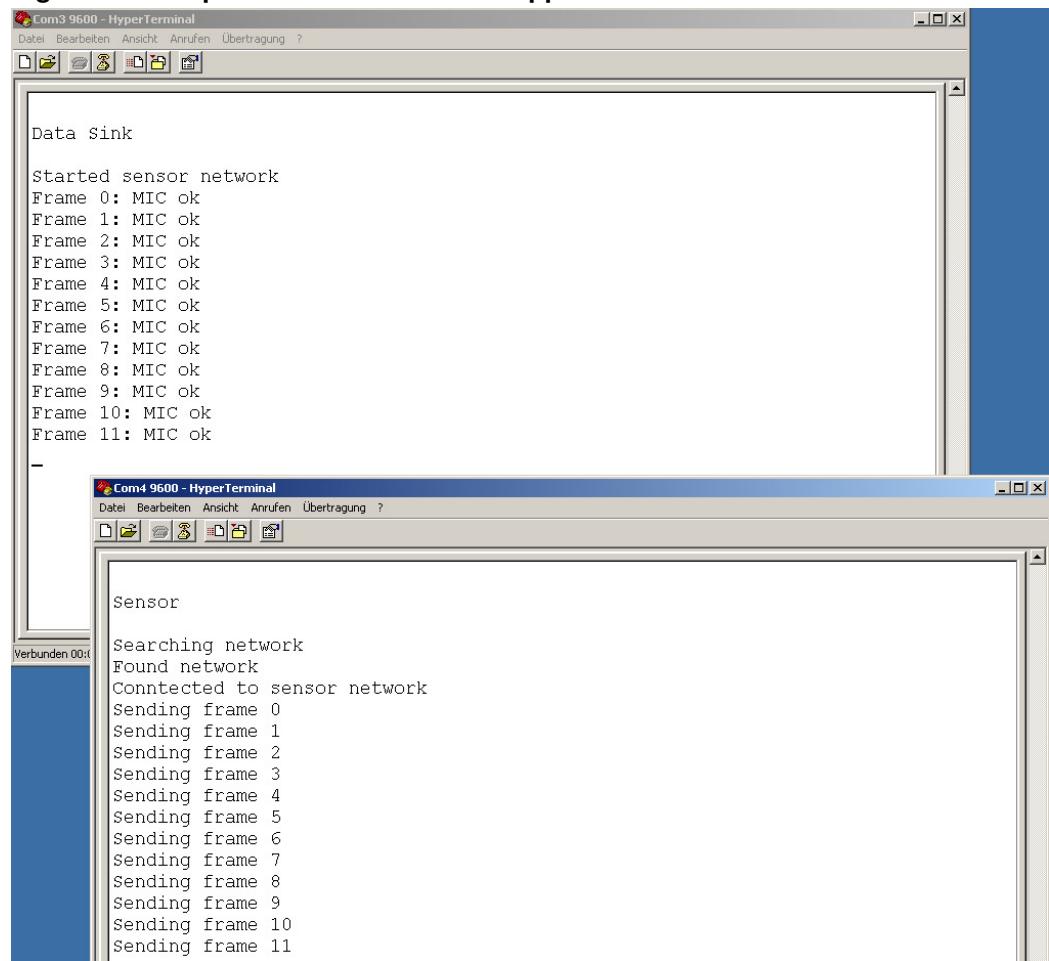
Sensor (Device):

- LED 0 on: Application is running.
- LED 1 on: Sensor (Device) is associated to Data Sink (Coordinator).
- LED 2 blinking slowly: Data frames are sent (one every 2 seconds).

Data Sink (Coordinator):

- LED 0 on: Application is running.
- LED 1 on: Network is started.
- LED 2 blinking slowly: Data frames are received (one every 2 seconds).
- LED 0 blinking fast: Format error on received frame - wrong payload length, or security control byte has a wrong value.
- LED 1 blinking fast: Frame with too small frame counter received.
- LED 2 blinking fast: MIC of received frame is wrong.

Figure 8-9. Snapshot of Secure Sensor Application



8.2.3.2.4 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.
- It is not recommended operating more than two nodes with the application simultaneously.

8.3 Common SIO Handler

Applications that require SIO input or output (via UART or USB) in order to print data on a terminal or to get input from the end user can utilize a common SIO handler that provides helper functions for platform independent processing of serial information. These helper functions are

- *sio_putchar()*: print one character
- *sio_getchar()*: read one character via SIO (blocking function)
- *sio_getchar_nowait()*: read one character via SIO without blocking

The helper functions in return make use of the PAL related SIO functions such as *pal_sio_tx()* and *pal_sio_rx()*. This allows for easy implementation of SIO functionality in the application.

The SIO handler is located in directory

```
Applications/Helper_Files/SIO_Support
```

The implementation is located in file *sio_handler.c* in the *Src* directory, whereas the corresponding function declaration are located in file *sio_handler.h* in the *Inc* directory.

For the IAR compiler the file *write.c* located in the *IAR_Support* needs to be added as well.

In order to use these helper functions, the application needs to do the following things:

- Include the header file in its source files

```
#include "sio_handler.h"
```

- Extend the include search path for GCC Makefiles, e.g.

```
PATH_SIO_SUPPORT = $(MAIN_DIR)/Applications/Helper_Files/SIO_Support
...
## Include directories for SIO support
INCLUDES += -I $(PATH_SIO_SUPPORT)/Inc
```

- Extend the include search path for IAR project files, e.g.

```
<name>newCCIncludePaths</name>
```

```
...
```

```
<state>$PROJ_DIR$\..\..\..\..\Helper_Files\SIO_Support\Inc</state>
```

- Add file *sio_handler.c* to the list of source and object files in GCC Makefiles, e.g.

```
...
```

```
$(TARGET_DIR)/sio_handler.o\
```

```
...
```

```
$(TARGET_DIR)/sio_handler.o: $(PATH_SIO_SUPPORT)/Src/sio_handler.c
```

```
$(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $<
```

Add file *sio_handler.c* to the list of source and object files in GCC Makefiles, e.g.

```
<file>
```

```
<name>
```

```
$PROJ_DIR$\..\..\..\..\Helper_Files\SIO_Support\Src\sio_handler.c
```

```
</name>
```

```
</file>
```

```
<file>
```

```
<name>
```

```
$PROJ_DIR$\..\..\..\..\Helper_Files\SIO_Support\IAR_Support\write.c
```

```
</name>
```

```
</file>
```

8.4 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.

8.4.1 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. All required MAC stub functions are already implemented in the files *usr_mcps_*.c* or *usr_mlme_*.c* in directory *MAC/Src*.

So whenever such a callback is not used by the application, simply add the required *usr_*.c* stub files to your Makefiles or IAR project files.

8.4.2 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.

So whenever such a callback is not used by the TAL based application, simply add the required *tal_*.c* stub files to your Makefiles or IAR project files.

8.4.3 Example for MAC Callbacks

The handling of callback stub functions shall be more illustrated by the example of the MAC based application Star_Nobeacon. When opening the main source file of this application Applications/MAC_Examples/Star_Nobeacon/Src/main.c, the source code indicates that the following used MAC callback functions are actually used and filled with dedicated application code:

- *usr_mlme_reset_conf* – Node can perform a MAC reset
- *usr_mlme_scan_conf* – Node can perform scanning
- *usr_mlme_set_conf* – Node can change PIB attributes
- *usr_mlme_start_conf* – Node can set up a new network
- *usr_mlme_associate_ind* – Node can accept associations from other nodes
- *usr_mlme_comm_status_ind* – Required to accept associations from other nodes
- *usr_mcps_data_ind* – Node can receive data frames
- *usr_mlme_associate_conf* – Node can perform association to other nodes
- *usr_mcps_data_conf* – Node can transmit data frames

The remaining callbacks

- *usr_mcps_purge_conf*
- *usr_mlme_beacon_notify_ind*
- *usr_mlme_disassociate_conf*
- *usr_mlme_disassociate_ind*
- *usr_mlme_get_conf*
- *usr_mlme_orphan_ind*
- *usr_mlme_poll_conf*
- *usr_mlme_rx_enable_conf*
- *usr_mlme_sync_loss_ind*

are not used by the application, since this example does not deal with this functionality. In order to avoid implementing empty stubs in the application, the way of using the delivered stub functions in the files *usr_*.c* shall be used. This is done by adding these stub files to the GCC Makefiles or IAR project files.

In the corresponding GCC Makefile (*Applications/MAC_Examples/Star_Nobeacon/any_platform/GCC/Makefile*) the following lines can be found in order to add the required object files to the link process:

```
## Objects that must be built in order to link
OBJECTS = $(TARGET_DIR)/main.o \
          $(TARGET_DIR)/pal_uart.o \
          $(TARGET_DIR)/mac_api.o \
          $(TARGET_DIR)/usr_mcps_purge_conf.o \
```





```
$ (TARGET_DIR) /usr_mlme_beacon_notify_ind.o \
$ (TARGET_DIR) /usr_mlme_disassociate_conf.o \
$ (TARGET_DIR) /usr_mlme_disassociate_ind.o \
$ (TARGET_DIR) /usr_mlme_get_conf.o \
$ (TARGET_DIR) /usr_mlme_orphan_ind.o \
$ (TARGET_DIR) /usr_mlme_poll_conf.o \
$ (TARGET_DIR) /usr_mlme_rx_enable_conf.o \
$ (TARGET_DIR) /usr_mlme_sync_loss_ind.o
```

In the corresponding IAR project file (*Applications/MAC_Examples/Star_Nobeacon/any_platform/Star.ewp*) the following lines can be found in order to add the required source files to the build process:

```
<group>
  <name>MAC_API</name>
  <file>
    <name>$PROJ_DIR$..\..\..\..\..\MAC\Src\mac_api.c</name>
  </file>
  <file>
    <name>$PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mcps_purge_conf.c</name>
  </file>
  <file>
    <name>
      $PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_beacon_notify_ind.c
    </name>
  </file>
  <file>
    <name>
      $PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_disassociate_conf.c
    </name>
  </file>
  <file>
    <name>
      $PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_disassociate_ind.c
    </name>
  </file>
  <file>
    <name>
      $PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_get_conf.c</name>
    </name>
  </file>
  <file>
    <name>
      $PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_orphan_ind.c
    </name>
  </file>
  <file>
    <name>$PROJ_DIR$..\..\..\..\..\MAC\Src\usr_mlme_poll_conf.c</name>
  </file>
  <file>
```

```
<name>
    $PROJ_DIR$\\..\\..\\..\\..\\MAC\\Src\\usr_mlme_rx_enable_conf.c
</name>
</file>
<file>
<name>
    $PROJ_DIR$\\..\\..\\..\\..\\MAC\\Src\\usr_mlme_sync_loss_ind.c
</name>
</file>
</group>
```



9 Supported Platforms

This chapter describes which hardware platforms are currently supported with the AVR2025 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), and
- A specific Board or even several boards that contain the MCU or the transceiver chip.

The supported software for each platform can be found in the directory *PAL*.

9.1 Supported MCU Families

Currently the following generic MCU families are supported:

- ARM7: Atmel ARM7 platforms
- AVR: Atmel AVR 8-bit ATmega platforms
- MEGA_RF: Atmel AVR 8-bit ATmegaRF Single Chip platforms
- XMEGA: Atmel AVR 8-bit ATxmega platforms

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

9.2 Supported MCUs

Within each platform family a number of MCUs are supported. These are for example the ATmega1281, ATxmega128A1, ATmega128RFA1, AT90SAM7X256, etc.

For a complete list of the actually supported MCUs please refer to the AVR2025 release notes (*MAC_Release_Notes.txt* in directory *MAC_v_x_y_z\Doc*) or directly look into the various subdirectories as mentioned in section 9.1.

9.3 Supported Transceivers

For a complete list of all supported transceivers please refer to the AVR2025 release notes (*Release_Notes.txt* in directory *MAC_v_x_y_z\Doc*).

9.4 Supported Boards

The actually supported boards can be found in the corresponding PAL Boards directories *MAC_v_x_y_z/PAL/pal_family_name/mcu_name/Boards*. For example all supported boards for the ATmega128RFA1 Single Chip are located in directory *PAL/MEGA_RF/ATMEGA128RFA1/Boards*. Each board directory contains a file *pal_boardtypes.h*, which contains a list of all supported boards based on this specific MCU together with a short description.

The following sections describe the currently supported hardware platforms in more detail. All described hardware boards are available from Atmel (see [1]) or third party vendors (e.g. see [2]).

For a short list of the supported peripherals of each board see section 9.4.6.

9.4.1 Radio Controller Boards (RCB) based Platforms

The Radio Controller Board (RCB) is a radio module containing either a MCU (e.g. ATmega1281) and an Atmel transceiver, or an Atmel Single Chip. RCBs can be used

stand alone as plain RCB or in conjunction with a base board providing additional peripheral capabilities.

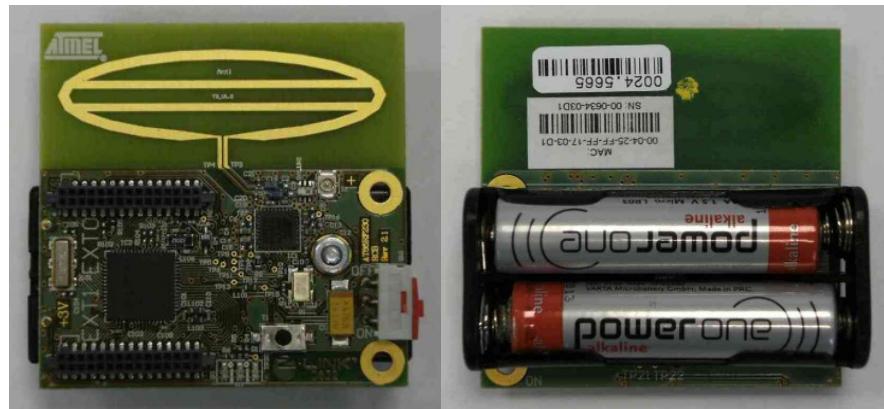
9.4.1.1 Plain Radio Controller Board RCB230 V3.1

For more information about earlier versions of the Plain Radio Controller Boards prior to V3.2 please contact Atmel support ([3]).

9.4.1.2 Plain Radio Controller Board RCB230 V3.2

- AT86RF230B and ATmega1281
- See PAL\AVR\ATMEGA1281\BOARDS\RCB_3_2_PLAIN

Figure 9-1. RCB V3.2 with AT86RF230B and ATmega1281



9.4.1.3 Plain Radio Controller Board RCB231 V4.0

- AT86RF231 with and ATmega1281
- See PAL\AVR\ATMEGA1281\BOARDS\RCB_4_0_PLAIN

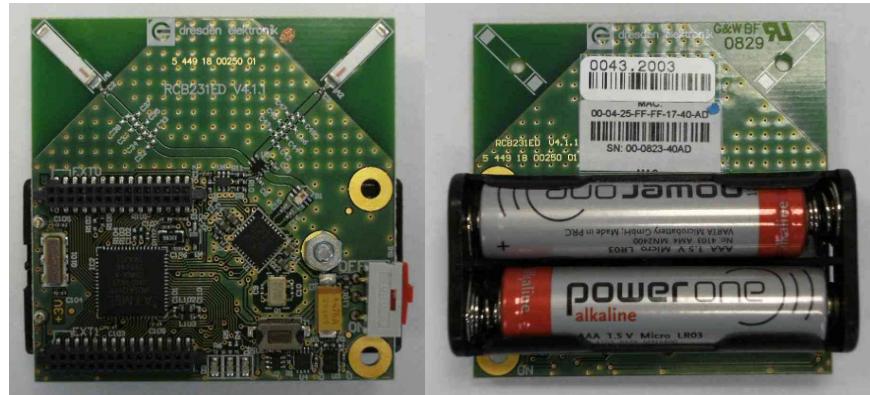
Figure 9-2. RCB V4.0 with AT86RF231 and ATmega1281



9.4.1.4 Plain Radio Controller Board RCB231ED V4.1.1

- AT86RF231 with Antenna Diversity and ATmega1281
- See PAL\AVR\ATMEGA1281\BOARDS\RCB_4_1_PLAIN

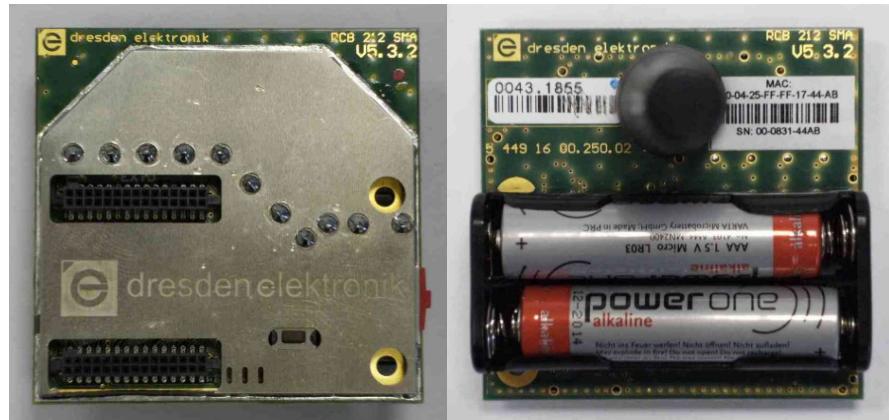
Figure 9-3. RCB V4.1 with AT86RF231 and ATmega1281



9.4.1.5 Plain Radio Controller Board RCB212SMA V5.3.2

- AT86RF212 with and ATmega1281
- See PAL\AVR\ATMEGA1281\BOARDS\RCB_5_3_PLAIN

Figure 9-4. RCB V5.3 with AT86RF212 and ATmega1281



9.4.1.6 Plain Radio Controller Board RCB V6.3 with ATmega128RFA1

- ATmega128RFA1 Single Chip only
- See PAL\AVR\ATMEGA1281\BOARDS\RCB_5_3_PLAIN

Figure 9-5. RCB V6.3 with ATmega128RFA1



9.4.1.7 Sensor Terminal Board

The Sensor Terminal Board can be used as baseboard for a Plain RCB to enable SIO capabilities via USB. It provides one button and two LEDs for user interaction. It can be used for existing RCBs mentioned in section 9.4.1.

The following pictures depict a Sensor Terminal Board with and without an RCB, and also indicate how the Sensor Terminal Board is connected to the JTAG-ICE.

Figure 9-6. Sensor Terminal Board without RCB

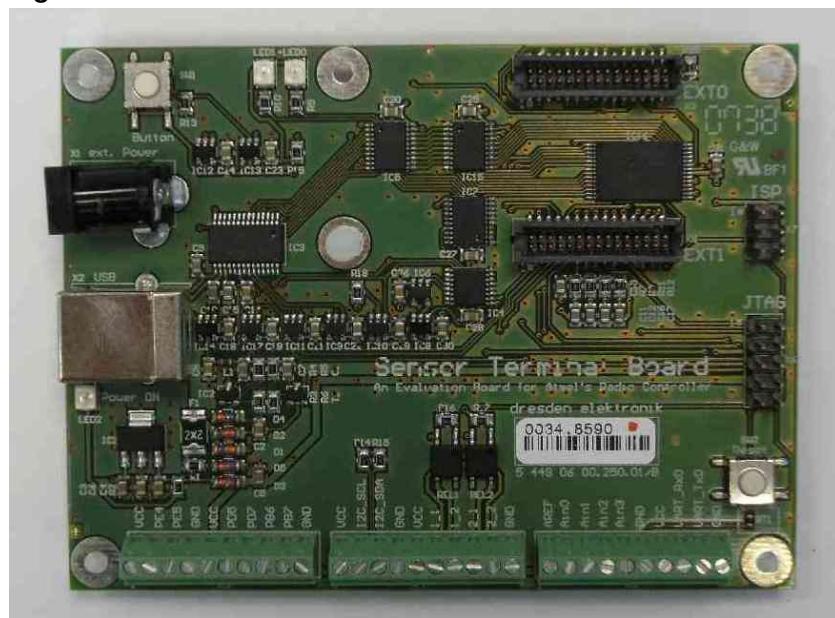


Figure 9-7. Sensor Terminal Board with RCB V6.3 with ATmega128RFA1

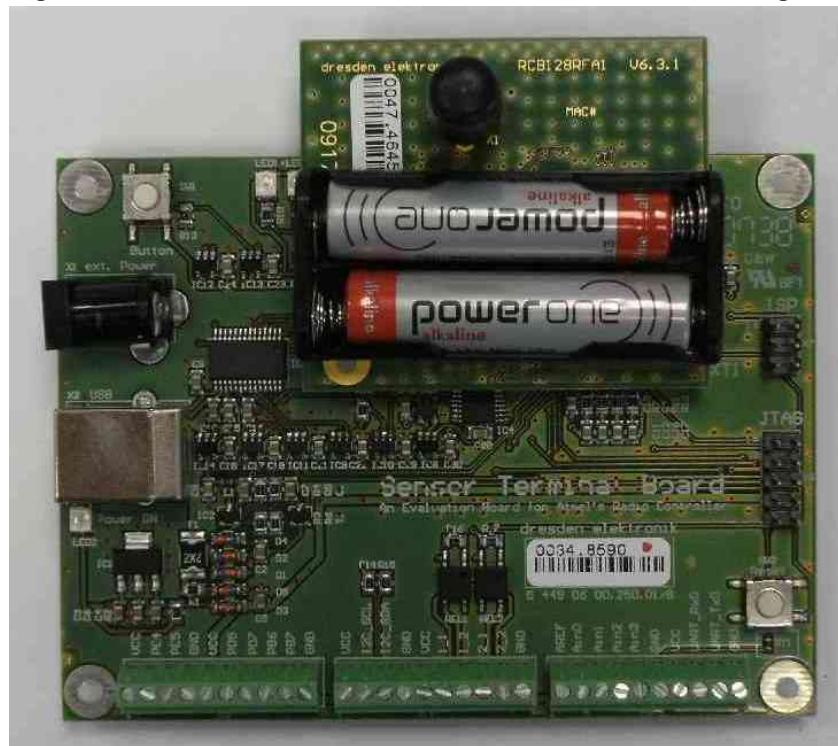
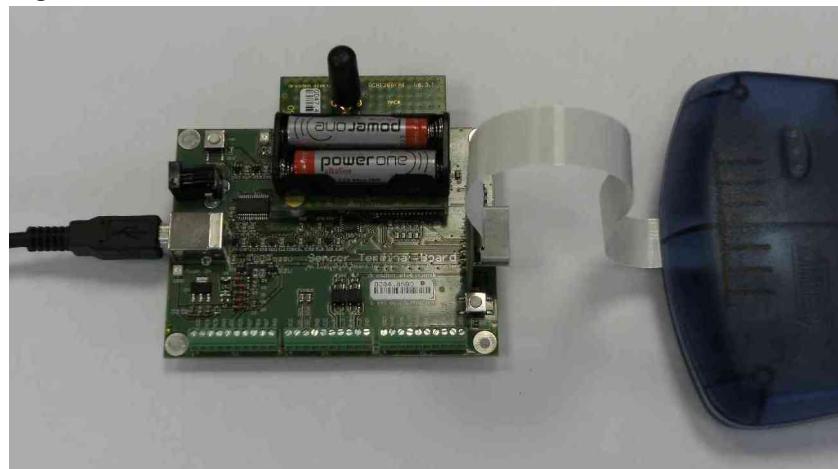


Figure 9-8. Sensor Terminal Board connected to JTAG-ICE and USB



The Sensor Terminal Board can be powered with USB, although it is recommended to use a powered hub in case the board is not directly connected to a PC.

9.4.1.8 Breakout Board (Light)

The Breakout Board (Light) can be used as baseboard for a Plain RCB to enable SIO capabilities via UART. It can be used for existing RCBs mentioned in section 9.4.1**Error! Reference source not found.**. Its main purpose is to be a simple baseboard for the RCB for image flashing and debugging purpose.

The following pictures depict a Breakout Board Light with and without an RCB, and also indicate how the Breakout Board Light is connected to the JTAG-ICE.

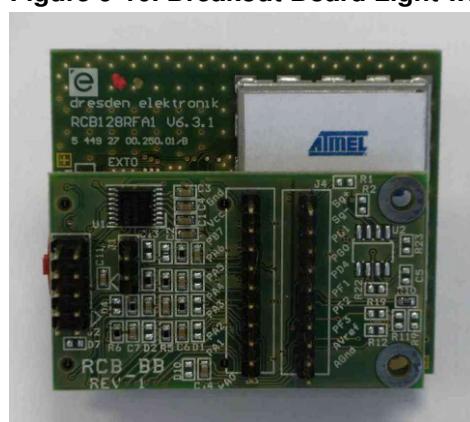
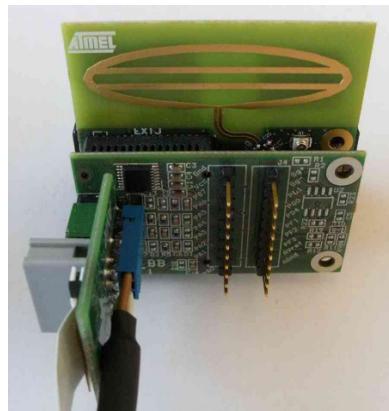
Figure 9-9. Breakout Board Light without RCB**Figure 9-10. Breakout Board Light with RCB****Figure 9-11. Breakout Board Light connected to JTAG-ICE and UART**

Figure 9-12. Close-up View of Breakout Board Light connected to JTAG-ICE and UART



9.4.2 Radio Extender Boards (REB)

The Radio Extender Board (REB) is a radio module containing an Atmel transceiver. REBs cannot be used stand alone but require an additional baseboard for the MCU, such as STK600, STK500, AT91SAM7X-EK, AT91SAM7XC-EK, etc.

9.4.2.1 Radio Extender Board REB230 V2.1

For more information about earlier versions of the Radio Extender Boards prior to V2.3 please contact Atmel support ([3]).

9.4.2.2 Radio Extender Board REB230 V2.3

- Atmel transceiver: AT86RF230B
- The REB230B V2.3 is supported on several platforms:
 - See PAL\AVR\ATMEGA1281\Boards\REB_2_3_STK500_STK501
 - See PAL\AVR\ATMEGA2561\Boards\REB_2_3_STK500_STK501
 - See PAL\AVR\ATMEGA644P\Boards\REB_2_3_STK500
 - See PAL\XMEGA\ATXMEGA128A1\Boards\REB_2_3_STK600
 - See ARM7\AT91SAM7X256\Boards\REB_2_3_REX_ARM_REV_2
 - See ARM7\AT91SAM7XC256\Boards\REB_2_3_REX_ARM_REV_2

Figure 9-13. REB V2.3 with AT86RF230B**9.4.2.3 Radio Extender Board REB231 V4.0.1**

- Atmel transceiver: AT86RF231
- The REB231 V4.0 is supported on several platforms:
 - See PAL\AVR\ATMEGA1281\Boards\REB_4_0_STK500_STK501
 - See PAL\XMEGA\ATXMEGA128A1\Boards\REB_4_0_STK600

Figure 9-14. REB V4.0.1 with AT86RF231**9.4.2.4 Radio Extender Board REB231ED V4.1.1**

- Atmel transceiver: AT86RF231 using Antenna Diversity
- The REB231ED V4.1 is supported on several platforms:
 - See PAL\AVR\ATMEGA1281\Boards\REB_4_1_STK500_STK501

- See PAL\XMEGA\ATXMEGA128A1\Boards\REB_4_1_STK600
- See PAL\XMEGA\ATXMEGA256A3\Boards\REB_4_1_STK600

Figure 9-15. REB231ED V4.1.1 with AT86RF231



9.4.2.5 Radio Extender Board REB212 V5.0.2

- Atmel transceiver: AT86RF212
- The REB212 V5.0 is supported on several platforms:
 - See PAL\AVR\ATMEGA1281\Boards\REB_5_0_STK500_STK501
 - See PAL\XMEGA\ATXMEGA128A1\Boards\REB_5_0_STK600

Figure 9-16. REB212 V5.0.2 with AT86RF212



9.4.3 Radio Extender Boards (REB) based Platforms

9.4.3.1 STK600 and REB to STK600 Adapter

The STK600 in conjunction with an REB to STK600 Adapter can be used as a baseboard for an REB to create platforms using AVR 8-bit MCUs (such as ATxmega or ATmega MCUs). It provides eight buttons and LEDs for user interaction. It can be used for existing REBs mentioned in section 9.4.2.

The following pictures depict an REB to STK600 Adapter and an STK600 with an REB with REB to STK600 Adapter.

Figure 9-17. REB to STK600 Adapter

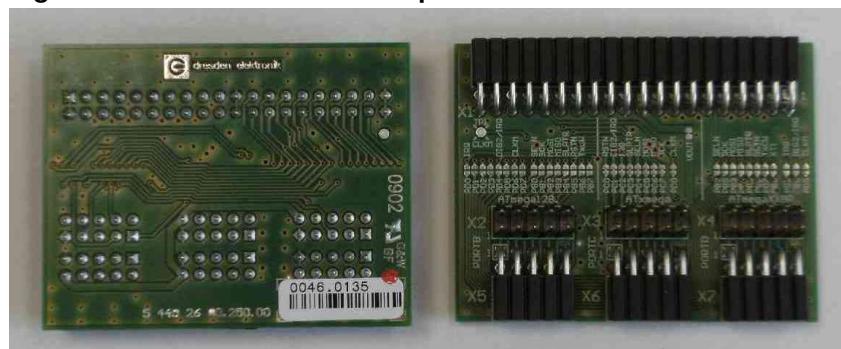


Figure 9-18. STK600 with REB to STK600 Adapter and REB231ED V4.1.1



The next pictures indicates how the STK600 is connected to the JTAG-ICE and UART, and how the LEDs and buttons are enabled.

Figure 9-19. STK600 (with ATxmega128A1) with REB to STK600 Adapter and REB connected to JTAG-ICE and UART, and LED and Button Cable

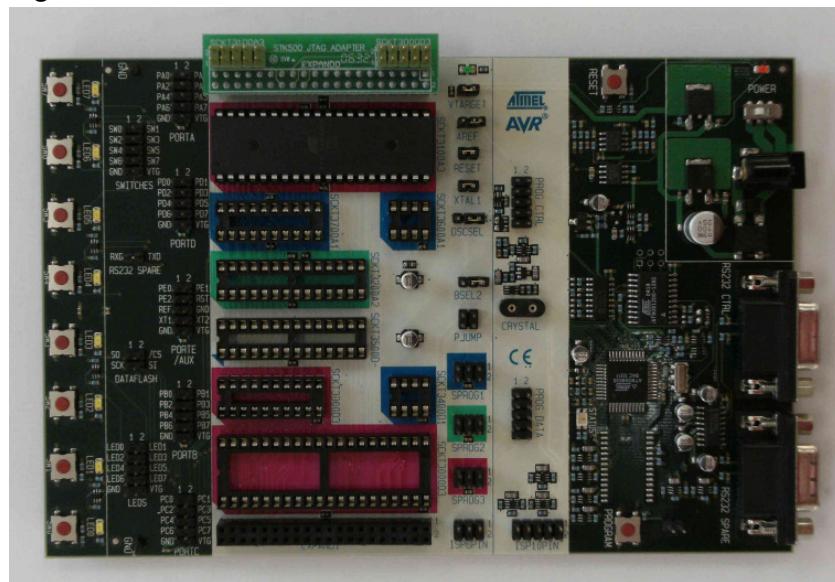


9.4.3.2 STK500

The STK500 can be used as a baseboard for an REB to create platforms using AVR 8-bit MCUs with a 40-pin PDIP package (such as ATmega644p MCUs). It provides two buttons (there are actually eight buttons but only the first two are used for this platform) and eight LEDs for user interaction. It can be used for existing REBs mentioned in section 9.4.2.

The following picture depicts an STK500 with an ATmega644p.

Figure 9-20. STK500



The next pictures indicates how the STK500 is connected to the JTAG-ICE and UART, and how the LEDs and buttons are enabled.

Figure 9-21. STK500 (with ATmega644p) with REB connected to JTAG-ICE and UART, and LED and Button Cable

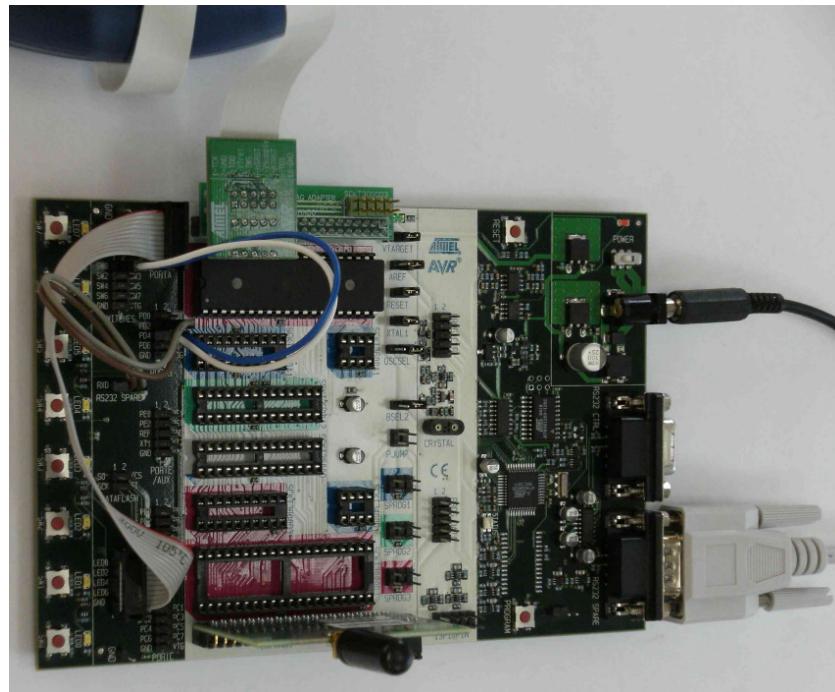
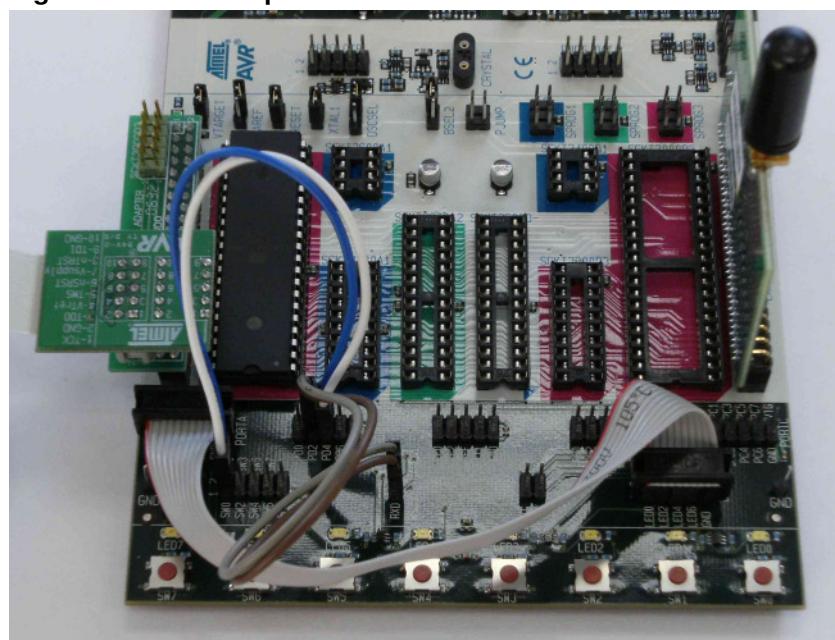


Figure 9-22. Close-up View of Cable Connection for LEDs and Buttons



9.4.3.3 STK500 + STK501

The STK500 in conjunction with an STK501 can be used as a baseboard for an REB to create platforms using AVR 8-bit MCUs with a 64-pin TQFP package (such as

ATmega1281 MCUs). It provides eight buttons and LEDs for user interaction. It can be used for existing REBs mentioned in section 9.4.2.

The following picture depicts an STK500 with an STK501.

Figure 9-23. STK500 with STK501

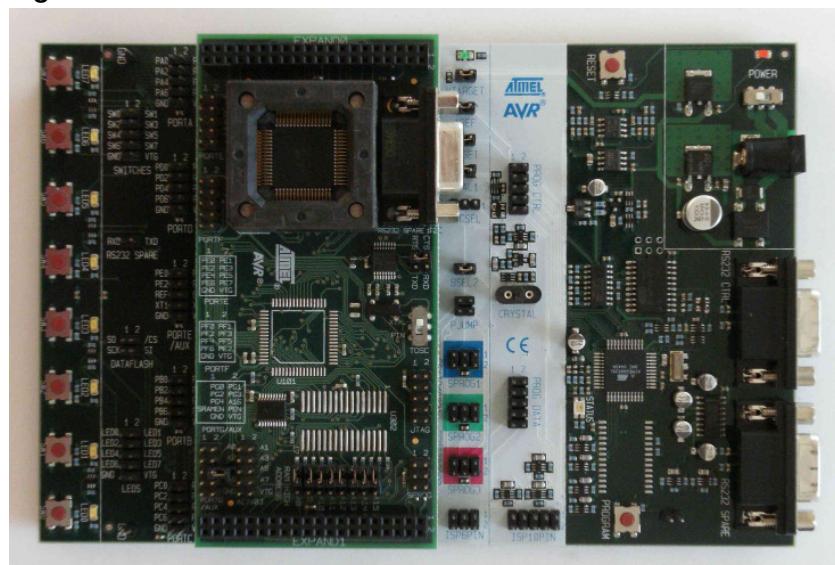


Figure 9-24. STK500 with STK501 (with ATmega1281) and REB connected to JTAG-ICE and UART, and LED and Button Cable

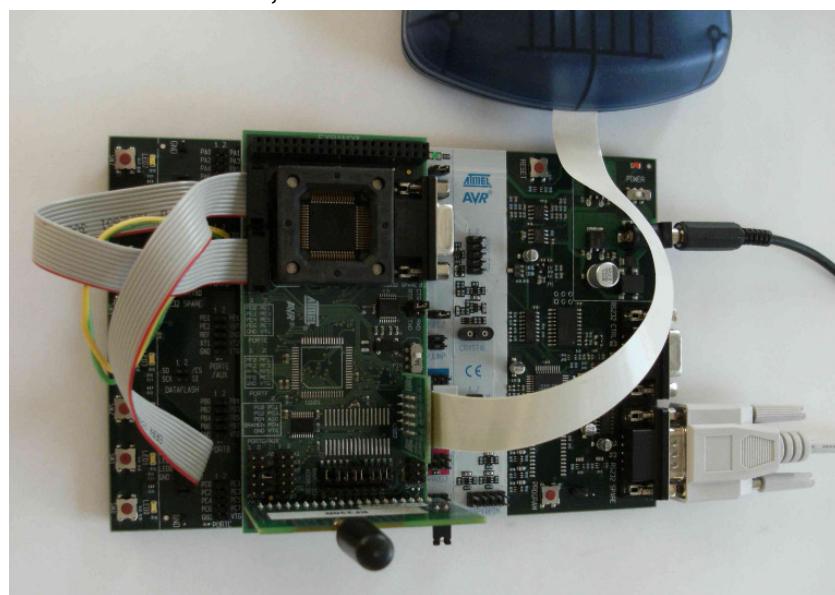
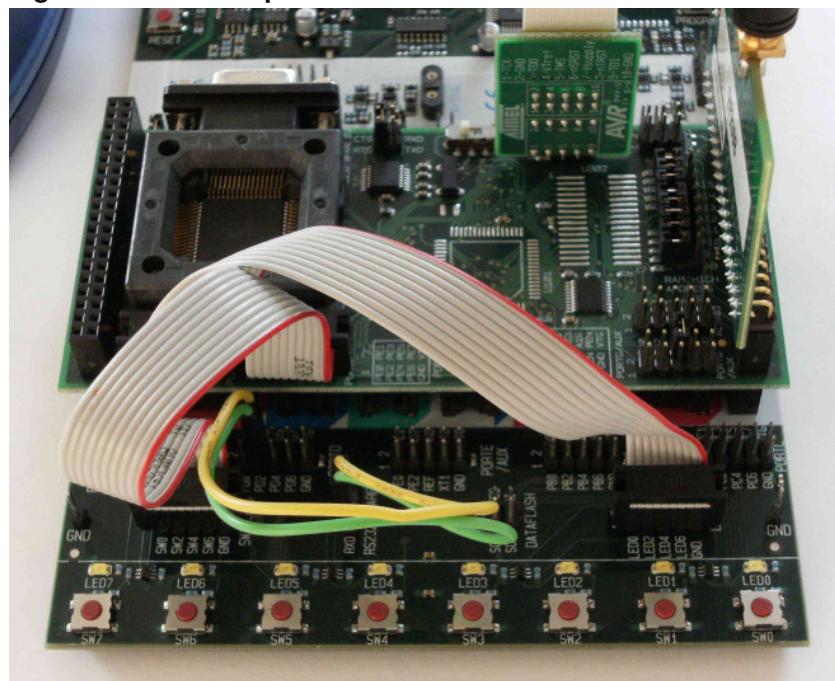


Figure 9-25. Close-up View of Cable Connection for LEDs and Buttons

9.4.3.4 AT91SAM7X-EK

The AT91SAM7X-EK in conjunction with an REB to ARM Adapter can be used as a baseboard for an REB to create platforms using Atmel ARM7 MCUs (such as AT91SAM7X256 MCUs). It provides currently 4 LEDs and a Joystick for user interaction.

Currently supported are:

- REB230B V2.3 (with AT86RF230) in conjunction with REB to ARM Adapter Rev. 2 (REV_ARM REV 2).
- REB231 V4.0.1/V4.0.2 (with AT86RF231) in conjunction with REB to ARM Adapter Rev. 3 (REV_ARM REV 3)
- REB212 V5.0.2 (with AT86RF212) in conjunction with REB to ARM Adapter Rev. 3 (REV_ARM REV 3)

The following pictures depict an AT91SAM7X-EK board with AT91SAM7X256 and an AT91SAM7X-E with an REV_ARM REV 2 with REB230B V2.3.

Figure 9-26. AT91SAM7X-EK Board with AT91SAM7X256



Figure 9-27. AT91SAM7X-EK Board with AT91SAM7X256



Figure 9-28. AT91SAM7X-EK Board with AT91SAM7X256 connected to SAM-ICE and UART



9.4.4 AT91SAM7XC-EK

The AT91SAM7XC-EK is handled similar to the AT91SAM7X-EK.

9.4.5 RZ USBstick

The RZ USBstick (as part of the ATAVRRZRAVEN 2.4 GHz Evaluation and Starter Kit) is based on AT90USB1287:

- See PAL\AVR\AT90USB1287\Boards\USBSTICK_C
- The USB Bootloader is currently not supported
- Images need to be flashed using JTAG-ICE and the JTAG connector
- In case the 10 pin header for the JTAG connector (included in the ATAVRRZRAVEN kit) is not on the board, this needs to be added
- A USB driver providing a virtual COM port for the USBstick is provided in the software package (see *rzusbstick_cdc.inf* in directory PAL\AVR\AT90USB1287\Boards\USBSTICK_C)

The following picture depicts an RZ USBstick with the 10 pin header for the JTAG connector and the JTAG-ICE.

Figure 9-29. RZ USBstick with 10 Pin Header for JTAG Connector

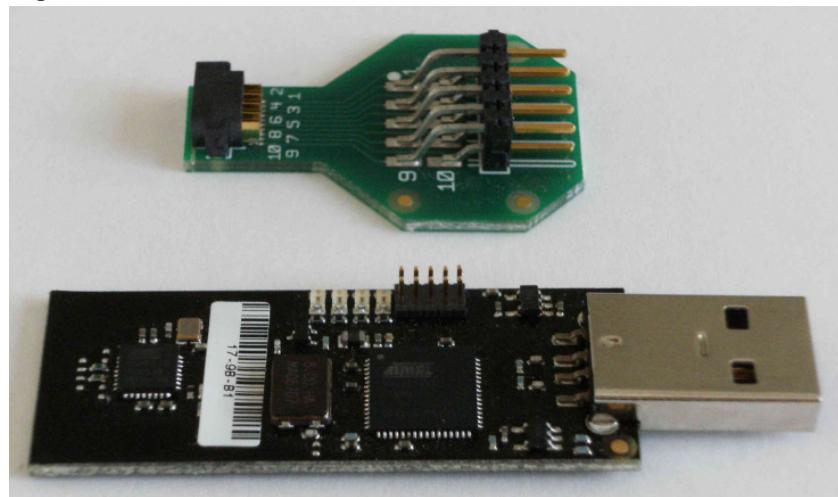
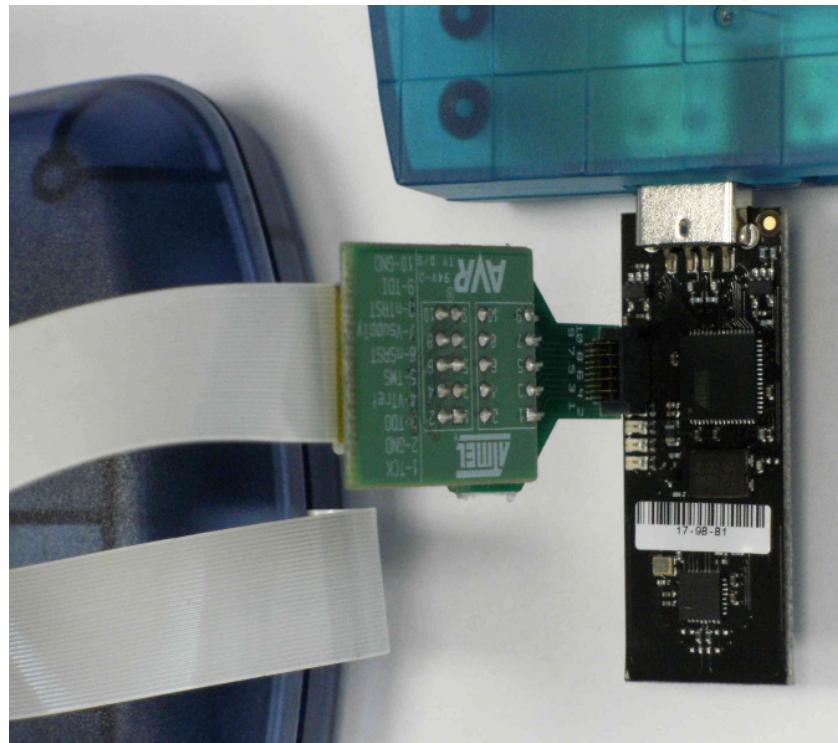


Figure 9-30. RZ USBstick with JTAG-ICE and USB Connection



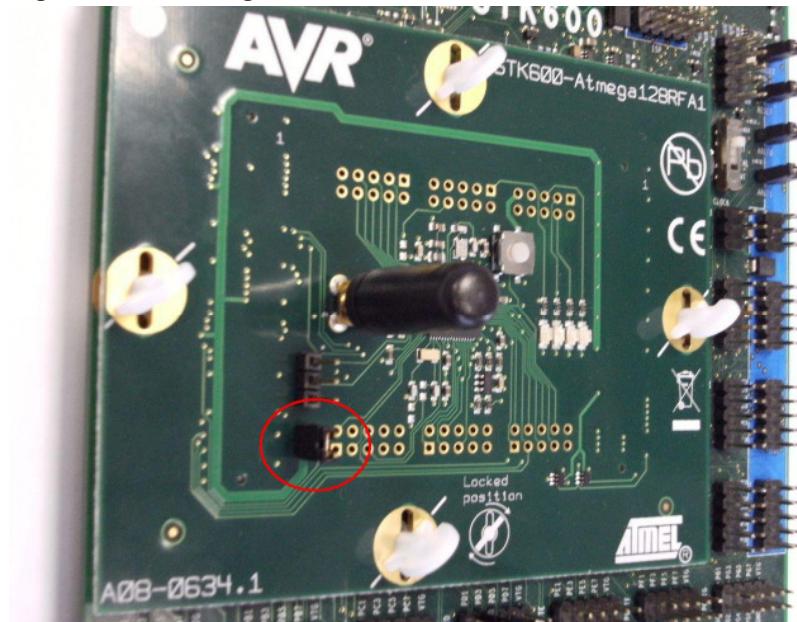
9.4.6 ATmega128RFA1-EK1 Evaluation Kit

The ATmega128RFA1-EK1 Evaluation Kit provides an STK600-ATmega128RFA1 Top Card that can be used in conjunction with the STK600 board. No further Routing Cards are required.

- See PAL\MEGA_RF\ATMEGA128RFA1\Boards\EK1
- It provides currently 3 LEDs and 1 button for user interaction (similar to RCBs)
- The buttons and LEDs from the STK600 base board are not supported
- Both UART0 and UART1 are supported (UART1 is default in provided example applications)

The following picture depicts an ATmega128RFA-EK1 Top Card placed on an STK600 base board and the board setup connected to a JTAG-ICE and set-up for utilization of UART1.

Figure 9-31. ATmega128RFA1-EK1 on STK600



Please note that jumper J11 (see red circle) needs to be placed as shown above (if no current measurements are done) in order to provide the proper voltage to the board.

Figure 9-32. ATmega128RFA1-EK1 on STK600 with JTAG-ICE using UART1

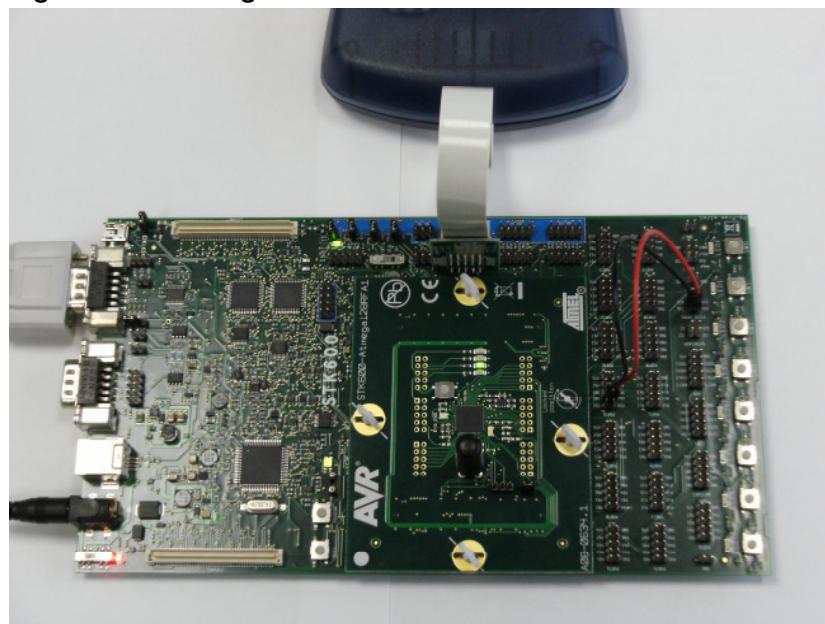
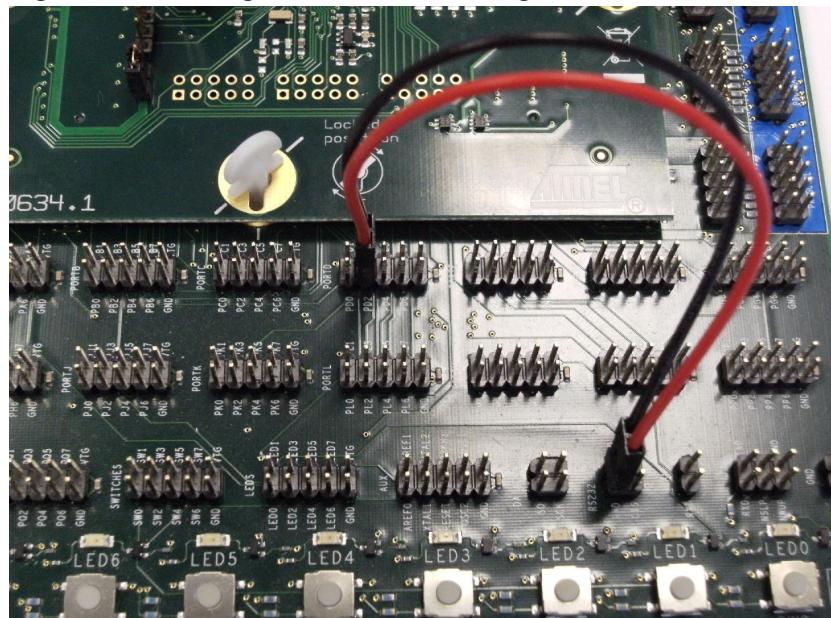


Figure 9-33. ATmega128RFA1-EK1 wiring for UART1



Please note that the ATmega128RFA1-EK1 is not optimized for performance.

9.4.7 Peripherals

Each board or combination of boards provides a variety of peripherals that determine which application can be executed to which extent of this board. This section provides an overview about the differences for the various platforms for the following peripherals:

- Buttons
- LEDs
- SIO support (UART, USB)

Table 9-34. Peripherals supported by Platform Type

	Buttons	LEDs	SIO Support
Plain RCB	1	3	No
Sensor Terminal Board + RCB	1	2	USB
Breakout Board + RCB	0	0	UART
Breakout Board (Light) + RCB	0	0	UART
STK600 + REB to STK600 Adapter + REB	8	8	UART
STK500 + REB	2	8	UART
STK500 + STK501 + REB	8	8	UART
AT91SAM7X(C)-EK + REX_ARM + REB	Joystick	4	UART
RZ USBstick	0	4	USB
ATmega128RFA1-EK1	1	3	UART

10 Platform Porting

10.1 Porting to a new Platform

In case a new platform that is not supported yet, needs to be utilized, the task to bring-up the new platform can be performed as described in the following section. Generally this task can be split into different subtasks:

1. Bring-up of a new PAL or of a board within an already existing PAL
2. Bring-up of an existing application on the new platform
3. Bring-up of new applications on the new platform (if required)

Each of these subtasks will be explained subsequently and described on an example.

Before the actual porting is described a number of terms need to be defined:

- Target MCU: MCU that is not yet supported and shall be utilized within a new platform
- Base MCU: MCU that is already supported within the MAC package and it's PAL implementation is used as base code for the target MCU
- Target board: Board based on the target MCU that is not yet supported and shall be utilized within a new platform
- Base board: Board that is already supported within the base MCU PAL directory and is used as base code for the target board
- Target platform: Platform consisting of target MCU and target board
- Base platform: Platform consisting of base MCU and base board

10.2 Bring-up of a new PAL

If the target platform is not provided by the MAC software package this platform needs to be brought-up. A new target platform can be one of the following cases:

- a) Bring-up of a new customized board for an already supported MCU: This is usually the case if the customer has designed its own hardware board using a standard Atmel MCU (e.g. ATmega1281). This is the simplest case and is described in section 10.3.
- b) Bring-up of a new platform based on a not yet supported MCU but within a supported MCU family: This is usually the case if the customer wants to use a dedicated MCU (e.g. with different memory resources such as using the ATxmega256A3), that is based on an existing MCU family (ATxmega family). This case is described in more detail in section 10.4.
- c) Bring-up of a new platform based on a not yet supported MCU within a not yet supported MCU family, such as an ARM device not based on the ARM7 family.

10.3 Bring-up of a new Hardware Board

10.3.1 Implementation of PAL for Target Platform

The task of bringing-up a new Platform ("target platform") for an already supported MCU is explained in general within this section and furthermore by the example of a

platform based on the AT91SAM7X256 in section 10.3.2. The same steps are to be performed for all other platforms or boards respectively that shall be used.

10.3.1.1 Phase1: General Preparation for Target Platform

The first phase is a simple preparation phase. It provides the required directory and file structure for the target platform/board and defines proper build switches required for this target platform.

- Step 1: Identify the MCU for the target board
 - Each supported MCU is identified by a specific value of the build switch PAL_TYPE (and is part of a specific MCU family specified by the build switch PAL_GENERIC_TYPE).
 - For a list of all currently supported MCUs with a given MCU family check file PAL/Inc/pal_types.h for defined values of PAL_TYPE for each MCU family.
 - For more information see also section 9.2.
- Step 2: Add the target board to the selected MCU (i.e. PAL_TYPE) in the corresponding file PAL/MCU_FAMILY_NAME/MCU_NAME/Boards/pal_boardtypes.h. As an example for a new board for the AT81SAM7X256, add the target board to file PAL/ARM7/AT91SAM7X256/Boards/pal_boardtypes.h (see 10.3.2).
- Step 3: Identify an already supported board that best fits the target board to start the porting ("base platform").
 - Each supported board based on a given MCU is identified by a specific value of the build switch BOARD_TYPE.
 - For a list of all currently supported boards check file PAL/MCU_FAMILY_NAME/MCU_NAME/Boards/pal_boardtypes.h for defined values of BOARD_TYPE.
 - For more information see also section 9.4.
- Step 4: Copy the PAL board directory of the base platform in a separate directory within the same board directory (of the target MCU) and name it according to the target platform.
- Step 5: Rename all occurrences of base platform to target platform.

10.3.1.2 Phase2: Actual Porting to Target Platform

This section describes the actual porting phase once the directory and file structure of the target platform has been established.

The board directory of the target platform contains currently three files (same as for the base platform):

- pal_board.c
- pal_irq.c
- pal_config.h

Within the next phase all hardware resources need to be adjusted from the base platform to fit the resources of the target platform, such as timers, IRQs, ports, LEDs, buttons, ports and registers for SIO support, etc. This is explained in the subsequent steps.

All of these files are now adapted to the target platform needs step by step.

- Step 6: File pal_irq.c

This source file contains functions to initialize, enable, disable and install handler for the transceiver interrupts. It needs to be updated to match the requirements of the target





board when handling the transceiver interrupts (see examples in section 10.3.2.6 and in section 10.4.2.8).

- Step 7: File pal_board.c

This source file contains board specific functions to initialize and handle peripherals (such as LEDs, buttons, GPIO to the transceiver). It needs to be updated to match the requirements of the target board when handling these peripherals (see 10.3.2.7).

- Step 8: File pal_config.h

This header file contains configuration parameters for the target platform such as CPU frequency for this particular board, IRQ pins, pins between transceiver and MCU, LED pins, button pins, timer clock source definitions, debug macros, etc. It needs to be updated to match the requirements of the target board (see example in section 10.3.2.8).

10.3.2 Example Implementation of PAL for AT91SAM7X256 based Platform

This section describes the porting activities explained in the previous sections in general more specifically for the example of porting an existing software package based on the AT91SAM7X256 ARM 7 MCU to the new target board AT91SAM7X-EK with Radio Extender board REB231 V4.0 on REX_ARM adapter Revision 3 (REB_4_0_2_REX_ARM_REV_3) in order to support the AT86RF231 transceiver on this MCU.

10.3.2.1 Step 1 - Identify the MCU of the new Board

The target board is based on the MCU AT91SAM7X256. It is a member of the ARM7 family. Therefore new code of the target board is based on the PAL_GENERIC_TYPE=ARM7 and PAL_TYPE= AT91SAM7X256.

10.3.2.2 Step 2 - Add the new Board to File pal_boardtypes.h

To add support for the target board “AT91SAM7X-EK with Radio Extender board REB231 V4.0 on REX_ARM adapter Revision 3”, open file PAL/ARM7/AT91SAM7X256/Boards/pal_boardtypes.h and add the target board definition to the proper section of this file. Since the REB_4_0_2_REX_ARM_REV_3 is based on the transceiver AT86RF231, the new target board shall be added to the section “Boards for AT86RF231” where all AT86RF231 based boards for this MCU are listed. Use a not existing unique value for the target board.

Example:

```
/* Boards for AT86RF230B */
#define REB_2_3_REB_TO_SAM7EK          (0x01)
#define REB_2_3_REX_ARM_REV_2         (0x02)

/* Boards for AT86RF231 */
#define REB_4_0_2_REB_TO_SAM7EK3      (0x11)
/* AT91SAM7X-EK and AT91SAM7XC-EK boards with Radio Extender board
REB231 V4.0 on REX_ARM adapter Revision 3 */
#define REB_4_0_2_REX_ARM_REV_3       (0x12)
```

The actual numerical value of each define is not important, as long as each board within a specific PAL_TYPE has a unique value.

10.3.2.3 Step 3 - Identify an already supported Board that best fits the new Board as base Platform

From all currently supported boards based on the AT91SAM7X256 MCU the board “REB_2_3_REX_ARM_REV_2” (AT91SAM7X-EK boards with Radio Extender board REB230B V2.3 on REX_ARM adapter Revision 2) is selected as base platform to start the porting to the target platform “REB_4_0_2_REX_ARM_REV_3”.

10.3.2.4 Step 4 - Copy the Board directory of the base Platform

Example:

Copy the entire board directory

PAL/ARM7/AT91SAM7X256/Boards/REB_2_3_REX_ARM_REV_2

and rename it to directory

PAL/ARM7/AT91SAM7X256/Boards/REB_4_0_2_REX_ARM_REV_3.

Your directory PAL/ARM7/AT91SAM7X256/Boards now contains the following entries:

```
REB_2_3_REX_ARM_REV_2
...
REB_4_0_2_REX_ARM_REV_3
...
```

The directory REB_4_0_2_REX_ARM_REV_3 currently contains the identical code from the directory REB_2_3_REX_ARM_REV_2 (from the base platform). It comprises of the following files:

```
pal_board.c
pal_irq.c
pal_config.h
```

10.3.2.5 Step 5 - Rename all occurrences of base Platform to target Platform

Within the entire directory

PAL/ARM7/AT91SAM7X256/Boards/REB_4_0_2_REX_ARM_REV_3

all occurrences of the string “REB_2_3_REX_ARM_REV_2” are searched and replaced by the target Platform “REB_4_0_2_REX_ARM_REV_3”. In this example this needs to be done in the following files:

- pal_board.c
- pal_irq.c
- pal_config.h

10.3.2.6 : Step 6 - File pal_irq.c

This file contains functions to initialize, enable, disable and install handler for the transceiver interrupts. In this porting example it is a file dedicated to a board utilizing the transceiver AT86RF231, but is derived from a board utilizing the transceiver AT86RF230B.

While the AT86RF230B provides only one transceiver interrupt, the AT86RF231 provides usually two transceiver interrupts. This approach is followed for most boards for the AVR and Xmega MCU family.

Since the AT91SAM7X256 provides on a limited number of external interrupts, this approach is not followed in this example. The AT86RF231 is only used with one transceiver interrupt enabled. The Timestamp interrupt (based on DIG2 pin from the transceiver) is not used. Timestamping is done similar for AT86RF230B systems.



Because of this limitation for this particular board, no changes within file pal_irq.c during the porting from the base platform to the target platform are to be done (except the board name change explained in section 10.3.2.5).

10.3.2.7 Step 7 - File pal_board.c

This source file contains board specific functions to initialize and handle peripherals (such as LEDs, buttons, GPIO to the transceiver).

The following items within this file need to be updated according to the target board's needs or at least checked for correctness.

- Function pal_generate_rand_seed()
 - Each board needs a random seed value for
 - The random seed for the CSMA-CA algorithm
 - The generation of a random IEEE address for demo applications in case no valid IEEE address is provided for this board (e.g. by means of an external EEPROM)
 - While the AT86RF230B does not provide a random number generator (and thus this random seed needs to be generated for the corresponding board by different means), the AT86RF231 provides such a feature that is automatically enabled within the TAL for AT86RF231.
 - This implies the mechanism used for the base platform to generate a random seed by utilizing the ADC of the MCU, is not required for the target platform.
 - Therefore function pal_generate_rand_seed() can be removed entirely for this platform.
- Functions
 - adc_irq_handler()
 - adc_get_data()
 - adc_is_channel_irq_status_set()
 - adc_initialize()
 - These functions are helper functions for the obsolete function pal_generate_rand_seed() and can be removed completely as well.
- Function timer_init_non_generic()
- Function trx_interface_init()
 - While the board of the base platform connects the transceiver to the MCU via SPI0, the target platform uses SPI1. The pins of the AT91SAM7X256 used for SPI1 are still controlled via PIO-A, but used as "Peripheral A". For more information about the used pins for the SPI see 10.3.2.8. In order to use SPI1, change

```
/**  
 * @brief Initializes the transceiver interface  
 *  
 * This function initializes the transceiver interface.  
 * This board uses SPI0.  
 */  
void trx_interface_init(void)  
{
```

```

/*
...
* Peripheral A.
*/
AT91C_BASE_PIOA->PIO_ASR = (MISO | MOSI | SCK);
AT91C_BASE_PIOA->PIO_PDR = (MISO | MOSI | SCK);

AT91C_BASE_PIOA->PIO_ASR = TRX_INTERRUPT_PIN;
AT91C_BASE_PIOA->PIO_PDR = TRX_INTERRUPT_PIN;
...
/* Set SEL as output pin. */
AT91C_BASE_PIOA->PIO_OER = SEL;
AT91C_BASE_PIOA->PIO_PER = SEL;

/*
* Used peripheral interface is SPI0.
* The clock to the utilized SPI 0 peripheral is enabled.
*/
AT91C_BASE_PMC->PMC_PCER = _BV(AT91C_ID_SPI0);
to

/***
* @brief Initializes the transceiver interface
*
* This function initializes the transceiver interface.
* This board uses SPI1.
*/
void trx_interface_init(void)
{
/*
...
* Peripheral B.
*/
AT91C_BASE_PIOA->PIO_BSR = (MISO | MOSI | SCK);
AT91C_BASE_PIOA->PIO_PDR = (MISO | MOSI | SCK);

AT91C_BASE_PIOA->PIO_ASR = TRX_INTERRUPT_PIN;
AT91C_BASE_PIOA->PIO_PDR = TRX_INTERRUPT_PIN;
...
/* Set SEL as output pin. */
AT91C_BASE_PIOA->PIO_OER = SEL;
AT91C_BASE_PIOA->PIO_PER = SEL;

/*
* Used peripheral interface is SPI1.
* The clock to the utilized SPI 1 peripheral is enabled.
*/
AT91C_BASE_PMC->PMC_PCER = _BV(AT91C_ID_SPI1);

```





10.3.2.8 Step 8 - File pal_config.h

This header file contains configuration parameters for the target platform such as CPU frequency for this particular board, IRQ pins, pins between transceiver and MCU, LED pins, button pins, timer clock source definitions, debug macros, etc.

The following items within this file need to be updated according to the target board's needs or at least checked for correctness.

- Enum definition for PIOs, LEDs, and buttons: Remain unchanged
- Clock frequency selection (F_CPU) and corresponding defines: Remain unchanged
- Mapping of TRX IRQs to MCU pins (TRX_INTERRUPT_PIN)
 - This defines the interrupt pin used for the transceiver interrupt and is derived from the actual pin on the MCU where the transceiver interrupt line is connected to
 - In case the transceiver interrupt is routed to a different MCU pin (providing interrupt handling) this interrupt vector needs to be updated too
 - Remain unchanged
- Configuration of Advanced Interrupt Controller (AIC_CPNFIGURE()): Remain unchanged
- IRQ Macros: Change the comments

```
/*
 * AT86RF230B:
 *
 * TRX_MAIN_IRQ_HDLR_IDX
 *           TRX interrupt mapped to MCU IRQ0 pin and TIOA line of
 *           timer channel 0
 * TRX_TSTAMP_IRQ_HDLR_IDX
 *           Not used
 */
```

to

```
/*
 * AT86RF231:
 *
 * TRX_MAIN_IRQ_HDLR_IDX
 *           TRX interrupt mapped to MCU IRQ0 pin and TIOA line of
 *           timer channel 0
 * TRX_TSTAMP_IRQ_HDLR_IDX
 *           Time stamping interrupt, not used for this board,
 *           since the DIG2 pin of the 231 transceiver is not routed
 *           as an additional interrupt to the MCU.
 *           Timestamping is done similar to the 230 implementation.
 *           Make sure that the build switch "DISABLE_TSTAMP_IRQ"
 *           is set in the corresponding project files.
 */
```

- Number of transceiver interrupts (NO_OF_TRX_IRQS):
This is set to 1 for all AT86RF230B platforms and remains unchanged. Add the following comment:

```
/* Number of used TRX IRQs in this implementation */
```

```

/*
 * Even if the 231 transceiver generally provides
 * an additional interrupt for convenient timestamping,
 * it is not used for this board.
 */

```

```
#define NO_OF_TRX_IRQS (1)
```

- Macros for handling transceiver interrupts (enabling, disabling, clearing of transceiver interrupts): Remain unchanged
 - Macro for critical region with respect to transceiver interrupts: Remain unchanged
 - PAL_USE_SPI_TRX: Remains unchanged, since AT91SAM7X256 uses SPI as interface to the transceiver
 - SPI registers and pins: In case the transceiver is connected differently than on the base platform, these SPI registers and pins used on the MCU need to be updated.
- Change:

```

/*
 * SPI Base Register:
 * SPI0 is used with REX ARM Rev. 2.
 */
#define AT91C_BASE_SPI_USED (AT91C_BASE_SPI0)
/* RESET pin is pin 9 of PIOA. */
#define RST (AT91C_PIO_PA9)
/* Sleep Transceiver pin is pin 8 of PIOA. */
#define SLP_TR (AT91C_PIO_PA8)
/*
 * Slave select pin is PA14
 */
#define SEL (AT91C_PA14_SPI0_NPCS2)
/*
 * SPI Bus Master Output/Slave Input pin is PA17
 */
#define MOSI (AT91C_PA17_SPI0_MOSI)
/*
 * SPI Bus Master Input/Slave Output pin is PA16
 */
#define MISO (AT91C_PA16_SPI0_MISO)
/*
 * SPI serial clock pin is PA18
 */
#define SCK (AT91C_PA18_SPI0_SPCK)

```

to

```

/*
 * SPI Base Register:
 * SPI1 is used with REX ARM Rev. 3.
 */
#define AT91C_BASE_SPI_USED (AT91C_BASE_SPI1)
/* RESET pin is pin 9 of PIOA. */
#define RST (AT91C_PIO_PA9)

```





```
/* Sleep Transceiver pin is pin 8 of PIOA. */
#define SLP_TR          (AT91C_PIO_PA8)
/*
 * Slave select pin is PA21
 */
#define SEL             (AT91C_PA21_SPI1_NPCS0)
/*
 * SPI Bus Master Output/Slave Input pin is PA23
 */
#define MOSI            (AT91C_PA23_SPI1_MOSI)
/*
 * SPI Bus Master Input/Slave Output pin is PA24
 */
#define MISO            (AT91C_PA24_SPI1_MISO)
/*
 * SPI serial clock pin is PA22
 */
#define SCK              (AT91C_PA22_SPI1_SPCK)
```

in order to reflect the utilization of SPI as transceiver interface.

- TRX GPIO pins: Remain unchanged
- Short PAL waiting delays (PAL_WAIT_65_NS(),PAL_WAIT_500_NS, PAL_WAIT_1_US): Remains unchanged
- Timeout macros (MIN_TIMEOUT, MAX_TIMEOUT, MIN_DELAY_VAL): Remains unchanged since this platform uses the same timer implementation as the base platform
- Timer source macros:
 - TIMER_SRC_DURING_TRX_AWAKE
 - TIMER_SRC_DURING_TRX_SLEEP
 - These macros specify which timer source is used on this system when the transceiver is awake or sleeping
 - Remains unchanged for this platform since the same timer sources are used as for the base platform
- TIME_STAMP_REGISTER: Remains unchanged, since the target platform uses the same register for time stamping as the base platform
- TRX Access macros for SPI: Remain unchanged
- LED pins and joystick: Remains unchanged
- Alert initialization and indication macros: Remain unchanged since the same ports for the LEDs are used as for the base platform
- ADC Initialization values: Remove completely since ADC is not used for random number generation

10.3.3 Bring-up of an existing (MAC) Application on the Target Platform

The task of bringing-up an already existing application on the new target platform is explained in this section by the example of a target platform “REB_4_0_2_REX_ARM_REV_3” based on the ARM7 MCU AT91SAM7X256. A similar example based on the MCU ATxmega256A3 during the course of bringing-up a new MCU is explained in detail in section 10.5.

The same steps are to be performed for all other target platforms/boards that shall be used. Each step is first described generally and then specifically described in detail for porting the existing code from the base platform “REB_2_3_REX_ARM_REV_2” (using AT86RF230B with AT91SAM7X256) to the target platform “REB_4_0_2_REX_ARM_REV_3” using AT86RF231 with AT91SAM7X256.

- Step 1: Identify the application that shall be ported. This requires that all peripherals required by the application need to be supported (such as SIO support, LEDs, buttons). For more information about the requirements of existing applications see section 8.2.
- Step 2: Identify the best matching base platform already supported within this application. Duplicate the directory of the base platform within this application and rename it according to your target platform.
- Step 3: Update the GCC Makefile. Independent from whether the target application is built from command line using `make` or AVRStudio project file (APS-files), which also use external Makefiles themselves, the Makefile needs to be updated to cope with the target platform. This includes
 - Build specific properties such as
 - TAL_TYPE
 - PAL_TYPE
 - PAL_GENERIC_TYPE
 - BOARD_TYPE
 - The MCU type
 - The selected SIO channel in case stream I/O is used within the particular application
 - If the build switch SIO_HUB is set in the Makefile (-DSIO_HUB), also one of the following (currently supported) SIO channels needs to be enabled as well:
-DUART0
-DUART1
-DUSBO
 - If the build switch SIO_HUB is not set, no further SIO channel needs to be defined
 - Other specific build and link options if required
- Step 4: Update the IAR project files. Usually all required changes for the IAR Workbench are to done in the `ewp` file. This includes changing of the paths for includes and source files referring to the proper MCU. Also the proper MCU needs to be selected within the options for this project. Other changes may include updating the proper SIO channel, etc.

10.3.3.1 Step 1 - Identify the Application to be ported to the Target Platform

The MAC application Star_Nobeacon shall be enabled on the target board AT91SAM7X-EK with Radio Extender board REB231 V4.0 on REX_ARM adapter Revision 3 (REB_4_0_2_REX_ARM_REV_3). For more information about the platform bring-up for this target platform see the previous sections. This MAC application is located in directory

Application/MAC_Examples/Star_Nobeacon.





10.3.3.2 Step 2 - Identify the best matching Base Platform

The best matching base platform for the target board AT91SAM7X-EK with Radio Extender board REB231 V4.0 on REX_ARM adapter Revision 3 (REB_4_0_2_REX_ARM_REV_3) is the AT91SAM7X-EK with Radio Extender board REB230B V2.3 on REX_ARM adapter Revision 2 (REB_2_3_REX_ARM_REV_2). This entire build files for this particular base platform are located in directory

AT86RF230B_AT91SAM7X256_REB_2_3_REX_ARM_REV_2

within the application directory

Applications/MAC_Examples/Star_Nobeacon/.

The entire directory

Applications/MAC_Examples/Star_Nobeacon/

AT86RF230B_AT91SAM7X256_REB_2_3_REX_ARM_REV_2

is copied and renamed

Applications/MAC_Examples/Star_Nobeacon/

AT86RF231_AT91SAM7X256_REB_4_0_2_REX_ARM_REV_3

Usually the directory name for applications consists of the following items to uniquely identify the target platform:

- TAL_TYPE (= AT86RF231)
- PAL_TYPE (= AT91SAMX256)
- PAL_GENERIC_TYPE (= ARM7)
- BOARD_TYPE (= REB_4_0_2_REX_ARM_REV_3)

Attention: Make sure that the build switch “HIGHEST_STACK_LAYER” is not changed. This build switch always needs to reflect the proper highest stack layer that the application is residing on. In case of a MAC application (such as the Star_Nobeacon) the application is residing on top of the MAC, so “HIGHEST_STACK_LAYER” needs to be MAC. If this is not properly set, this leads to undefined behavior during the build process or during application usage.

10.3.3.3 Step 3 – Update the GCC Makefile

Currently the GCC build is not supported for ARM7 based boards. For an example how to port applications for MCU families with GCC support within this software package (such as AVR or Xmega MCUs) see section 10.4.3.3.

10.3.3.4 Step 4 – Update the IAR project files

The IAR project files for an ARM7 based project (Star.ewd, Star.ewp, Star.eww, and flash.icf) of the target board which were duplicated from the base platform/board are now located in the application build directory of the target platform, i.e.

Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_AT91SAM7X256_REB_4_0_2_REX_ARM_REV_3.

In order to support the target platform the file Star.ewp needs to be updated as follows:

- Change the TAL_TYPE from

<state>TAL_TYPE=AT86RF230B</state>

to

<state>TAL_TYPE=AT86RF231</state>

- Change the BOARD_TYPE from


```
<state>BOARD_TYPE=REB_2_3_REX_ARM_REV_2</state>
```

 to


```
<state>BOARD_TYPE=REB_4_0_2_REX_ARM_REV_3</state>
```
- Since the target board does not use the Timestamp interrupt from the AT86RF231 add the following build option


```
<state>DISABLE_TSTAMP_IRQ</state>
```
- Change all occurrences of


```
REB_2_3_REX_ARM_REV_2
```

 to


```
REB_4_0_2_REX_ARM_REV_3
```
- Change all occurrences of


```
AT86RF230B
```

 to


```
AT86RF231
```

Since the MCU type is note changed, the build switches PAL_GENERIC_TYPE and PAL_TYPE remain unchanged.

10.4 Bring-up of a new MCU based on a supported MCU Family

10.4.1 Implementation of PAL for Target Platform

The task of bringing-up a new MCU (“target MCU”) within an already supported MCU family is explained in general within this section and furthermore by the example of a platform based on the ATxmega256A3 in section 10.4.2. The same steps are to be performed for all other MCUs that shall be used.

10.4.1.1 Phase1: General Preparation for Target Platform

The first phase is a simple preparation phase. It provides the required directory and file structure for the target MCU and defines proper build switches required for the target platform.

- Step 1: Identify the MCU family for the target MCU
 - Each supported MCU family is identified by a specific value of the build switch PAL_GENERIC_TYPE.
 - For a list of all currently supported MCU families check file PAL/Inc/pal_types.h for defined values of PAL_GENERIC_TYPE.
 - For more information see also section 9.1.
- Step 2: Add the target MCU to the selected PAL_GENERIC_TYPE in file PAL/Inc/pal_types.h.
- Step 3: Identify an already supported MCU that best fits the target MCU to start the porting (“base MCU”).
 - Each supported MCU is identified by a specific value of the build switch PAL_TYPE.
 - For a list of all currently supported MCU check file PAL/Inc/pal_types.h for defined values of PAL_TYPE within the selected PAL_GENERIC_TYPE.
 - For more information see also section 9.2.





- Step 4: Copy the PAL directory of the base MCU in a separate directory within the same MCU family and name it according to the target MCU.
- Step 5: Identify an already existing board type ("base board") within the directory of the target MCU that best fits the new board to be supported ("target board").
 - Each supported board with the newly created directory contains a "board" directory including file pal_boardtypes.h and least one specific board directory.
 - Select the base board for the target board.
 - Other boards within target board directory that are obsolete may be removed.
- Step 6: Rename the target board (optional).
 - In case the selected target board shall to be renamed, file pal_boardtypes.h with in target MCU directory needs to be updated with the new board type. Also the target directory needs to be renamed matching the new board type. For more information see section 10.3.
- Step 7: Rename all occurrences of base MCU to target MCU.

10.4.1.2 Phase2: Actual Porting to Target Platform

This section describes the actual porting phase once the directory and file structure of the target platform has been established.

The directory of the target platform contains currently three directories (same as for the base platform):

- Boards
- Inc
- Src

Within the next phase all hardware resources need to be adjusted from the base platform to fit the resources of the target platform, such as timers, IRQs, ports, LEDs, buttons, ports and registers for SIO support, etc. This is explained in the subsequent steps.

The main changes for a new platform need to be done in directory Boards in the subdirectory for the target board. In the current example change to directory

PAL/XMEGA/ATXMEGA256A3/Boards/REB_4_1_STK600.

In this directory the following three files are located:

- pal_board.c
- pal_irq.c
- pal_config.h

All of these files are now adapted to the target platform needs step by step.

- Step 8: File pal_irq.c

This source file contains functions to initialize, enable, disable and install handler for the transceiver interrupts. It needs to be updated to match the requirements of the target board when handling the transceiver interrupts (see example in section 10.4.2.8).

- Step 9: File pal_board.c

This source file contains board specific functions to initialize and handle peripherals (such as LEDs, buttons, GPIO to the transceiver). It needs to be updated to match the requirements of the target board when handling these peripherals.

- Step 10: File pal_config.h

This header file contains configuration parameters for the target platform such as CPU frequency for this particular board, IRQ pins, pins between transceiver and MCU, LED pins, button pins, timer clock source definitions, debug macros, etc. It needs to be updated to match the requirements of the target board (see example in section 10.4.2.10).

- Step 11: File pal_sio_hub.c in directory Src

- The file pal_sio_hub.c within the Src directory of the target MCU is a source containing the hub functionality for all serial I/O related functionality. This included currently UART and/or USB.
- Depending on the available and utilized serial I/O peripherals on the target MCU (please check the corresponding data sheet of the target MCU) and board this file needs to be updated.

10.4.2 Example Implementation of PAL for ATxmega256A3

This section describes the porting activities explained in the previous sections in general more specifically for the example of porting an existing software package to the new target MCU ATxmega256A3 (on the STK600 board with REB to STK600 Adapter and Radio Extender board REB231ED V4.1.1).

10.4.2.1 Step 1 - Identify the MCU Family of the new MCU

The target MCU is the ATxmega256A3. It is a member of the ATxmega AVR family. Therefore new code of this MCU is based on the PAL_GENERIC_TYPE=XMEGA.

10.4.2.2 Step 2 - Add the new MCU to File pal_types.h

To add support for ATxmega256A3, open file PAL/Inc/pal_types.h and add the ATxmega256A3 to the section where all ATxmega MCUs are listed. Use a non-existing unique value for the target MCU.

Example:

```
#elif (PAL_GENERIC_TYPE == XMEGA)
    /* PAL_TYPE for XMEGA MCUs */
#define ATXMEGA128A1                (0x01)
#define ATXMEGA256A3                (0x02)

#endif
```

For better readability the target MCU has been added to reflect the alphabetical order of the MCU directory name. Also the existing values of the defines have been updated. The actual numerical value of each define is not important, as long as each MCU within a specific PAL_GENERIC_TYPE has a unique value.

10.4.2.3 Step 3 - Identify an already supported MCU that best fits the new MCU as base MCU

From all currently supported MCUs within the ATxmega family the ATXMEGA128A1 is selected as base MCU to start the porting to the target MCU ATXMEGA256A3.

10.4.2.4 Step 4 - Copy the PAL directory of the base MCU

Example:





Copy the entire directory PAL/XMEGA/ATXMEGA128A1 and rename it to directory PAL/XMEGA/ATXMEGA256A3. Your directory PAL/XMEGA now contains the following entries:

ATXMEGA128A1
ATxmega256A3
Generic

The directory ATxmega256A3 currently contains the identical code from the directory ATXMEGA128A1 (from the base MCU). It comprises of the following directories:

Boards
Inc
Src

10.4.2.5 Step 5 - Identify the Base Board best fitting the Target Board

Within the board directory of the target MCU (PAL/XMEGA/ATXMEGA256A3/Boards) a variety of subdirectories are existing. Please remove all directories except REB_4_1_STK600. Now the following entries are existing:

- Directory REB_4_1_STK600
- File pal_boardtypes.h

The directory REB_4_1_STK600 contains the board implementation for the transceiver AT86RF231 based on the board REB231ED V4.1.1 on top of an STK600 board with REB to STK600 Adapter. For more information about this platform see sections 9.4.2.4 and 9.4.3.1.

For simplicity reason it is assumed that the same platform is used, and only the MCU shall be replaced. For more information about how to bring-up a new board, see also section 10.3.

10.4.2.6 Step 6 - Rename the target board

The file pal_boardtypes.h contains entries inherited form the base platform. All obsolete entries (except of REB_4_1_STK600) may be removed if desired. In this example all entries are kept since this allows for the easy extension of the boards later.

10.4.2.7 Step 7 - Rename all occurrences of base MCU to target MCU

Within the entire directory PAL/XMEGA/ATXMEGA256A3 all occurrences of the string "ATXMEGA128A1" are searched and replaced by the target MCU "ATXMEGA256A3". In this example this needs to be done in the following files:

- PAL/XMEGA /ATXMEGA256A3/Boards/pal_boardtypes.h
- PAL/XMEGA /ATXMEGA256A3/Boards/REB_4_1_STK600/pal_config.h

10.4.2.8 Step 8 - File pal_irq.c

Since the target platform is close very close to the base platform, no changes need to be done in this file.

10.4.2.9 Step 9 - File pal_board.c

Since the target platform is close very close to the base platform, no changes need to be done in this file.

10.4.2.10 Step 10 - File *pal_config.h*

Since the target platform is close very close to the base platform, no changes need to be done in this file.

10.4.2.11 Step 11 - File *pal_sio_hub.c* in directory *Src*

No changes need to be done in this file.

10.4.3 Bring-up of an existing Application on the Target Platform

The task of bringing-up an already existing application on a new target platform is explained by the example of a platform based on the ATxmega256A3. The same steps are to be performed for all other MCUs that shall be used. Each step is first described generally and then specifically described in detail for porting the existing code to the ATxmega256A3.

Please note that the opposite task of bringing up a new application for an existing platform is explained in section 10.5.

- Step 1: Identify the application that shall be ported. This requires that all peripherals required by the application need to be supported (such as SIO support, LEDs, buttons). For more information about the requirements of existing applications see section 8.2.
- Step 2: Identify the best matching base platform already supported within this application. Duplicate the directory of the base platform within this application and rename it according to your target platform.
- Step 3: Update the GCC Makefile. Independent from whether the target application is built from command line using `make` or AVRStudio project file (APS-files), which also use external Makefiles themselves, the Makefile needs to be updated to cope with the target platform. This includes
 - Build specific properties such as
 - TAL_TYPE
 - PAL_TYPE
 - PAL_GENERIC_TYPE
 - BOARD_TYPE
 - The MCU type
 - The selected SIO channel in case stream I/O is used within the particular application
 - If the build switch SIO_HUB is set in the Makefile (-DSIO_HUB), also one of the following (currently supported) SIO channels needs to be enabled as well:
-DUART0
-DUART1
-DUSBO
 - If the build switch SIO_HUB is not set, no further SIO channel needs to be defined
 - Other specific build and link options if required
- Step 4: Update the IAR project files. Usually all required changes for the IAR Workbench are to done in the ewp file. This includes changing of the paths for includes and source files referring to the proper MCU. Also the proper MCU needs to be selected within the options for this project. Other changes may include updating the proper SIO channel, etc.





- Step 5: Update the AVR Studio Project files (aps files). This includes changing of the paths for includes and source files referring to the proper MCU. All other items are already within the external Makefiles that are called during the build process.

10.4.3.1 Step 1 - Identify the Application to be ported to the Target Platform

The MAC application Star_Nobeacon shall be enabled on the target platform ATxmega256A3 MCU based on REB231ED V4.1.1 on top of an STK600 board. For more information about the platform bring-up for this target platform see the previous sections. The MAC or TAL must not be changed, since these layers residing on top of the PAL are completely independent from the selected platform. This MAC application is located in directory

Application/MAC_Examples/Star_Nobeacon.

10.4.3.2 Step 2 - Identify the best matching Base Platform

The best matching base platform for an ATxmega256A3 placed on the STK600 board with REB to STK600 Adapter and Radio Extender board REB231ED V4.1.1 (with AT86RF231) is the ATxmega128A1 on the same hardware setup (STK600 board with REB to STK600 Adapter and Radio Extender board REB231ED V4.1.1, see section 9.4.2.4 and 9.4.3.1). This entire build files for GCC and IAR for this particular base platform are located in directory

AT86RF231_ATXMEGA128A1_REB_4_1_STK600

within the application directory

Applications/MAC_Examples/Star_Nobeacon/.

The entire directory

Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA128A1_REB_4_1_STK600

is copied and renamed to

Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600

Usually the directory name for applications consists of the following items to uniquely identify the target platform:

- TAL_TYPE (= AT86RF231)
- PAL_TYPE (= ATXMEGA128A1)
- PAL_GENERIC_TYPE (= XMEGA)
- BOARD_TYPE (= REB_4_1_STK600)

Attention: Make sure that the build switch "HIGHEST_STACK_LAYER" is not changed. This build switch always needs to reflect the proper highest stack layer that the application is residing on. In case of a MAC application (such as the Star_Nobeacon example) the application is residing on top of the MAC, so "HIGHEST_STACK_LAYER" needs to be MAC. If this is not properly set, this leads to undefined behavior during the build process or during application usage.

10.4.3.3 Step 3 – Update the GCC Makefile

The Makefile of the target platform which was duplicated from the base platform is now located in the GCC directory of the target platform, i.e.

Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.

In order to support the target platform this Makefile needs to be updated as follows:

- Update the build specific properties from

```
_TAL_TYPE = AT86RF231
_PAL_TYPE = ATXMEGA128A1
_PAL_GENERIC_TYPE = XMEGA
_BOARD_TYPE = REB_4_1_STK600

to

_TAL_TYPE = AT86RF231
_PAL_TYPE = ATXMEGA256A3
_PAL_GENERIC_TYPE = XMEGA
_BOARD_TYPE = REB_4_1_STK600
```

In the described example only the PAL_TYPE needs to be changed. Make sure that the PAL_TYPE (i.e. the variable in the Makefile) is set identical to the target directory in the PAL that was created for the new platform (see sections 10.4.2.2 and 10.4.2.4), since the PAL_TYPE is used within directory names.

- Update the MCU type from

```
MCU = atxmega128a1

to
```

```
MCU = atxmega256a3
```

- Updating the SIO channel (e.g. UART1, etc.) is not required here, since this application does not use SIO functionality.

10.4.3.4 Step 4 – Update the IAR project files

The IAR project files (Star.eww and Star.ewp) of the target platform which were duplicated from the base platform are now located in the application build directory of the target platform, i.e.

```
Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.
```

In order to support the target platform the file Star.ewp needs to be updated as follows:

- Select the proper MCU (ATxmega256A3) for this project within IAR Workbench.
- Correct the proper used SIO channel is not required for here, since this application does not use SIO functionality. This can be done using an XML editor or within IAR Workbench.
- Change all occurrences of “ATXMEGA128A1”, which are used as path names for directories and files to “ATXMEGA256A3”. This can be done using an XML editor, a regular text editor, or within IAR Workbench.

10.4.3.5 Step 5 - Update the AVR Studio Project files

The AVR Studio project file (i.e. aps file) of the target platform which were duplicated from the base platform are now located in the application build directory of the target platform, i.e.

```
Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.
```

In order to support the target platform the file Star.aps needs to be updated as follows:

- Change all occurrences of “ATXMEGA128A1”, which are used as path names for directories and files to “ATXMEGA256A3”. This can be done using an XML editor or a regular text editor.



10.5 Bring-up of a new Application on an existing Platform

The task of bringing-up a new application on an existing platform is explained by the MAC example application Promiscuous_Mode_Demo for a supported platform based on the ATxmega256A3. The same steps are to be performed for all other MCUs that shall be used, or all other applications respectively. Each step is first described generally and then specifically explained in detail for bringing-up the Promiscuous_Mode_Demo.

Please note that the opposite task of bringing up a new platform on an existing application is explained in section 10.4.3.

- Step 1: Identify a matching base application already supported for this platform. Duplicate the directory of the base application for this platform to the corresponding target platform.
- Step 2: Update the GCC Makefiles. Independent from whether the target application is built from command line using `make` or AVRStudio project file (APS-files), which also use external Makefiles themselves, the Makefile needs to be updated to cope with the target application. This includes the following steps:
 - Replace occurrences of the name of base application with the target application (e.g. replace "Star" with "Promiscuous_Mode_Demo").
 - Update the required path variables ("# Path variables")
 - Update the required compiler options ("CFLAGS"); this includes updating the required SIO handling switches
 - Update the required include directories ("INCLUDES")
 - Update the required list of object files for the linker ("OBJECTS")
 - Update the compile section for the source files ("## Compile")
- Step 3: Update the IAR project files. Usually all required changes for the IAR Workbench are to done in the ewp file. This includes updating of the paths for includes and updating of the required source files. Other changes may include updating the proper SIO channel, etc.
- Step 4: Update the AVR Studio Project files (aps files). This includes replacing occurrences of the name of base application with the target application (e.g. replace "Star" with "Promiscuous_Mode_Demo"). All other items are already within the external Makefiles that are called during the build process.

10.5.1 Step 1 - Identify a matching Base Application

The MAC application Promiscuous_Mode_Demo (target application) shall be enabled on the existing platform ATxmega256A3 MCU based on REB231ED V4.1.1 on top of an STK600 board. For more information about the platform bring-up for this target platform see section 10.4. This target application is located in directory

Application/MAC_Examples/Promiscuous_Mode_Demo.

A matching base application for this MAC application is the Star_Nobeacon application. The platform ATxmega256A3 MCU based on REB231ED V4.1.1 is already supported for this base application.

All required build files for GCC and IAR for this particular base application are located in directory

Applications/MAC_Examples/Star_Nobeacon/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600.

This entire directory is copied to the target application directory. Now the following directory exists:

Applications/MAC_Examples/Promiscuous_Mode_Demo/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600.

This directory now contains the following entries:

- GCC/Makefile
- GCC/Makefile_Debug
- Star.aps
- Star.ewp
- Star.eww

Now rename the project files as required for the target application, i.e. rename all files Star.* to Promiscuous_Mode_Demo.*.

10.5.2 Step 2 – Update the GCC Makefile

The Makefiles of the target application which was duplicated from the base application is now located in the GCC directory of the target application, i.e.

Applications/MAC_Examples/Promiscuous_Mode_Demo/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.

In order to support the target application these Makefiles need to be updated as follows:

- Replace occurrences of the name of base application “Star” with the name of the target application Promiscuous_Mode_Demo “Promiscuous_Mode_Demo”. This is required in comments and for specifying the generated output files (“PROJECT = ...”).
- Add the path for the generic SIO support functions (see section 8.3):
`PATH_SIO_SUPPORT = $(MAIN_DIR)/Applications/Helper_Files/SIO_Support`
- Enable SIO support for this application via the supported SIO channel. In this example UART1 is used. Change

```
CFLAGS += -Wall -Werror -g -Wundef -std=c99 -Os  
to
```

```
CFLAGS += -Wall -Werror -g -Wundef -std=c99 -DSIO_HUB -DUART1 -Os
```

- Update the used and/or unused compiler flags. The base example application (Star_Nobeacon) used the compiler switches FFD and REDUCED_PARAM_CHECK that are not required anymore for the target application. On the other hand the target application requires the build switch PROMISCUOUS_MODE. Therefore change

```
CFLAGS += -DDEBUG=0  
CFLAGS += -DFFD  
CFLAGS += -REDUCED_PARAM_CHECK  
CFLAGS += -DTAL_TYPE=$(TAL_TYPE)  
CFLAGS += -DPAL_GENERIC_TYPE=$(PAL_GENERIC_TYPE)  
CFLAGS += -DPAL_TYPE=$(PAL_TYPE)  
CFLAGS += -DBOARD_TYPE=$(BOARD_TYPE)  
CFLAGS += -DHIGHEST_STACK_LAYER=$(HIGHEST_STACK_LAYER)
```

to

```
CFLAGS += -DDEBUG=0  
CFLAGS += -DPROMISCUOUS_MODE  
CFLAGS += -DTAL_TYPE=$(TAL_TYPE)  
CFLAGS += -DPAL_GENERIC_TYPE=$(PAL_GENERIC_TYPE)  
CFLAGS += -DPAL_TYPE=$(PAL_TYPE)
```





```
CFLAGS += -DBOARD_TYPE=$(BOARD_TYPE)
CFLAGS += -DHIGHEST_STACK_LAYER=$(HIGHEST_STACK_LAYER)
```

- Update the used and/or unused include directories in the Makefile. The base example application (Star_Nobeacon) did not use SIO support, so this needs to be added to the target application in order to allow for the inclusion of sio_handler.h. Add the following include path by changing

```
## Include directories for application
INCLUDES = -I $(APP_DIR)/Inc
## Include directories for general includes
INCLUDES += -I $(MAIN_DIR)/Include
```

to

```
## Include directories for application
INCLUDES = -I $(APP_DIR)/Inc
## Include directories for SIO support
INCLUDES += -I $(PATH_SIO_SUPPORT)/Inc
## Include directories for general includes
INCLUDES += -I $(MAIN_DIR)/Include
```

- Update the used and/or unused object files to the list of OBJECTS in the Makefile. Since the Promiscuous_Mode_Demo application only uses three MAC callback functions itself (usr_mcps_data_ind(), usr_mlme_reset_conf(), and usr_mlme_set_conf()), all other callbacks need to be added as stubs by adding the following lines. For more information about MAC stub functions see 8.4. Change

```
## Objects that must be built in order to link
OBJECTS = $(TARGET_DIR)/main.o \
          $(TARGET_DIR)/pal_uart.o \
          . . .
          $(TARGET_DIR)/mac_api.o \
          $(TARGET_DIR)/usr_mcps_purge_conf.o \
          $(TARGET_DIR)/usr_mlme_beacon_notify_ind.o \
          $(TARGET_DIR)/usr_mlme_disassociate_conf.o \
          $(TARGET_DIR)/usr_mlme_disassociate_ind.o \
          $(TARGET_DIR)/usr_mlme_get_conf.o \
          $(TARGET_DIR)/usr_mlme_orphan_ind.o \
          $(TARGET_DIR)/usr_mlme_poll_conf.o \
          $(TARGET_DIR)/usr_mlme_rx_enable_conf.o \
          $(TARGET_DIR)/usr_mlme_sync_loss_ind.o
```

to

```
## Objects that must be built in order to link
OBJECTS = $(TARGET_DIR)/main.o \
          $(TARGET_DIR)/sio_handler.o \
          $(TARGET_DIR)/pal_uart.o \
          . . .
          $(TARGET_DIR)/mac_api.o \
          $(TARGET_DIR)/usr_mcps_data_conf.o \
          $(TARGET_DIR)/usr_mcps_purge_conf.o \
          $(TARGET_DIR)/usr_mlme_associate_conf.o \
```

```

$(TARGET_DIR)/usr_mlme_associate_ind.o \
$(TARGET_DIR)/usr_mlme_beacon_notify_ind.o \
$(TARGET_DIR)/usr_mlme_comm_status_ind.o \
$(TARGET_DIR)/usr_mlme_disassociate_conf.o \
$(TARGET_DIR)/usr_mlme_disassociate_ind.o \
$(TARGET_DIR)/usr_mlme_get_conf.o \
$(TARGET_DIR)/usr_mlme_orphan_ind.o \
$(TARGET_DIR)/usr_mlme_poll_conf.o \
$(TARGET_DIR)/usr_mlme_rx_enable_conf.o \
$(TARGET_DIR)/usr_mlme_scan_conf.o \
$(TARGET_DIR)/usr_mlme_start_conf.o \
$(TARGET_DIR)/usr_mlme_sync_loss_ind.o

```

- Update the used and/or unused list of files to be compiled. Change

```

## Compile
$(TARGET_DIR)/main.o: $(APP_DIR)/Src/main.c
    $(CC) $(INCLUDES) $(CFLAGS) -c -o $@ $<
$(TARGET_DIR)/pal_uart.o:
$(PATH_PAL)/$_PAL_GENERIC_TYPE/Generic/Src/pal_uart.c
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $<

to

## Compile
$(TARGET_DIR)/main.o: $(APP_DIR)/Src/main.c
    $(CC) $(INCLUDES) $(CFLAGS) -c -o $@ $<
$(TARGET_DIR)/sio_handler.o:
$(PATH_SIO_SUPPORT)/Src/sio_handler.c
    $(CC) $(INCLUDES) $(CFLAGS) -c -o $@ $<
$(TARGET_DIR)/pal_uart.o:
$(PATH_PAL)/$_PAL_GENERIC_TYPE/Generic/Src/pal_uart.c
    $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $<

```

10.5.3 Step 3 – Update the IAR project files

The IAR project files (Promiscuous_Mode_Demo.eww and Promiscuous_Mode_Demo.ewp) of the target application are now located in the directory

Applications/MAC_Examples/Promiscuous_Mode_Demo/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.

In order to support the target application these project files need to be updated as follows:

- Replace occurrences of the name of base application “Star” with the name of the target application Promiscuous_Mode_Demo “Promiscuous_Mode_Demo”. This is required in both project files.
- Enable SIO support for this application via the supported SIO channel. In this example UART1 is used. Also update the used and/or unused compiler flags. The base example application (Star_Nobeacon) used the compiler switches FFD and REDUCED_PARAM_CHECK that are not required anymore for the target application. On the other hand the target application requires the build switch PROMISCUOUS_MODE. In file Promiscuous_Mode_Demo.ewp change

<name>CCDefines</name>





```
<state>DEBUG=0</state>
<state>FFD</state>
<state>REDUCED_PARAM_CHECK</state>
<state>TAL_TYPE=AT86RF231</state>
<state>PAL_TYPE=ATXMEGA256A3</state>
<state>PAL_GENERIC_TYPE=XMEGA</state>
<state>BOARD_TYPE=REB_4_1_STK600</state>
<state>HIGHEST_STACK_LAYER=MAC</state>
```

to

```
<name>CCDefines</name>
<state>DEBUG=0</state>
<state>PROMISCUOUS_MODE</state>
<state>SIO_HUB</state>
<state>UART1</state>
<state>TAL_TYPE=AT86RF231</state>
<state>PAL_TYPE=ATXMEGA256A3</state>
<state>PAL_GENERIC_TYPE=XMEGA</state>
<state>BOARD_TYPE=REB_4_1_STK600</state>
<state>HIGHEST_STACK_LAYER=MAC</state>
```

- Update the used and/or unused include directories in file Promiscuous_Mode_Demo.ewp. The base example application (Star_Nobeacon) did not use SIO support, so this needs to be added to the target application in order to allow for the inclusion of sio_handler.h. Add the following include path by changing

```
<name>newCCIIncludePaths</name>
<state>$PROJ_DIR$\\..\\Inc</state>
<state>$PROJ_DIR$\\..\\..\\..\\Include</state>
```

to

```
<name>newCCIIncludePaths</name>
<state>$PROJ_DIR$\\..\\Inc</state>
<state>$PROJ_DIR$\\..\\..\\Helper_Files\\SIO_Support\\Inc</state>
<state>$PROJ_DIR$\\..\\..\\..\\Include</state>
```

- Update the used and/or unused list of source files. Add the following callback stub files in file Promiscuous_Mode_Demo.ewp (For more information about MAC stub functions see 8.4.)

```
<name>MAC_API</name>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\..\\MAC\\Src\\mac_api.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\..\\MAC\\Src\\usr_mcps_purge_conf.c</name>
</file>
```

```

<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_notify_ind.c
</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_disassociate_conf.c
</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_disassociate_ind.c<
/n
ame>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_get_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_orphan_ind.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_poll_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_rx_enable_conf.c<n
ame>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_sync_loss_ind.c</na
me>
</file>
to
<name>MAC_API</name>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\mac_api.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mcps_data_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mcps_purge_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_associate_conf.c<n
ame>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_associate_ind.c</na
me>
</file>
<file>
<name>$PROJ_DIR$\\..\\..\\..\\MAC\\Src\\usr_mlme_notify_ind.c
</name>

```





```
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_comm_status_ind.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_disassociate_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_disassociate_ind.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_get_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_orphan_ind.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_poll_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_rx_enable_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_scan_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_start_conf.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\..\MAC\Src\usr_mlme_sync_loss_ind.c</name>
</file>
```

- Add the sio_handler source files in file Promiscuous_Mode_Demo.ewp

```
<file>
<name>$PROJ_DIR$..\Src\main.c</name>
</file>
to
<file>
<name>$PROJ_DIR$..\Src\main.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\Helper_Files\SIO_Support\Src\sio_handler.c</name>
</file>
<file>
<name>$PROJ_DIR$..\..\..\Helper_Files\SIO_Support\IAR_Support\write.c</name>
</file>
```

</file>

10.5.4 Step 4 - Update the AVR Studio Project files

The AVR Studio project file (i.e. .aps file) of the target application is located in the directory of the target platform, i.e.

Applications/MAC_Examples/Promiscuous_Mode_Demo/
AT86RF231_ATXMEGA256A3_REB_4_1_STK600/GCC.

In order to support the target application the file Promiscuous_Mode_Demo.aps needs to be updated as follows:

- Replace all occurrences of the name of base application with the target application (e.g. replace “Star” with “Promiscuous_Mode_Demo”). All other items are already within the external Makefiles that are called during the build process.

10.6 Customizing the Platform Clock Speed

10.6.1 Customizing ATxmega128A1 Platforms

The following section describes how a specific ATxmega128A1 based platform can be customized to run at various clock speeds. The default clock speed for these platforms is currently 16MHz, but user may want to adapt this to their specific needs. Due to the platform abstraction implemented within the MAC Software package, this is very simple and straightforward.

10.6.1.1 Introduction to the ATxmega128A1 Platform Software Design

The system clock (i.e. the MCU clock frequency) controls a variety of blocks within the MCU.

The MCU for the ATxmega128A1 platform is always clocked by the 32MHz Internal RC Oscillator in conjunction with the corresponding System Clock Prescaler to derive the selected system clock.

The event system is always selected as source for the hardware timer used by software. Since the hardware timer requires a tick of 1µs (i.e. timer runs at speed of 1MHz), the event system channel used as timer source, is divided to the proper speed by means of the event system Prescaler.

The SPI speed is derived from the system clock with the constraint that the SPI speed in asynchronous mode (i.e. the MCU is not clocked by the CLKM from the transceiver, see the data sheet for the corresponding transceiver) must be smaller than 8MHz. This implies that the largest usable SPI speed is 4MHz.

So based on this design and depending on the selected clock speed the following hardware blocks need to be initialized accordingly:

- MCU clock setting
- Event System initialization and thus setting proper timer source
- SPI speed between MCU and transceiver
- UART baud rate

In order to change the implementation of the ATxmega128A1 platform and to select a specific system clock, the Platform Abstraction Layer (PAL) for this platform needs to be customized.





10.6.1.2 Customizing the ATxmega128A1 Platform Abstraction

The PAL for all ATxmega128A1 based platforms can be found in directory `MAC_v_x_y_z\PAL\XMEGA\ATXMEGA128A1`.

The directory `PAL\XMEGA` contains the code that is common to all ATxmega family based systems.

The directory `PAL\XMEGA\ATXMEGA128A1` contains the code that is common to all ATxmega128A1 MCU platforms.

The entire code that is dedicated to a specific hardware platform (based on a specific MCU **and** on a specific board with specific settings) is located in the board directory for this particular hardware configuration. All supported platforms (i.e. boards with specific settings) can be found in directory `PAL\XMEGA\ATXMEGA128A1\Boards`.

The selectable clock speed will be explained for the board **REB_4_1_STK600**, which implements the platform abstraction for a Radio Extender Board version 4.1 with AT86RF231 (with Antenna Diversity) placed on a STK600 (with ATxmega128A1). The explanations given for this board apply for each board configuration and can easily be adapted to other boards as well.

The directory for the specific board implementation for **REB_4_1_STK600** contains three files:

- `pal_board.c`
- `pal_irq.c`
- `pal_config.h`

Whenever a configuration parameter (such as system clock speed) is changed, only these files shall be adapted. All other files generally remain unchanged.

In the case of the selectable system clock speed, only `pal_board.c` and `pal_config.h` are customized.

10.6.1.2.1 MCU Clock Setting

Depending on the selected system clock, the MCU clock needs to be set. As described for ATxmega128A1 based platforms currently always the 32MHz Internal RC Oscillator is utilized. In order to gain the proper system clock, the corresponding Prescaler needs to be set.

The corresponding implementation can be found in file `pal_board.c` in function `clock_init()` which is called during PAL initialization.

10.6.1.3 Event System Initialization

The hardware timer used for this board is always driven by the event system. This is implemented in file `pal_board.c` in function `timer_init_non_generic()` by using the macro `TIMER_SRC_DURING_TRX_SLEEP()`. This macro is implemented in file `pal_config.h`. Currently always the Event Channel 0 is used as timer source.

Since the timer runs at 1MHz speed, the Event Channel 0 is configured to provide events at 1 μ s rate. This is implemented in file `pal_board.c` in function `event_system_init()` where the proper Prescaler for this particular Event Channel is set.

10.6.1.3.1 SPI Speed Selection

As already mentioned the SPI between the MCU and the transceiver needs to be initialized properly with the border condition that the SPI rate must not reach 8MHz. Thus the largest (trivially) selectable SPI speed is 4MHz.

The SPI is initialized in file *pal_config.h* in function macro `TRX_INIT()`. This macro is called by function `trx_interface_init()` (see file *pal_trx_access.c* for the ATxmega family).

10.6.1.3.2 Small Blocking Delays

In order to comply with the data sheet for the transceivers and the corresponding Software Programming Model, certain very small delays are required (such as 1µs or even 500ns). Since it is not reasonable for such small delays to call the function `pal_timer_delay()`, these delays are implemented as macros in file *pal_config.h*: `PAL_WAIT_1_US` and `PAL_WAIT_500_NS`. These macros are based on nop operations, which depend on the MCU clock. This is implemented in file *pal_config.h* as well.

10.6.1.3.3 UART Baud Rate

The UART baud rate is selected by properly setting the USART baud rate register, which in return is depending on the MCU clock speed. This is implemented in file *pal_uart.h* in macro `UART_BAUD()`. This macro does not have to be updated if the clock speed changes.

10.6.1.4 Selecting the proper System Clock Speed

The desired clock speed can be selected at compile time by using the build switch `F_CPU`. It is not intended to change the clock during operation, since this would add additional code size.

The currently supported clock speed is 32MHz, 16MHz, 8MHz, or 4MHz.

The standard frequency of the system clock is 16MHz. If the build switch `F_CPU` is not set during the build process, the clock speed of 16MHz will be applied. In case other clock speed shall be selected, the build switch needs to be set accordingly.

For the GCC tool chain example Makefiles for changing the system clock are provided for the MAC application *Star_Nobeacon* for the above described hardware platform. Please check directory

MAC_v_x_y_z\Applications\MAC_Examples\Star_Nobeacon\AT86RF231_ATXMEGA128A1_REB_4_1_STK600

for the following Makefiles:

- Makefile: Standard clock speed of 16MHz
- Makefile_32/8/4_MHZ: Makefiles for customized clock speed with proper build switch `F_CPU`

For the IAR tool chain Example project files for changing the system clock are provided for the MAC application *Star_Nobeacon* for the above described hardware platform. Please check directory

MAC_v_x_y_z\Applications\MAC_Examples\Star_Nobeacon\AT86RF231_ATXMEGA128A1_REB_4_1_STK600

for the following IAR project files:

- Star.eww: Standard clock speed of 16MHz
- Star_32/8/4_MHZ.eww/ewp: IAR project files for customized clock speed with proper build switch `F_CPU`

In order to review the changes for the selectable clock speed feature, search for occurrences of this build switch in the corresponding files *pal_board.c* and *pal_config.h*.



Other clock speeds than the currently supported speed (such as 1MHz or 2MHz) could be implemented easily by updated the dedicated code snippets pointed out by F_CPU.

11 Protocol implementation conformance statement (PICS)

This chapter lists the conformance of the AVR2025 MAC implementation with the requirements and optional features as defined by the standard specified in 162] in section D.7.

11.1 Major roles for devices compliant with IEEE Std 802.15.4-2006

Table 11-1. 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.					

11.2 Major capabilities for the PHY

Table 11-2. 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: M O		X (FFD only)	
PLF6	Link quality indication (LQI)	M		X	
PLF7	Channel selection	M		X	



			Support		
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.					

11.3 Major capabilities for the MAC Sublayer

11.3.1 MAC Sublayer Functions

Table 11-3. MAC Sublayer 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	

			Support		
MLF8	Association and disassociation	M		X	
MLF9	Security	M			X (No MAC security, only application security)
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
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.					

11.3.2 MAC Frames

Table 11-4. 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)





		Transmitter		Receiver	
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

12 Abbreviations

- API Application Programming Interface
- BMM Buffer Management
- GPIO General Purpose Input/Output
- IRQ Interrupt Request
- MAC Medium Access Control
- MCL MAC Core Layer
- MCPS MAC Common Part Sublayer
- MCU Microcontroller Unit
- MHR MAC Header
- MIC Message Integrity Code
- MLME MAC Sublayer Management 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
- WPAN Wireless Personal Area Network





13 References

- [1] Atmel Wireless MCU Software Website
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4373
- [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)

14 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.

14.1 Rev. 2025F-MCU Wireless-02/10

Released with AVR2025 MAC Version 2.4.2

1. Support of program code larger than 128 KByte added
2. PAN-Id conflict detection handling added
3. Support of AT91SAM7XC added
4. Description of application security added
5. Description of security build switches updated

14.2 Rev. 2025E-MCU Wireless-01/10

Released with AVR2025 MAC Version 2.4.0

6. Support for AT86RF231 and AT86RF212 with AT91SAM7X256 added
7. Support for ATmega128RFA1-EK1 added
8. Build switch DISABLE_TSTAMP_IRQ added
9. Support for ATmega1284P removed
10. Support for ATxmega256A3
11. MAC Porting Guide using ATxmega256A3 as example added
12. MAC Examples App 3 (Beacon Payload) and App 4 (Beacon Broadcast Data) added
13. Build switch SYSTEM_CLOCK_MHZ renamed to F_CPU
14. Updated handling of MAC PIB attribute macRxOnWhenIdle and MAC power management
15. New build switch BAUD_RATE added
16. Support for AT86RF230A and related hardware platforms discontinued
17. Handling of callback stub functions added
18. Handling of callback stub functions added
18. PICS Table added

14.3 Rev. 2025B-MCU Wireless-09/09

Released with AVR2025 MAC Version 2.3.1

1. Name of Radio Controller Board (RCB) changed: transceiver number suffix is replaced by board suffix
2. Support for ATmega128RFA1 added
3. Handling of Promiscuous Mode updated.
4. Migration Guide for previous MAC versions removed
5. Filter tuning section added
6. Description of Performance test application extended
7. Support for ATxmega MCU based AES within SAL
8. Handling of MAC components updated
9. Initial Support for AT91SAM7X256 added
10. DISABLE_IEEE_ADDR_CHECK added





- 11. Chapter Supported Platforms added
- 12. Chapter Topics on Platforms Porting added

14.4 Rev. 2025-AVR-04/09

Released with AVR2025 MAC Version 2.2.0

- 1. Initial Version

15 Table of Contents

AVR2025: IEEE 802.15.4 MAC Software Package - User Guide -	1
Features	1
1 Introduction	1
2 General Architecture	2
2.1 Main Stack Layers	2
2.1.1 Platform Abstraction Layer (PAL).....	2
2.1.2 Transceiver Abstraction Layer (TAL).....	3
2.1.3 MAC Core Layer (MCL).....	4
2.1.4 Usage of the Stack	5
2.2 Other Stack Components	6
2.2.1 Resource Management	6
2.2.2 Security Abstraction Layer.....	6
2.2.3 Security Toolbox.....	7
2.2.4 Transceiver Feature Access.....	7
2.3 Application	7
3 Understanding the Software Package.....	8
3.1 MAC Package Directory Structure	8
3.2 Header File Naming Convention	13
4 Understanding the Stack.....	15
4.1 Configuration Files	15
4.1.1 Application Resource Configuration – app_config.h.....	16
4.1.2 Stack Resource Configuration – stack_config.h.....	17
4.1.3 PAL Resource Configuration – pal_config.h.....	17
4.1.4 TAL Resource Configuration – tal_config.h	17
4.1.5 MAC Resource Configuration – mac_config.h.....	18
4.1.6 NWK Resource Configuration – nwk_config.h.....	18
4.1.7 Build Configuration File – mac_build_config.h.....	18
4.1.8 User Build Configuration File – mac_user_build_config.h	18
4.2 MAC Components	18
4.2.1 MAC_INDIRECT_DATA_BASIC	19
4.2.2 MAC_INDIRECT_DATA_FFD	20
4.2.3 MAC_PURGE_REQUEST_CONFIRM.....	21
4.2.4 MAC_ASSOCIATION_INDICATION_RESPONSE	21
4.2.5 MAC_ASSOCIATION_REQUEST_CONFIRM	22
4.2.6 MAC_DISASSOCIATION_BASIC_SUPPORT	22
4.2.7 MAC_DISASSOCIATION_FFD_SUPPORT	23
4.2.8 MAC Scan Components	23
4.2.9 MAC_ORPHAN_INDICATION_RESPONSE.....	23
4.2.10 MAC_START_REQUEST_CONFIRM	24
4.2.11 MAC_RX_ENABLE_SUPPORT	25
4.2.12 MAC_SYNC_REQUEST	26
4.2.13 MAC_SYNC_LOSS_INDICATION	26
4.2.14 MAC_BEACON_NOTIFY_INDICATION.....	27
4.2.15 MAC_GET_SUPPORT	27
4.2.16 MAC_PAN_ID_CONFLICT_AS_PC.....	28
4.2.17 MAC_PAN_ID_CONFLICT_NON_PC	28



4.3 Support of AVR Platforms larger than 128 KByte Program Memory	28
4.3.1 General.....	28
4.3.2 Stack Implementation	28
4.3.3 Application Support	30
4.4 Application Security Support	30
5 MAC Power Management	32
5.1 Understanding MAC Power Management.....	32
5.2 Reception of Data at Nodes applying Power Management	33
5.2.1 Setting of macRxOnWhenIdle to True	33
5.2.2 Enabling the Receiver	33
5.2.3 Handshake between End Device and Coordinator.....	34
5.2.4 Indirect Transmission from Coordinator to End Device	34
5.3 Application Control of MAC Power Management.....	35
5.3.1 MAC PIB Attribute macRxOnWhenIdle	35
5.3.2 Handling the Receiver with wpan_rx_enable_req()	35
5.4 TAL Power Management API.....	36
6 Application and Stack Configuration	37
6.1 Build Switches	37
6.1.1 Global Stack Switches.....	39
6.1.2 Standard and User Build Configuration Switches.....	42
6.1.3 Platform Switches.....	43
6.1.4 Transceiver specific Switches	47
6.1.5 Security Switches	51
6.1.6 Test and Debug Switches.....	52
6.2 Build Configurations	53
6.2.1 Standard Build Configurations.....	53
6.2.2 User Build Configurations – MAC_USER_BUILD_CONFIG.....	56
7 Tool Chain	60
7.1 General Prerequisites.....	60
7.2 Building the Applications	60
7.2.1 Using GCC Makefiles	60
7.2.2 Using AVR Studio.....	60
7.2.3 Using IAR Embedded Workbench.....	61
7.2.4 Batch Build	62
7.3 Downloading an Application	62
7.3.1 Using AVR Studio directly	62
7.3.2 Using AVR Studio after Command Line Build of Application.....	63
7.3.3 Using IAR Embedded Workbench.....	69
8 Example Applications.....	77
8.1 Walking through a Basic Application	77
8.1.1 Implementation of the Coordinator	77
8.1.2 Implementation of the Device	81
8.2 Provided Examples Applications	86
8.2.1 MAC Examples.....	86
8.2.2 TAL Examples	95
8.2.3 STB Examples.....	97

8.3 Common SIO Handler	102
8.4 Handling of Callback Stubs	103
8.4.1 MAC Callbacks.....	104
8.4.2 TAL Callbacks	104
8.4.3 Example for MAC Callbacks.....	105
9 Supported Platforms	108
9.1 Supported MCU Families	108
9.2 Supported MCUs	108
9.3 Supported Transceivers	108
9.4 Supported Boards	108
9.4.1 Radio Controller Boards (RCB) based Platforms	108
9.4.2 Radio Extender Boards (REB).....	114
9.4.3 Radio Extender Boards (REB) based Platforms.....	117
9.4.4 AT91SAM7XC-EK	123
9.4.5 RZ USBstick.....	123
9.4.6 ATmega128RFA1-EK1 Evaluation Kit.....	124
9.4.7 Peripherals	126
10 Platform Porting.....	128
10.1 Porting to a new Platform	128
10.2 Bring-up of a new PAL	128
10.3 Bring-up of a new Hardware Board.....	128
10.3.1 Implementation of PAL for Target Platform	128
10.3.2 Example Implementation of PAL for AT91SAM7X256 based Platform.....	130
10.3.3 Bring-up of an existing (MAC) Application on the Target Platform	136
10.4 Bring-up of a new MCU based on a supported MCU Family	139
10.4.1 Implementation of PAL for Target Platform	139
10.4.2 Example Implementation of PAL for ATxmega256A3	141
10.4.3 Bring-up of an existing Application on the Target Platform.....	143
10.5 Bring-up of an new Application on an existing Platform.....	146
10.5.1 Step 1 - Identify a matching Base Application	146
10.5.2 Step 2 – Update the GCC Makefile	147
10.5.3 Step 3 – Update the IAR project files.....	149
10.5.4 Step 4 - Update the AVR Studio Project files	153
10.6 Customizing the Platform Clock Speed.....	153
10.6.1 Customizing ATxmega128A1 Platforms.....	153
11 Protocol implementation conformance statement (PICS).....	157
11.1 Major roles for devices compliant with IEEE Std 802.15.4-2006	157
11.2 Major capabilities for the PHY	157
11.3 Major capabilities for the MAC Sublayer	158
11.3.1 MAC Sublayer Functions.....	158
11.3.2 MAC Frames	159
12 Abbreviations	161
13 References.....	162
14 User Guide Revision History.....	163





14.1 Rev. 2025F-MCU Wireless-02/10	163
14.2 Rev. 2025E-MCU Wireless-01/10	163
14.3 Rev. 2025B-MCU Wireless-09/09	163
14.4 Rev. 2025-AVR-04/09	164
15 Table of Contents.....	165
16 Table of Figures	169

16 Table of Figures

Figure 2-1. MAC Architecture	2
Figure 2-2. Stack Usage	6
Figure 4-1. Configuration File #include-Hierarchy	16
Figure 4-2. Essential and Supplementary MAC Components	19
Figure 4-3. Example of provided Functionality for MAC_INDIRECT_DATA_BASIC and MAC_INDIRECT_DATA_FFD	21
Figure 4-4. Provided Functionality for MAC_ASSOCIATION_INDICATION_ RESPONSE and MAC_ASSOCIATION_REQUEST_CONFIRM	22
Figure 4-5. Provided Functionality for MAC_ORPHAN_INDICATION_RESPONSE and MAC_SCAN_ORPHAN_REQUEST_CONFIRM (Orphan Scan Procedure)	24
Figure 4-6. Start of Nonbeacon Network and Active Scan	25
Figure 4-7. Enabling of Receiver and proper Data Reception	26
Figure 4-8. Synchronization and Loss of Synchronization	27
Figure 6-1. Build Configuration Example	39
Figure 6-2. Handling of Promiscuous Mode	42
Figure 7-1. AVR Studio - Verifying and Setting of IEEE Address	63
Figure 7-2. AVR Studio "Connect" Dialog	63
Figure 7-3. AVR Studio "Select AVR Programmer" Dialog	64
Figure 7-4. AVR Studio "Select JTAGICE mkII" Dialog	64
Figure 7-5. AVR Studio "JTAGICE mkII" Dialog with "Main" Tab	65
Figure 7-6. AVR Studio "JTAGICE mkII" Dialog with "Fuses" Tab	65
Figure 7-7. AVR Studio "JTAGICE mkII" Dialog with "Program" Tab	66
Figure 7-8. AVR Studio "JTAGICE mkII" Dialog with "Program" Tab Download Status	66
Figure 7-9. AVR Studio "Open File" Dialog	67
Figure 7-10. AVR Studio – Select elf-File	67
Figure 7-11. AVR Studio "Select debug platform and device" Window	68
Figure 7-12. AVR Studio "JTAGICE mkII" Dialog with "Debug" Tab	68
Figure 7-13. AVR Studio after successful Debug Build Download	69
Figure 7-14. IAR Embedded Workbench – "Options" -> "Debugger" -> "Setup"	70
Figure 7-15. IAR Embedded Workbench – "Options" -> "Debugger" -> "Plugins"	70
Figure 7-16. IAR Embedded Workbench – "Options" -> "JTAGICE mkII" -> "JTAGICE mkII 2"	71
Figure 7-17. IAR Embedded Workbench – "Options" -> "Debugger" -> "Setup"	72
Figure 7-18. IAR Embedded Workbench – "Options" -> "Debugger" -> "Plugins"	72
Figure 7-19. IAR Embedded Workbench – "Options" -> "JTAGICE mkII" -> "JTAGICE mkII 2"	73
Figure 7-20. IAR Embedded Workbench - Start "Download and Debug"	74
Figure 7-21. IAR Embedded Workbench - Successful Download of Debug Build	75
Figure 7-22. IAR Embedded Workbench – Verifying and Setting of IEEE Address	76
Figure 8-1. Tree Network Example	90
Figure 8-2. Promiscuous_Mode_Demo Terminal Program Snapshot	93
Figure 8-3. Terminal Program Snapshot of Performance_Test Application	96
Figure 8-4. Terminal Program Snapshots of Performance_Test Application	97
Figure 8-5. Secure Remote Control Application – Both Nodes in Secure Mode	99
Figure 8-6. Secure Remote Control Application – Both Nodes in Unsecure Mode	99
Figure 8-7. Secure Remote Control Application – Transmitter in Secure Mode, Receiver in Unsecure Mode	99
Figure 8-8. Secure Remote Control Application – Transmitter in Unsecure Mode, Receiver in Secure Mode	100
Figure 8-9. Snapshot of Secure Sensor Application	102
Figure 9-1. RCB V3.2 with AT86RF230B and ATmega1281	109
Figure 9-2. RCB V4.0 with AT86RF231 and ATmega1281	109
Figure 9-3. RCB V4.1 with AT86RF231 and ATmega1281	110
Figure 9-4. RCB V5.3 with AT86RF212 and ATmega1281	110





Figure 9-5. RCB V6.3 with ATmega128RFA1	111
Figure 9-6. Sensor Terminal Board without RCB	111
Figure 9-7. Sensor Terminal Board with RCB V6.3 with ATmega128RFA1.....	112
Figure 9-8. Sensor Terminal Board connected to JTAG-ICE and USB.....	112
Figure 9-9. Breakout Board Light without RCB.....	113
Figure 9-10. Breakout Board Light with RCB.....	113
Figure 9-11. Breakout Board Light connected to JTAG-ICE and UART.....	113
Figure 9-12. Close-up View of Breakout Board Light connected to JTAG-ICE and UART	114
Figure 9-13. REB V2.3 with AT86RF230B	115
Figure 9-14. REB V4.0.1 with AT86RF231	115
Figure 9-15. REB231ED V4.1.1 with AT86RF231	116
Figure 9-16. REB212 V5.0.2 with AT86RF212.....	116
Figure 9-17. REB to STK600 Adapter.....	117
Figure 9-18. STK600 with REB to STK600 Adapter and REB231ED V4.1.1	117
Figure 9-19. STK600 (with ATxmega128A1) with REB to STK600 Adapter and REB connected to JTAG-ICE and UART, and LED and Button Cable.....	118
Figure 9-20. STK500.....	118
Figure 9-21. STK500 (with ATmega644p) with REB connected to JTAG-ICE and UART, and LED and Button Cable.....	119
Figure 9-22. Close-up View of Cable Connection for LEDs and Buttons	119
Figure 9-23. STK500 with STK501	120
Figure 9-24. STK500 with STK501 (with ATmega1281) and REB connected to JTAG- ICE and UART, and LED and Button Cable.....	120
Figure 9-25. Close-up View of Cable Connection for LEDs and Buttons	121
Figure 9-26. AT91SAM7X-EK Board with AT91SAM7X256.....	122
Figure 9-27. AT91SAM7X-EK Board with AT91SAM7X256.....	122
Figure 9-28. AT91SAM7X-EK Board with AT91SAM7X256 connected to SAM-ICE and UART	123
Figure 9-29. RZ USBstick with 10 Pin Header for JTAG Connector	124
Figure 9-30. RZ USBstick with JTAG-ICE and USB Connection	124
Figure 9-31. ATmega128RFA1-EK1 on STK600.....	125
Figure 9-32. ATmega128RFA1-EK1 on STK600 with JTAG-ICE using UART1	125
Figure 9-33. ATmega128RFA1-EK1 wiring for UART1	126
Table 9-34. Peripherals supported by Platform Type	127



Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN ATTEL'S TERMS AND CONDITIONS OF SALE LOCATED ON ATTEL'S WEB SITE, ATTEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATTEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATTEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.