CC13x0 SimpleLink™ TI 15.4-Stack 2.x.x Embedded

Developer's Guide

TEXAS INSTRUMENTS

Literature Number: SWRU489A
September 2016–Revised December 2016

Contents

List of Figures

List of Tables

Overview

The SimpleLink™ TI 15.4-Stack is part of the CC13x0 SimpleLink SDK. The CC13x0 SimpleLink SDK includes the software stack from Texas Instruments that implements the standard IEEE 802.15.4e and 802.15.4g specification. The TI 15.4-Stack also implements a frequency-hopping scheme adopted from Wi-SUN® field area network (FAN) specification. The CC13x0 SimpleLink SDK also provides the required tools, real-time operating system (RTOS), and example applications for the TI 15.4-Stack to help developers quickly get started developing their own star-topology-based wireless network products.

The purpose of this document is to give an overview of the SimpleLink TI 15.4-Stack to help developers run the out-of-box example applications and enable creation of custom TI 15.4-Stack-based wireless star-topology-based networking solutions. This document introduces the essential need-to-know technology details for developing a wireless network based on the IEEE™ 802.15.4 and Wi-SUN FAN specification supported by the TI 15.4-Stack.

NOTE: Do not use this document as a substitute for the complete specification. For more details, see the IEEE 802.15.4 specification and Wi-SUN FAN specification.

1.1 Introduction

The Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 standard defines the physical (PHY) and media access control (MAC) layers of the Open Systems Interconnection (OSI) model of network operations. The PHY defines the wireless link conditions like modulation, frequency, and power, while the MAC defines the format of the data.

TI implementation of this standard combines the following:

- The basic standard (802.15.4-2006) with the most recent updates
- The 802.15.4e for industrial applications and 802.15.4g for the smart utility networks (SUN)
- An implementation of Wi-SUN frequency hopping

SimpleLink, TI-RTOS, Code Composer Studio, Sensor Controller Studio, SmartRF, BoosterPack are trademarks of Texas Instruments.
ARM, Cortex are registered trademarks of ARM Ltd.
Wireshark is a trademark of CACE Technologies, LLC.
IEEE is a trademark of International Electrical and Electronics Engineers.
Node.js is a registered trademark of Joyent, Inc.
Linux is a registered trademark of Linus Torvalds.
Windows is a registered trademark of Microsoft Corporation.
Wi-SUN is a registered trademark of Wi-SUN Alliance.
All other trademarks are the property of their respective owners.

TI 15.4-Stack Software Development Platform

TI's royalty-free TI 15.4-Stack is a complete software platform for developing applications that require extremely low-power, long-range, reliable, robust and secure wireless star-topology-based networking solutions. This kit is based on the CC13x0 SimpleLink ultra-low power wireless microcontroller unit (MCU). The CC13x0 device combines a sub-1 GHz RF transceiver with 128KB of in-system programmable memory, 20KB of SRAM, and a full range of peripherals. The CC13x0 device is centered on an ARM® Cortex®-M3 series processor that handles the application layer, the TI 15.4-Stack, and an autonomous radio core centered on an ARM Cortex-M0 processor, which handles all the low-level radio control and processing associated with the physical layer and parts of the link layer. The sensor controller block provides additional flexibility by allowing autonomous data acquisition and control independent of the Cortex-M3 processor, which further extends the low-power capabilities of the CC13x0 device.

Figure 2-1 shows the block diagram. For more information on the CC13x0 device, see the *CC13xx, CC26xx SimpleLink Wireless MCU Technical Reference Manual* (TRM).

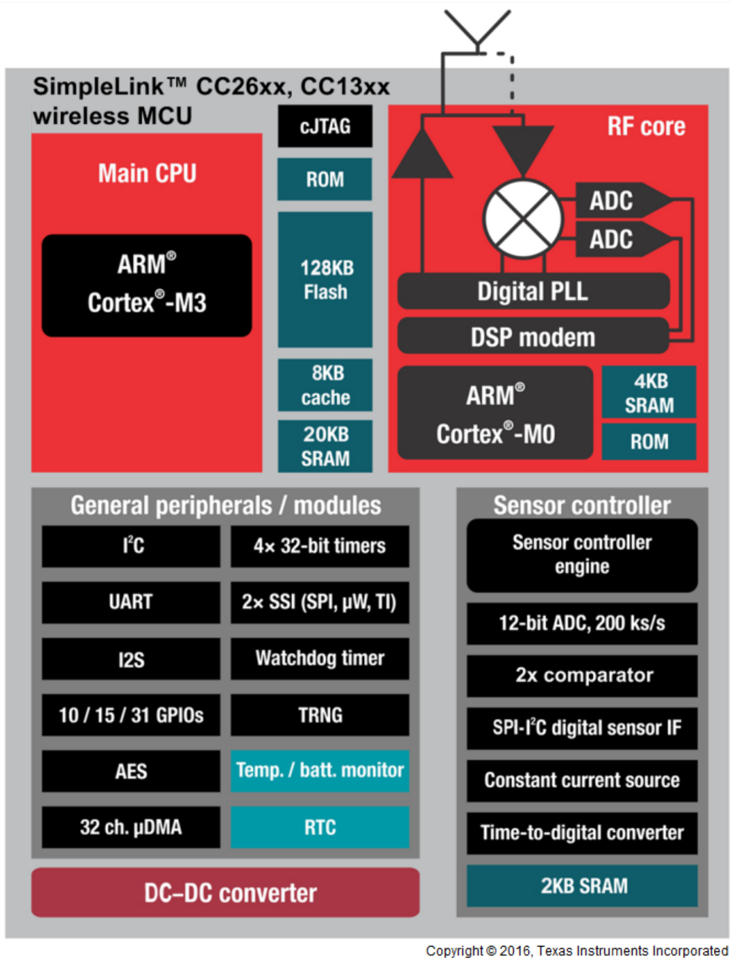


Figure 2-1. SimpleLink™ CC13x0 Block Diagram

2.1 Protocol Stack and Application Configurations

Figure 2-2 shows the two different system architectures enabled by the TI 15.4-Stack.

- A single device is shown in Figure 2-2 (a). The application and protocol stack are both implemented on the CC13x0 device as a true single-chip solution. This configuration is the simplest and most common when using the CC13x0 device for network nodes and also using the CC13x0 device as a personal area network (PAN) coordinator node. This configuration is the most cost-effective technique and provides the lowest-power performance.
- A coprocessor is shown in Figure 2-2 (b). The protocol stack runs on the CC13x0 device while the application is executed on an external MPU or MCU. The application interfaces with the CC13x0 device using the network protocol interface (NPI) over a serial universal asynchronous receiver/transmitter (UART) connection. The description of the API interface is provided in the *TI-15.4 Stack CoP Interface Guide.pdf* document found in the <docs/ti154stack/guides> folder of the TI 15.4-Stack install. This configuration is useful for applications that must add long-range wireless connectivity or peripheral applications, which execute on another device (such as an external MCU) or on a PC without the requirement to implement the complexities associated with a wireless networking protocol. In these cases, the application can be developed externally on a host processor while running the TI 15.4-Stack on the CC13x0 device, which provides ease of development and quickly adds long-range wireless connectivity to existing products.

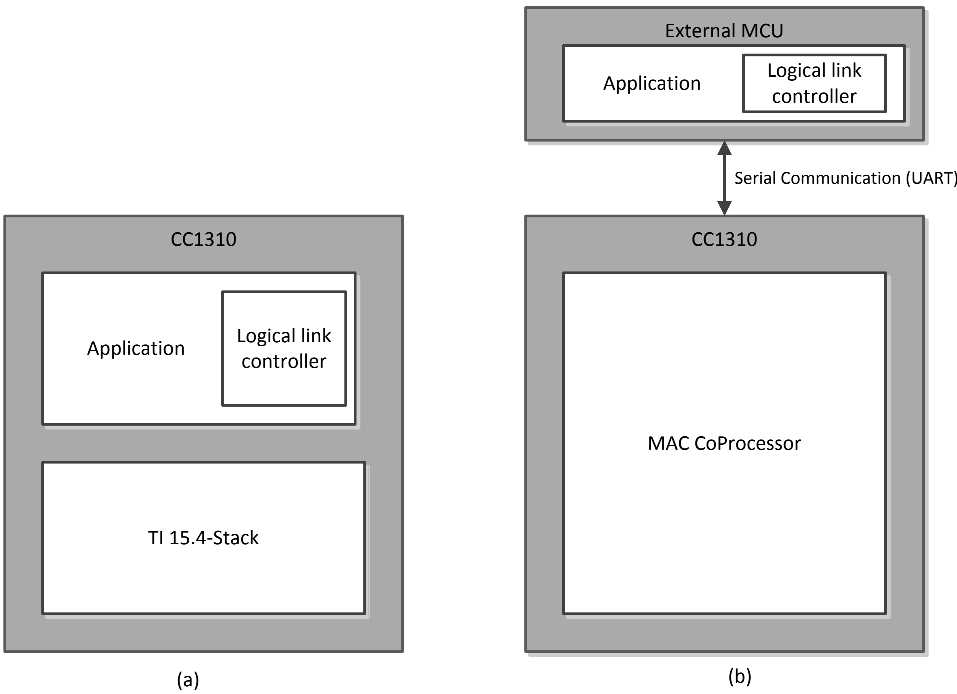


Figure 2-2. Single Device and Coprocessor Configuration

2.2 Solution Platform

This section describes the various components that are installed with the TI 15.4-Stack, the directory structure of the protocol stack, and any tools required for development. Figure 2-3 shows the TI 15.4-Stack development system.

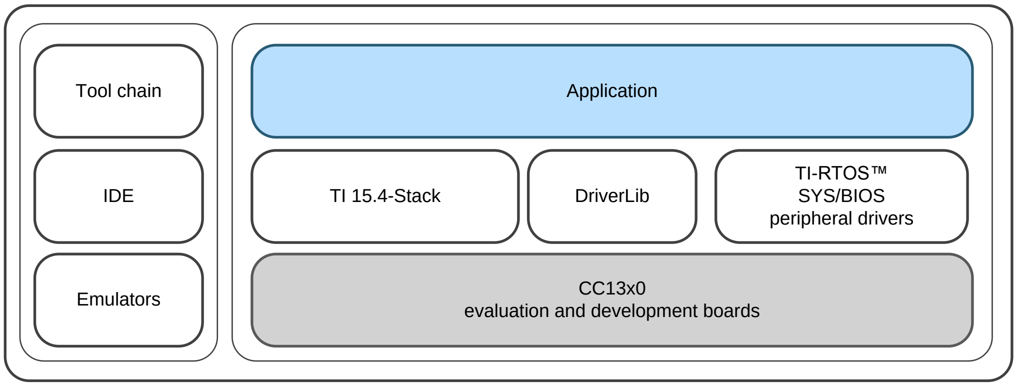


Figure 2-3. TI 15.4-Stack Development System

The following components are included in the solution platform:

- Real-time operating system (RTOS) with the TI-RTOS™ SYS/BIOS kernel, optimized power management support, and peripheral drivers (serial peripheral interface [SPI], UART and so forth)
- The CC13xxware driverLib provides a register abstraction layer that is used by software and drivers to control the CC1310 MCU.
- The TI 15.4-Stack is provided in library form.
- Example applications make the beginning stages of development easier. Example applications are provided for the CC13x0 platform and Linux® example applications are provided for the AM335x device running the processor SDK.
- Code Composer Studio™ (CCS) is the supported IDE for the example applications for the CC13x0 platform.

2.3 Directory Structure

The CC13x0 SimpleLink SDK installer includes all the files needed to start evaluating example applications and to later create custom applications using the TI 15.4-Stack. The installed SDK provides the following content at the indicated default locations on the development computer:

- Documents: detailed API, developer's guide, and user's guide documentation
  - C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\docs\ti154stack
- Examples: complete application examples for collector, sensor, and coprocessor devices, as well as prebuilt hex files.
  - If using a CC1310 then examples are here:
    - C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1310_LAUNCHXL\ti154stack
  - If using a CC1350 then examples are here:
    - C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1350_LAUNCHXL\ti154stack
- Tools: support files for the Wireshark protocol analyzer
  - C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\tools\ti154stack

2.4 Projects

The TI 15.4-Stack component within the CC13x0 SimpleLink SDK includes several projects that range from providing core IEEE 802.15.4 MAC functionality to use-case specific applications such as *Collector* and *Sensor*. The following projects can be used directly out of the box to demonstrate basic wireless applications and can be used later as a starting point for new application development.

The *Coprocessor* project can be used to build a MAC coprocessor device that works with a host processor in a 2-chip scenario. The coprocessor project provides full-function MAC capability over serial interface to the application running on the host. This device allows TI 15.4-Stack wireless functionality to be added to systems that are not suited to single-chip solutions. A prebuilt hex file for the coprocessor is provided in the SDK. If changes are needed, such as addition of a custom API command, the coprocessor project can be used to generate a new hex file.

The *Collector* project builds a full-function device (FFD) that performs the functions of a network coordinator (starting a network and permitting devices to join that network) and also provides an application to monitor and collect sensor data from one or more sensor devices. Prebuilt hex files for the collector project (demonstrating several communication scenarios) are provided in the SDK. The collector project is used to build these hex files and can be modified to alter communication or application functionality.

The *Sensor* project builds an reduced-function device (RFD) that performs the functions of a network device (joining a network and polling the coordinator for messages) and also provides an application to collect and send sensor data to the collector device. Prebuilt hex files for the sensor project are provided in the SDK to demonstrate operation in several communication scenarios. The sensor project is used to build these hex files and can be modified to alter communication or application functionality.

The *Linux Collector* and *Gateway* Applications are provided as part of the TI 15.4-Stack Linux SDK installer. The TI 15.4-Stack Linux SDK is a separate SDK that can be downloaded online at http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK. The Linux Collector Example Application interfaces with the CC13x0 device running the MAC coprocessor through UART. The Linux Collector Example Application provides the same functionality as the Embedded Collector Application with the addition of providing a socket server interface to the Linux Gateway Application. The Linux Gateway Application implemented within the Node.js® framework connects as a client to the socket server created by the Linux Collector Example Application and establishes a local web server to which the user can connect through a web browser (in the local network) to monitor and control the network devices. The Collector and Gateway Applications that provide IEEE 802.15.4 to the IP Bridge are a great starting point for creating Internet of Things (IoT) applications with the TI 15.4-Stack.

The *Linux Serial Bootloader Application* is included inside the TI 15.4-Stack Linux SDK installer. This application demonstrates how to upgrade the firmware of the CC13x0 MCU through the CC13x0 ROM bootloader.

NOTE: Specific documentation for detailed Linux example applications can be found in the ${linux_sdk_root}/doc folder after running the Linux Installer.

The Linux installer requires an x86 64-bit machine running Ubuntu.

2.5 Setting Up the Integrated Development Environment

All embedded software for the TI 15.4-Stack is developed using TI's CCS on a Windows® 7 or later PC. To browse through the SDK projects and view the code as it is referenced in this document, it is necessary to install and set up the CCS integrated development environment (IDE). This section provides information on where to find this software and how to properly configure the workspace for the IDE.

Path and file references in this document assume that the CC13x0 SimpleLink SDK has been installed to the default path, hereafter referred to as *<INSTALL_DIR>*. Projects do not build properly if paths below the top-level directory are modified.

2.5.1 Installing the SDK

To install the CC13x0 SimpleLink SDK, run the installer:

Simplelink-CC13x0-SDK-1.00.00.xx.exe

NOTE: The xx indicates the SDK build revision number at the time of release.

The default TI 15.4-Stack install path is:
C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1350_LAUNCHXL\ti154stack
or
C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1310_LAUNCHXL\ti154stack
(if using a CC1310).

In addition to TI 15.4-Stack code, documentation, and projects, installing the SDK also installs the TI-RTOS bundle and the XDC tools, if not already installed. Table 2-1 lists the software and tools that are supported and tested with this SDK. Check the TI 15.4-Stack Wiki page [3] for the latest supported tool versions.

Table 2-1. Supported Tools and Software

Tool Or Software	Version	Install Path
TI 15.4-Stack SDK Installer	1_00_00_xx	C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1350_LAUNCHXL\ti154stack
Core SDK	3.01.00.04	C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\kernel\tirtos
XDC Tools	3.32.0.6	C:\ti\xdctools_3_32_00_06_core
CCS IDE	7.0	C:\ti\ccsv7
Sensor Controller Studio™	1.0.1	Windows default
SmartRF™ Flash Programmer 2	1.7.2	Windows default
SmartRF Studio 7	2.1.0	Windows default

2.5.2 Code Composer Studio

Code Composer Studio (CCS) provides a suite of tools that are used to develop applications that run on TI's MCUs and embedded processors. CCS contains many features that go beyond the scope of this document—more information can be found on the CCS website. Check the CC13x0 SimpleLink SDK release notes to see which CCS version to use.

The following describes installing and configuring the correct version of CCS and the necessary tools.

1. Download CCS 7.0 from the Download_CCS wiki page.
2. Launch the CCS installer (for example, *ccs_setup_win32.exe*).
3. On the *Processor Support* menu (see Figure 2-4), expand *SimpleLink Wireless MCUs* and select SimpleLink CC13xx and CC26xx Wireless MCUs

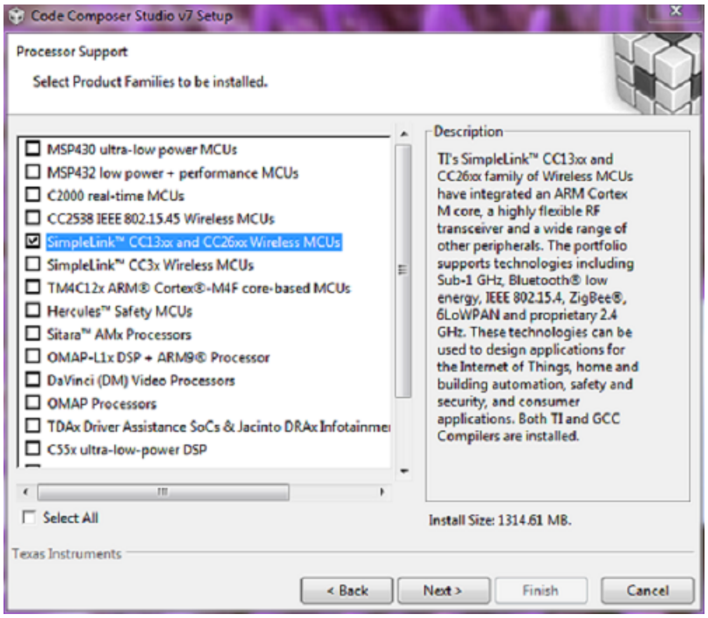


Figure 2-4. Processor Support Menu Selections

4. Click the Next button, then click the Finish button.
5. After CCS has installed, apply all available updates by selecting *Help → Check for Updates.*

NOTE: This step may require restarting CCS as each update is applied.

2.5.2.1 Configure CCS

This section explains how to configure CCS for development and debugging. It also provides information on useful CCS IDE settings (see Section 2.5.2.8)

2.5.2.2 Using CCS

This section describes how to open and build an existing project. The Sensor project is used as an example. However, all of the CCS projects included in the development kit have a similar structure.

2.5.2.3 Importing SDK Projects

Launch the CCS IDE and prepare to import the TI 15.4 projects from the installed SDK:

1. Select *Project → Import CCS*
2. Select *Select search-directory:* and click the Browse… button.
3. Navigate to *<INSTALL_DIR\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1350_LAUNCHXL\ti154stack* and click the OK button (see Figure 2-5).

   If using CC1310, navigate to *C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1310_LAUNCHXL\ti154stack*.

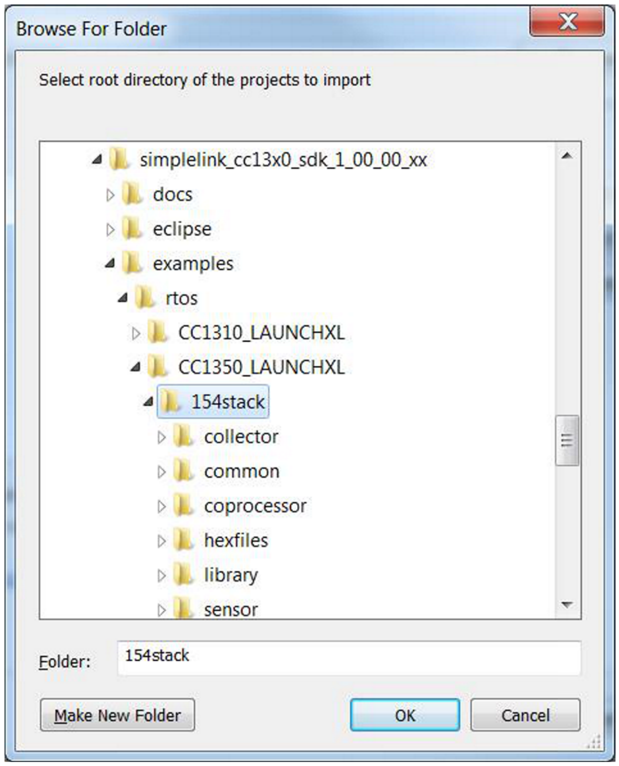


Figure 2-5. Import SDK Projects Menu Selection

4. The Discovered projects box now lists the projects that have been found. Select the Copy projects into workspace checkbox, click the Select All button, and last click the Finish button to import a copy of each SDK project into the workspace (see Figure 2-6).

NOTE: In the following sections, the project names for the CC1310 and CC1350 platforms are referred to as CC13x0. Replace *x* with either *1* or *5* depending on the wireless MCU being used.

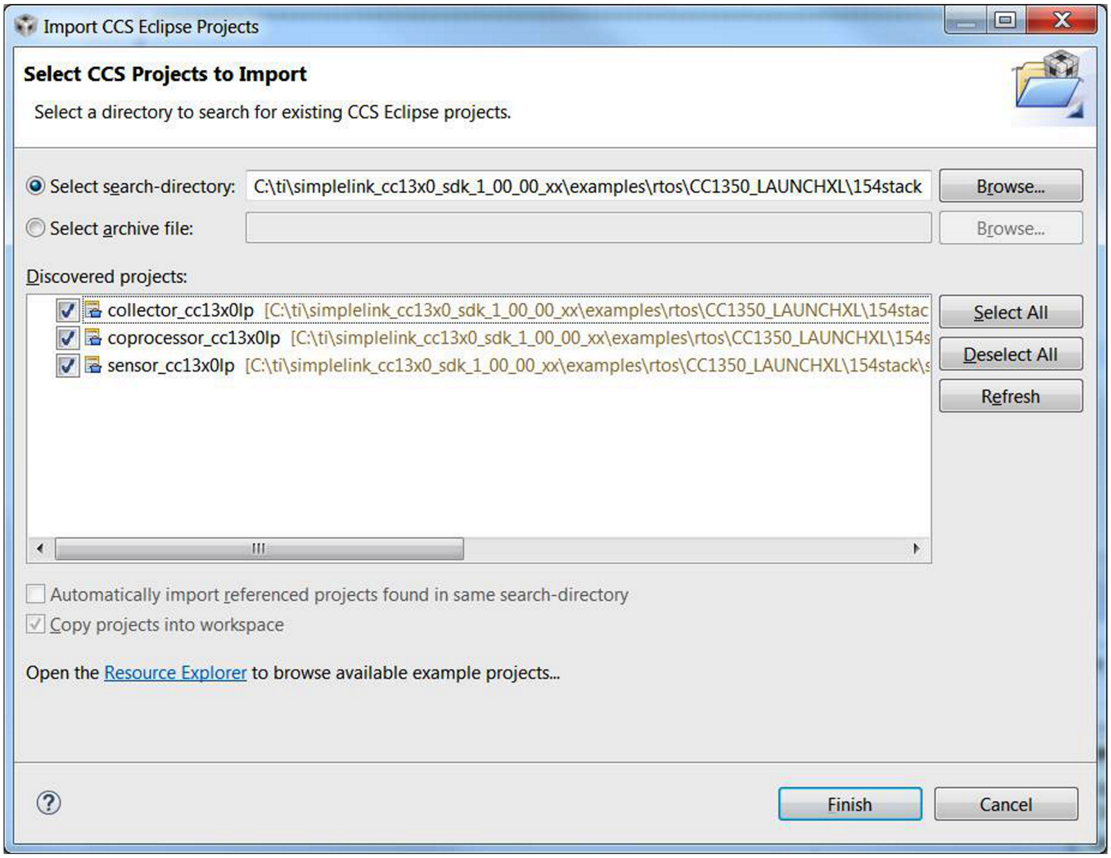


Figure 2-6. CCS Project Import Pane

2.5.2.4 Workspace Overview

The workspace now contains all projects needed to build downloadable images. The following examples build a Sensor image, to produce a functional sensor device.

In the Project Explorer pane, click on the *sensor_cc13x0lp* project to make it active. Click on the arrow to the left of the project to expand its contents, as shown in Figure 2-7.

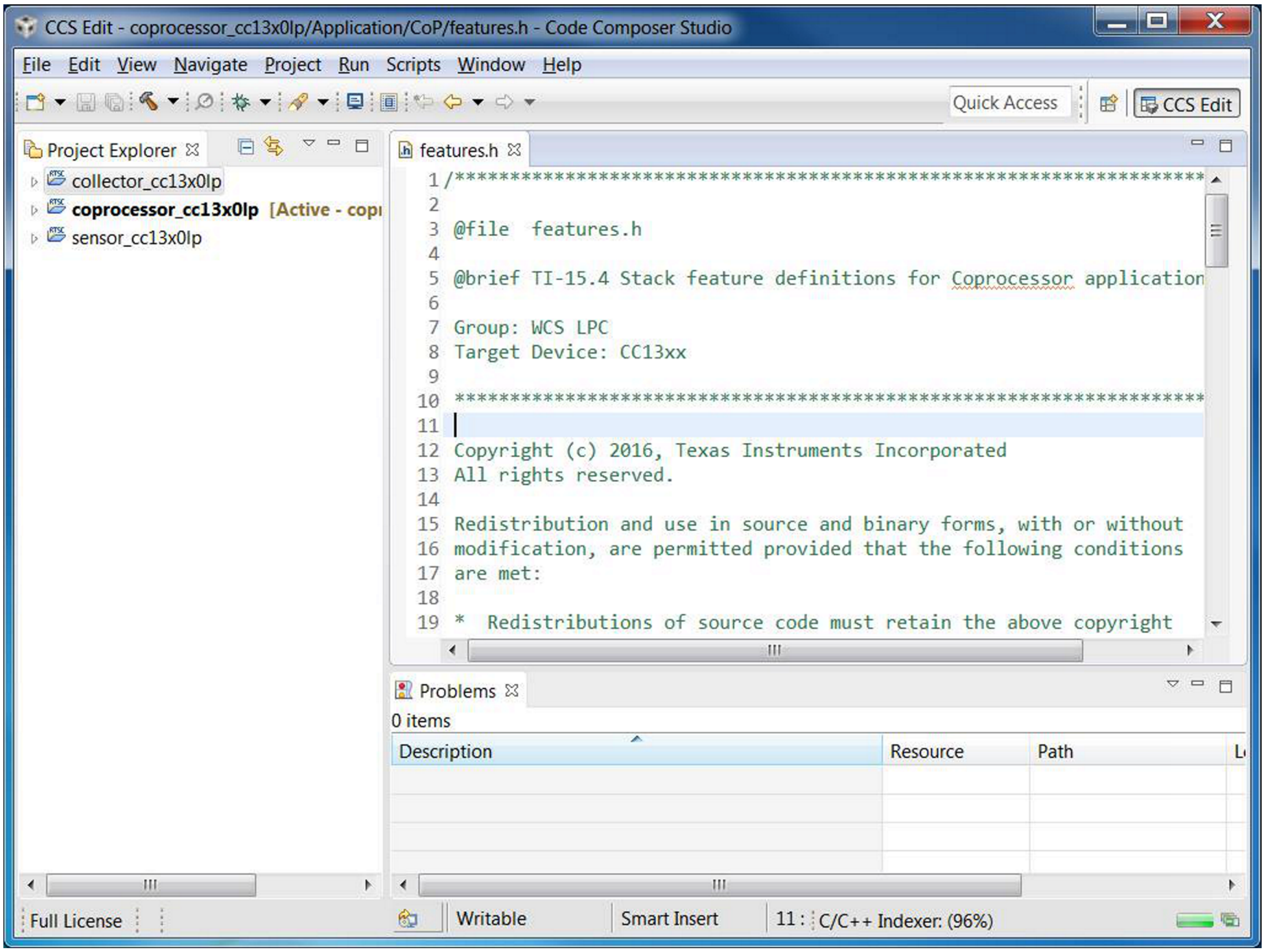


Figure 2-7. CCS Project Explorer Pane

In Figure 2-7, all folders (under the sensor_cc13x0lp project) except for sensor_cc13x0lp can be considered input folders, which contain source code files, header files, and configuration files used to compile and link the application. The sensor_cc13x0lp folder contains output files, which include programmable images and the linker map.

2.5.2.5 Compiling and Linking

All example projects in the SDK are ready to build out-of-the-box meaning they are preconfigured for use with a specific target board, in this case the CC13x0 LaunchPad. To build a Sensor Application image that is ready to program onto a LaunchPad board, select *Project → Rebuild Project*. The Console pane of the IDE displays the individual results of source file compilations, followed by the linker results, as shown in Figure 2-8.

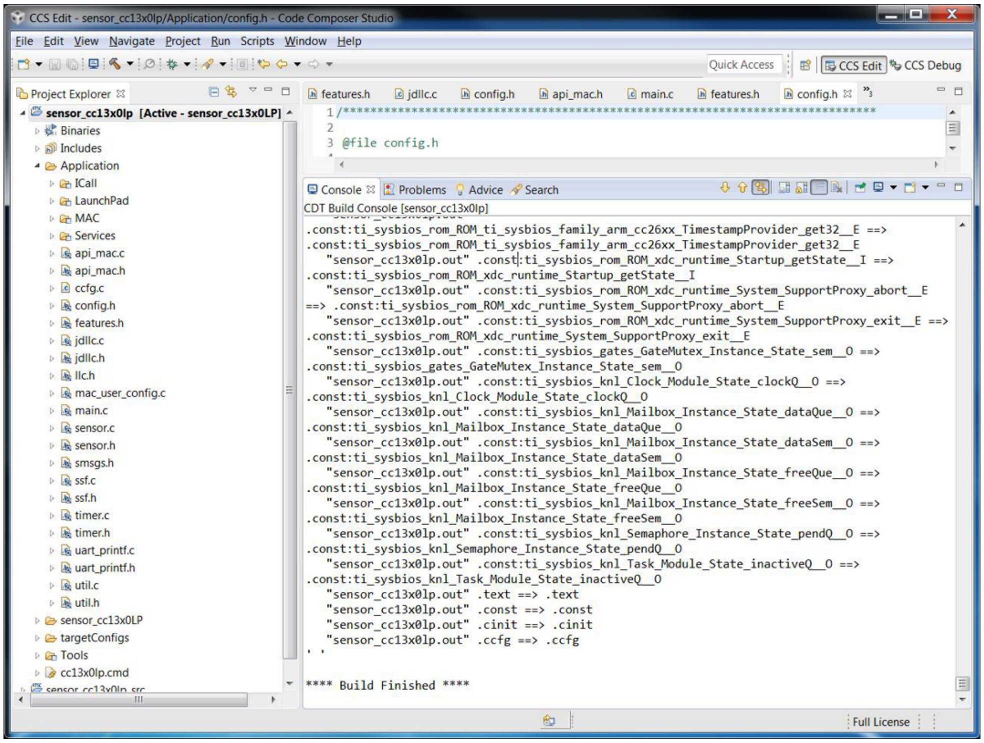


Figure 2-8. CCS Project Console Pane

In Figure 2-8, the output folder sensor_cc13x0lp is populated with results from a successful build of the Sensor Application. Two programmable output files have been produced, sensor_cc13x0lp.out and sensor_cc13x0lp.hex, along with a detailed linker map file, sensor_cc13x0lp.map.

2.5.2.6 Downloading Hex Files

As shown in Section 2.5.2.5, the CCS linker produces .hex files that are ready for direct download to the target hardware. The hex image can be downloaded to a target device by a stand-alone tool, such as the SmartRF Flash Programmer 2, shown in Figure 2-9.

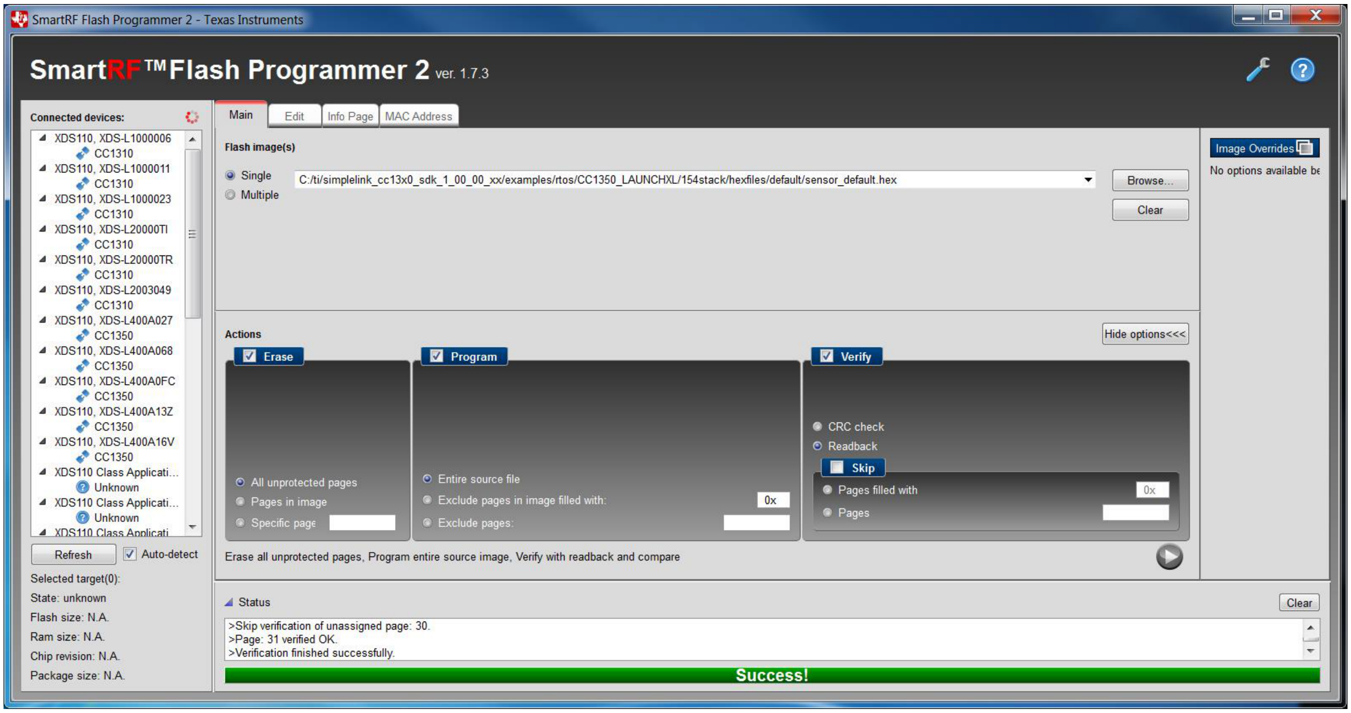


Figure 2-9. Programming Hex Files

As shown in Figure 2-9, SmartRF Flash Programmer 2 can be used to program the prebuilt hex file using the *Flash image(s)* → *Single* feature. Use the *Browse…* button to select the following files:

<INSTALL_DIR>\simplelink_cc13x0_sdk_1_00_00_xx\examples\rtos\CC1350_LAUNCHXL\ti154stack\hexfiles\default\sensor_default.hex

After connecting a CC1310 LaunchPad target board to one of the USB ports of the PC, the CC1310 XDS110 instance becomes listed in the *Connected devices* panel. Select the listed device by clicking on the *CC1310 icon*. Programming the CC1310 LaunchPad is accomplished through the following sequence (see Figure 2-9).

1. Select the *Erase* action, using the *Pages in image* option.
2. Select the *Program* action, using the *Entire source file* option.
3. Select the *Verify* action, using the *Readback* option.
4. Press the *Go* button (blue icon at the lower right corner of the Actions panel).
5. Observe the device programming feedback in the *Status* panel. It shows *Success!* when finished.
6. Exit the SmartRF Flash Programmer 2 and power cycle the LaunchPad to run the application.

2.5.2.7 Debugging

When debugging is necessary, *.out* files produced by the linker are downloaded and run from the CCS IDE. The following procedure would typically be used to debug the program.

Continuing with the Sensor example project, the compiled program can be downloaded to the target and a debug session initiated by selecting: *Run → Debug* on the IDE, as shown in Figure 2-10.

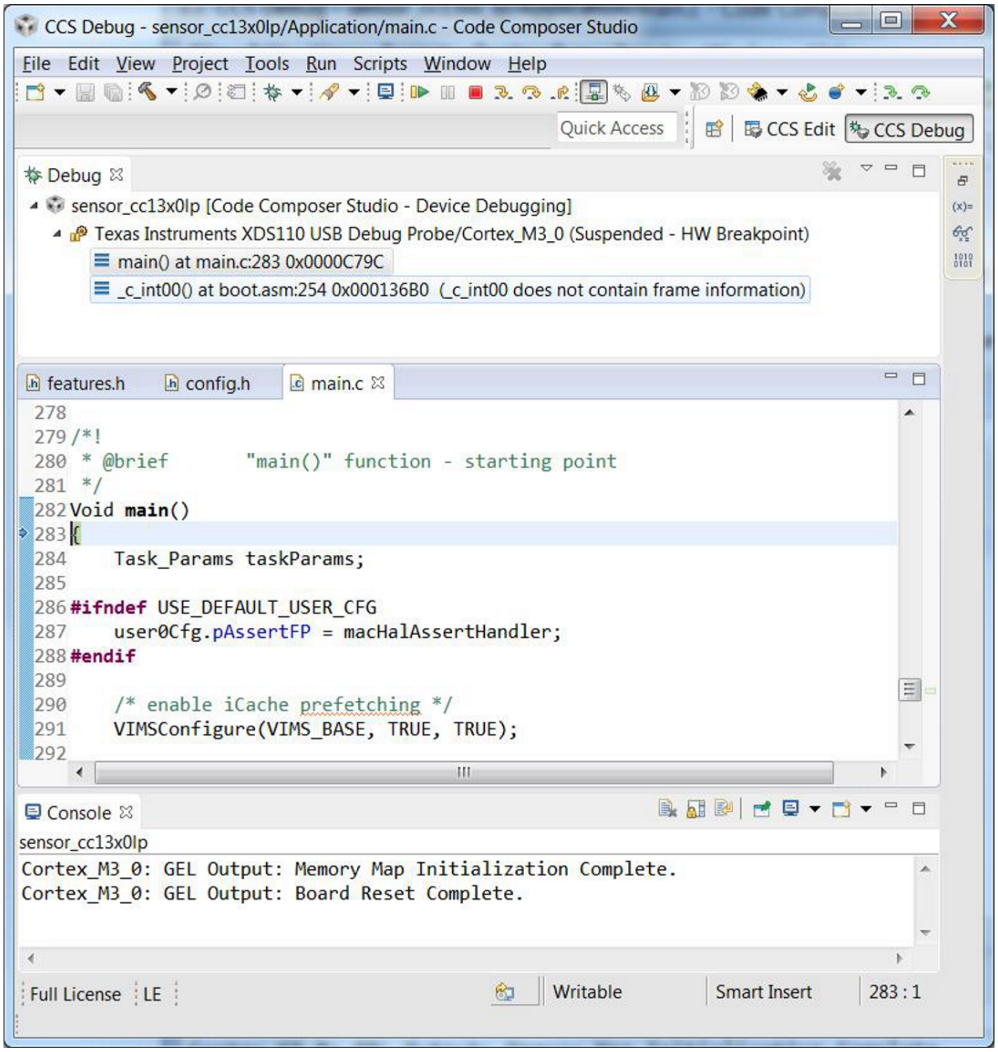


Figure 2-10. Debugging Sensor Application

In Figure 2-10, the IDE has switched from the CCS Edit perspective to *CCS Debug* and shows the program counter stopped at main(). From this starting point, the developer can single-step through source code, set and run-to breakpoints, and run the program using icons at the top of the display. During a debug session, the user can switch between the CCS Edit and CCS Debug perspectives, as necessary, to view project files and perform debugging operations.

2.5.2.8 Useful CCS IDE Settings

The CCS provides a large number of configurable settings that can be used to customize the IDE and individual projects. The following examples do not alter the generated program code, but they can improve the developer's experience when working with CCS projects. The CCS can reduce project compilation time by taking advantage of multiple processor cores on the development computer.

To use this feature, navigate to *Project → Properties → Build → Behavior* and select Enable parallel build, as shown in Figure 2-11.

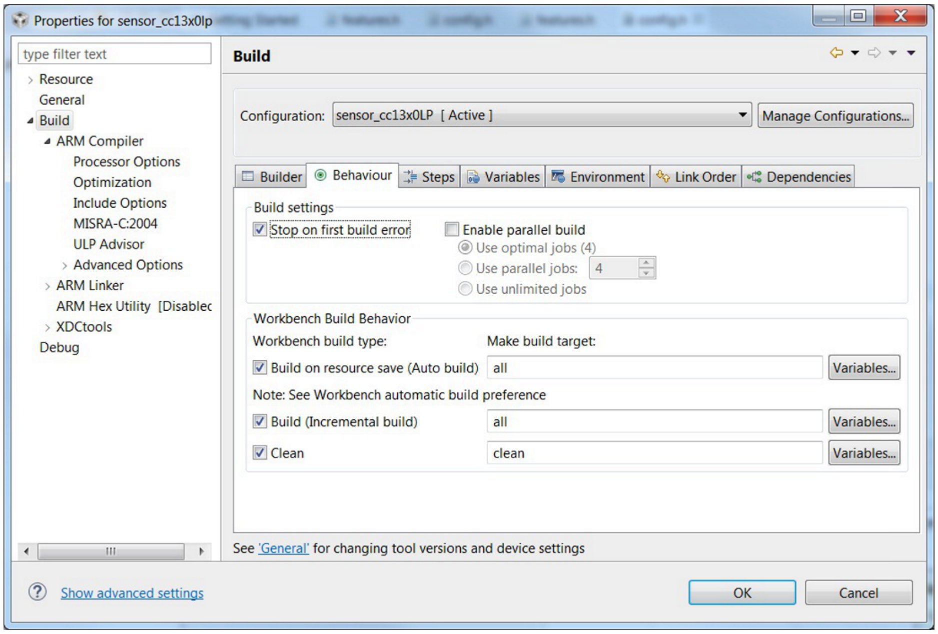


Figure 2-11. Properties for sensor_cc13x0lp

CCS users can control the amount of information that is displayed in the Console portion of the screen during project compilation and linking, ranging from Verbose to Super quiet. To change this setting, navigate to *Window → Preferences → Code Composer Studio → Build* and select an entry from the Console verbosity level drop-down, as shown in Figure 2-12.

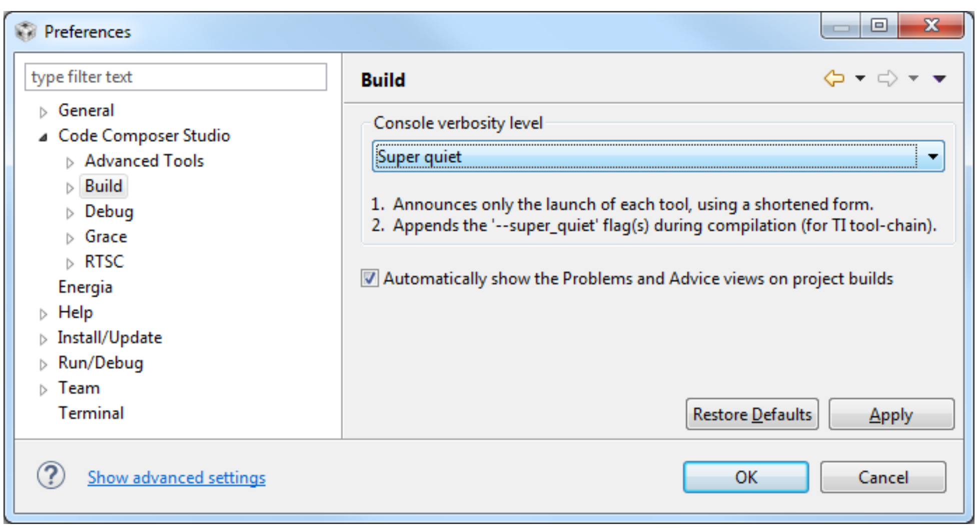


Figure 2-12. Console Verbosity Level Preferences

2.6 Accessing Preprocessor Symbols

Throughout this document and in the source code, various C preprocessor symbols may need to be defined or modified at the project level. Preprocessor symbols (also known as *Predefined Symbols*) are used to enable and disable features and set operational values to be considered when the program is compiled. A common way to disable an item without deleting it is to prefix an *x* to that item (see *xASSERT_LEDS* in Figure 2-13 for an example).

In CCS, preprocessor symbols are accessed by selecting and opening the appropriate Project Properties, then navigating to *CCS Build → ARM Compiler → Predefined Symbols*. To add, delete, or edit a preprocessor symbol, use one of the icons shown in the red box in Figure 2-13.

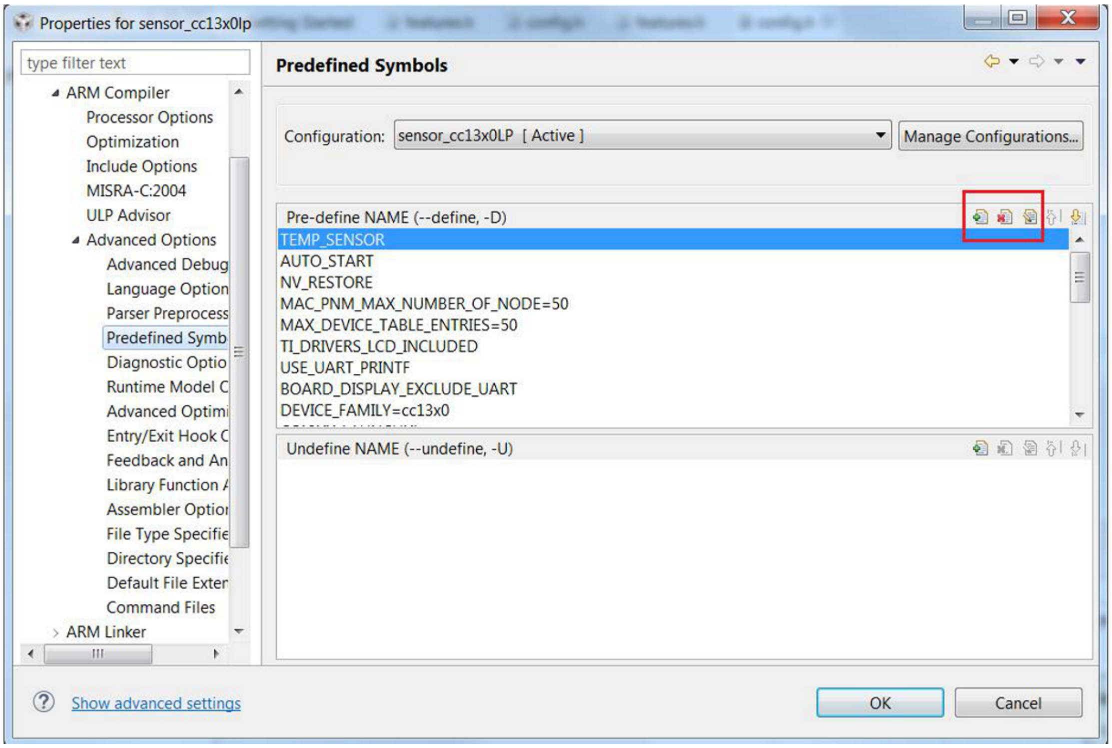


Figure 2-13. Predefined Symbols Pane

2.7 Top-Level Software Architecture

The TI 15.4-Stack software environment consists of three separate parts:

- A real-time operating system (RTOS)
- An Application
- A Stack

The TI-RTOS is a real-time, pre-emptive, multithreaded operating system that runs the software solution with task synchronization. Both the Application and MAC protocol stack exist as separate tasks within the RTOS, with the TI 15.4-Stack having the highest priority. A messaging framework, Indirect Call (ICall), is used for thread-safe synchronization between the Application and the Stack. Figure 2-14 illustrates the architecture.

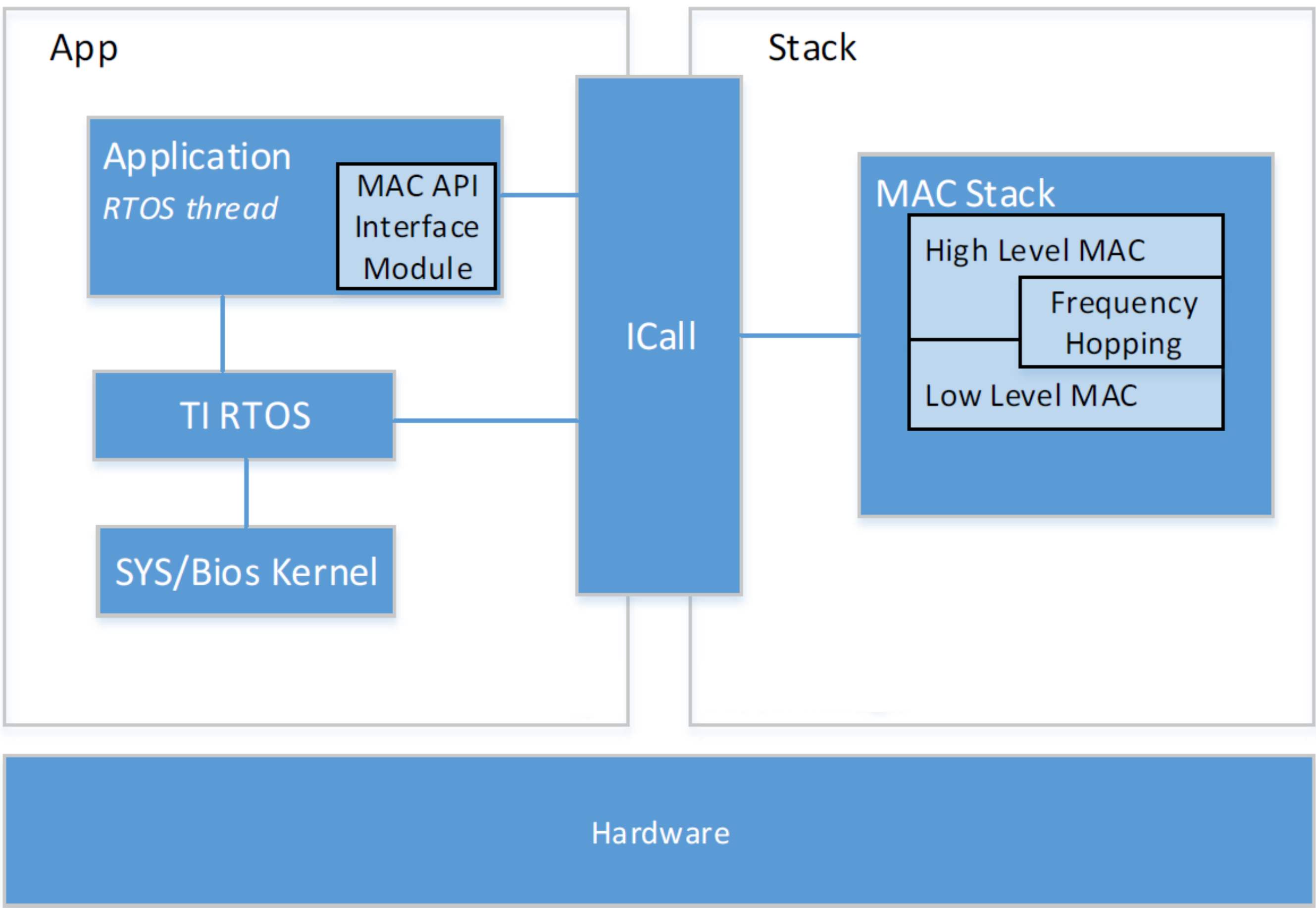


Figure 2-14. Software Architecture

- The Application
  – Includes the application code, drivers, TI-RTOS, and the ICall module
- The Stack
  – Includes the TI 15.4-Stack
    - High-Level MAC is the API with the application, handles protocol messaging and data queues, controls the personal area network information bases (PIBs).
    - Frequency Hopping maintains frequency-hopping schedules and neighbor tracking.
    - Low-Level MAC handles low level timing, encryption and decryption, and interfaces to the PHY.

RTOS Overview

TI-RTOS is the operating environment for CC13x0 SimpleLink SDK projects. The TI-RTOS kernel is a tailored version of the SYS/BIOS kernel and operates as a real-time, pre-emptive, multithreaded operating system with tools for synchronization and scheduling (XDCTools). The SYS/BIOS kernel manages four distinct levels of execution threads (see Figure 3-1).

- Hardware interrupt service routines (ISRs)
- Software interrupt routines
- Tasks
- Background idle functions

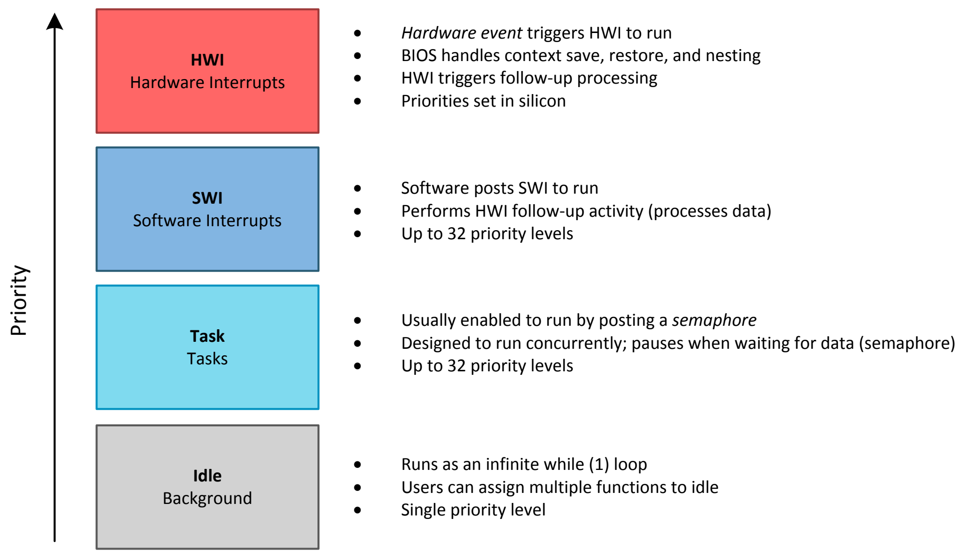


Figure 3-1. RTOS Execution Threads

This chapter describes the four execution threads and various structures used throughout the TI-RTOS for messaging and synchronization. In some cases in TI 15.4-Stack application projects, the underlying RTOS functions have been abstracted to higher-level functions (for example, in the *timer.c* file). The lower-level RTOS functions are described in the SYS/BIOS module section of the *TI SYS/BIOS API Guide*. This document also defines the packages and modules included with the TI-RTOS.

3.1 RTOS Configuration

The SYS/BIOS kernel provided with the installer can be modified using the RTOS configuration file (that is for example, app.cfg for the collector_cc13x0lp project). In the CCS project, this configuration file is in the application project workspace. This configuration file defines the various SYS/BIOS and XDCTools modules in the RTOS compilation, as well as system parameters such as exception handlers and timer-tick speed. The RTOS must then be recompiled for these changes to take effect by recompiling the project.

The default project configuration is to use elements of the RTOS from the CC13x0 ROM. In this case, some RTOS features are unavailable. If any ROM-unsupported features are added to the RTOS configuration file, use an RTOS in flash configuration. Using RTOS in flash consumes additional flash memory. The default RTOS configuration supports all the features required by the respective example projects in the SDK.

See the TI-RTOS documentation for a full description of configuration options.

NOTE: With the CC13x0 SimpleLink SDK 1.0 Release, TI recommends not changing the TI-RTOS version.

3.2 Semaphores

The kernel package provides several modules for synchronizing tasks such as the semaphore. Semaphores are the prime source of synchronization in the CC13x0 software and are used to coordinate access to a shared resource among a set of competing tasks (that is, the application and TI 15.4-Stack). Semaphores are used for task synchronization and mutual exclusion.

Figure 3-2 shows the semaphore functionality. Semaphores are either counting semaphores or binary semaphores. Counting semaphores keep track of the number of times the semaphore is posted with Semaphore_post(). When a group of resources are shared between tasks, this tracking function is useful. Such tasks might call Semaphore_pend() to see if a resource is available before using it. Binary semaphores can have only two states: available (count = 1) and unavailable (count = 0). Binary semaphores can be used to share a single resource between tasks or for a basic-signaling mechanism where the semaphore can be posted multiple times. Binary semaphores do not keep track of the count; instead, they track only whether the semaphore has been posted.

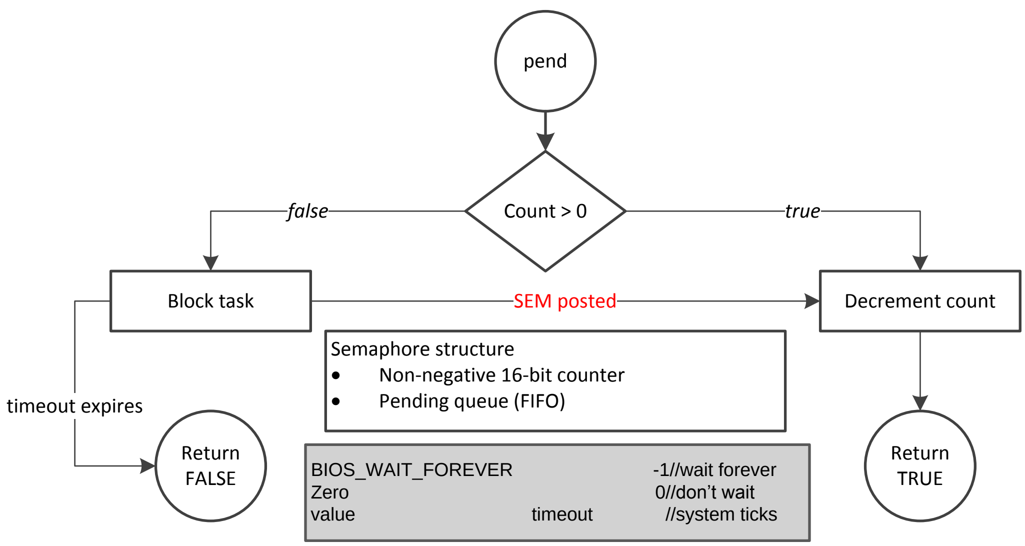


Figure 3-2. Semaphore Functionality

3.2.1 Initializing a Semaphore

The following code depicts how a semaphore is initialized in RTOS. An example of this process in the collector_cc13x0lp project is when a task is registered with the ICall module: ICall_registerApp(), which eventually calls ICall_primRegisterApp(). These semaphores coordinate task processing. Section 4.2 further describes this coordination.

```
Semaphore_Handle sem;
sem = Semaphore_create(0, NULL, NULL);
```

3.2.2 Pending a Semaphore

Semaphore_pend() is a blocking call that lets another task run while waiting for a semaphore. The time-out parameter lets the task wait until a time-out, wait indefinitely, or not wait at all. The return value indicates if the semaphore was signaled successfully.

```
Semaphore_pend(sem, timeout);
```

3.2.3 Posting a Semaphore

Semaphore_post() signals a semaphore. If a task is waiting for the semaphore, this call removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, Semaphore_post() increments the semaphore count and returns. For a binary semaphore, the count is always set to 1.

```
Semaphore_post(sem);
```

3.3 RTOS Tasks

RTOS tasks are equivalent to independent threads that conceptually execute functions in parallel within a single C program. In reality, switching the processor from one task to another helps achieve concurrence. Each task is always in one of the following modes of execution:

- Running: task is currently running
- Ready: task is scheduled for execution
- Blocked: task is suspended from execution
- Terminated: task is terminated from execution
- Inactive: task is on inactive list

Only one task is always running, even if that task is the idle task (see Figure 3-1). The currently running task can be blocked from execution by calling certain task-module functions as well as functions provided by other modules like semaphores. The current task can also terminate itself. In either case, the processor is switched to the highest priority task that is ready to run. See the task module in the package *ti.sysbios.knl* section of the *TI SYS/BIOS API Guide* for more information on these functions.

Numeric priorities are assigned to tasks and multiple tasks can have the same priority. Tasks are readied to execute by highest-to-lowest priority level (5 is the highest and 1 is the lowest); tasks of the same priority are scheduled in the order of arrival. The priority of the currently running task is never lower than the priority of any ready task. The running task is preempted and rescheduled to execute when there is a ready task of higher priority. In the collector_cc13x0lp application, the TI 15.4-Stack protocol stack task is given the highest priority (5) and the application task is given the lowest priority (1).

Each RTOS task has an initialization function, an event processor, and one or more callback functions.

3.3.1 Creating a Task

When a task is created, the task has its own runtime stack for storing local variables and further nesting of function calls. All tasks executing within one program share a common set of global variables that are accessed according to the standard rules of scope for C functions. This set of memory is the context of the task. The following is an example of the application task being created in the collector_cc13x0lp project.

```
/* Configure task. */
Task_Params_init(&taskParams);
taskParams.stack = myTaskStack;
taskParams.stackSize = APP_TASK_STACK_SIZE;
taskParams.priority = 1;
Task_construct(&myTask, taskFxn, &taskParams, NULL);
```

The task creation is done in the main() function, before the SYS/BIOS scheduler is started by BIOS_start(). The task executes at the assigned priority level after the scheduler is started. TI recommends using the existing application task for application-specific processing. When adding an additional task to the application project, the priority of the task must be assigned a priority within the RTOS priority-level range, which is defined in the app.cfg RTOS configuration file.

```
/* Reduce number of Task priority levels to save RAM */
Task.numPriorities = 6;
```

Do not add a task with a priority equal to or higher than the TI 15.4-Stack protocol stack task. Ensure the task has a minimum task stack size of 512 bytes of predefined memory. At a minimum, each stack must be large enough to handle normal subroutine calls and one task preemption context. A task preemption context is the context that is saved when one task preempts another as a result of an interrupt thread readying a higher priority task. Using the TI-RTOS profiling tools of the IDE, the task can be analyzed to determine the peak usage of the task stack.

NOTE: The term *created* describes the instantiation of a task. The actual TI-RTOS method is to construct the task. See Section 3.11.3 for details on constructing RTOS objects.

3.3.2 *Creating the Task Function*

When a task is constructed, a function pointer to a task function (for example, taskFxn) is passed to the Task_Construct function. When the task first gets a chance to process, this is the function which the RTOS runs. Figure 3-3 shows the general topology of this task function.

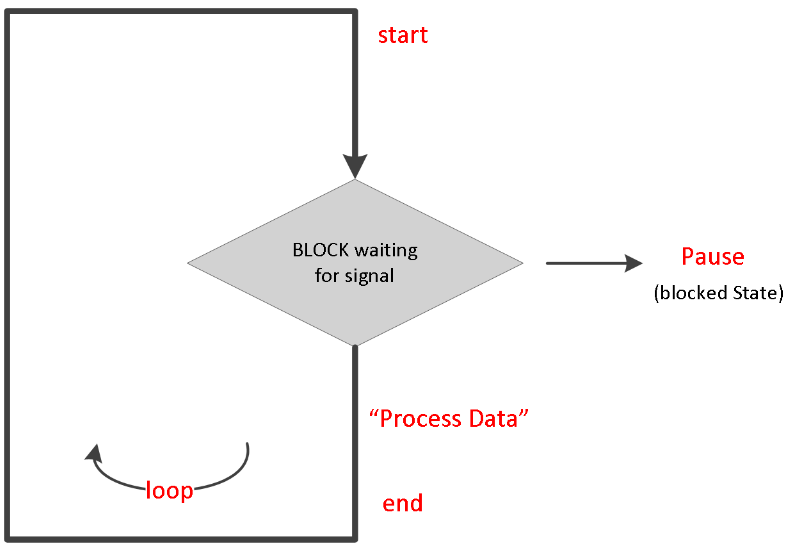


Figure 3-3. General Task Topology

In the collector_cc13x0lp task, the task spends most of its time in the blocked state, where it is pending a semaphore. When the semaphore of the task is posted to from an ISR, callback function, queue, and so forth, the task becomes ready, processes the data, and returns to this paused state. See Section 4.2.1 for more detail on the functionality of the collector_cc13x0lp task.

3.4 Clocks

Clock instances are functions that can be scheduled to run after a certain number of clock ticks. Clock instances are either one-shot or periodic. These instances start immediately upon creation, are configured to start after a delay, and can be stopped at any time. All clock instances are executed when they expire in the context of a software interrupt. The following example shows the minimum resolution is the RTOS clock tick period set in the RTOS configuration

```
/* 10us tick period */
Clock.tickPeriod = 10;
```

Each tick, which is derived from the RTC, launches a clock software interrupt (SWI) that compares the running tick count with the period of each clock to determine if the associated function should run. For higher-resolution timers, TI recommends using a 16-bit hardware timer channel or the sensor controller.

3.4.1 API

Developers can use the RTOS clock module functions directly (see the clock module in the *TI SYS/BIOS API Guide*). For usability, these functions have been extracted to various functions in the timer.c file. Refer to the timer.h file in the Application folder of the application projects for the available APIs.

3.4.2 Functional Example

The following example from the collector_cc13x0lp project details the creation of a clock instance, and describes how to handle the expiration of the instance.

1. Define the clock handler function to service the clock expiration SWI. csf.c

```
/* Join permit timeout handler function */
static void processJoinTimeoutCallback(UArg a0)
{
    (void)a0; /* Parameter is not used */

    Cllc_events |= CLLC_JOIN_EVT;

    /* Wake up the application thread when it waits for clock event */
    Semaphore_post(collectorSem);
}
```

2. Create the clock instance.

```
STATIC Clock_Struct joinClkStruct;
STATIC Clock_Handle joinClkHandle;
void Csf_initializeJoinPermitClock(void)
{
    /* Initialize join permit timer */
    joinClkHandle = Timer_construct(&joinClkStruct,
                                    processJoinTimeoutCallback,
                                    JOIN_TIMEOUT_VALUE,
                                    0,
                                    false,
                                    0);
}
```

3. Wait for the clock handler to expire and process in the application context (in Figure 3-4 green corresponds to the processor running in the application context and red corresponds to an SWI).

```
/*csf.c*/
/* join permit clock handler function */
static void processJoinTimeoutCallback(UArg a0)
{
    (void)a0; /* Parameter is not used */

    Cllc_events |= CLLC_JOIN_EVT;

    /* Wake up the application thread when it waits for clock event */
    Semaphore_post(collectorSem);
}
/*Cllc.c*/
/* Process join permit event */
  if(Cllc_events & CLLC_JOIN_EVT)
  {
      joinPermitExpired();
      /* Clear the event */
      Cllc_events &= ~CLLC_JOIN_EVT;
  }
```

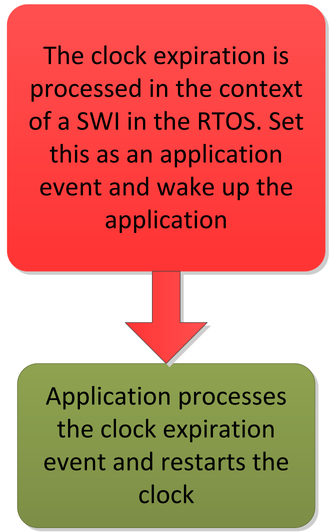


Figure 3-4. Clock Expiration Flow Diagram

3.5 Queues

Queues let applications process events in sequence by providing a FIFO ordering for event processing. A project may use a queue to manage internal events coming from application profiles or another task. Clocks must be used when an event must be processed in a time-critical manner. Queues are more useful for events that must be processed in a specific sequence.

The Queue module provides a unidirectional method of message passing between tasks using a FIFO. In Figure 3-5, a queue is configured for unidirectional communication from task A to task B. Task A pushes messages onto the queue and task B pops messages from the queue in sequence. Figure 3-5 shows the queue messaging process.

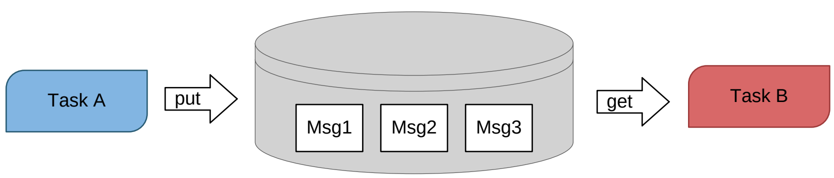


Figure 3-5. Queue Messaging Process

3.5.1 Queue API

Refer to the Queue module in the *TI SYS/BIOS API Guide* for details on the APIs.

3.6 Idle Task

The Idle module specifies a list of functions to be called when no other tasks are running in the system. In the CC13x0 software, the idle task runs the Power Policy Manager.

3.7 Power Management

Power-management functionality is handled by the peripheral drivers and the TI 15.4-Stack protocol stack project. The TI 15.4-Stack protocol stack project is configured to always use low power and allow the device to enter sleep mode whenever possible. More information on power-management functionality, including the API and a sample use case for a custom UART driver, are in TI-RTOS Power Management for CC13x0 included in the RTOS install. These APIs are required only when using a custom driver.

3.8 Hardware Interrupts

Hardware interrupts (HWIs) handle critical processing that the application must perform in response to external asynchronous events. The SYS/BIOS device-specific HWI modules are used to manage hardware interrupts. Specific information on the nesting, vectoring, and functionality of interrupts can be found in the *TI CC13xx ,CC26xx SimpleLink Wireless MCU Technical Reference Manual*. The *SYS/BIOS User Guide* details the HWI API and provides several software examples.

HWIs are abstracted through the peripheral driver to which they pertain (see the relevant driver in Chapter 6). Chapter 9 provides an example of using GPIOs as HWIs. Abstracting through the peripheral driver to which they pertain is the preferred method of using interrupts. Using the Hwi_plug() function, ISRs which do not interact with SYS/BIOS can be written. These ISRs must do their own context preservation to prevent breaking the time-critical TI 15.4-Stack.

For the TI 15.4-Stack to meet RF time-critical requirements, all application-defined HWIs execute at the lowest priority. TI does not recommend modifying the default HWI priority when adding new HWIs to the system. No application-defined critical sections should exist to prevent breaking the RTOS or time-critical sections of the TI 15.4-Stack. Code that executes in a critical section prevents processing of real-time interrupt-related events.

3.9 Software Interrupts

See the *TI SYS/BIOS API Guide* for detailed information about the software interrupts (SWIs) module. Software interrupts have priorities that are higher than tasks, but lower than the priorities of hardware interrupts (see Figure 3-6). The amount of processing in an SWI must be limited, because this processing takes priority over the TI 15.4-Stack task. As described in Section 3.4, the clock module uses SWIs to preempt tasks. The only processing the clock handler SWI does is set an event and post a semaphore for the application to continue processing outside of the SWI. Whenever possible, the clock module should be used to implement SWIs. An SWI can be implemented with the SWI module as described in the TI SYS/BIOS API Guide.

NOTE: To preserve the RTOS heap, the amount of dynamically created SWIs must be limited as described in Section 3.11.3.

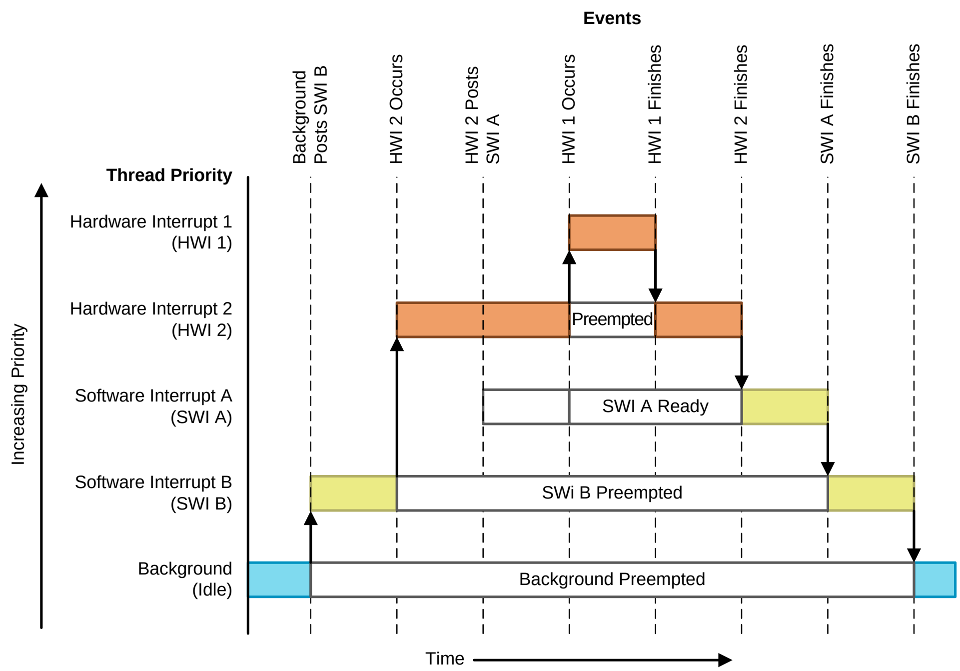


Figure 3-6. Preemption Scenario

3.10 Flash

The flash is split into erasable pages of 4 KB. The various sections of flash and the associated linker configuration file (cc13x0lp.cmd).

- Application space: contains example application (or your application), MAC stack, RTOS, drivers, and so on
- Nonvolatile (NV) area used for NV memory storage by the Application. See Section 3.10.1 for configuring NV.
- Customer Configuration Area (CCA): the last sector of flash used to store customer specific chip configuration (CCFG) parameters

3.10.1 Using Nonvolatile Memory

The NV area of flash is used for storing persistent data for the application. The last page in flash is the CCA page, the two pages before the last page (CCA) are defined as the NV area. The NV driver uses one page to store the persistent data and one page as an erase page, so the application has 4 KB (one page) of storage. The Collector and Sensor projects use the NV On-Chip Two-Page (NVOCTP) NV driver (nvoctp.c) with the API defined in nvintf.h

The NV driver is set up in main.c

```
#ifdef NV_RESTORE
    /* Setup the NV driver */
    NVOCTP_loadApiPtrs(&Main_user1Cfg.nvFps);

    if(Main_user1Cfg.nvFps.initNV)
    {
        Main_user1Cfg.nvFps.initNV( NULL);
    }
#endif
```

Then the applications use the function pointers in Main_user1Cfg to call the NV functions defined in nvintf.h

```
//! Structure of NV API function pointers
typedef struct nvintf_nvfuncts_t
{
    //! Initialization function
    NVINTF_initNV initNV;
    //! Compact NV function
    NVINTF_compactNV compactNV;
    //! Create item function
    NVINTF_createItem createItem;
    //! Delete NV item function
    NVINTF_deleteItem deleteItem;
    //! Read item function
    NVINTF_readItem readItem;
    //! Write item function
    NVINTF_writeItem writeItem;
    //! Write existing item function
    NVINTF_writeItemEx writeItemEx;
    //! Get item length function
    NVINTF_getItemLen getItemLen;
} NVINTF_nvFuncts_t;
```

The following is an example of a write from csf.c

```
static void updateDeviceListItem(Llc_deviceListItem_t *pItem)
{
    if((pNV != NULL) &&  (pItem != NULL))
    {
        int idx;
        idx = findDeviceListIndex(&pItem->devInfo.extAddress);
        if(idx != DEVICE_INDEX_NOT_FOUND)
        {
            NVINTF_itemID_t id;

            /* Setup NV ID for the device list record */
            id.systemID = NVINTF_SYSID_APP;
            id.itemID = CSF_NV_DEVICELIST_ID;
            id.subID = (uint16_t)idx;

            /* write the device list record */
            pNV->writeItem(id, sizeof(Llc_deviceListItem_t), pItem);
        }
    }
}
```

The following is an example of a read from csf.c

```
bool Csf_getNetworkInformation(Llc_netInfo_t *pInfo)
{
    if((pNV != NULL) && (pNV->readItem != NULL) && (pInfo != NULL))
    {
        NVINTF_itemID_t id;

        /* Setup NV ID */
        id.systemID = NVINTF_SYSID_APP;
        id.itemID = CSF_NV_NETWORK_INFO_ID;
        id.subID = 0;

        /* Read Network Information from NV */
if(pNV->readItem(id, 0, sizeof(Llc_netInfo_t), pInfo) == NVINTF_SUCCESS)
        {
            return(true);
        }
    }
    return(false);
}
```

The NV system is a collection of NV items. Each item is unique and have the following pieces to it (defined in nvintf.h).

```
/**
 * NV Item Identification structure
 */
typedef struct nvintf_itemid_t
{
    //! NV System ID - identifies system (ZStack, BLE, App, OAD...)
    uint8_t systemID;
    //! NV Item ID
    uint16_t itemID;
    //! NV Item sub ID
    uint16_t subID;
} NVINTF_itemID_t;
```

3.11 Memory Management (RAM)

Space for RAM is configured in the linker configuration file (cc13x0lp.cmd).

Application Image: RAM space for the Application and shared heaps. This space is configured in the linker config file of the Application, cc13x0lp.cmd

3.11.1 System Stack

Besides the RTOS and ICall heaps previously mentioned, there are other sections of memory to consider. As described in Section 3.3.1, each task has its own runtime stack for context switching. Furthermore, another runtime stack is used by the RTOS for main(), HWIs, and SWIs. This system stack is allocated in the Application linker file, to be placed at the end of the RAM of the Application.

For CCS, the RTOS system stack is defined by the Program.stack parameter in the app.cfg RTOS configuration file.

```
/* main() and Hwi, Swi stack size */
Program.stack = 1280;
```

Then the RTOS system stack is placed by the linker in the RAM space of the Application:

```
/* Create global constant that points to top of stack */
/* CCS: Change stack size under Project Properties    */
__STACK_TOP = __stack + __STACK_SIZE;
```

3.11.2 Dynamic Memory Allocation

The system uses two heaps for dynamic memory allocation. It is important to understand the use of each heap so that the application designer maximizes the use of available memory. The RTOS is configured with a small heap in the app.cfg RTOS configuration file.

```
var HeapMem = xdc.useModule('xdc.runtime.HeapMem');

BIOS.heapSize = 1724;
```

This heap (HeapMem) is used to initialize RTOS objects as well as allocate the TI 15.4-Stack task runtime stack. This size of this heap has been chosen to meet the system initialization requirements. Due to the small size of this heap, TI does not recommend allocating memory from the RTOS heap for general application use. For more information on the TI-RTOS heap configuration, refer to the *Heap Implementations* section of the *TI-RTOS SYS/BIOS Kernel User's Guide*.

Instead, a separate heap must be used by the Application. The ICall module statically initializes an area of Application RAM, heapmgrHeapStore, which can be used by the various tasks. The size of this ICall heap is defined by the preprocessor definition of the Application HEAPMGR_SIZE, and is set to 0 by default for the Collector and Sensor projects; a value of 0 means that all unused RAM is given to the ICall heap. Although the ICall heap is defined in the Application project, it is also shared with the TI 15.4-Stack APIs which allocate memory from the ICall heap. To manually change the size of the ICall heap, adjust the value of the preprocessor symbol HEAPMGR_SIZE in the Application project to a value other than 0.

To profile the amount of ICall heap used, define the HEAPMGR_METRICS preprocessor symbol in the Application project. Refer to heapmgr.h in Components\applib\heap for available heap metrics. The following is an example of dynamically allocating a variable length (n) array using the ICall heap.

```
//define pointer
uint8_t *pArray;

// Create dynamic pointer to array.
if (pArray = (uint8_t*)ICall_malloc(n*sizeof(uint8_t)))
{
  //fill up array
}
else
{
  //not able to allocate
}
```

The following is an example of freeing the previous array.

```
ICall_free(pMsg->payload);
```

3.11.3 A Note on Initializing RTOS Objects

Due to the limited size of the RTOS heap, TI strongly recommends that users construct and not create RTOS objects. To illustrate this recommendation, consider the difference between the Clock_construct() and Clock_create() functions. Figure 3-7 shows the definitions of these functions from the SYS/BIOS API.

```
Clock_Handle Clock_create(Clock_FuncPtr clockFxn, UInt timeout, const Clock_Params *params, Error_Block *eb);
    // Allocate and initialize a new instance object and return its handle

Void Clock_construct(Clock_Struct *structP, Clock_FuncPtr clockFxn, UInt timeout, const Clock_Params *params);
    // Initialize a new instance object inside the provided structure
```

Figure 3-7. Definitions of Functions from the SYS/BIOS API

By declaring a static Clock_Struct object and passing this object to Clock_construct(), the .DATA section for the actual Clock_Struct is used, not the limited RTOS heap. Conversely, Clock_create() causes the RTOS to allocate Clock_Struct using the limited heap of the RTOS. As much as possible, this method is how clocks and RTOS objects in general, should be initialized throughout the project. If creating RTOS objects must be used, the size of the RTOS heap may need to be adjusted in app.cfg.

TI 15.4-Stack Overview

This chapter explains in detail the three different network-configuration modes supported by the TI 15.4-Stack for application development. Useful information is presented for developers using the TI 15.4-Stack for their custom application development, which lets developers quickly understand the basics of the selected configuration mode and develop their end products more quickly.

4.1 Beacon Enabled Mode

NOTE: In the following sections, the project names for the CC1310 and CC1350 platforms are referred to as CC13x0. Replace *x* with either *1* or *5* depending on the wireless MCU being used.

4.1.1 Introduction

The IEEE 802.15.4 specification defines beacon-enabled mode of operation where the personal area network (PAN) coordinator device transmits periodic beacons to indicate its presence and allows other devices to perform PAN discovery and synchronization. The beacons provide beacon-related information and also mark the start of the new superframe. The beacon has information on the superframe specification, which helps the device intending to join the network to synchronize timing- and network-related parameters before starting the join process. The beacon helps the existing device in the PAN to maintain the network synchronization. The superframe is divided into an active and an inactive period. During the active period, devices communicate using the CSMA/CA procedure. The inactive period allows the devices in the network to conserve energy.

4.1.2 Network Operations

This section describes critical network operations for the beacon-enabled mode of operation.

4.1.2.1 Network Start-Up

A network is always started by a fully functional device after performing a MAC sublayer reset. The network operates on a single channel (frequency hopping is not available in this configuration, although frequency agility may be implemented by application-specific means). To select the most optimal channel of operation, the fully functional device (before starting the network) can optionally scan for the channels with the least amount of radio interference by first performing the energy-detect scan to identify the list of channels with the least amount of RF energy. When a list of channels is identified, the fully functional device can (optionally) perform active scan to find the channel with the least number of active networks. When the channel with the least RF energy and lowest number of active networks is selected, the PAN coordinator must set its short address (the PAN identifier) beacon payload and turn on the associate permit flag. The network starts upon the following actions:

- Call to ApiMac_mlmeStartReq() API
  - With the PAN coordinator parameter set to *TRUE*
  - With the desired superframe configuration
  - Coordinator realignment parameter set to *FALSE*

Figure 4-1 shows the interaction between the application and the TI 15.4-Stack to start the beacon-enabled network by the PAN coordinator.

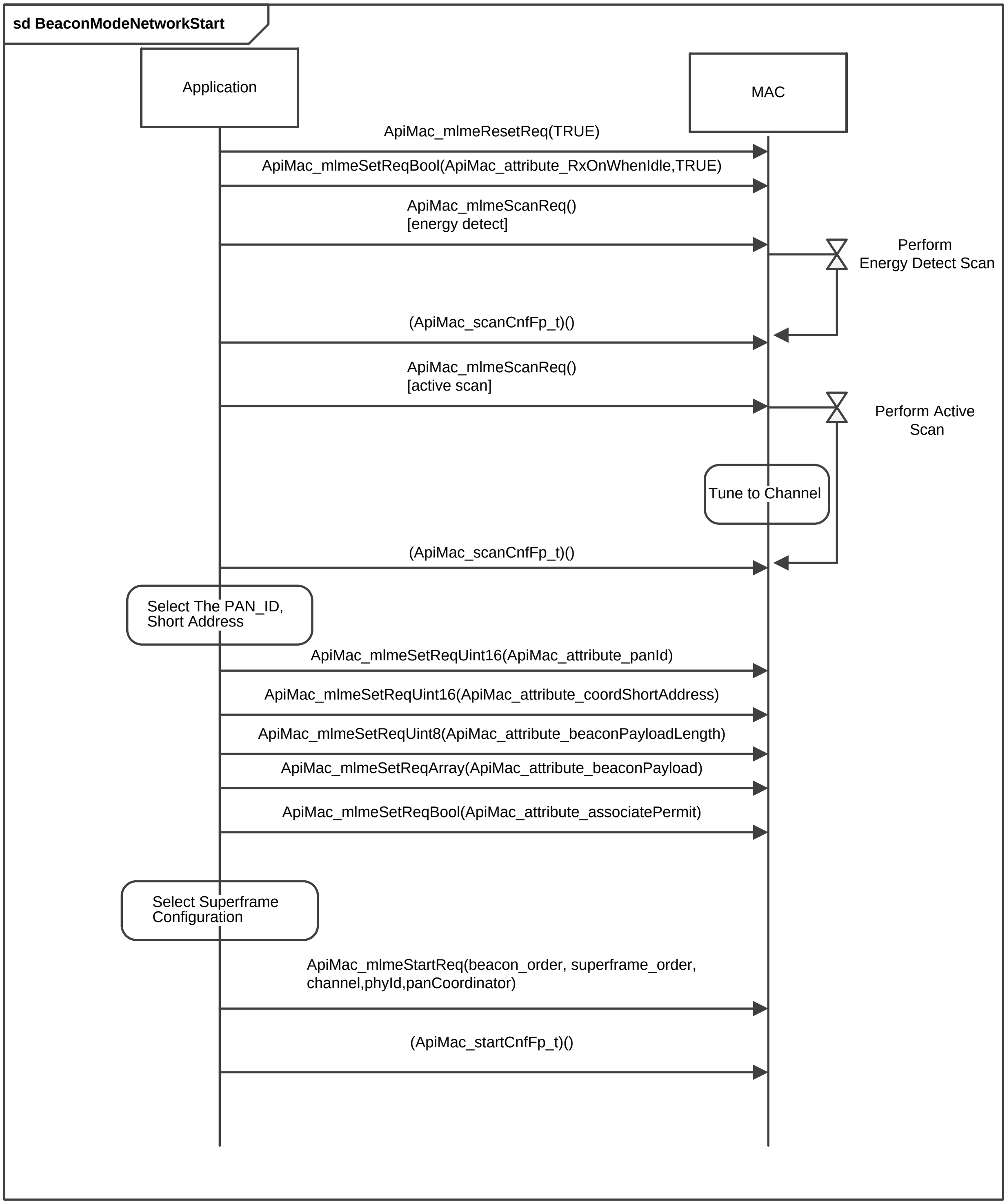


Figure 4-1. Beacon Mode Network Start-Up Sequence

4.1.2.2 Network Association

A device that is intended to join the beacon-enabled network must first perform a passive channel scan. The results of the channel scan can then be used to choose a suitable network. Following the selection of an association network, the application should set the following PIB items:

- ApiMac_attribute_beaconOrder
  – Set to the value received in the beacon superframe specification of the chosen PAN
- ApiMac_attribute_superframeOrder
  – Set to the value received in the beacon superframe specification of the chosen PAN
- ApiMac_attribute_panId
  – PAN identifier of the PAN
- ApiMac_attribute_coordShortAddress
  – Short address of the PAN coordinator

The next step is to perform beacon synchronization to track the beacon and to detect any pending messages. Synchronization is requested by using the ApiMac_mlmeSyncReq() API call and setting the following parameters:

- Channel
- PHY identifier
- Channel page
- Setting track beacon to TRUE

To acquire beacon synchronization, the device searches for the maximum time calculated as follows:

```
aBaseSuperframeDuration* (2n + 1), where n is the value of the beacon order
```

and

```
aBaseSuperframeDuration = aBaseSlotDuration × aNumSuperframeSlots = 60*16 = 960 symbols
```

Refer to the IEEE 802.15.4 specification for the definition of previous constants.

The device must to wait for the previously stated time period for the synchronization process to complete. Alternatively the device can turn off the Auto Request by setting the ApiMac_attribute_autoRequest attribute item to FALSE, which forces the MAC sublayer to send the beacon notification to the upper layer. If the application receives beacon notification indications for the normal beacon and no sync loss indication, it is a good indication that the device has synchronized with the coordinator beacons. TI recommends waiting for at least two beacon notifications before turning on the Auto Request.

When the device is synchronized to the network and is tracking the beacon, the application can perform the network association. The *ApiMac_mlmeAssociateReq()api* call is used to send the association request message to the coordinator. The association process is successful when the application receives the association confirmation.

Figure 4-2 shows a device performing the network association.

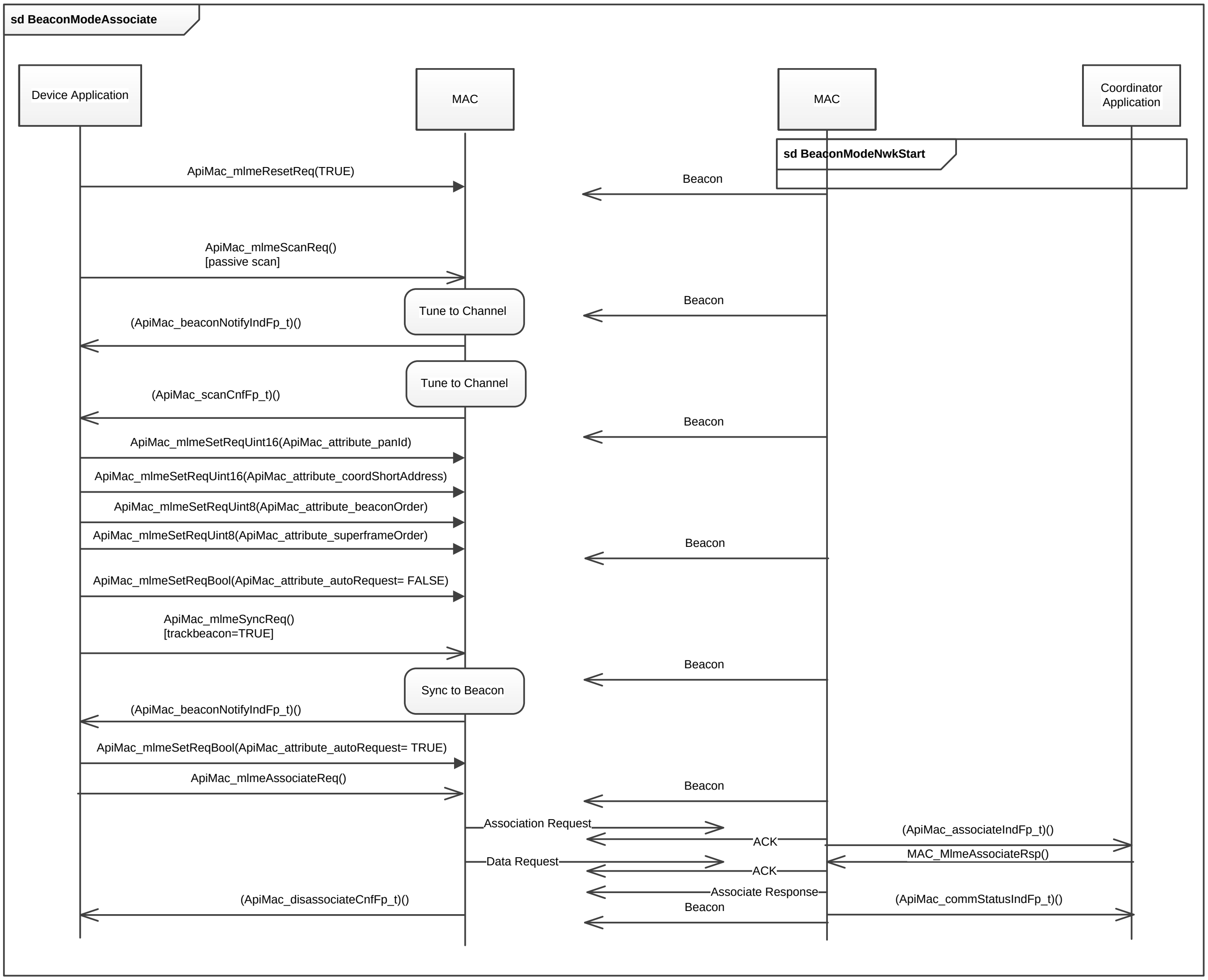


Figure 4-2. Beacon Mode Device Association Sequence

4.1.2.3 Data Exchange

The sequence diagram in Figure 4-3 depicts the various direct data transactions between an always-on (mains powered) device and a coordinator in a beacon-enabled network.

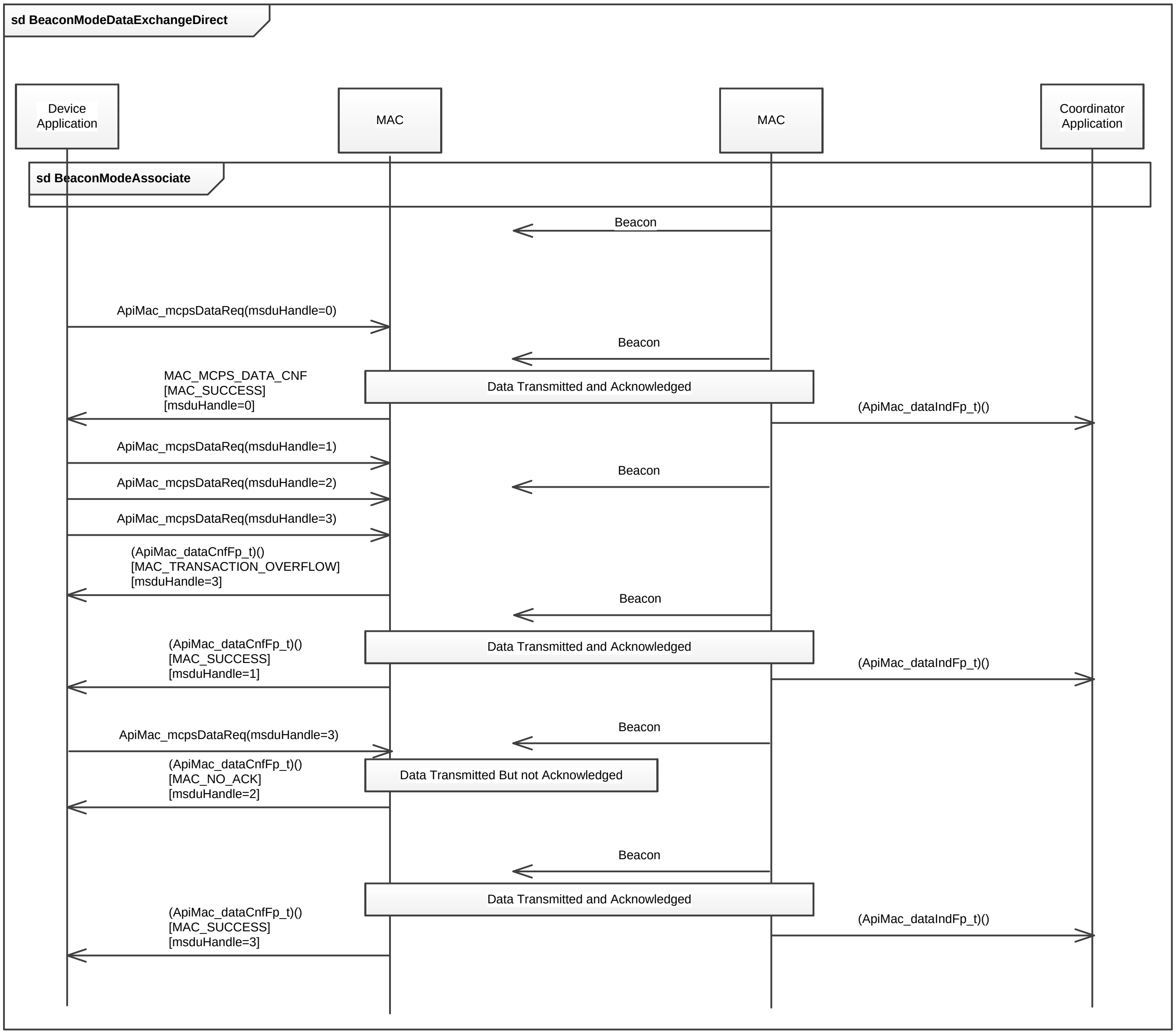


Figure 4-3. Beacon Mode Direct Data Exchange Sequence

The sequence diagram in Figure 4-4 depicts the indirect data transaction in a beacon-enabled network.

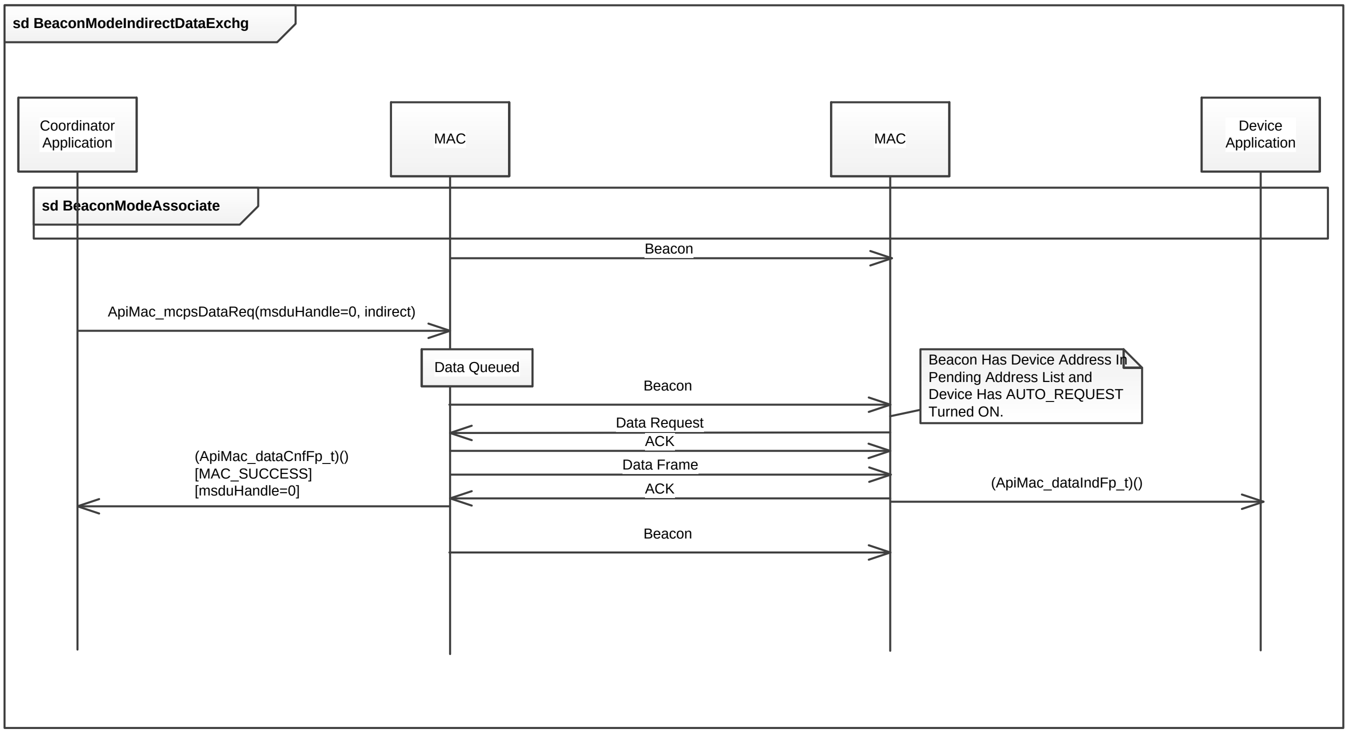


Figure 4-4. Beacon Mode Indirect Data Exchange Sequence

4.1.2.4 Maintaining a Connection for End Nodes

All devices operating on a beacon-enabled PAN must acquire beacon synchronization with a coordinator. Synchronization is performed by receiving and decoding the beacon frame information. The beacon frame contains the superframe specification which lets the device sync its timing information with the coordinator and detect any pending messages.

During the network association phase, the end device calls the *Api_mlmeSyncReq()* API with the *trackBeacon* parameter set to TRUE to acquire beacon and keep track of it (see Figure 4-5). With the track beacon set to TRUE, the MAC sublayer shall enable its receiver at a time before the next expected beacon frame transmission. If the number of consecutive beacons missed by the MAC sub layer reaches the maximum allowed (four beacons), the TI 15.4-Stack makes a callback *ApiMac_syncLossIndFp_t()* with a status of *ApiMac_status_beaconLoss* to the application. The application tries to resynchronize with the coordinator by calling *Api_mlmeSyncReq()* with the trackBeacon set to TRUE.

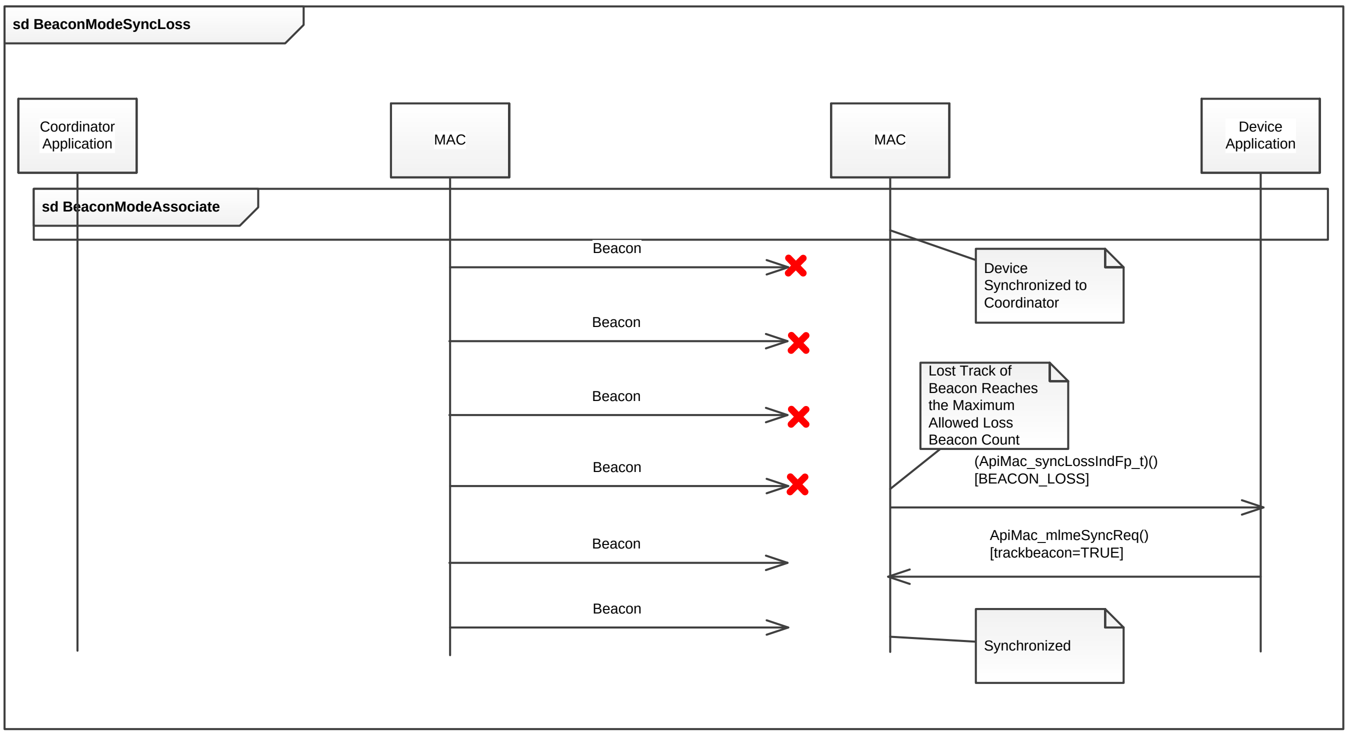


Figure 4-5. Beacon Mode Sync Loss Sequence

4.1.2.5 Network Disassociation

This section describes three scenarios. The first two scenarios are initiated by the coordinator (one for the mains powered end device and the other for the battery powered end device). In the third scenario, the end device disassociates itself from the network.

When the coordinator application requires an associated device to leave the network, the coordinator application requests that the TI 15.4-Stack send the disassociation notification command by using the ApiMac_mlmeDisassociateReq() call. If the txIndirect parameter is set to TRUE, the TI 15.4-Stack sends the disassociation notification command to the device using indirect transmission; then, the disassociation notification command is added to the list of pending transactions stored on the coordinator and pulled by the device using a data request command (see Figure 4-6).

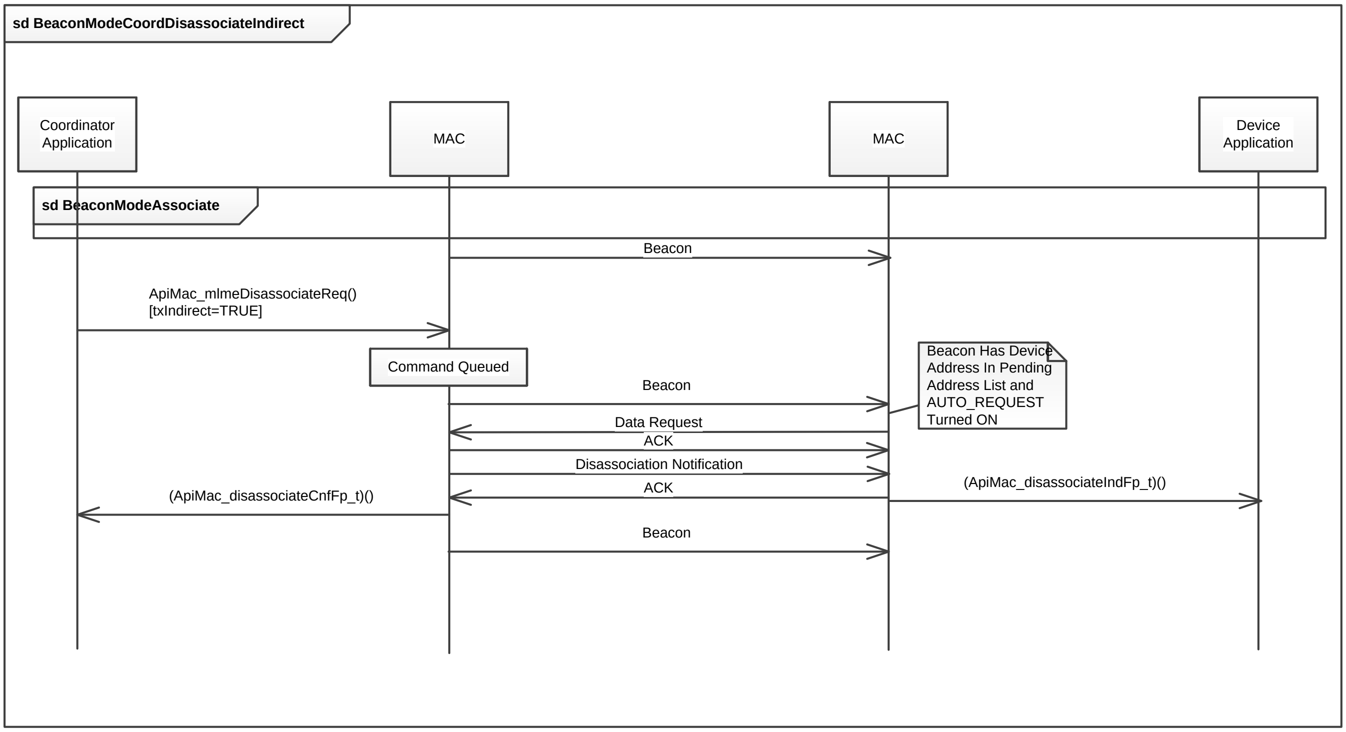


Figure 4-6. Beacon Mode Coordinator Initiated Indirect Disassociation Sequence

If the txIndirect parameter is set to FALSE, the TI 15.4-Stack sends the disassociation notification command frame to the device using direct transmission (see Figure 4-7). The TI 15.4-Stack layer at the coordinator makes a callback to the application using the registered function pointer of type ApiMac_disassociateCnfFp_t after completion of the disassociation. The TI 15.4-Stack at the device makes a callback to the application using the registered function pointer of type ApiMac_disassociateIndFp_t on reception of the disassociation notification command frame from the coordinator.

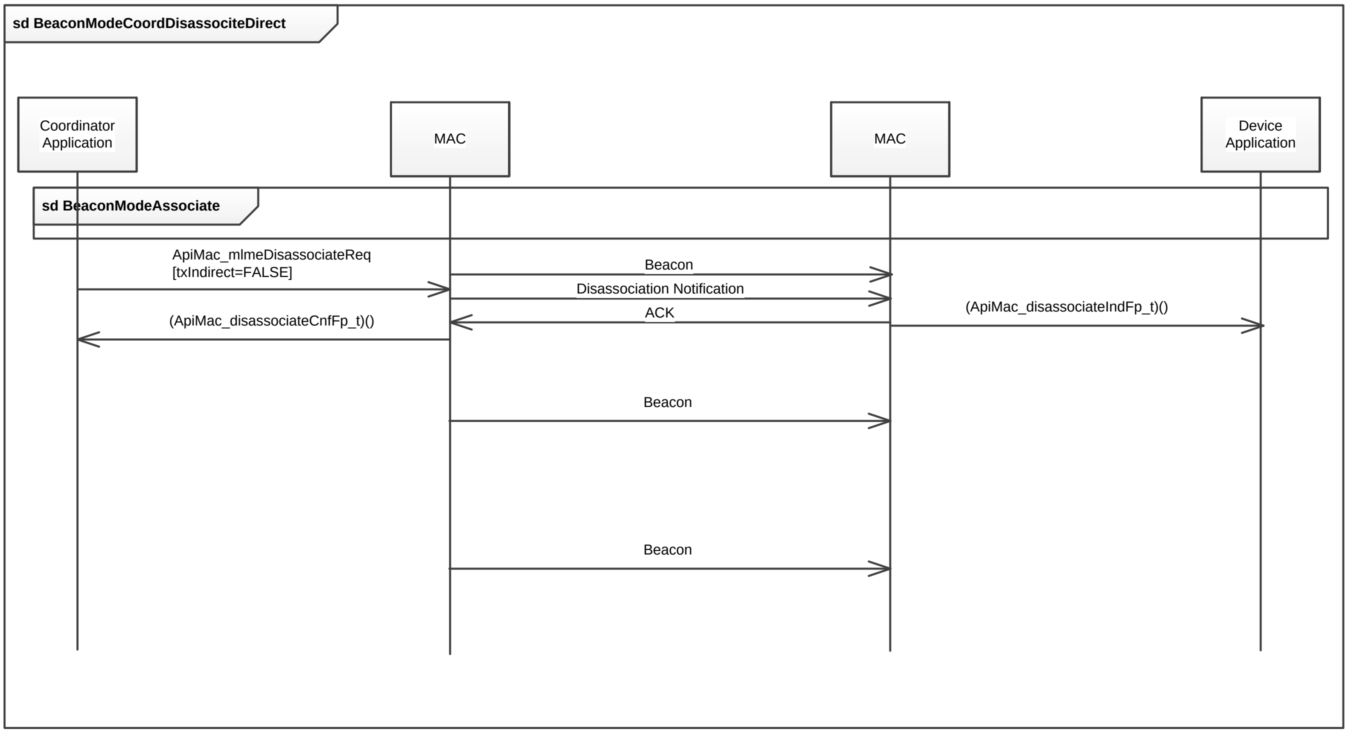


Figure 4-7. Beacon Mode Coordinator-Initiated Direct Disassociation Sequence

The end device application can also initiate the disassociation process as described in Figure 4-8.

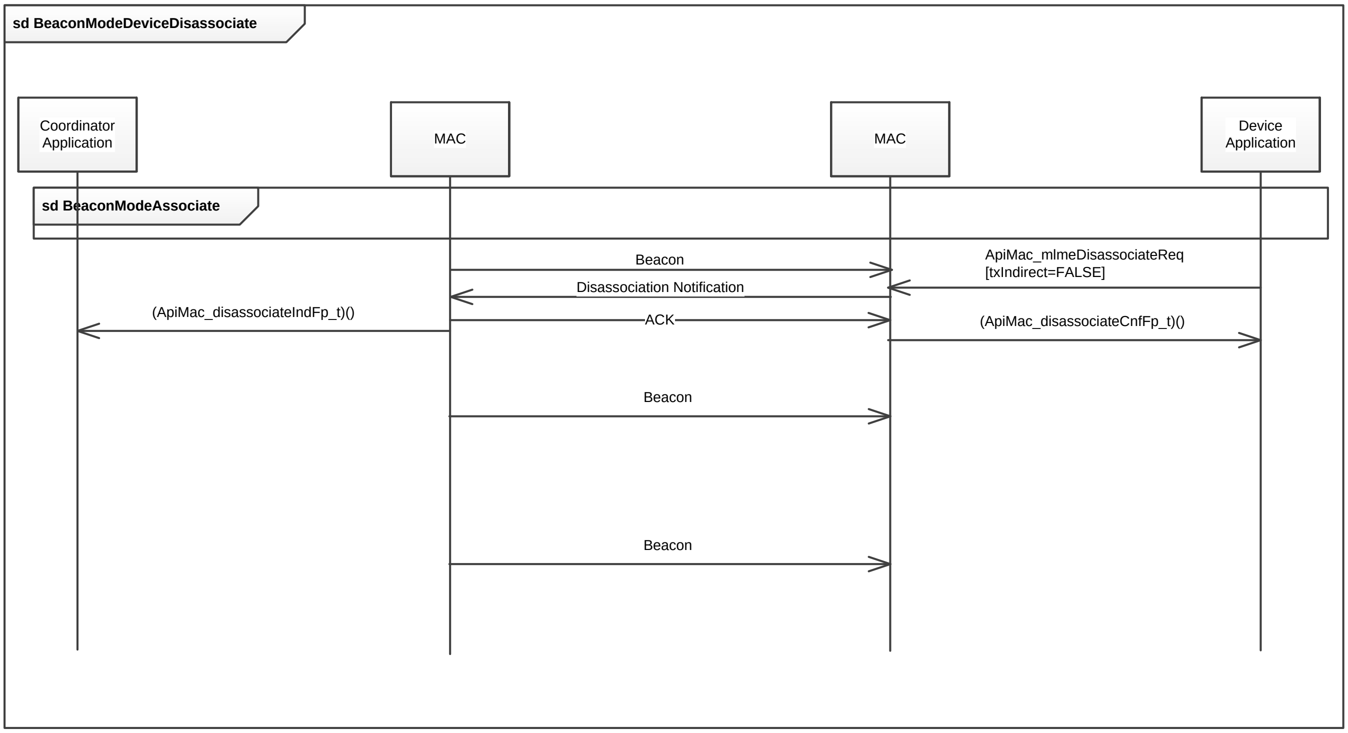


Figure 4-8. Beacon Mode Device-Initiated Disassociation Sequence

4.1.3 Stack Configuration Knobs

4.1.3.1 Attribute Configuration

Table 4-1 lists the attribute configuration for beacon mode.

Table 4-1. Attribute Configuration for Beacon Mode

Name	Type	Range	API Number	Description
ApiMac_attribute_associatePermit	bool	TRUE, FALSE	0x41	TRUE if a coordinator is currently allowing association
ApiMac_attribute_autoRequest	bool	TRUE, FALSE	0x42	TRUE if a device automatically sends a data request if its address is listed in the beacon frame
ApiMac_attribute_beaconOrder	uint8	0–15	0x47	How often the coordinator transmits a beacon
ApiMac_attribute_RxOnWhenIdle	bool	TRUE, FALSE	0x52	TRUE if the MAC enables its receiver during idle periods
ApiMac_attribute_superframeOrder	uint8	0–15	0x54	This specifies the length of the active portion of the superframe.

The ApiMac_attribute_associatePermit is used by the coordinator application to indicate to the joining devices whether it allows association or not. Setting this attribute item to TRUE by the coordinator indicates to the joining devices in its beacon frame that the coordinator application allows association.

The ApiMac_attribute_RxOnWhenIdle, if set to TRUE, enables the receiver during the idle period.

The ApiMac_attribute_RxOnWhenIdle, if set to TRUE, enables the receiver during the idle in the contention period of the superframe. The coordinator application sets this item to TRUE.

The ApiMac_attribute_beaconOrder item is used by the device application to set the beacon order during the joining phase, after the passive scan of beacons, and after the device makes the decision on which coordinator to join. This attribute shall be set to the selected coordinators beacon order value.

The ApiMac_attribute_superframeOrder item is used by the device application to set the superframe order during the joining phase, after the passive scan of beacons, and after the device makes the decision on which coordinator to join. This attribute shall be set to the selected coordinators beacon order value.

4.1.3.2 Configuration Constants

The TI 15.4-Stack uses a structure containing various user-configurable parameters (at compile time). This structure, called *macCfg_t*, is in the mac_cfg.c file. Table 4-2 describes the configuration elements.

Table 4-2. Configuration Constants

Name	Description	Range	Default
txDataMax	Maximum number of data frames queued in the transmit data queue.	1–255	2
txMax	Maximum number of frames of all types queued in the transmit data queue.	1–255	5
rxMax	Maximum number of frames queued in the receive data queue.	1–255	2
dataIndOffset	Allocate additional bytes in the data indication for application-defined headers.	0–127	0
appPendingQueue	When TRUE, registered callback of type ApiMac_pollIndFp_t will be made to the application when a data request command frame is received from another device.	TRUE or FALSE	FALSE

4.2 Nonbeacon Mode

4.2.1 Introduction

The IEEE 802.15.4 specification defines the nonbeacon mode of network operation where the coordinator does not send out periodic beacons. The nonbeacon mode is an asynchronous network mode of operation where the devices communicate by using the CSMA/CA mechanism.

4.2.2 Network Operations

4.2.2.1 Network Start-Up

A network is always started by a full function device. The procedure to start the network begins with a MAC layer reset. The application can directly start the network on a desired channel with a desired or random PAN-ID, or it can first check for a channel with lowest RF energy by performing a energy detect scan to select the channel with lowest energy or least interference (see Figure 4-9). The application then performs an active scan to detect the existing networks in the area, and select network parameters for its own network which do not conflict. After selecting the channel, the PAN Coordinator application must set the short address and PAN-ID, and then set the beacon payload and (optionally) turn on the associate permit flag if it wants new devices to join the network. The network is then started using the API ApiMac_mlmeStartReq() with the panCoordinator parameter set to TRUE.

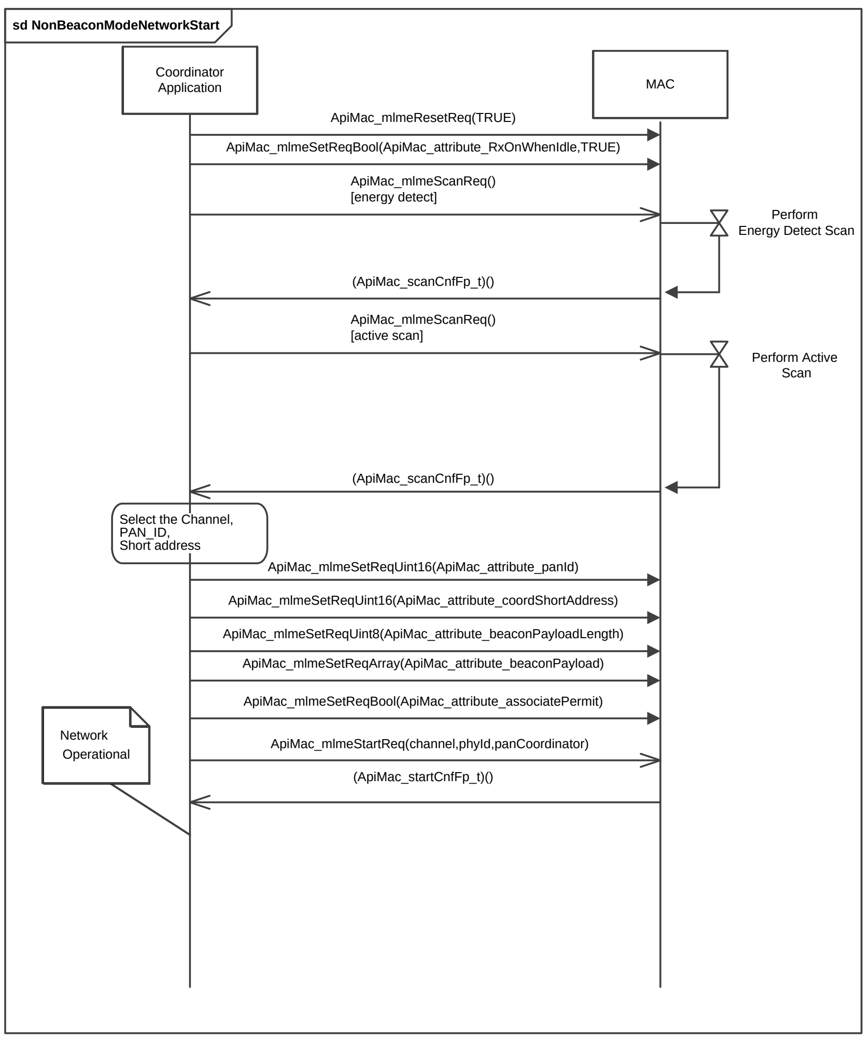


Figure 4-9. Nonbeacon Mode Network Start-Up Sequence

4.2.2.2 Network Join

When a device is ready to join a nonbeacon network, it must first perform an active scan broadcasting a beacon request. After receiving the beacon request, the nonbeacon PAN coordinators in the radio range of the device respond with their beacons. When the scan is complete, the TI 15.4-Stack calls the registered callback of type ApiMac_scanCnfFp_t with the PAN descriptors of the beacons it has received during the scan to the device application. The device application examines the PAN descriptors and selects a coordinator.

The next step is to perform the network association (see Figure 4-10). The device application calls the ApiMac_mlmeAssociateReq() API to send the association request message to the coordinator. The association process is successful when the device application receives the association confirmation from the TI 15.4-Stack layer.

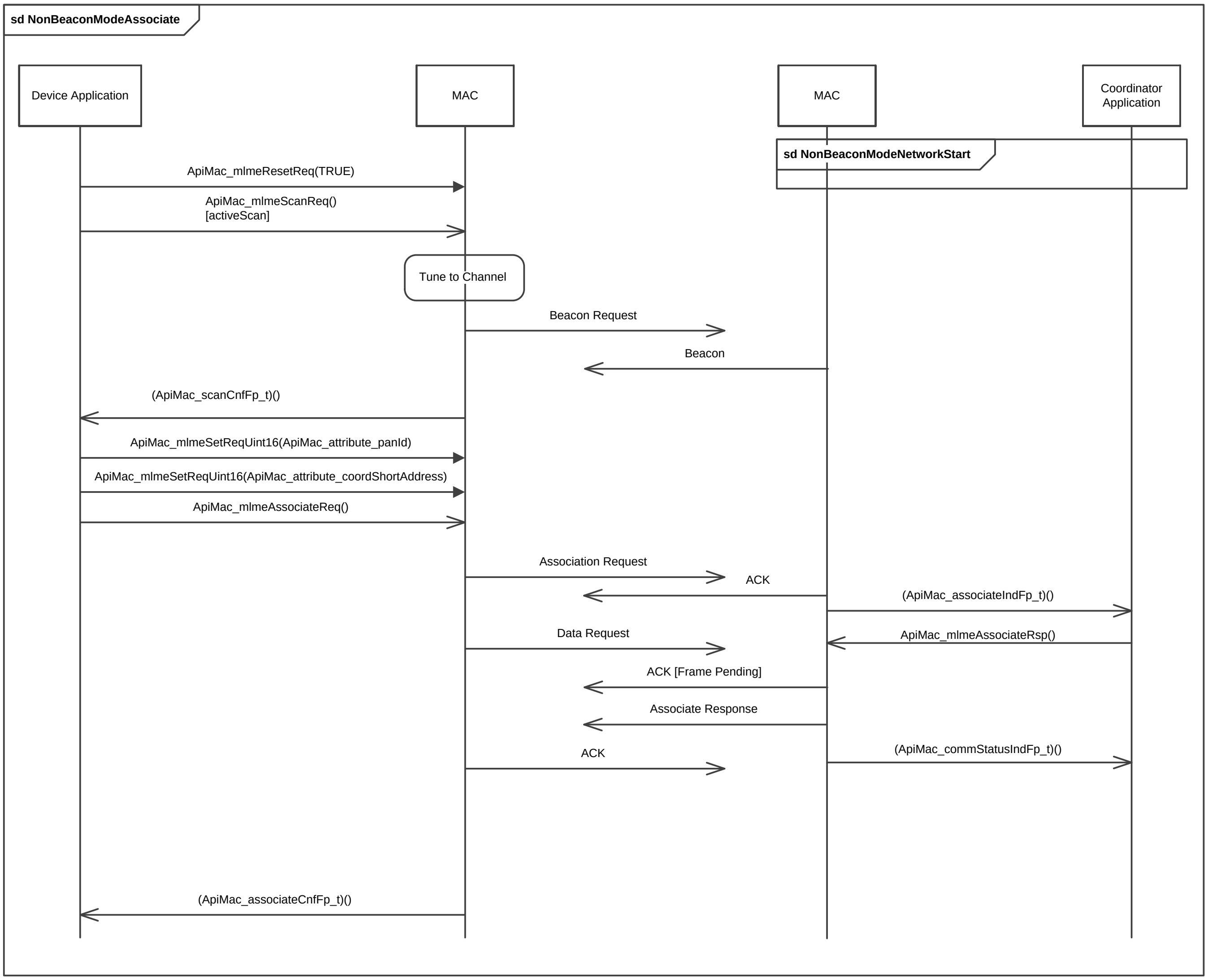


Figure 4-10. Nonbeacon Mode Device Association Sequence

4.2.2.3 Data Exchange

The sequence diagram in Figure 4-11 depicts the various direct data transactions between a device and a coordinator in a nonbeacon-enabled network.

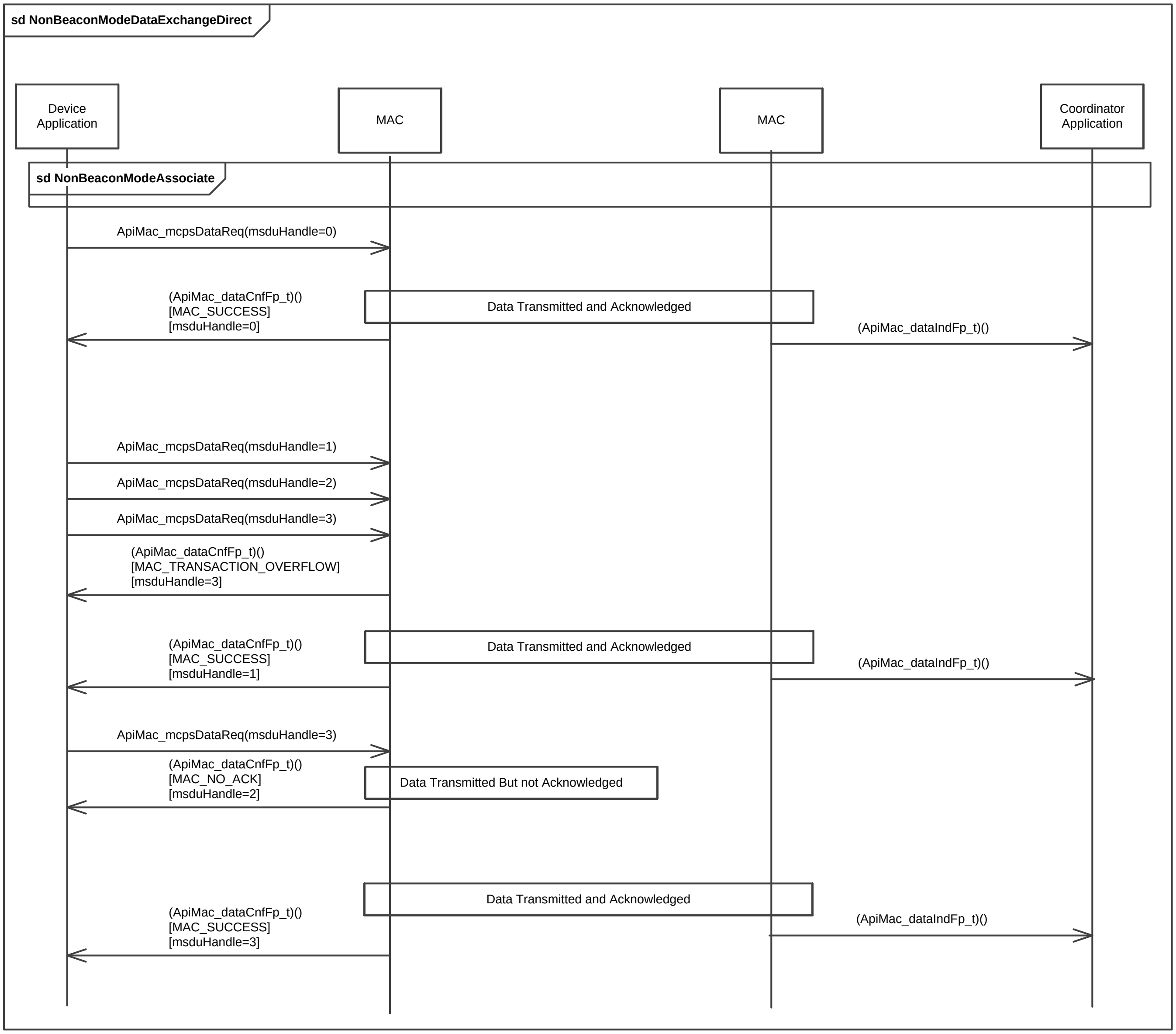


Figure 4-11. Nonbeacon Mode Direct Data Exchange Sequence

The sequence diagram in Figure 4-12 depicts the indirect data transaction in a nonbeacon-enabled network.

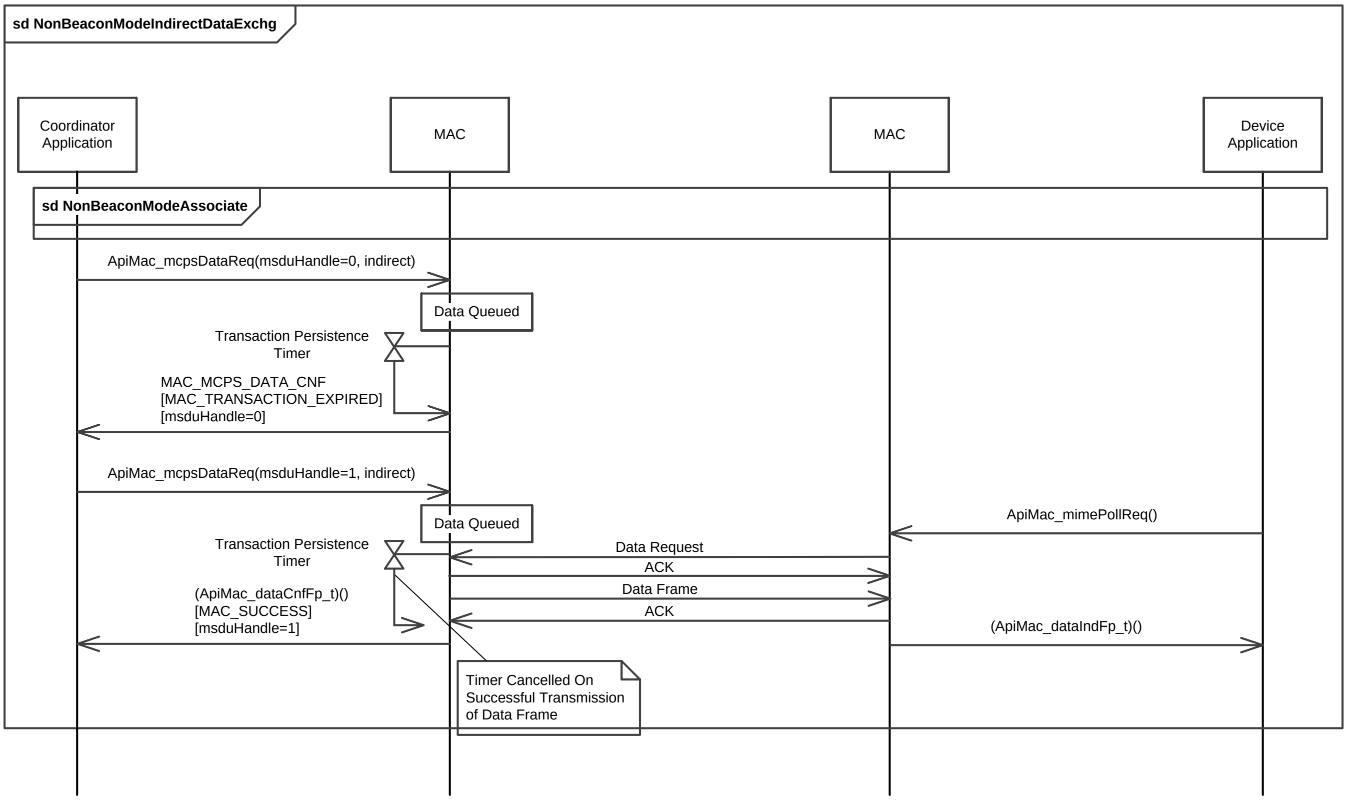


Figure 4-12. Nonbeacon Mode Indirect Data Exchange Sequence

4.2.2.4 Maintaining a Connection for End Nodes

If the device application receives repeated communication failures following requests to transmit data, the device application may conclude that it has been orphaned and can initiate an orphaned-device realignment procedure. Figure 4-13 shows the nonbeacon mode orphan sequence.

In the orphan realignment procedure, the device application requests the TI 15.4-Stack to perform the orphan scan over a specified set of channels by using the ApiMac_MlmeScanReq() API with the scan-type parameter set to orphan scan. For each channel specified, the TI 15.4-Stack at the device switches to the channel and then sends an orphan notification command. After successfully transmitting the orphan notification command, the MAC layer enables the receiver for at most ApiMac_attribute_responseWaitTime. If the device successfully receives a coordinator realignment command, the device terminates the scan and calls the registered callback of type ApiMac_scanCnfFp_t. At the coordinator side, the reception of the orphan notification command results in the call of the registered callback of type ApiMac_orphanIndFp_t by the TI 15.4-Stack. If the coordinator application finds the record of the device, it sends a coordinator realignment command to the orphaned device by using the ApiMac_MlmeOrphanRsp() call.

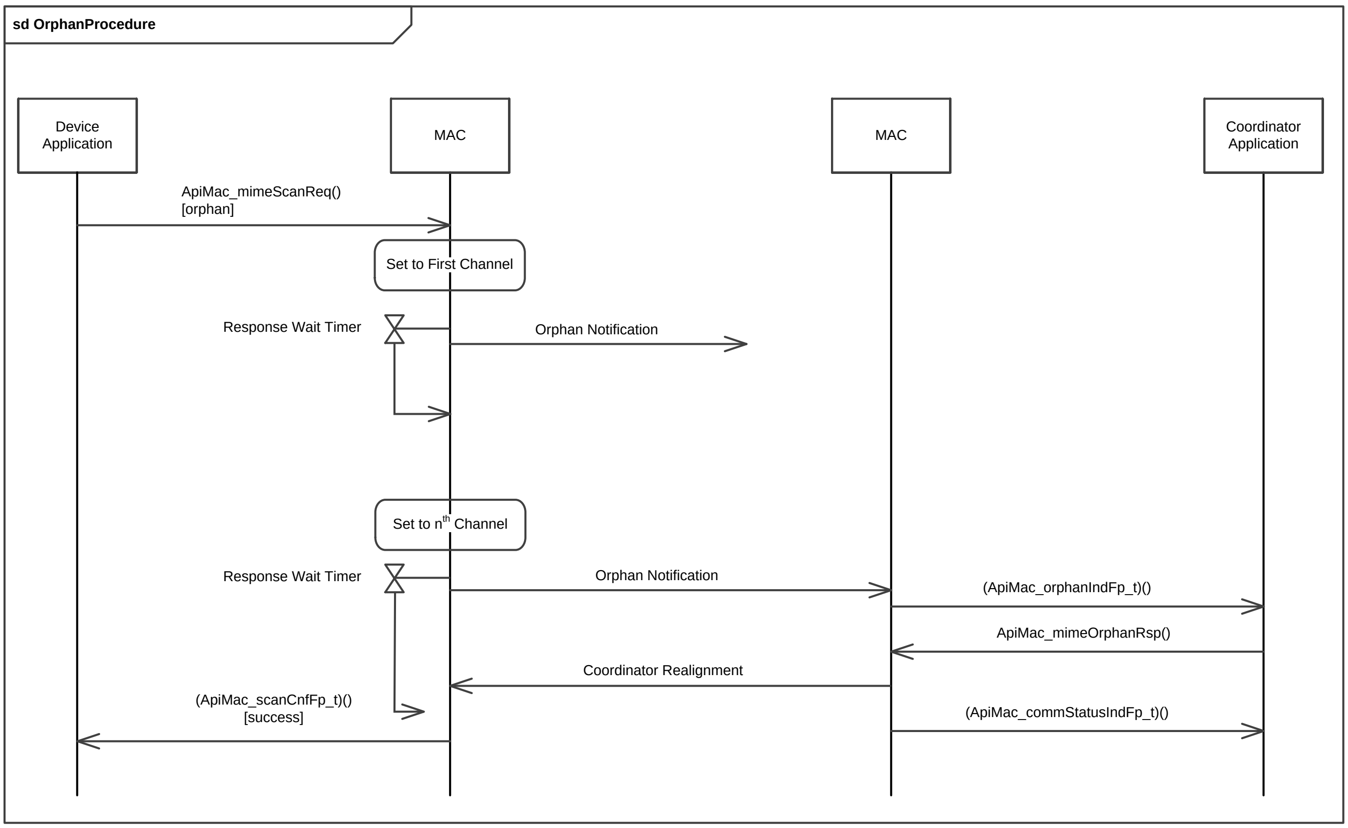


Figure 4-13. Nonbeacon Mode Orphan Sequence

4.2.2.5 Disassociating

Two scenarios are described in the following: the first is initiated by the coordinator and the second is initiated by the device. Figure 4-14 shows the indirect disassociation sequence initiated by the nonbeacon mode coordinator.

When the coordinator application wants one of the associated devices must leave the PAN, the coordinator application requests that the TI 15.4-Stack send the disassociation notification command by using the ApiMac_mlmeDiassociateReq() call. If the txIndirect parameter is set to TRUE, the TI 15.4-Stack sends the disassociation notification command to the device using indirect transmission; then, the disassociation notification command is added to the list of pending transactions stored on the coordinator and pulled by the device using data request command.

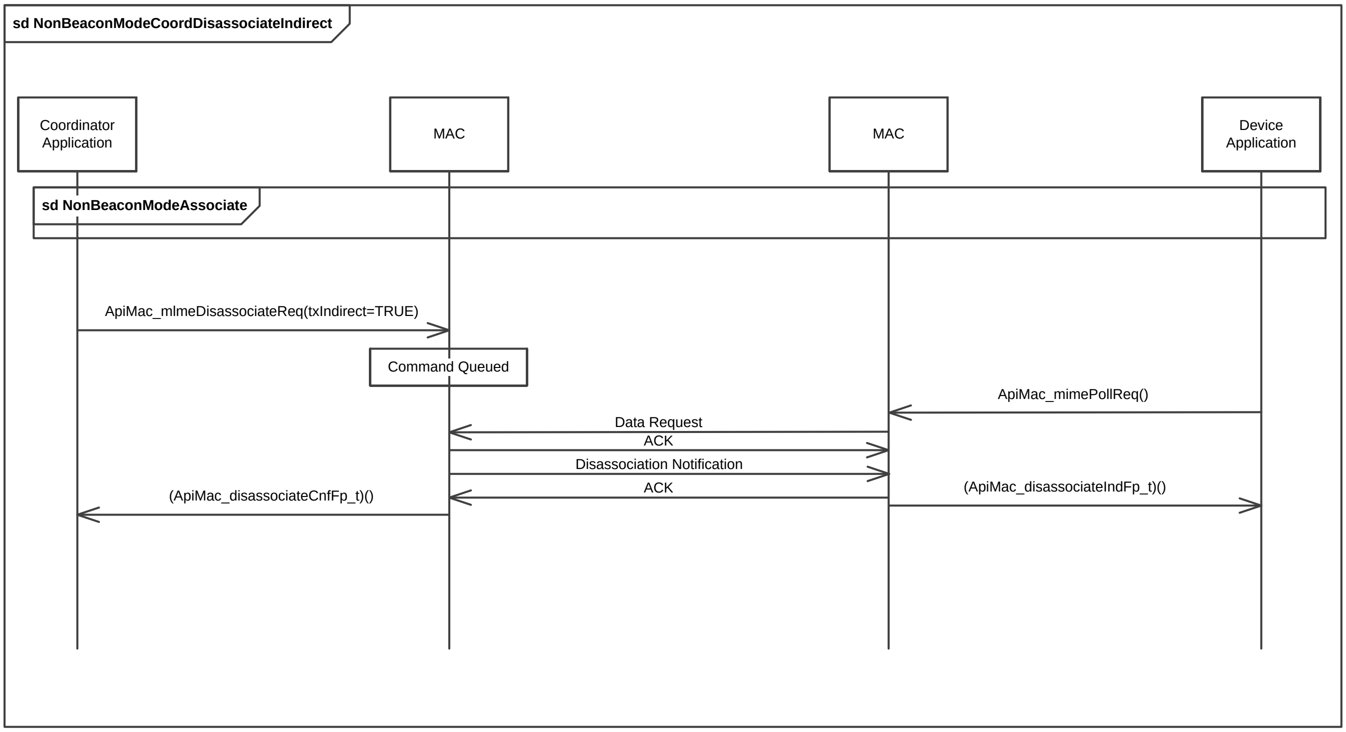


Figure 4-14. Indirect Disassociation Sequence Initiated by the Nonbeacon Mode Coordinator

The end device application can also initiate the disassociation process as described in Figure 4-15, which shows the sequence of messages exchanged when the end device initiates the disassociation process in the non-beacon network.

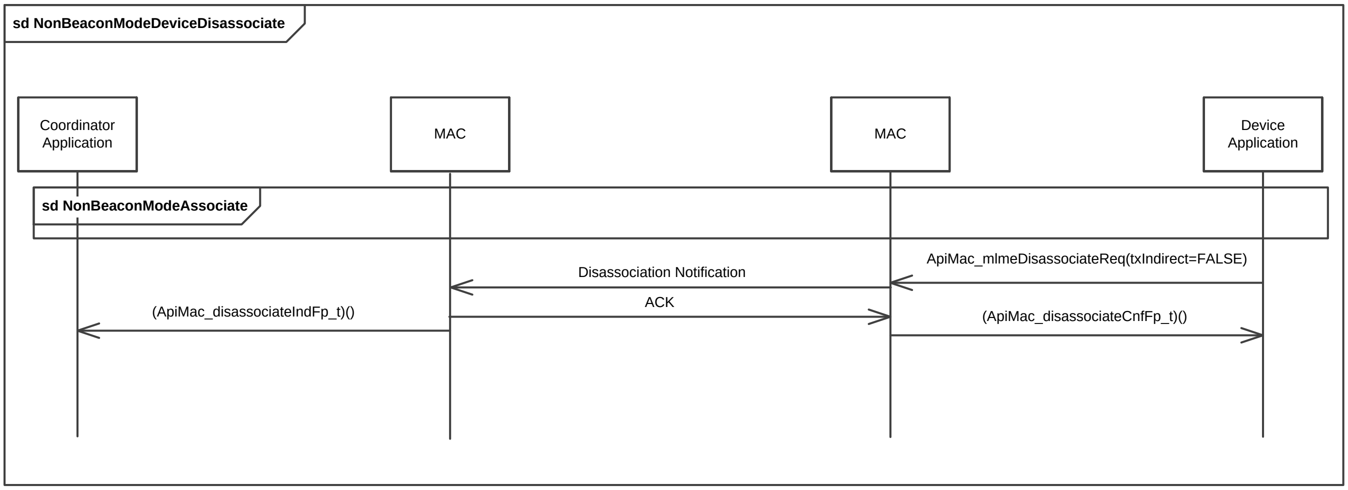


Figure 4-15. Disassociation Sequence Initiated by the Nonbeacon Mode Device

4.2.3 Stack Configuration Knobs

4.2.3.1 Attribute Configuration

The ApiMac_attribute_associatePermit is used by the coordinator application to indicate to the joining devices whether association is allowed or not. When the coordinator sets this attribute item to TRUE, this indicates to the joining devices within the beacon frame that association is allowed. Table 4-3 lists the attribute configurations that apply to beacon mode.

If set to TRUE, the ApiMac_attribute_RxOnWhenIdle enables the receiver during the idle period.

Table 4-3. Attribute Configuration Applicable to Beacon Mode

Name	Type	Range	Number	Description
ApiMac_attribute_associatePermit	Bool	TRUE, FALSE	0x41	TRUE if a coordinator is currently allowing association.
ApiMac_attribute_RxOnWhenIdle	Bool	TRUE, FALSE	0x52	TRUE if the MAC enables its receiver during idle periods.

4.2.3.2 Configuration Constants

The TI 15.4-Stack uses a structure containing various user-configurable parameters (at compile time). This structure, called *macCfg_t*, is in the mac_cfg.c file. Table 4-4 lists the configuration elements.

Table 4-4. Configuration Constants

Name	Description	Range	Default
txDataMax	Maximum number of data frames queued in the transmit data queue.	1 – 255	2
txMax	Maximum number of frames of all types queued in the transmit data queue.	1 – 255	5
rxMax	Maximum number of frames queued in the receive data queue.	1 – 255	2
dataIndOffset	Allocate additional bytes in the data indication for application-defined headers.	0 – 127	0
appPendingQueue	When TRUE, registered callback of type ApiMac_pollIndFp_t will be made to the application when a data request command frame is received from another device.	TRUE – FALSE	FALSE

4.3 Frequency-Hopping Mode

4.3.1 Introduction

Applications that are developed using the TI 15.4-Stack can be configured to operate the network in frequency-hopping configuration where the network devices hop on different frequencies. The TI 15.4-Stack supports an unslotted channel-hopping feature only in the 902-MHz to 928-MHz frequency band based on the directed frame exchange (DFE) mode of the Wi-SUN FAN specification v1.0 (see Chapter 15). This feature can be operated in fixed channel mode at any specified channel, or it can be operated in channel-hopping mode where the channel hopping sequence is based on direct hash channel function (DH1CF) (see [8]). DH1CF generates a pseudo-random sequence of channels on which to hop based on the extended address of the node; thus, the pseudo-random sequence of channels is unique to each node. Each node supports two types of channel-hopping sequences:

- Unicast
- Broadcast

Frequency hopping for each node is based on the unicast channel hopping sequence of those nodes (see Figure 4-16).

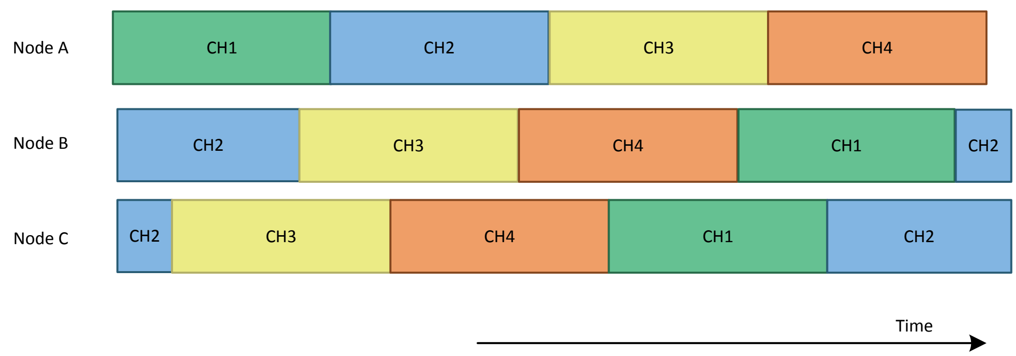


Figure 4-16. Unicast Hopping Sequence

To enable broadcast transmissions, the coordinator starts a broadcast schedule (see Figure 4-17). Every other device follows the broadcast-hopping sequence received from the PAN coordinator. A device performs unicast hopping until the next broadcast dwell time. Then, the device switches to the broadcast-hopping channel for the broadcast dwell time and resumes unicast hopping at the end of the broadcast dwell interval.

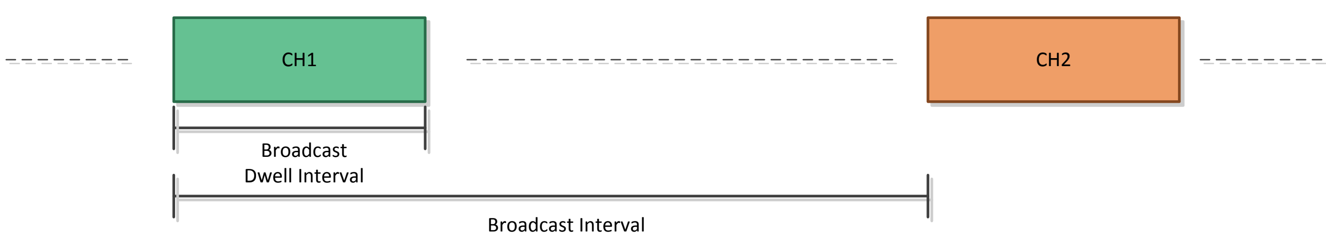


Figure 4-17. Broadcast Channel Hopping Sequence

The application can specify the broadcast dwell interval (default is 250 ms), channel hopping function (default is Fixed Channel), and the list of channels to hop (based on PHY Descriptor ID; for example, the default value of 1 represents 129 channels). Additionally, the broadcast interval (default is 4250 ms) can be specified on the PAN coordinator. When the hopping function is set to *Fixed*, the node must stay on the channel set by ApiMac_FHAttribute_unicastFixedChannel or ApiMac_FHAttribute_broadcastFixedChannel during the unicast and broadcast dwell times, respectively.

NOTE: A set of channels can also be excluded from the hopping channels by using the ApiMac_FHAttribute_unicastExcludedChannels and ApiMac_FHAttribute_broadcastExcludedChannels PIBs as defined in Section 4.3.3.

A special type of transmission called *async transmission* is also supported. In frequency-hopping mode, a device transmits any one of the Async frame types (as defined in the Wi-SUN FAN specification) in all of the requested channels (see [8]). This enables a hopping device to receive such a frame irrespective of the hopping sequence. Thus, the async transmission can be used to exchange channel-hopping information. This feature is especially useful in initial network formation as explained in Section 4.3.2.3.

When channel-hopping information of a neighbor is received, the TI 15.4-Stack tracks the hopping sequences of the neighbor hopping devices and enables successful unicast and broadcast transmissions; thus, hiding the complexities of maintaining synchronization from the application and easing the task of developing applications with frequency hopping feature. A key difference from operating in nonfrequency hopping mode is that the devices use only the extended address (that is, **not** the short address) over the air. Optionally, the short address can be assigned during the association phase but are not used for data exchange.

4.3.2 Network Operations

In frequency hopping mode, nodes operate as one of the following device types:

- PAN coordinator
- Nonsleepy device
- Sleepy device

A typical star topology has a single PAN coordinator connected to a set of nonsleepy or sleepy devices. Each network is identified by a specific network name, which is an ASCII value of 32 bytes and a 16-bit PAN Identifier. The network name (NetName) is a unique network identifier that is configured by the application using frequency hopping and PAN information base (FH-PIB) attributes. Maintenance of the NetName is beyond the scope of the MAC stack and is not used by the stack to filter frames.

Section 4.3.2.1 through Section 4.3.2.7 explain various network operations important to understand when developing a frequency hopping-enabled network over the TI 15.4-Stack.

4.3.2.1 Network Start-Up

Figure 4-18 shows how a frequency-hopping network is started by starting the PAN coordinator in frequency-hopping mode.

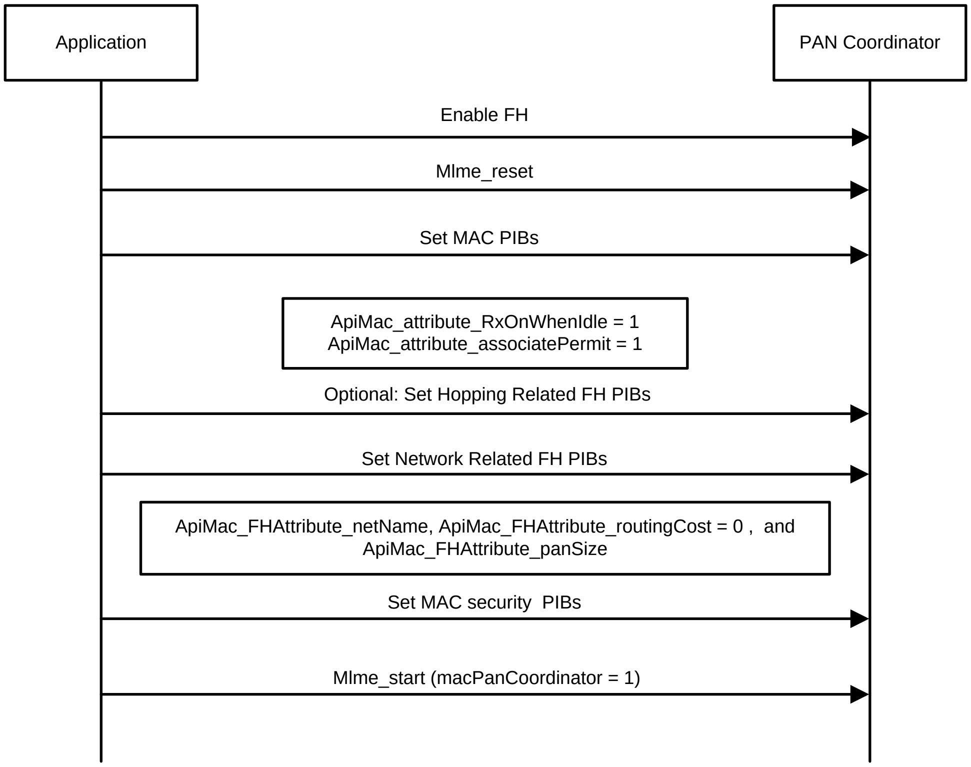


Figure 4-18. Start-Up Sequence of PAN Coordinator

The PIB attributes that are related to frequency-hopping configuration are explained in Section 4.3.3. The NetName is a 32-bit ASCII value to be set by the application. The routing cost must be set to zero. Initially, the PAN size must be set to zero; later, the PAN size must be updated based upon the number of nodes joined.

NOTE: This NetName value is not used by the TI 15.4-Stack to make any decision; instead, the value is carried in a PAN Advertisement frame that can be parsed by a receiving application. Similarly, the GTK HASH 0, GTK HASH 1, GTK HASH2, and GTK HASH 3 can be used to determine the validity of GMK keys that are in use and are beyond the scope of the TI MAC protocol stack.

Finally, start MAC using the macStart API, which specifies that the node is a coordinator.

4.3.2.2 Device Start-Up

Figure 4-19 shows the start-up sequence of the device.

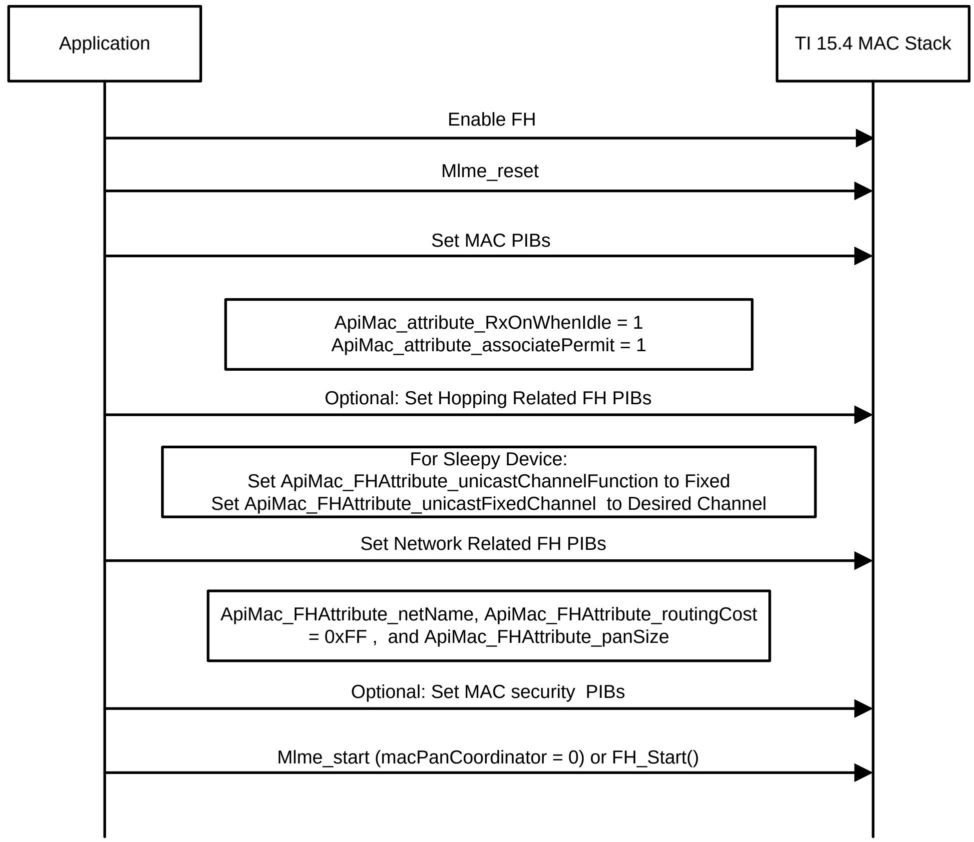


Figure 4-19. Start-Up Sequence of the Device

Wi-SUN FAN v1.0 does not specify sleep mode operation, but the TI 15.4-Stack implements a proprietary extension over the behavior defined in the Wi-SUN FAN and IEEE 802.15.4 MAC protocols (see Section 4.3.2.5) to enable sleepy devices in the frequency-hopping networks based on the TI 15.4-Stack. The channel-hopping function for a sleepy device must be set to *Fixed*, and the fixed channel can be set to any desired channel. The security keys can be set at start-up (if the security keys are already preconfigured for the network), or the security keys can be set after obtaining the same through a key exchange. The key exchange protocol must be handled above the MAC layer.

The routing cost must be set to a high value (0xFFFF) to indicate that the device has not joined the network; later, it can be updated by the application based on the routing metric used.

NOTE: The sequence to start the sleepy and nonsleepy devices is the same until they join a network. A sleepy device is configured to be sleepy by setting the MAC PIB (macRxOnWhenIdle) to zero only after it joins the network (see Section 4.3.2.3). In other words, the sleep mode operation uses low-power mode only for data exchange after successfully joining the network.

4.3.2.3 Network Join

To join to a network, a node must go through the two phases described as follows.

4.3.2.3.1 Phase 1: Exchange of Channel-Hopping Sequence Information Through Asynchronous Messages

Asynchronous messages are sent back-to-back over a specified channel list. This action enables a receiver to receive such frames with high probability, irrespective of the hopping sequence. Four different asynchronous messages are supported by the TI 15.4-Stack as defined in the Wi-SUN FAN specification. All asynchronous frames are transmitted based on a trickle timer [RFC 6206]. The I_{min}, I_{max}, and K for the trickle algorithm are recommended to be set at 1 min, 16 min, and 1, respectively.

Brief descriptions of the four types of asynchronous messages follow:

- PAN Advertisement Solicit (PAS):
  - PAS messages are used by a device to request a coordinator or other joined nodes to transmit a PAN Advertisement frame.
  - Upon reception of the PAN Advertisement frame, a joining application can detect the NetName IE in the frame and then use the name to determine whether or not to reset PA trickle timer.
- PAN Advertisement (PA):
  - PA frames can be transmitted by a coordinator or by a joined node to inform neighbors about the PAN size, Routing cost, and PAN ID.
  - The trickle timer associated with PA transmissions is programmed to be reset on reception of a PAS frame.
  - Upon, reception of the PAS frame, nodes communicate with the transmitter of the PA frames (note the hopping sequence is carried in the PA frame).
  - The device can choose one of the source nodes of the PA frame as relay to perform an *Authentication and Secure Key Exchange* protocol that must be implemented by the application running over the TI 15.4-Stack. Example applications (collector and sensor) included in TI 15.4-Stack **do not** demonstrate this feature.
- PAN Configuration Solicit (PCS):
  - When a device has the group master key (GMK) keys used in the network, the device can request the transmission of a PAN Configuration frame.
  - PCS messages are transmitted by a node to request neighbors or the coordinator transmit a PAN Config frame.
- PAN Configuration (PC):
  - PC messages are transmitted by the coordinator or a joined node based on a trickle timer that must be reset upon reception of a PCS frame.
  - PC frames carry the broadcast-hopping sequence and the hash values of the list of GMK keys that are actively used.
  - Upon reception of a PC frame, a device detects that the channel-hopping exchanges are completed.

When using the frequency-hopping configuration on star-network topology, TI recommends the following:

- The PAN coordinator transmits PA and PC frames based on separate trickle timers. TI recommends that developers refer to the collector example application implementation (located under the examples folder in the TI 15.4-Stack installation directory).
- Devices transmit PAS and PCS frames for the purpose of joining; then, the devices suspend the trickle timer after a successful join. TI recommends that developers refer to the sensor example application (located under the examples folder in the CC13x0 SimpleLink SDK installation directory) as a reference on how to implement this action, or that they use the implementation in the sensor example application *as-is* as a starting point for custom applications.
- Devices must also implement the suppression mechanism to limit the number of PAS and PCS frames transmitted. TI recommends developers refer to the sensor example application (examples folder in the TI 15.4-Stack installation directory) as a reference for implementing this action, or use the implementation in the sensor example application *as-is* as a starting point for custom applications.

4.3.2.3.2 Phase 2: Proprietary Association Procedure to Inform Coordinator of the Network Join (This is an Optional Step)

Because the frequency hopping join procedure (defined by the Wi-SUN specification) is silent, the PAN coordinator cannot detect if the device has successfully joined the network. The TI 15.4-Stack example applications use an additional step for network join. The MAC layer association procedure described in the IEEE 802.15.4 specification is used after the PCS indicates to the PAN coordinator that the device has successfully joined the frequency-hopping network.

In addition to informing the coordinator that the device has successfully joined the network, the optional mechanism allows the PAN coordinator to detect if the joining node is sleepy or always-on through the *capability information* field of the association request message sent by the device to the PAN coordinator. This optional mechanism is required for the PAN coordinator application to determine the following:

- If the application must buffer the message until the device polls for some configured amount of time in case the message is for a sleepy device
- If the application must send the message OTA as soon as the message-transmit request is generated

Although this step is not required for data exchange (because EUI addresses are used instead of short addresses for communication in frequency-hopping mode), TI recommends using this optional procedure in the applications using the frequency-hopping configuration of the TI 15.4-Stack to enable the coordinator application to build the list of joined nodes and to detect if the newly joined device is sleepy or is an always-on device.

NOTE: The association procedure must be started at least 2 seconds after reception of a PAN configuration frame. Upon failure, the association procedure can be independently retried up to the retry limit of the application. For a sleepy device, the ApiMac_attribute_RxOnWhenIdle should be set to zero only after a successful completion of the mac-association procedure.

Figure 4-20 and Figure 4-21 shows the procedure for sleepy and non-sleepy devices, respectively.

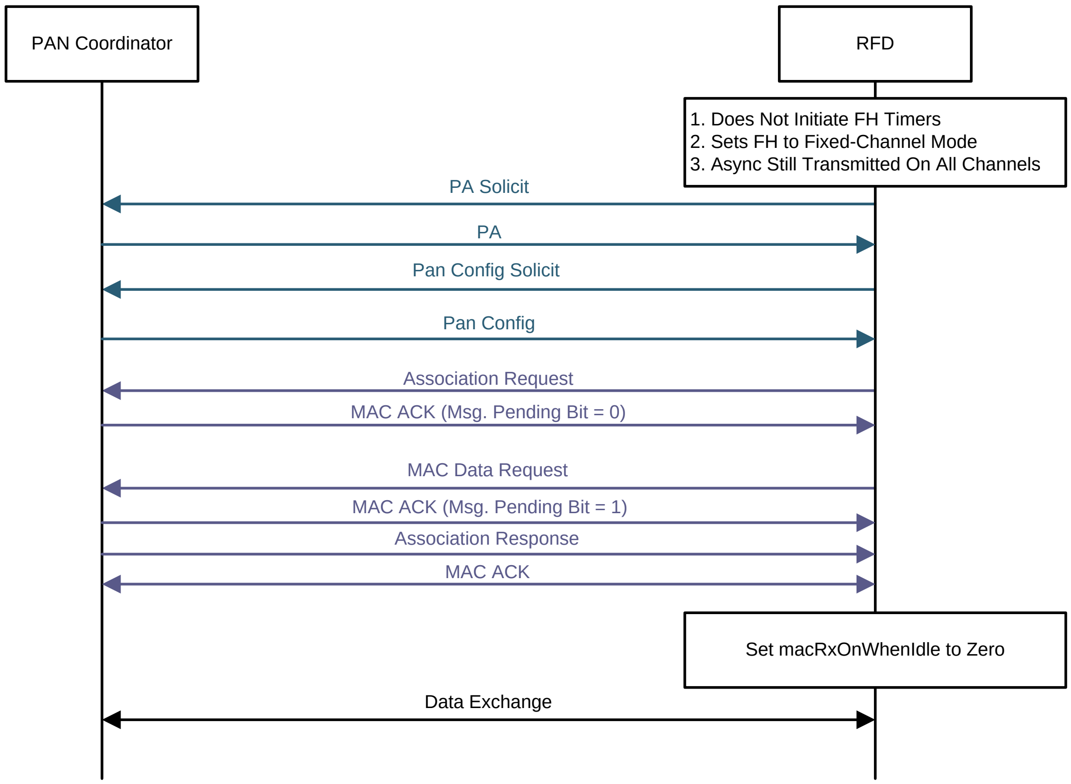


Figure 4-20. Joining Procedure for a Sleepy Frequency-Hopping Device

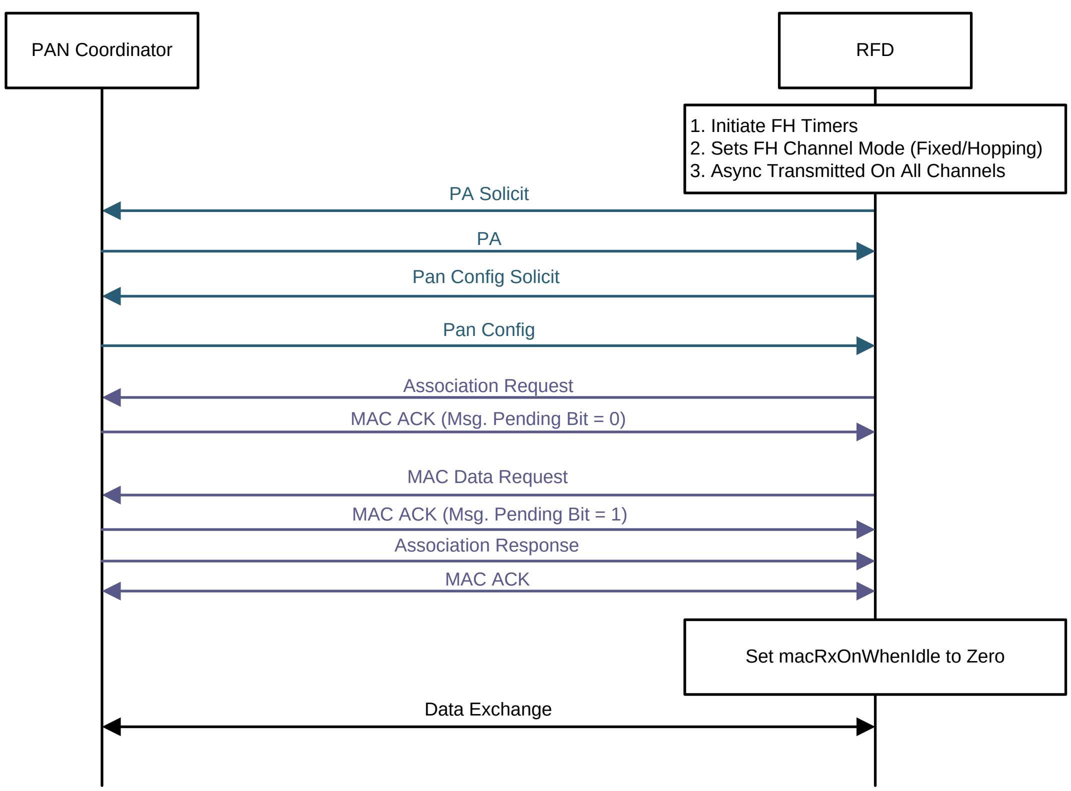


Figure 4-21. Joining Procedure for a Nonsleepy Frequency-Hopping Device

4.3.2.4 Data Exchange

Three types of data-exchange mechanisms are supported by the TI 15.4-Stack:

- Unicast data exchange
- Broadcast data exchange
- Asynchronous frame exchange

All frame exchanges are required to use the address type of 3 (extended addresses). To initiate a unicast or broadcast frame exchange, the API MAC API (ApiMac_mcpsDataReq) should be used. To transmit an asynchronous frame, ApiMac_mlmeWSAsyncReq should be used. The determination of whether the message is unicast or broadcast is done based on the destination address mode used in the data request parameter type ApiMac_mcpsDataReq_t (see Table 4-5).

Table 4-5. Addressing Modes for Unicast and Broadcast Message With TI 15.4-Stack in Frequency-Hopping Configuration

Message Type	Source Address Mode	Destination Address Mode	Destination Address
Unicast	3	3	Specified by the application
Broadcast	3	0	Ignored by stack

Unicast Data Exchange

Unicast data exchange in frequency-hopping mode occurs on the channel of the destination node. A node transmits the frame on the expected receive channel of the destination node. The entire frame exchange occurs on the same channel (see Figure 4-22). Subsequent data exchange occurs on the channel on which the receiver is hopping at the time of transmission; this subsequent data exchange is independent from earlier transmissions.

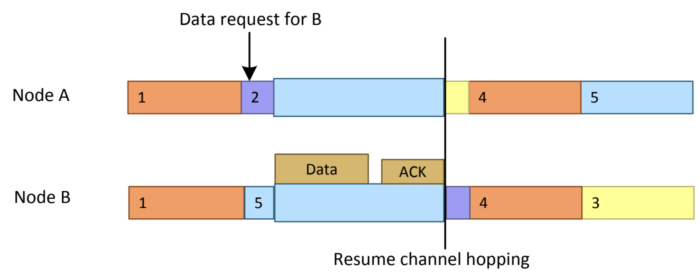


Figure 4-22. Data Exchange With TI 15.4-Stack in Frequency-Hopping Configuration

To transmit a unicast frame to a neighbor, the hopping information was received by the node in some earlier frame. The hopping information could have been received through the reception of any type of asynchronous frames from the destination node. Hence, an application should ensure that such a frame is received from destination node before initiating a data request.

If a data request is issued to a node whose entry is not in the neighbor table, the error code ApiMac_status_fhNotInNeighborTable (0x64) specifies that the node that is not in the neighbor table is returned to the application by the TI 15.4-MAC protocol stack. Also, an expiry is associated with each neighbor table entry. The default time of the expiry is 2 hours for hopping neighbors. The 2 hour default expiry is set because the hopping information stored in the neighbor table may not be useful beyond that time limit for a successful data exchange (due to loss in synchronization caused by inherent clock drifts). Any data exchange helps the node to resynchronize the entry; thus, the entry is considered active for the next 2 hours. If a data request is sent for an expired neighbor, the message is not sent to the destination and a status of ApiMac_status_fhExpiredNode (0x6C) is returned to the application.

NOTE: The lifetime of a hopping neighbor can be changed to any other desired value in the range of 5 minutes to 600 minutes (10 hours) by using the PIB attribute, ApiMac_FHAttribute_neighborValidTime, which is specified in minutes.

Broadcast Frame Exchange

Broadcast frames are transmitted only during the broadcast dwell interval as shown in Figure 4-17. Broadcast frames have a higher priority than unicast frames in such a dwell time and will preempt a unicast frame when present. This priority difference can lead to situations where frames are out of order (that is, the order in which frames are requested to be transmitted by the application could be different from the order in which they are transmitted over the air). Thus, the order in which the ApiMac_mcpsDataReq confirm is received may be different from the order in which ApiMac_mcpsDataReq is sent. The msdu-handle should be used to match the request primitive to the corresponding .confirm primitive by the application.

An application on the PAN coordinator can transmit a broadcast frame any time because the PAN coordinator starts the broadcast hopping as soon as the application is started. All other nodes should wait for the reception of a PAN Configuration frame from the PAN coordinator to start the broadcast-hopping sequence. An application on these other nodes should wait for the reception of the PAN Configuration frames; then the application can set the source address of the PAN Configuration frame (if it selects to use that node as a Parent) to the FH_MAC_TRACK_PARENT PIB before issuing a request of broadcast frame exchange. If an application issues a broadcast data request while the node has not yet started following a broadcast hopping sequence, the stack returns an ApiMac_status_badState (0x19) error code. A sleepy FH device does not track broadcast dwell times and therefore cannot receive broadcast frames. Broadcast frame exchanges are only to be done between nonsleepy devices and coordinators.

Asynchronous Frame Exchange

Asynchronous frames are transmitted by a device on the list of channels specified by user in the Async request. Figure 4-23 shows that the device deviates from the hopping sequence and performs this operation.

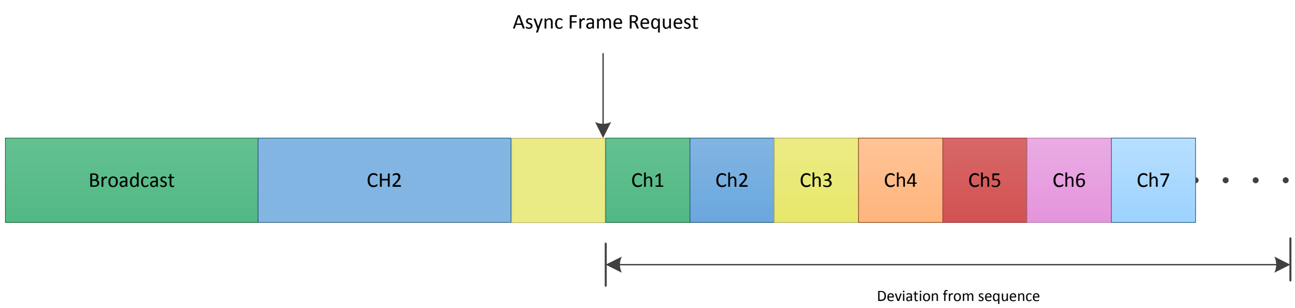


Figure 4-23. Asynchronous Frame Exchange

The objective of asynchronous frame exchange is to transmit data on all available channels (default = 129); thus, asynchronous frame exchange can take a few seconds to complete (worst case is approximately 4 seconds). Such transmissions are typically controlled by trickle timers and are not recommended to be transmitted frequently (refer to [8]). Optionally, a device may issue an Async Request with a Stop command, which will stop an ongoing Async frame exchange.

4.3.2.5 Sleep Mode Operation

Wi-SUN FAN v1.0 does not define a sleep mode operation. However, the TI 15.4-MAC protocol stack supports a proprietary sleep mode operation using a mechanism similar to the TI 15.4-MAC protocol stack nonfrequency-hopping configuration operation, which uses indirect transmissions. TI recommends that the application change the device's fixed channel of operation before initiating a join request. It can help when the current fixed channel of operation is affected by interference.

The sleep mode operation is explained in Figure 4-24. Because the joining procedure is explained in Section 4.2.2.2, the data exchange mechanism is emphasized in this section.

On successful association, the MAC PIB macRxOnWhenIdle should be set to zero, which enables the sleepy device to enter into low-power operation. The sleepy device transmits frames to the PAN coordinator based on the hopping sequence. The MAC stack on the sleepy device operates on a fixed channel and will not hop independently.

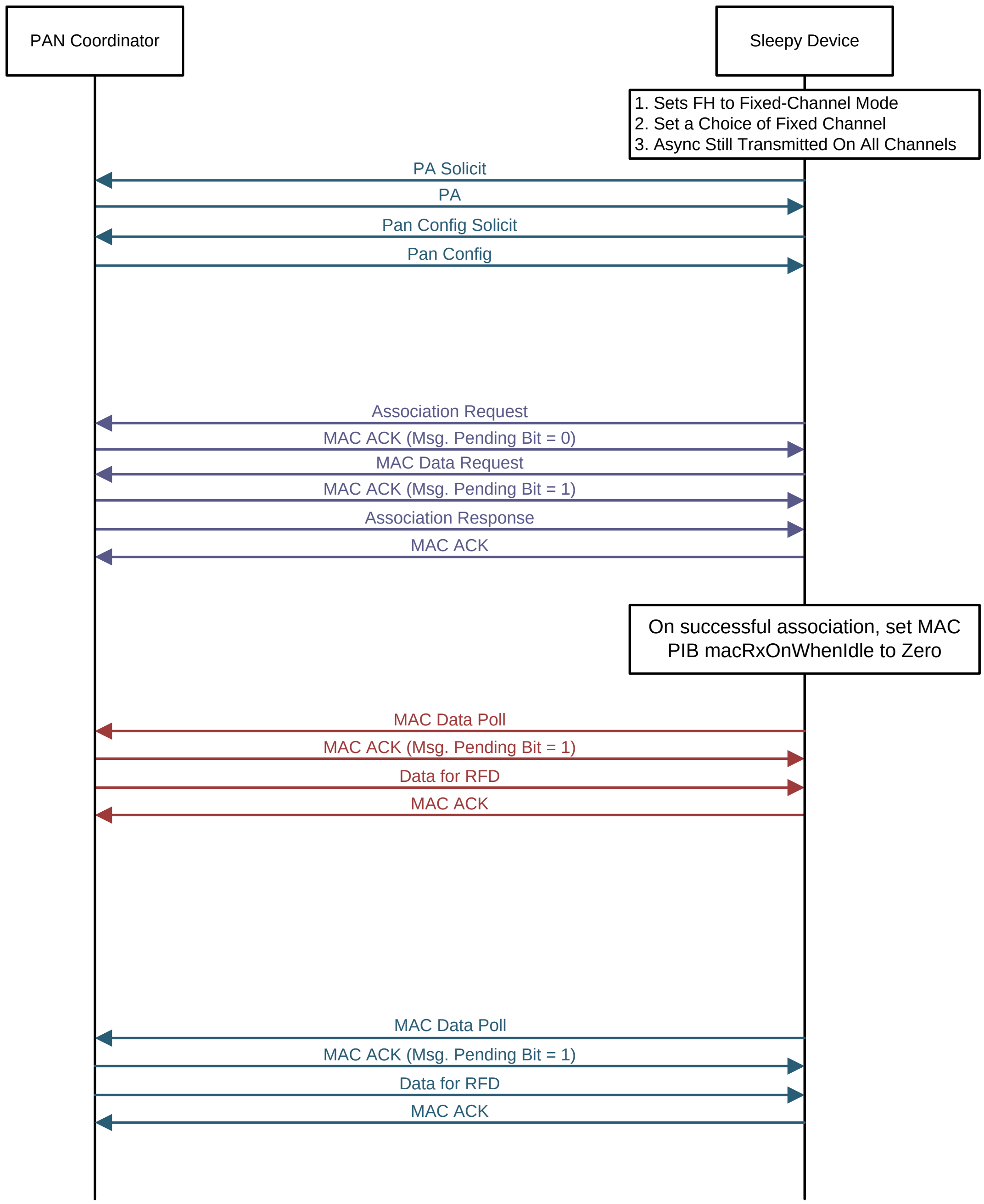


Figure 4-24. Sleep Mode Operation in Frequency-Hopping Mode

4.3.2.6 Maintaining a Connection for End Nodes

In a typical star network, the devices have to keep track of unicast and broadcast timing of Coordinator's hopping sequences while the coordinator has to do for the unicast timing information of all the connected devices.

The timing information of the unicast and broadcast sequence of a device is carried in unicast timing and frame type information element (UTT-IE) and broadcast timing Information element (BT-IE), respectively. All data frames carry UTT-IE and BT-IE. The ACK frames from PAN-Coordinator contain both the UTT-IE and BT-IE, while those from other devices carry UTT-IE alone. The timing information corresponding to the source of the received Data/ACK frame is updated based on received frames.

The lifetime of a neighbor table is based on the last time the entry was updated. As long as a frame is received from a neighbor once every *neighbor valid time*, it is kept active. The neighbor valid time for a hopping neighbor is set to 2 hours by default. Neighbor valid time can be changed using the MAC_FHPIB_NEIGHBOR_VALID_TIME PIB attribute. After that period the entry is considered as expired. Any neighbor table entry is deleted if it is not updated within the last 10 hours.

The period within which at least one data frame should be exchanged to maintain reliable communication depends on the dwell time value used by the PAN coordinator. TI recommends keeping this period for at most 10 minutes or 25 minutes, for a PAN coordinator dwell time of 100 ms and 250 ms respectively.

4.3.2.7 Disassociating

The frequency-hopping mode also supports the disassociation command defined in IEEE 802.15.4, similar to the nonfrequency-hopping mode.

4.3.3 Stack Configuration Knobs

The frequency-hopping mode features can be controlled through a set of MAC FHPIB attributes. Some of these PIBs affect the TI 15.4-Stack operation directly, while others are provided to help applications generate the required Asynchronous frames. This section explains the MAC FHPIB attributes.

4.3.3.1 Parameters Controlling the Unicast Channel-Hopping Sequence of the Node

The parameters controlling the unicast hopping must be set after the FH is enabled and before the MAC or FH-start API is called (see Table 4-6). These values must not be changed after the node starts the hopping sequence. To change these values the nodes have to be power cycled or reset.

Table 4-6. Unicast Channel-Hopping PIB Parameters

PIB	PIB ID	Range	Default	Description
ApiMac_FHAttribute_unicastDwellInterval	0x2004	15 to 250 (ms)	250	Amount of time spent on each channel
ApiMac_FHAttribute_unicastChannelFunction	0x2008	0 or 2	0	Whether to hop or not. Only two values are supported: 0 – Fixed 2 – DH1CF-based hopping
ApiMac_FHAttribute_unicastFixedChannel	0x200C	0 – maximum channel based on PHY configuration	0	The channel to use during unicast hopping when the channel function is fixed
ApiMac_FHAttribute_unicastExcludedChannels	0x2002	17 bytes	All zeros	The list of channels to avoid when channel function is 2. Each bit represents a channel, starting from the LSB of the first byte which, represents Channel 0

4.3.3.2 Parameters Controlling the Broadcast Channel-Hopping Sequence

These parameters must only be set on the PAN coordinator (see Table 4-7). The parameters must be set after the FH is enabled and before the MAC or FH-start API is called. Other devices obtain this information on reception of a PC (an Asynchronous message) message from the PAN- Coordinator. Devices then perform their broadcast hopping based on the received configuration. The received configuration can be read from these PIBs after the reception of a PAN Config frame from the parent of a node.

Table 4-7. Broadcast Channel-Hopping PIB Parameters

PIB	PIB Id	Range	Default	Description
ApiMac_FHAttribute_broadcastInterval	0x2001	15 to 16777215 ms	4250	The interval between two different broadcast dwell interval
ApiMac_FHAttribute_broadcastDwellInterval	0x2005	15 to 250 ms	250	Amount of time spent during broadcast dwell interval
ApiMac_FHAttribute_broadcastChannelFunction	0x2009	0 or 2	0	Whether to hop or not. Only two values are supported: 0 – Fixed 2 – DH1CF based hopping
ApiMac_FHAttribute_broadcastFixedChannel	0x200D	0 – maximum channel based on PHY configuration	0	The channel to use during broadcast dwell interval when the channel function is fixed
ApiMac_FHAttribute_broadcastExcludedChannels	0x2003	17 bytes	All zeros	The list of channels to avoid when the channel function is 2. Each bit represents a channel, starting from the LSB of the first byte, which represents Channel 0.

NOTE: A large value of broadcast interval implies a higher delay in transmitting broadcast frames. An application could decide to increase or decrease this interval based on the perceived requirement for handling broadcast frames.

On the device side, an application must set the source address of the chosen parent to the MAC TRACK PARENT PIB (see Table 4-8). FH stack follows the broadcast hopping sequence of the chosen parent. An application can choose a parent based on the received source address of the PAN configuration frames. However, performance loss may occur due to loss in broadcast synchronization, which is corrected based on the subsequent PAN Configuration frame received from the new parent.

Table 4-8. Frequency-Hopping Parent Address PIB Attribute

PIB	PIB Id	Range	Default	Description
ApiMac_FHAttribute_trackParentEUI	0x2000	Any	0xFFFFFFFF	Source address of the parent.

4.3.3.3 Changing Broadcast Sequence Values in the Middle of Network Operation

A PAN coordinator may choose to modify the broadcast interval during a network operation. To do so the application of the PAN coordinator must set the values as required, and then increment the value of the MAC_FHPIB_BROCAST_SCHED_ID PIB (see Table 4-9).

Table 4-9. Broadcast Interval PIB Attribute

PIB	PIB Id	Range	Default	Description
ApiMac_FHAttribute_broadcastSchedId	0x200B	0 to 65535	0	A value representing a given broadcast configuration. It must be incremented when broadcast configurations are changed.

Transmit PAN Config frames more frequently to enable the dissemination of this information to the network.

NOTE: The performance of the network may be affected during this change in configuration time as it requires some time for the nodes to update their hopping sequences.

4.3.3.4 Parameters to Control Frequency of the Operation of Hopping Mode

The following parameters can be set to control specific functions, as defined in Table 4-10.

Table 4-10. Frequency Hopping Control PIB Attributes

PIB	PIB Id	Range	Default	Description
ApiMac_FHAttribute_clockDrift	0x2006	0 to 255	20	Represents the accuracy of the system clock in ppm. A value of 255 implies that the information is not provided.
ApiMac_FHAttribute_timingAccuracy	0x2007	0 to 255	0	Accuracy of provided timing information in 10s of micro seconds.
ApiMac_FHAttribute_neighborValidTime	0x2019	5 to 600 (minutes)	120	The time in minutes for which a hopping neighbor is considered valid after reception of a Data/ACK from it.

TI recommends not changing the values listed in Table 4-10, and using the default values.

4.3.3.5 Parameters to Control Neighbor Table Size

The amount of heap memory occupied by the FH neighbor table can be controlled through FH PIB attributes. The total number of end devices supported must be less than 50. If a deployment only requires a lesser number of devices, a lower number of neighbor table entries can be specified, thereby allowing more heap for the application. When configuring the number of neighbor table entries, both non-sleepy and sleepy devices must be changed together with MAC_FHPIB_NUM_NON_SLEEPY_DEVICES set first.

Table 4-11. Frequency Hopping Neighbor Control PIB Attributes

PIB	PIB Id	Range	Default	Description
MAC_FHPIB_NUM_NON_SLEEPY_DEVICES	0x201b	0 to 50	2	Total number of non-sleepy neighbors supported.
MAC_FHPIB_NUM_ SLEEPY_DEVICES	0x201c	0 to 50	48	Total number of sleepy neighbors supported.

NOTE: The number of non-sleepy and sleepy neighbors can only be configured before issuing a network start or FHAPI_start API. The total number of end devices supported must be less than 50. When configuring the number of neighbor table entries, both non-sleepy and sleepy devices must be changed together with MAC_FHPIB_NUM_NON_SLEEPY_DEVICES set first.

Table 4-12. Frequency Hopping Backoff PIB Attributes

PIB	PIB Id	Range	Default	Description
MAC_FHPIB_BASE_BACKOFF	0x201a	0 to 16	8	Additional back off parameter on target channel to account for interference (ms).
MAC_FHPIB_NEIGHBOR_VALID_TIME	0x2019	0 to 65535 (minutes)	120	The time in minutes for which a hopping neighbor is considered valid after reception of a Data/ACK from it.

The MAC_FHPIB_BASE_BACKOFF enables FH devices to mitigate interference, which causes a higher delay for packet transmission when interference is observed. The interference mitigation feature can be disabled by setting this parameter to zero (although it is not recommended to do so). TI recommends not changing these values and using the default values.

4.3.3.6 Parameters to Enable Application Generate and Process Asynchronous Frames

The following PIB attributes represent different fields of the Asynchronous frames (see Table 4-13). The attributes are used to generate the required IEs when an async request is made by application based on the async frame type. The TI 15.4-Stack is responsible for only using these PIBs to encode the async frames and does not use these values to make any decisions on its operation. It is up to the application to use these fields if needed to perform any relevant operation.

Table 4-13. PIB Attributes for Asynchronous Messages

PIB	PIB Id	Range	Default	Description
ApiMac_FHAttribute_panSize	0x200E	0 to 65535	0	The size of PAN network
ApiMac_FHAttribute_routingCost	0x200F	0 to 255	0	Zero for PAN Coordinator and Non-Zero for other devices. Actual metric used is beyond the scope of MAC. This can be used to choose a parent.
ApiMac_FHAttribute_routingMethod	0x2010	0 or 1	1	Specify the type of routing protocol used. Typical values are 0 – MHDS, 1 – RPL
ApiMac_FHAttribute_eapolReady	0x2011	0 or 1	1	Specify whether the node can support EAPOL to perform authentication and key exchange
ApiMac_FHAttribute_fanTPSVersion	0x2012	0 to 255	1	Wi-SUN FAN version number
ApiMac_FHAttribute_netName	0x2013	32 bytes	All zeros	Null terminated string
ApiMac_FHAttribute_panVersion	0x2014	0 to 65535	00000	Must be incremented whenever a configuration changes such as broadcast information or GTK Hash values are changed.
ApiMac_FHAttribute_gtk0Hash	0x2015	8 bytes	All zeros	The Hash value that can be used by the application to decide the validity of an exchanged GMK key with ID 0.
ApiMac_FHAttribute_gtk1Hash	0x2016	8 bytes	All zeros	The Hash value that can be used by the application to decide the validity of an exchanged GMK key with ID 1.
ApiMac_FHAttribute_gtk2Hash	0x2017	8 bytes	All zeros	The Hash value that can be used by the application to decide the validity of an exchanged GMK key with ID 2.
ApiMac_FHAttribute_gtk3Hash	0x2018	8 bytes	All zeros	The Hash value that can be used by the application to decide the validity of an exchanged GMK key with ID 3.

4.4 Security

The TI 15.4-Stack supports AES encryption as defined by the IEEE 802.15.4 Specification. The application is responsible for management of the keys. The out-of-box example application of the TI 15.4-Stack demonstrates how to use security with the TI 15.4-Stack.

4.5 Configuring Stack: Selecting the Network Mode of Operation

The TI 15.4-Stack offers three modes of network operations that follow (and as discussed in this chapter):

- Beacon mode
- Nonbeacon mode
- Frequency-hopping mode

The *features.h* file allows developers to compile-in or compile-out different 15.4-Stack features for different applications. The TI 15.4-Stack allows support for all three modes or allows user to select just one desired mode of network operation. The TI 15.4-Stack can be configured in four different modes of operation using the features.h file. Depending on the mode selection, considerable savings in the executable-image space can be achieved. Table 4-14 and Table 4-15 provide a summary of Flash and RAM use of the out-of-box Collector and Sensor Example Application with different compile options enabled.

1. FEATURE_ALL_MODES: When this compile flag is defined, the image is compiled with all the three modes of operation (frequency-hopping mode, beacon-enabled mode and nonbeacon mode) and the configuration file (config.h) can be used to select the specific mode for network operation. This feature allows flexibility to select any mode for the device. For the out-of-box collector and sensor example application, this feature is enabled.

```
/*! If defined, builds the image with all the modes of operation     (frequency hopping,
beacon mode and non beacon mode) */
#define FEATURE_ALL_MODES
```

2. FEATURE_FREQ_HOP_MODE: Defining this compile flag will compile only the frequency hopping mode of operation in the final executable image. For out of box example application, you would need to disable the compile option FEATURE_ALL_MODES and then enable this compile option as in the following:

```
/*! If defined, builds the image with all the modes of operation     (frequency hopping,
beacon mode and non beacon mode) */
#undef FEATURE_ALL_MODES

/*! If defined, builds the image with the frequency mode of operation */
#define FEATURE_FREQ_HOP_MODE
```

3. FEATURE_BEACON_MODE: Defining this compile flag will compile only the beacon mode of operation in the final executable image. For out of box example application, you would need to disable the compile option FEATURE_ALL_MODES and then enable this compile option as in the following:

```
/*! If defined, builds the image with all the modes of operation     (frequency hopping,
beacon mode and non beacon mode) */
#undef FEATURE_ALL_MODES

/*! If defined, builds the image with beacon mode of operation */
#define FEATURE_BEACON_MODE
```

4. FEATURE_NON_BEACON_MODE: Defining this compile flag will compile only the non-beacon mode of operation in the final executable image.

```
/*! If defined, builds the image with all the modes of operation     (frequency hopping,
beacon mode and non beacon mode) */
#undef FEATURE_ALL_MODES

/*! If defined, builds the image with non beacon mode of operation */
#define FEATURE_NON_BEACON_MODE
```

In addition to the compile flags listed previously, the FEATURE_FULL_FUNCTION_DEVICE compile flag is required for the PAN Coordinator device (see the out-of-box Collector Example Application) to perform the role as a central node in the network.

Also, the FEATURE_MAC_SECURITY compile flag is added in the features.h file to allow the ability to turn the MAC layer security on and turn off in the compile executable image. If the mac layer security is turned off, you will need the version of the stack library with no MAC security.

To build the image with no security, perform the steps that follow:

1. Select the linker *File Search Path* option.
2. Modify to include the maclib_nosecure.a instead of maclib_secure.a library file (shown in Figure 4-25).

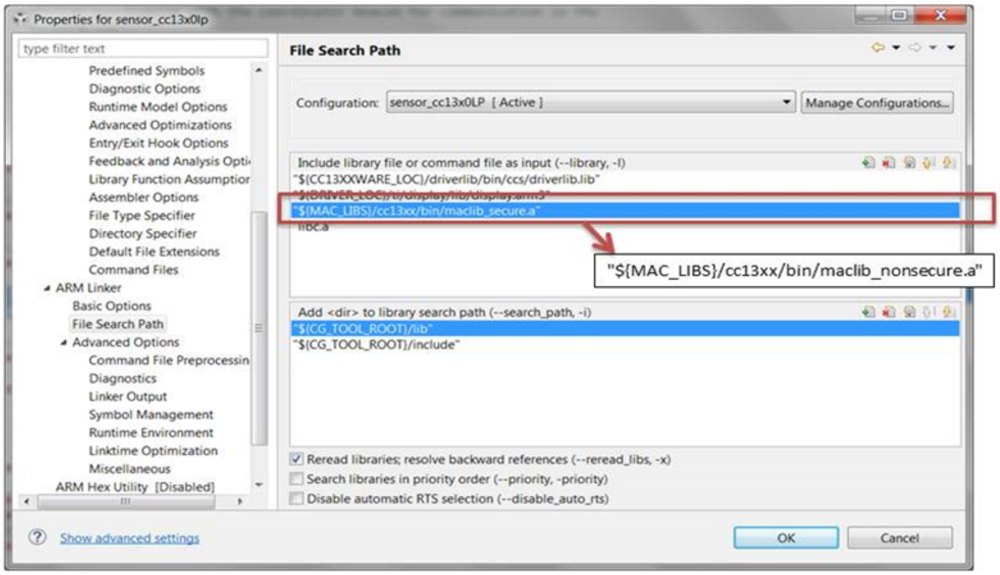


Figure 4-25. Changing TI 15.4 Stack library

Table 4-14. Out-of-Box Collector Example Application Flash and RAM Usage Summary With Various Compile-Option Combinations

Compile Option Enabled	FLASH	RAM
FEATURE_ALL_MODES	108k	14k
FEATURE_FREQ_HOP_MODE	104k	13k
FEATURE_NON_BEACON_MODE	89k	13k
FEATURE_BEACON_MODE	94k	13k

Table 4-15. Out-of-Box Sensor Example Application Flash and RAM Usage Summary With Various Compile-Option Combinations

Compile Option Enabled	FLASH	RAM
FEATURE_ALL_MODES	103k	13k
FEATURE_FREQ_HOP_MODE	100k	13k
FEATURE_NON_BEACON_MODE	86k	12k
FEATURE_BEACON_MODE	88k	12k

Application Overview

The TI 15.4-Stack example applications are designed to enable faster end-product development by providing implementation of various common-protocol stack-specific tasks, and other essential features such as nonvolatile memory storage, saving information over power cycles, in addition to protocol functionality. This chapter explains the example application implementation to help developers quickly modify the TI 15.4-Stack out-of-box example applications for customized development. The following sections detail the example applications of the TI 15.4-Stack projects.

- Pre-RTOS initialization
- Application architecture: the Application task which is the lowest priority task in the system. The code for this task resides in the Application IDE folder.
- Indirect Call Framework: an interface module which abstracts communication between the Stack and other tasks.

Section 5.1 describes overall architecture of the example applications. Section 5.4 provides more specific information about the application task implementation.

5.1 Application Architecture

Figure 5-1 shows the block diagram of the Sensor and Collector example applications on the CC13x0. Refer to the *Linux Developer's Guide* for details on the Linux example applications.

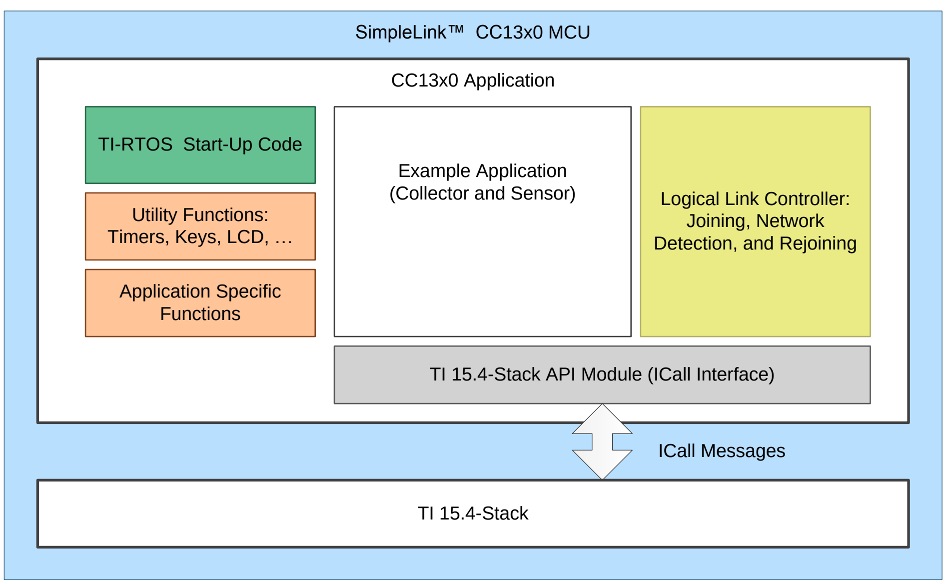


Figure 5-1. Example Application Block Diagram

High-level descriptions of various blocks in Figure 5-1 follow.

Example Application: the platform-independent implementation of the example use case. The TI 15.4-Stack out-of-box demonstrates two use cases – Collector and Sensor. Developers can modify the code in this module in out-of-box example applications for custom application requirements, to quickly develop end products. This is platform-independent code, used as in the Linux example application and also the CC13x0 platform example applications.

Logical Link Controller: implements various essential IEEE 802.15.4 specific or Wi-SUN (for frequency-hopping configuration) specific tasks, such as network formation, network joining, and rejoining. This block intends to offload various protocol-specific implementations from the developers, and enable faster custom application development. This is platform-independent code, as used in the Linux example application and also the CC13x0 platform example applications.

TI-RTOS Start-up Code: initializes the application (see Section 5.2 for more details).

Utility Functions: provides various platform utilities which the application can use for example LCD, timers, keys, and so on.

Application-Specific Functions: implements platform-specific functions such as data storage over power cycles (nonvolatile), and provides user interface functions such as handling button presses or displaying essential information on the LCD, and so on.

TI 15.4-Stack API Module (API MAC Module): this module provides an interface to the management and data services of the 802.15.4 stack through the Indirect Call Framework (ICALL) module. The TI 15.4-Stack API is listed in Chapter 13, and the ICALL module is described in Section 5.3.

5.2 Start-Up in main()

The main() function inside of main.c in the IDE Start-up folder is the application starting point at runtime. This point is where the board is brought up with interrupts disabled and board-related components are initialized. Tasks in this function are configured by initializing the necessary parameters, setting its priority, and initializing the stack size for the application. In the final step, interrupts are enabled and the SYS/BIOS kernel scheduler is started by calling BIOS_start(), which does not return. See CC13xx TRM for information on the start-up sequence before main() is reached.

```
void main()
{
  Task_Params taskParams;

#ifndef USE_DEFAULT_USER_CFG
    user0Cfg.pAssertFP = macHalAssertHandler;
#endif

    /* enable iCache prefetching */
    VIMSConfigure(VIMS_BASE, TRUE, TRUE);

    /* Enable cache */
    VIMSModeSet( VIMS_BASE, VIMS_MODE_ENABLED);

    CPU_WriteBufferDisable();

    /*      Initialization for board related stuff such as LEDs      following TI-
RTOS convention      */
    PIN_init(BoardGpioInitTable);

    /* Configure task. */
    Task_Params_init(&taskParams);
    taskParams.stack = myTaskStack;
    taskParams.stackSize = APP_TASK_STACK_SIZE;
    taskParams.priority = 1;
    Task_construct(&myTask, taskFxn, &taskParams, NULL);

#ifdef DEBUG_SW_TRACE
    IOCPortConfigureSet(IOID_8, IOC_PORT_RFC_TRC, IOC_STD_OUTPUT
                | IOC_CURRENT_4MA | IOC_SLEW_ENABLE);
#endif /* DEBUG_SW_TRACE */

    BIOS_start(); /* enable interrupts and start SYS/BIOS */
    return 0;
}
```

In terms of the IDE workspace, main.c exists in the Application project – meaning that when compiled it is placed in the allocated section of the application's flash.

5.3 Indirect Call Framework

ICALL is a module that provides a mechanism for the Application to interface with the TI 15.4-Stack services (such as TI 15.4-Stack APIs), as well as certain primitive services (such as thread synchronization) provided by the real-time operating system (RTOS). ICALL allows both the Application and protocol stack tasks to efficiently operate, communicate, and share resources in a unified RTOS environment.

The central component of the ICALL architecture is the dispatcher, which facilitates the application program interface between the Application and the TI 15.4-Stack task across the dual-image boundary. Although most of the ICALL interactions are abstracted within the TI 15.4-Stack APIs, it is important for the application developer to understand the underlying architecture so that proper TI 15.4-Stack protocol stack operation is achieved in the multithreaded RTOS environment. The source code of the ICALL module is provided in the ICALL IDE folder in the Application project.

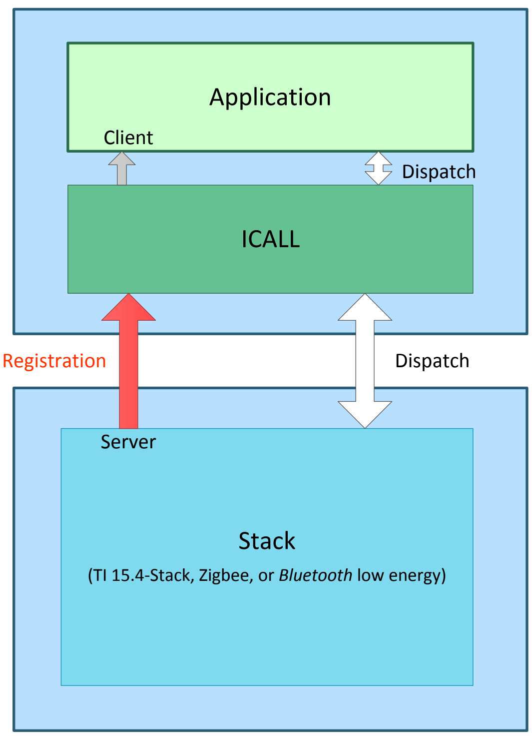


Figure 5-2. ICALL Application – Protocol Stack Abstraction

5.3.1 ICALL TI 15.4-MAC Protocol Stack Service

As depicted in Figure 5-2, the ICALL core use case involves messaging between a server entity (the TI 15.4-Stack task) and a client entity (the Application task). The reasoning for this architecture is twofold: to enable independent updating of the application and TI 15.4-Stack, and also to maintain API consistency as the software is ported from legacy platforms (for example OSAL for the CC253x) to the CC13x0 TI-RTOS. The ICALL TI 15.4-Stack Service serves as the Application interface to all TI 15.4-Stack APIs. Internally, when a TI 15.4-Stack protocol stack API is called by the Application, the ICALL module routes (dispatches) the command to the TI 15.4-Stack, and where appropriate, routes messages from the TI 15.4-Stack to the Application.

Because the ICALL module is part of the Application project, the Application task can access the ICALL with direct function calls. User modifications to the ICALL source are not encouraged. Also, because the TI 15.4-Stack executes at the highest priority, the Application task blocks until the response is received. Certain protocol stack APIs may respond immediately; however, the Application thread blocks because the API is being dispatched to the TI 15.4-Stack through the ICALL. Other TI 15.4-Stack APIs (such as event updates) may also respond asynchronously to the Application through the ICALL, with the response sent to the task event handler of the Application.

5.3.2 ICALL Primitive Service

ICALL includes a primitive service that abstracts various operating system-related functions. Due to shared resources, and to maintain interprocess communication, the Application must use the following ICALL primitive service functions.

- Messaging and Thread Synchronization
- Heap Allocation and Management

5.3.2.1 Messaging and Thread Synchronization

The messaging and thread synchronization functions provided by the ICALL let users design an application to protocol stack interface in the multithreaded RTOS environment. Within the ICALL, messaging between two tasks is achieved by sending a message block from one thread to the other using a message queue. The sender allocates memory, writes the content of the message into the memory block, and then sends (enqueues) the memory block to the recipient. Notification of message delivery is accomplished using a signaling semaphore. The receiver wakes up on the semaphore, copies the message memory block (or blocks), processes the message, and returns (frees) the memory block to the heap.

The Stack uses the ICALL for notifying and sending messages to the Application. These service messages (such as state change notifications) received by the Application task are delivered by the ICALL and processed in the task context of the Application.

5.3.2.2 Heap Allocation and Management

The ICALL provides the Application with global heap APIs for dynamic memory allocation. The size of the ICALL heap is configured with the HEAPMGR_SIZE preprocessor define in the Application project. See Section Section 3.11.2 for more details on dynamic memory management. ICALL uses this heap for all protocol stack messaging as well as to obtain memory for other ICALL services. TI recommends that the Application uses these ICALL APIs for dynamic memory allocation within the Application.

5.3.3 ICALL Initialization and Registration

To instantiate and initialize the ICALL service, the following functions must be called by the application in main() before starting the SYS/BIOS kernel scheduler.

```
/* Initialize ICall module */
    ICall_init();
    /* Start tasks of external images - Priority 5 */
    ICall_createRemoteTasks();
```

Calling ICall_init() initializes the ICALL primitive service (for example, heap manager) and framework. Calling ICall_createRemoteTasks() creates, but does not start, the TI 15.4-Stack protocol stack task.

Before using ICALL protocol services, both the server and client must enroll and register with the ICALL. The server enrolls a service which is enumerated at build time. Service function handler registration uses a globally defined unique identifier for each service. For example, TI 15.4-Stack uses ICALL_SERVICE_CLASS_TIMAC for receiving TI 15.4-Stack protocol stack messages through the ICALL.

The following is a call to enroll the TI 15.4-Stack protocol stack service (server) with the ICALL in MacStack.c

```
// ICall enrollment
/* Enroll the service that this stack represents */
ICall_enrollService(ICALL_SERVICE_CLASS_TIMAC, NULL, &entity, &sem);
```

The registration mechanism is used by the client to send and receive messages through the ICALL dispatcher. For a client (for example, Application task) to use the TI 15.4-Stack APIs, the client must first register its task with the ICALL. This registration is done for the application in ApiMac_init(), which is called by the applications initialization functions. The following is the call to the ICALL in ApiMac_init() in api_mac.c

```
/* Register the current thread as an ICall dispatcher application
 * so that the application can send and receive messages.
 */
ICall_registerApp(&ApiMac_appEntity, &sem);
```

api_mac.c supplies the ApiMac_appEntity and sem inputs which, upon return of ICall_registerApp(), are initialized for the client (for example, Application) task. These objects are subsequently used by the ICALL to facilitate messaging between the Application and server tasks. The sem argument represents the semaphore used for signaling, whereas the ApiMac_appEntity represents the task destination message queue. Each task registering with the ICALL has unique sem and ApiMac_appEntity identifiers.

NOTE: TI 15.4-Stack APIs defined in api_mac.c, and other ICALL primitive services, are not available for use before ICALL registration.

5.3.4 ICALL Thread Synchronization

The ICALL module switches between Application and Stack threads through the use of preemption and semaphore synchronization services provided by the RTOS. The two ICALL functions to retrieve and enqueue messages are not blocking functions. They check whether there is a received message in the queue and if there is no message, the functions return immediately with the ICALL_ERRNO_NOMSG return value. To allow a client or a server thread to block until it receives a message, ICALL provides the following function which blocks until the semaphore associated with the caller RTOS thread is posted.

```
//static inline ICall_Errno ICall_wait(uint_fast32_t milliseconds)
ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);
```

In the preceding function, milliseconds is the timeout period in ms, after which if the function has not already returned, the function returns with ICALL_ERRNO_TIMEOUT. If ICALL_TIMEOUT_FOREVER is passed as ms, the ICall_wait() shall block forever, or until the semaphore is posted. Allowing an application or a server thread to block is important to yield the processor resource to other lower priority threads, or to conserve energy by shutting down power and clock domains whenever possible. The semaphore associated with an RTOS thread is signaled by either of the following conditions.

- A new message is queued to the Application RTOS thread queue.
- ICall_signal() is called for the semaphore.

ICall_signal() is provided so that an application or a server can add its own event to unblock the ICall_wait() and synchronize the thread. ICall_signal() accepts a semaphore handle as its sole argument as follows.

```
//static inline ICall_Errno ICall_signal(ICall_Semaphore msgsem)
ICall_signal(sem);
```

The semaphore handle associated with the thread is obtained through either the ICall_enrollService() call or ICall_registerApp() call.

NOTE: It is not possible to call an ICALL function from a stack callback. This action causes the ICALL to abort (with ICall_abort()) and breaks the system.

5.3.5 Example ICALL Usage

Figure 5-3 shows an example command being sent from the application to the TI 15.4-Stack through the ICALL, with a corresponding return value passed back to the application. ICall_init() initializes the ICALL module instance itself and ICall_createRemoteTasks() creates a task per external image, with an entry function at a known address. After initializing the ICALL, the Application task registers with the ICALL using ICall_registerApp. After the SYS/BIOS scheduler starts and the Application task runs, the application sends a protocol command defined in api_mac.c such as ApiMac_mlmeSetReqArray(). The protocol command is not executed in the application thread. Instead the command is encapsulated in an ICALL message, and routed to the TI 15.4-Stack task through the ICALL. In other words, this command is sent to the ICALL dispatcher where it is dispatched and executed on the server side (TI 15.4-Stack). The Application thread meanwhile blocks (waits for) the corresponding command status message (status). When the TI 15.4-Stack protocol stack finishes executing the command, the command status message response is sent through the ICALL back to the application thread.

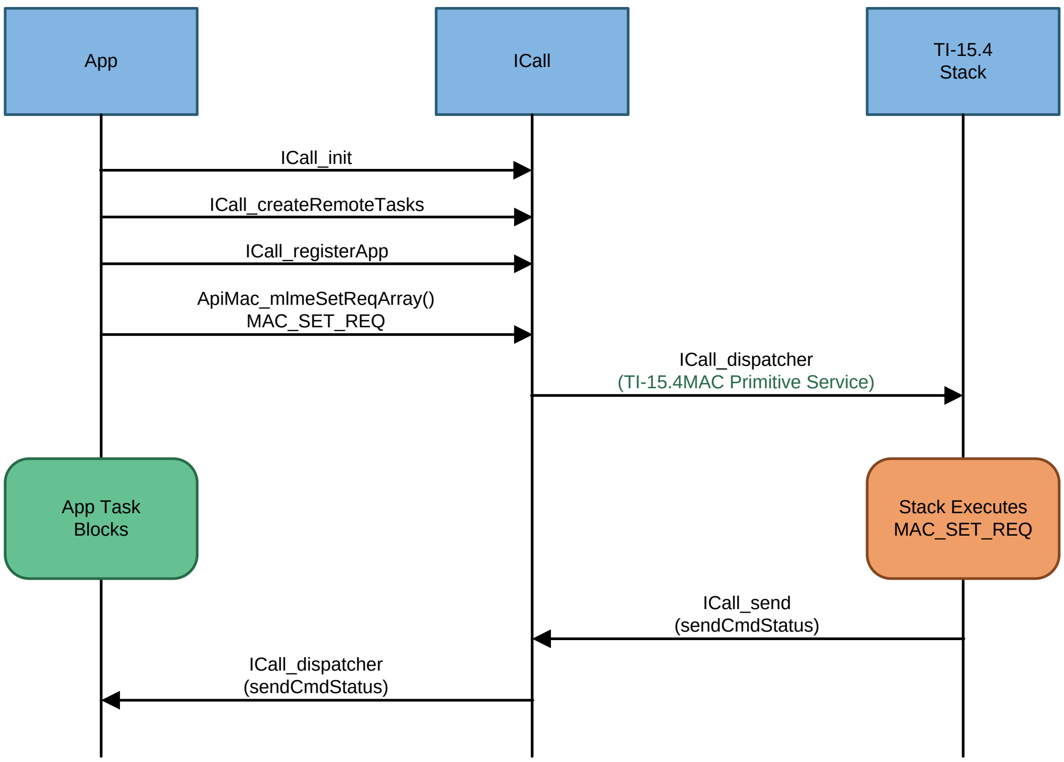


Figure 5-3. ICALL Messaging Example

5.4 General Application Architecture

This section describes how an Application task is structured in more detail.

5.4.1 Application Initialization Function

Section 3.3 describes how a task is constructed. After the task is constructed and the SYS/BIOS kernel scheduler is started, the function that was passed during task construction is run when the task is ready. Power-management functions are initialized here and the ICALL module is initialized through ICall_init(). The primary IEEE address (programmed by TI) is obtained from the CCFG area of the flash memory and NV drivers are initialized. The application task (Sensor application in Figure 5-4) is initialized and started.

```
Void taskFxn(UArg a0, UArg a1)
{
    /* Disallow shutting down JTAG, VIMS, SYSBUS during idle state      * since TIMAC requires
SYSBUS during idle. */
    Power_setConstraint(PowerCC26XX_IDLE_PD_DISALLOW);

    /* Initialize ICall module */
    ICall_init();

#ifdef FEATURE_MAC_SECURITY
    /*      * Copy the extended address from the CCFG area      * Assumption: the memory in
CCFG_IEEE_MAC_0 and CCFG_IEEE_MAC_1      * is contiguous and LSB first.      */
    memcpy(ApiMac_extAddr, (uint8_t *)&*)&(__ccfg.CCFG_IEEE_MAC_0),
           (APIMAC_SADDR_EXT_LEN));

    /* Check to see if the CCFG IEEE is valid */
    if(memcmp(ApiMac_extAddr, dummyExtAddr, APIMAC_SADDR_EXT_LEN) == 0)
    {
        /* No, it isn't valid.  Get the Primary IEEE Address */
        memcpy(ApiMac_extAddr, (uint8_t *)(FCFG1_BASE + EXTADDR_OFFSET),
               (APIMAC_SADDR_EXT_LEN));
    }
#endif


#ifdef NV_RESTORE
    /* Setup the NV driver */
    NVOCTP_loadApiPtrs(&Main_user1Cfg.nvFps);

    if(Main_user1Cfg.nvFps.initNV)
    {
        Main_user1Cfg.nvFps.initNV( NULL);
    }
#endif

    /* Start tasks of external images */
    ICall_createRemoteTasks();

    /* Initialize the application */
    Sensor_init();

    /* Kick off application - Forever loop */
    while(1)
    {
        Sensor_process();
    }
}
```

For example, in the sensor example application file main.c function taskfxn(), the initialization function Sensor_init() sets several software configuration settings as well as parameters. Some examples are:

- Initializing structures for sensor data
- Initializing TI 15.4-Stack
- Setting up the security and logical link controller
- Registering MAC callbacks

5.4.2 Event Processing in the Task Function

In the initialization function in the previous code snippet, the task function enters an infinite loop so to continuously process as an independent task and does not run to completion, seen in Figure 5-4.

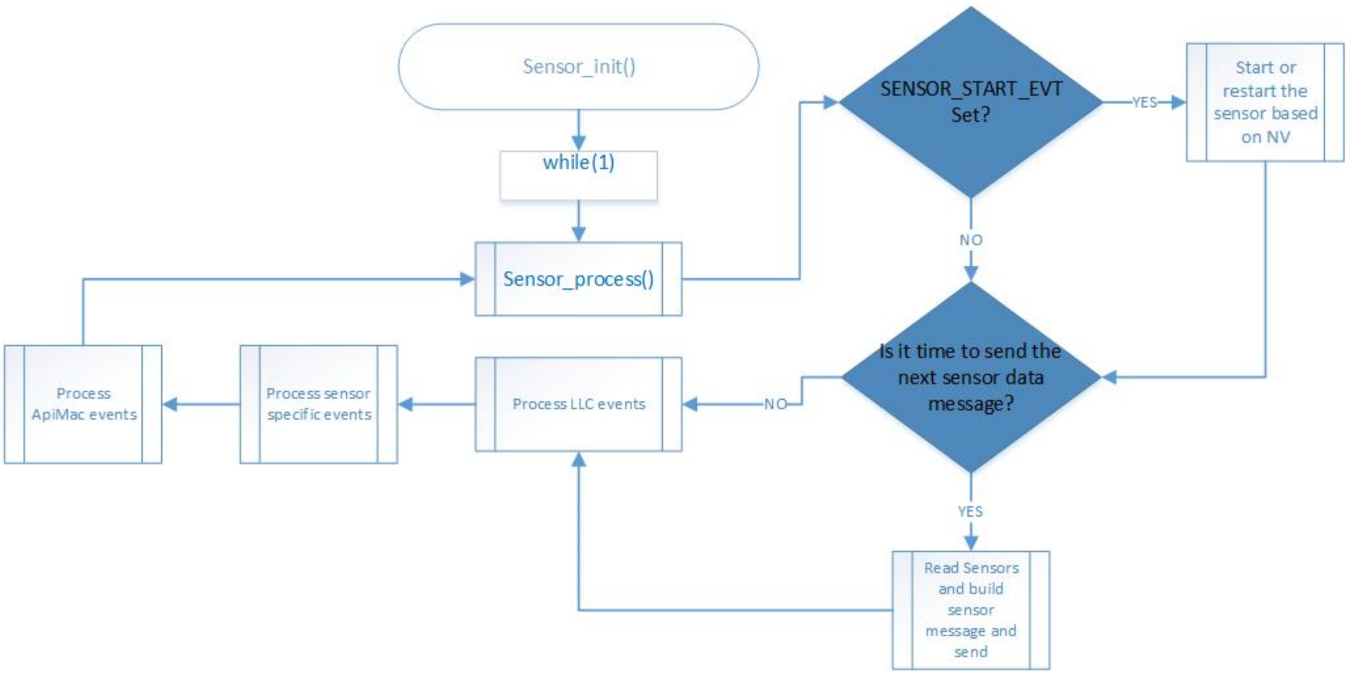


Figure 5-4. Sensor Example Application Task Flow Chart

Figure 5-4 shows various reasons for posting to the semaphore, causing the task to become active.

5.4.2.1 Events Signaled Through the Internal Event Variable

The Application task uses an event variable bit mask to identify what action caused the process to wake up, and takes appropriate action. Each bit of the event variable corresponds to a defined event such as:

```
/*! Event ID - Start the device in the network */
#define SENSOR_START_EVT 0x0001
/*! Event ID - Reading Timeout Event */
#define SENSOR_READING_TIMEOUT_EVT 0x0002
```

Whichever function sets this bit in the event variable must also ensure to post to the semaphore, to wake up the application for processing. An example of this is the clock handler which handles clock timeouts.

```
/* Is it time to send the next sensor data message? */
    if(Sensor_events &SENSOR_READING_TIMEOUT_EVT)
    {
        /* Setup for the next message */
        Ssf_setReadingClock(configSettings.reportingInterval);

        /* Read sensors */
        readSensors();

        /* Process Sensor Reading Message Event */
        processSensorMsgEvt();

        /* Clear the event */
        Sensor_events &= ~SENSOR_READING_TIMEOUT_EVT;
    }
```

When adding an event, it must be unique for the given task and be a power of 2 (so that only 1 bit is set). Because the event variable is initialized as uint16_t, this setup allows for a maximum of 16 internal events.

5.4.3 Callbacks

The application code also likely includes various callbacks from the protocol stack layer and RTOS modules. To ensure thread safety, processing should be minimized in the actual callback, and the bulk of the processing should be done in the application context. The following code snippet directs the callbacks through ApiMac_processIncoming() to the correct MAC API using the ICALL after all the application events are processed.

```
void Sensor_process(void)
{
..
..
    /* Start the collector device in the network */
    if(Sensor_events & SENSOR_START_EVT)
    {
    }
    /* Is it time to send the next sensor data message? */
    if(Sensor_events &SENSOR_READING_TIMEOUT_EVT)
    {
    }

    /* Process LLC Events */
    Jdllc_process();

    /* Allow the Specific functions to process */
    Ssf_processEvents();
    /*      Don't process ApiMac messages until all of the sensor events      are processed.
*/
    if(Sensor_events == 0)
    {
        /* Wait for response message or events */
        ApiMac_processIncoming();
    }
}
```

The previous code snippet directs the callbacks to the correct MAC API using ICALL. Two functions are defined per callback, one at the application level, the other in the MAC API. For example, consider the handling of a scan confirm in the following code snippet.

```
case MAC_MLME_SCAN_CNF:
     if(pMacCallbacks->pScanCnfCb)
     {
        processScanCnf(&(pMsg->scanCnf));
     }
     else
     {
        /*         If there's no callback, make sure the scanResults         are freed         */
        if(scanResults != NULL)
        {
           ICall_free(scanResults);
           scanResults = NULL;
        }
      }
break;
```

The MAC API callback is overwritten by the following application callback.

```
pMacCbs->pScanCnfCb = scanCnfCb;
```

At the application level:

```
/*!
 * @brief       Process  Scan Confirm  callback.
 *
 * @param       pData - pointer to Scan Confirm
 */
static void scanCnfCb(ApiMac_mlmeScanCnf_t *pData)
{
    if(pData->status == ApiMac_status_success)
    {
        if(pData->scanType == ApiMac_scantype_active)
        {
            /* set event to send Association Request */
            Jdllc_events |= JDLLC_ASSOCIATE_REQ_EVT;
        }
        else if(pData->scanType == ApiMac_scantype_passive)
        {
            /* send sync request for beacon enabled device */
            switchState(Jdllc_deviceStates_syncReq);

        }
        else if(pData->scanType == ApiMac_scantype_orphan)
        {
            /* coordinator realignment received, set event to process it */
            Jdllc_events |= JDLLC_COORD_REALIGN;
        }
    }
     …..

    if(macCallbacksCopy.pScanCnfCb != NULL)
    {
        macCallbacksCopy.pScanCnfCb(pData);
    }
}
```

The following code is at the MAC API level.

```
/*!
 * @brief       Process the incoming Scan Confirm callback.
 *
 * @param       pCnf - pointer MAC Scan Confirm info
 */
static void processScanCnf(macMlmeScanCnf_t *pCnf)
{
    /* Confirmation structure */
    ApiMac_mlmeScanCnf_t cnf;

    /* Initialize the structure */
    memset(&cnf, 0, sizeof(ApiMac_mlmeScanCnf_t));

    /* copy the message to the confirmation structure */
    cnf.status = (ApiMac_status_t)pCnf->hdr.status;

    cnf.scanType = (ApiMac_scantype_t)pCnf->scanType;
    cnf.channelPage = pCnf->channelPage;
    cnf.phyId = pCnf->phyID;
    memcpy(cnf.unscannedChannels, pCnf->unscannedChannels,
    APIMAC_154G_CHANNEL_BITMAP_SIZ);
    cnf.resultListSize = pCnf->resultListSize;

    if(cnf.resultListSize)
    {
        if(cnf.scanType == ApiMac_scantype_energyDetect)
        {
            cnf.result.pEnergyDetect = (uint8_t *)ICall_malloc(
                                cnf.resultListSize * sizeof(uint8_t));
            if(cnf.result.pEnergyDetect)
```

```
            {
                memcpy(cnf.result.pEnergyDetect, pCnf->result.pEnergyDetect,
                       cnf.resultListSize);
            }
            else
            {
                cnf.status = ApiMac_status_noResources;
                cnf.resultListSize = 0;
            }
        }
        else
        {
            cnf.result.pPanDescriptor = (ApiMac_panDesc_t *)ICall_malloc(
                            cnf.resultListSize * sizeof(ApiMac_panDesc_t));
            if(cnf.result.pPanDescriptor)
            {
                uint8_t x;
                ApiMac_panDesc_t *pDstPanDesc = cnf.result.pPanDescriptor;
                macPanDesc_t *pSrcPanDesc = pCnf->result.pPanDescriptor;

                for(x = 0; x < cnf.resultListSize;
                                x++, pDstPanDesc++, pSrcPanDesc++)
                {
                    copyMacPanDescToApiMacPanDesc(pDstPanDesc, pSrcPanDesc);
                }
            }
            else
            {
                cnf.status = ApiMac_status_noResources;
                cnf.resultListSize = 0;
            }
        }
    }

    /* We processed the scan confirm, so free the results */
    if(scanResults != NULL)
    {
        ICall_free(scanResults);
        scanResults = NULL;
    }

    /*
     * Initiate the callback, no need to check pMacCallbacks or the function
     * pointer for non-null, the calling function will check the function
     * pointer
     */
    pMacCallbacks->pScanCnfCb(&cnf);

    if(cnf.resultListSize)
    {
        if(cnf.scanType == ApiMac_scantype_energyDetect)
        {
            ICall_free(cnf.result.pEnergyDetect);
        }
        else
        {
            ICall_free(cnf.result.pPanDescriptor);
        }
    }
}
```

Example Applications

This section provides an overview of the TI 15.4-Stack out-of-box example applications and instructions on how to run them.

The TI 15.4-Stack-based star network consists of two types of logical devices: the PAN-Coordinator, and network devices (sleepy or nonsleepy). This separation of the device types derives from the IEEE 802.15.4 specification. The TI 15.4-Stack can be configured in either of the two roles by the application. The PAN-Coordinator is the device that starts the network, is the central node in the star network, and allows other devices to join the network. The network devices join the network and always communicate with the PAN-Coordinator.

The collector example application demonstrates how to implement a PAN-Coordinator device, while the sensor example application demonstrates how to implement the network devices.

The example applications provided in the TI 15.4-Stack are developed for the CC13x0 platform. In addition, the Linux example applications for the external host (AM335x) + MAC coprocessor configuration is included in the TI 15.4-MAC Linux SDK installer. All sample applications described in this section are intended to run on the CC13x0 LaunchPad platform. The Linux example application is described in the *TI 15.4-MAC Linux Developers' Guide* included in the TI 15.4-MAC Linux SDK installer.

The following hardware is required to run the TI 15.4-Stack OOB example applications:

- Two CC13x0 LaunchPad development kits (http://www.ti.com/tool/launchxl-cc1310)
- Optional – Two LCD BoosterPack™ modules (http://www.ti.com/tool/430boost-sharp96)

The OOB example applications are configured to operate in the nonbeacon configuration with security enabled. See Section 6.4 to understand the various parameters that application developers can configure to use the various configuration settings of the example applications.

NOTE: In the following sections, the project names for CC1310 and CC1350 platforms are referred to as CC13x0. Replace *x* with either *1* or *5* depending on the wireless MCU being used.

Topic **Page**

6.1 Collector Example Application

This example project implements a collector device: the PAN-Coordinator for the network. This device creates the TI 15.4-Stack network, allows sensor devices to join the network, collects sensor information sent by devices running the sensor example application, and tracks if the devices are on the network or not by periodically sending tracking request messages.

6.1.1 Running the Application

Perform the following steps to run the OOB collector example application.

1. Import the collector_cc13x0lp project as described in Section 2.5.2.3.
2. After importing, configure the following settings in the config.h file. To configure the settings on the collector application project: Select the collector_cc13x0lp project in the CCS Project Explorer window. Find the config.h file, as shown in Figure 6-1.

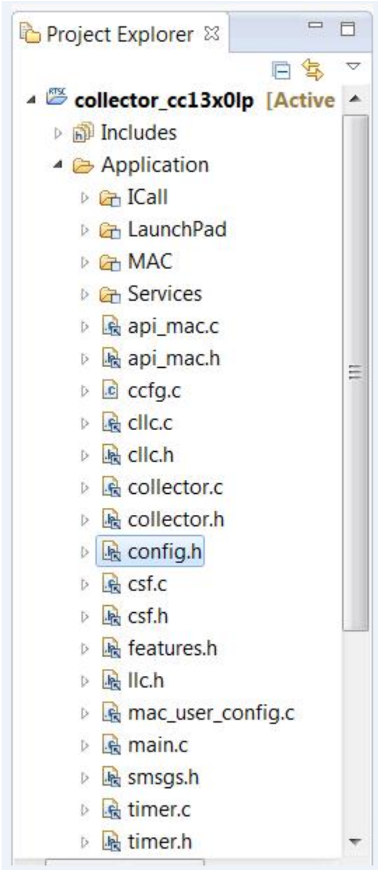


Figure 6-1. Collector Example Application Folder Project Explorer View

(a) Set #define CONFIG_PAN_ID to the desired value.

(b) Set the Phy ID according to region of interest:

- For a US or 915-MHz band of operation, use the OOB CONFIG_PHY_ID settings as:

```
/*! Setting for Phy ID */
#define CONFIG_PHY_ID                (APIMAC_STD_US_915_PHY_1)
```

- For ETSI PHY for Europe (or 868-MHz band operation), configure the CONFIG_PHY_ID parameter:

```
/*! Setting for Phy ID */
#define CONFIG_PHY_ID                (APIMAC_STD_US_915_PHY_3)
```

- Set the preferred channel of operation in the CONFIG_CHANNEL_MASK parameter:

```
/*!
 Channel mask used when CONFIG_FH_ENABLE is false
 Each bit indicates if the corresponding channel is to be
 scanned First byte represents channel 0 to 7 and the last byte represents
 channel 128 to 135
 */
#define CONFIG_CHANNEL_MASK           { 0x0F, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                        0x00, 0x00, 0x00, 0x00, 0x00 }
```

The channel numbers available in each band follow:

- 902 to 928 MHz (50 kbps): 0 to 128, such as when CONFIG_PHY_ID = APIMAC_STD_US_915_PHY_1
- 863 to 870 MHz (50 kbps): 0 to 33, such as when CONFIG_PHY_ID = APIMAC_STD_ETSI_863_PHY_3

In addition to the preceding configuration settings, note that when the devices join the network, the collector application configures the joining devices on how often to report the sensor data. To change or configure the interval at which the joining sensor devices report the sensor data, set the parameter CONFIG_REPORTING_INTERVAL to the desired value in milliseconds in the collector.c file. By default, the sensor reporting interval is set to 30 seconds, as shown by the following code snippet from the collector.c file:

```
/* Default configuration reporting interval, in milliseconds */
#define CONFIG_REPORTING_INTERVAL 30000
```

3. In the CCS Project Explorer, select the collector_cc13x0lp project.
4. Right-click on the collector_cc13x0lp project, and select the Build option. This builds the collector application project.
5. Download the project onto the CC13x0 LaunchPad by selecting Debug from the Run tab (see Figure 6-2).

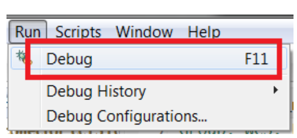


Figure 6-2. Debug Option

6. Terminate the debug session when the download is complete.

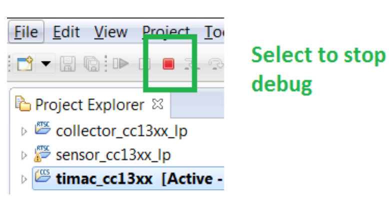


Figure 6-3. Select Terminate Option

7. If the AUTO_START compile flag is enabled, press BTN-1 to start the collector. The red LED on the LaunchPad turns on, and the display on the LCD should appear as in Figure 6-4.

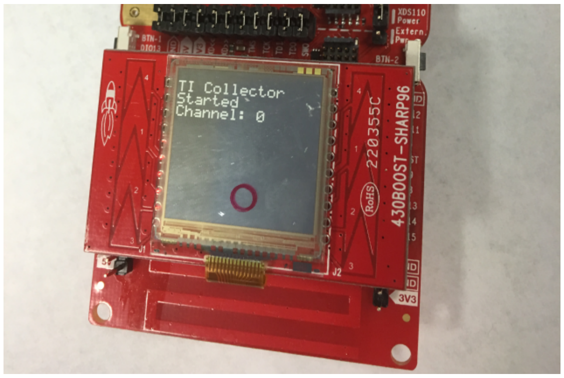


Figure 6-4. LCD Display (1 of 2)

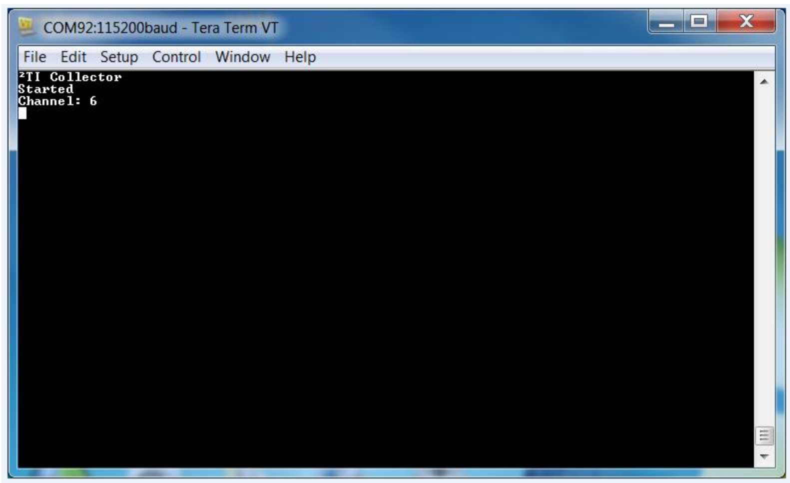


Figure 6-5. Hyperterminal When Collector is Started

NOTE: If you want to use LCD BoosterPack, remove USE_UART_PRINTF in the predefined symbols.

---

8. Press the BTN-2 button on the collector LaunchPad to allow new devices to join the network. Pressing the BTN-2 a second time closes the network, and new devices are not be able to join the network. Press button 2 a third time to allow new devices to join the network. When the network is open to new devices, the red LED blinks; when it does not blink, the network is closed to new devices. After the sensor successfully joins the network, the LCD on the collector LaunchPad is as shown in Figure 6-6.

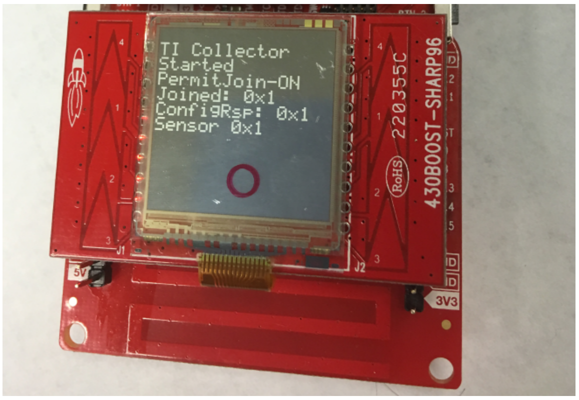


Figure 6-6. LCD Display (2 of 2)

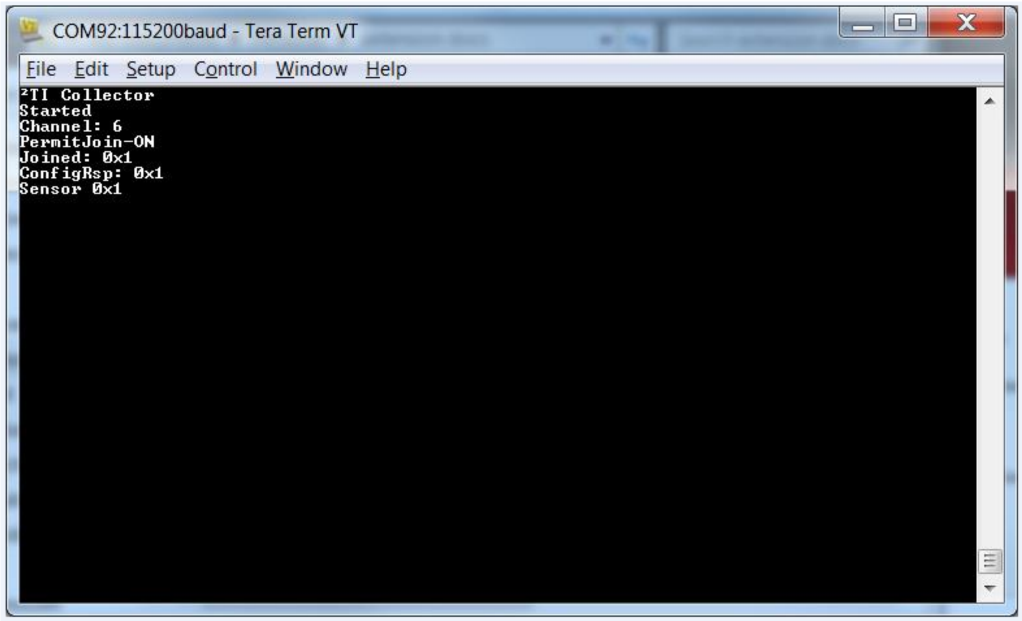


Figure 6-7. Hyperterminal When Sensor Joins Collector

NOTE: For instructions on programming and running the sensor application, see Section 6.2.

6.2 Sensor

This sample project implements a sensor, the end device which reads sensor information and sends it to the coordinator at a configured interval.

6.2.1 Running the Application

Perform these steps to run the out-of-box Sensor Example Application.

1. Import the sensor_cc13x0lp project, as described in Section 2.5.2.3.
2. After importing, configure the following settings in the config.h file. To configure the settings on the sensor application project:
   (a) Select the sensor_cc13x0lp project in the CCS Project Explorer window.
   (b) Find the config.h file, as shown in Figure 6-8.

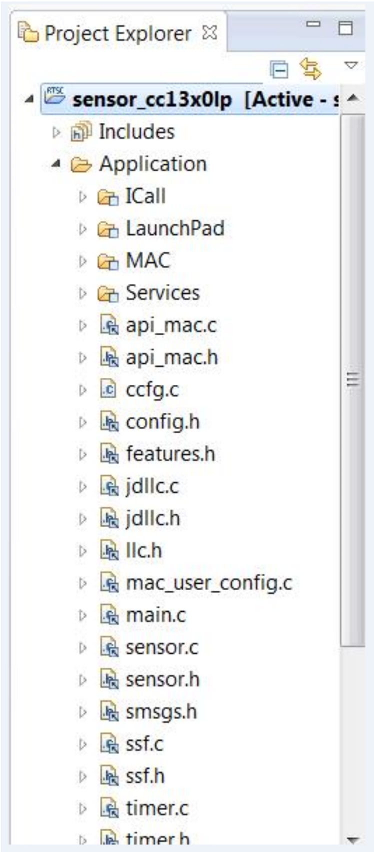


Figure 6-8. Config.h File

   (c) Set #define CONFIG_PAN_ID to the desired value to match the collector.
   (d) Set the Phy ID according to region of interest:
   - For a US or 915-MHz band of operation, use the OOB CONFIG_PHY_ID settings:

```
/*! Setting for Phy ID */
#define CONFIG_PHY_ID                (APIMAC_STD_US_915_PHY_1)
```

   - For ETSI PHY for Europe (or 868-MHz band operation), configure the CONFIG_PHY_ID parameter:

```
/*! Setting for Phy ID */
#define CONFIG_PHY_ID                (APIMAC_STD_US_915_PHY_3)
```

(e) Set the preferred channel of operation (matching the collector) in CONFIG_CHANNEL_MASK:

```
/*!
 Channel mask used when CONFIG_FH_ENABLE is false
 Each bit indicates if the corresponding channel is to be
 scanned First byte represents channel 0 to 7 and the last byte represents
 channel 128 to 135
 */
#define CONFIG_CHANNEL_MASK           { 0x0F, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, \
                                        0x00, 0x00, 0x00, 0x00, 0x00 }
```

The channel numbers available in each band follow:

- 902 to 928 MHz (50 kbps): 0 to 128, such as when CONFIG_PHY_ID = APIMAC_STD_US_915_PHY_1
- 863 to 870 MHz (50 kbps): 0 to 33, such as when CONFIG_PHY_ID = APIMAC_STD_ETSI_863_PHY_3

To configure other parameters, see Section 11.3.

3. Build the stack project, then connect a LaunchPad to the PC, and call it sensor-launchpad.
4. Download the stack project onto the sensor-launchpad, and terminate the debug session when the download is finished.
5. Build the sensor application project and download it onto the CC13x0 LaunchPad using the method previously used to download the stack and terminate the debug session once the download is complete.
6. Terminate the debug session when the download is complete. The initial state of the LCD before the sensor joins a network is as shown in Figure 6-9.

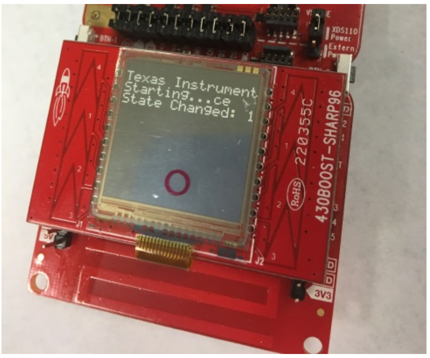


Figure 6-9. LCD Sensor Display (1 of 2)

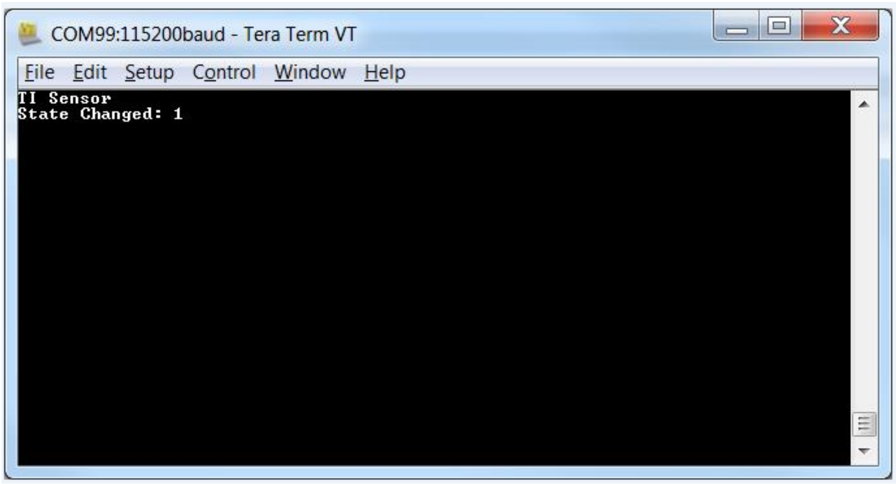


Figure 6-10. Hyperterminal When Sensor is Powered Up

After the sensor successfully joins the network, the LCD on sensor LaunchPad is as shown in Figure 6-11.

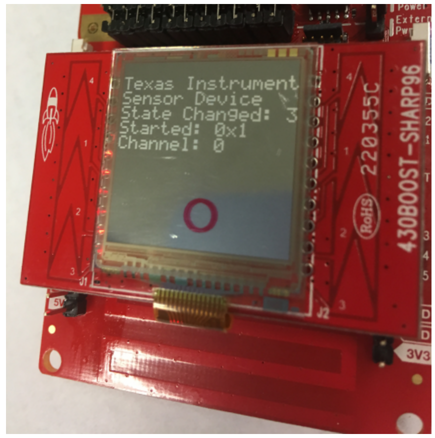


Figure 6-11. LCD Sensor Display (2 of 2)

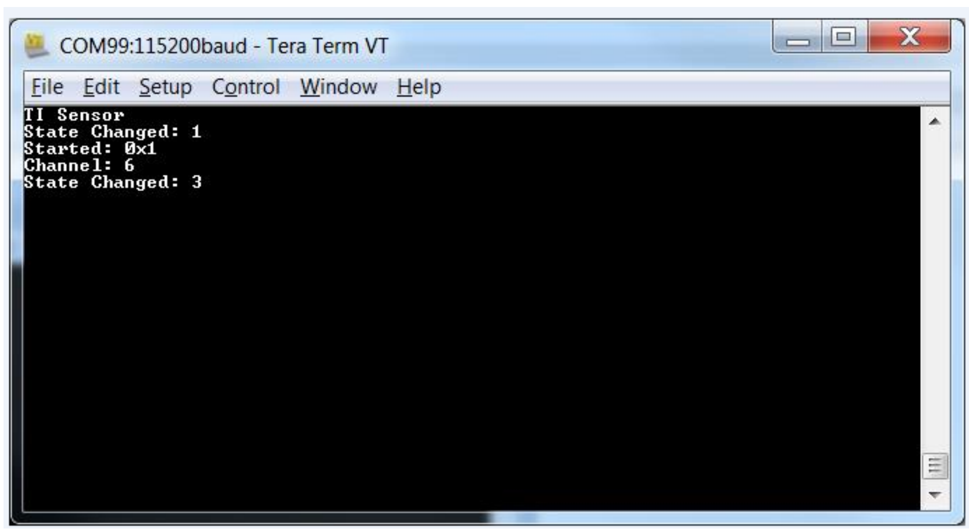


Figure 6-12. Hyperterminal When Sensor Joins The Network

NOTE: After the sensor node has successfully joined the network, it receives a configuration request message from the collector application. The node then configures the time interval on how often to report the sensor data to the collector application, and how often to poll for buffered messages in case of sleepy devices. After receiving the configuration request message, the green LED toggles whenever the device sends the message.

6.3 FH Conformance Certification Application Example

The FH conformance certification example application is provided to enable users to perform an FCC or ETSI compliance tests related to channel occupancy. FCC regulations states that a channel hopping device can transmit at a high power up to 30 dbm if using more than 50 channels for hopping and ensuring that the average channel occupancy time over a 20 second period is less than 400 ms per channel [FCC]. To verify this behavior, test labs perform a channel occupancy test [FCCTest]. The actual rate at which a node shall occupy a channel depends on the application traffic. To account for a generic application profile, a back-to-back data transmission mode can be used for the compliance test.

To enable back-to-back transmissions from a device to the collector, the following configurations must be set:

- CERTIFICATION_TEST_MODE = true
- CONFIG_FH_ENABLE = true

The rest of the configurations can be set to default. In this mode, the device joins the collector and transmits back to back frames to the collector. The collector does not generate any frames but simply acknowledges the transmissions from the sensor. The frames can be capture using a spectrum analyzer to perform the channel occupancy test. The mode can also be used for ETSI testing. Note that the example application is only provided for a general guidance and for ease in performing regulation tests. Any other alternate application profiles to better reflect the application needs can also be used for compliance tests.

[FCC] FCC Part 247 - 47 CFR 15.247 - Operation within the bands 902 to 928 MHz, 2400 to 2483.5 MHz, and 5725 to 5850 MHz

[FCCTest] C63.10-2013 - American National Standard of Procedures for Compliance Testing

6.4 Configuration Parameters

Table 6-1 lists the various configuration parameters available for the collector and sensor applications. Features.h allows the user to compile only the features needed for the mode of operation needed, which facilitates memory savings. Out of the box FEATURE_ALL_MODES is defined which enables all modes of operation and FEATURE_MAC_SECURITY is define which enable security.

The user can only define one of the features among the following options.

- FEATURE_FREQ_HOP_MODE – frequency hopping mode of operation
- FEATURE_BEACON_MODE – beacon mode of operation
- FEATURE_NON_BEACON_MODE – nonbeacon mode of operation
- FEATURE_MAC_SECURITY – enable security

Table 6-1. Configuration Parameters

Config Parameter	Description
Common Configuration Parameters	
CONFIG_SECURE	Turn security ON or OFF This value should match for both collector and sensor.
CONFIG_PAN_ID	Used to restrict the network to a certain PAN ID. If left as 0xFFFF, the collector starts with PAN ID 0x0001. If this parameter is set to a certain value for the collector, the value should be set to either the same value or 0xFFF for the sensor application, so that the sensor joins the intended parent.
CONFIG_FH_ENABLE	Used to turn frequency-hopping operation ON or OFF
CONFIG_MAX_BEACONS_RECD	Maximum number of received beacons to filter after the scan request is sent out
CONFIG_LINKQUALITY	The device responds to enhanced-beacon requests if mpduLinkQuality is equal to or higher than this value.
CONFIG_PERCENTFILTER	The device randomly determines if it is to respond to enhanced-beacon requests based on meeting this probability (0 to 100%).
CONFIG_SCAN_DURATION	Scan duration for scan request
CONFIG_MAX_DEVICES	Maximum number of children for coordinator
CONFIG_MAC_BEACON_ORDER	Beacon order according to mode of operation: For nonbeacon and frequency-hopping modes, set this value to 15 for both collector and sensor. For beacon mode, this value can be from 1 to 14, and must match for both collector and sensor.
CONFIG_MAC_SUPERFRAME_ORDER	Superframe order, according to mode of operation: For nonbeacon and frequency-hopping modes, set this value to 15 for both collector and sensor. For beacon mode, this value can be from 1 to 14, and must match for both collector and sensor.

Table 6-1. Configuration Parameters (continued)

Config Parameter	Description
CONFIG_CHANNEL_PAGE	The channel page on which to perform the scan
CONFIG_PHY_ID	PHY ID corresponding to the PHY descriptor to use based on region of operation
KEY_TABLE_DEFAULT_KEY	Default security key
CONFIG_CHANNEL_MASK	For the **collector application**: Each bit indicates if the corresponding channel is to be scanned. The first byte represents channels 0 to 7, and the last byte represents channels 128 to 135. In FH mode: represents the list of channels excluded from hopping. It is a bit string with LSB representing Ch0; for example, 0x01 0x10 represents Ch0 and Ch12 are excluded. Currently, the same mask is used for unicast and broadcast-hopping sequences. For the **sensor application**: For nonfrequency-hopping configuration: Channel mask – each bit indicates if the corresponding channel is to be scanned. The first byte represents channels 0 to 7, and the last byte represents channels 128 to 135. In FH mode: If CONFIG_RX_ON_IDLE = TRUE: represents the list of channels excluded from hopping. It is a bit string with LSB representing Ch0; for example, 0x01 0x10 represents Ch0 and Ch12 are excluded. The same mask is used for both unicast and broadcast-hopping sequences. If CONFIG_RX_ON_IDLE = FALSE: represents the list of channels to be used for hopping. It is a bit string with LSB representing Ch0; for example, 0x01 0x10 represents Ch0 and Ch12 are used for hopping. In this mode, the node hops in increasing order of the chosen channel.
FH_ASYNC_CHANNEL_MASK	List of channels to target the async frames. It is represented as a bit string with LSB representing Ch0; for example, 0x01 0x10 represents Ch0 and Ch12 are included. It must cover all channels that could be used by a target device in its hopping sequence. Channels marked beyond number of channels supported by PHY Config are excluded by stack. To avoid interference on a channel, remove it from async mask and add it to the exclude channels (CONFIG_CHANNEL_MASK).
CONFIG_DWELL_TIME	Duration of the unicast and broadcast slot of the node (in ms)
CONFIG_FH_NETNAME	Default value for FH PIB attribute netname
CONFIG_TRANSMIT_POWER	Value for transmit power in dBm. Default value is 14, allowed values are any value between 0 dBm to 14 dBm in 1 dB increments, and -10 dBm. When the nodes in the network are close to each other, lowering this value helps reduce saturation.

Table 6-1. Configuration Parameters (continued)

Config Parameter	Description
CERTIFICATION_TEST_MODE	If set to true, the device joins the collector and transmits back-to-back frames to it. The collector does not generate any frames, but simply acknowledges the transmissions from the sensor. The frames can be captured using a spectrum analyzer to perform the channel occupancy test. The mode can also be used for ETSI testing. The example application is only provided for a general guidance and for ease in performing regulation tests. **NOTE:** The FH conformance certification example application is provided to allow users to perform a FCC or ETSI compliance tests related to channel occupancy. FCC regulations state that a channel hopping device can transmit at a higher power of up to 30 dbm if it uses more than 50 channels for hopping and ensures that the average channel occupancy time over a 20 second period is less than 400 ms per channel [FCC]. To verify this behavior, test labs perform a channel occupancy test [FCC Test]. The actual rate at which a node occupies a channel depends on the application traffic. To account for a generic application profile, a back-to-back data transmission mode can be used for the compliance test. To enable back-to-back transmissions from a device to the collector, the following configurations are to be set: • CERTIFICATION_TEST_MODE = true • CONFIG_FH_ENABLE = true In this mode, the device will join the collector and transmit back to back frames to collector. Collector will not generate any frames but would simply acknowledge the transmissions from sensor. The frames can be capture using a spectrum analyzer to perform the channel occupancy test. The mode can also be used for ETSI testing. Note that the example application is only provided for a general guidance and for ease in performing regulation tests. Any other alternate application profiles to better reflect the application needs can also be used for compliance tests. [FCC] FCC Part 247 - 47 CFR 15.247 - Operation within the bands 902 to 928 MHz, 2400 to 2483.5 MHz, and 5725 to 5850 MHz [FCCTest] C63.10-2013 - American National Standard of Procedures for Compliance Testing
FH_NUM_NON_SLEEPY_NEIGHBOURS	The number of non-sleepy end devices to be supported. This value is limited to 50. This setting can be used for memory savings so that the stack allocates memory proportional to the number of end devices requested.
FH_NUM_SLEEPY_NEIGHBOURS	The number of sleepy end devices to be supported. This value is limited to 50. This setting can be used for memory savings so that the stack allocates memory proportional to the number of end devices requested.
Collector-Specific Configuration Parameters	
CONFIG_COORD_SHORT_ADDR	Short address for coordinator

Table 6-1. Configuration Parameters (continued)

Config Parameter	Description
CONFIG_TRICKLE_MIN_CLK_DURATION	The minimum trickle timer window for PAN advertisement and PAN configuration frame transmissions. Default is 0.5 minute. TI recommends setting this to half of the PAS/PCS MIN timer.
CONFIG_TRICKLE_MAX_CLK_DURATION	The maximum trickle timer window for PAN advertisement and PAN configuration frame transmissions. Default is 16 minutes.
CONFIG_FH_PAN_SIZE	Default value for PAN size PIB
CONFIG_DOUBLE_TRICKLE_TIMER	Enables doubling of PA or PC trickle time: used when the network has non-sleepy nodes and there is a requirement to use PA or PC to convey updated PAN information.
Sensor-Specific Configuration Parameters	
CONFIG_MAX_DATA_FAILURES	Maximum number of data failures before considering sync loss (this parameter is available only for the sensor)
CONFIG_PAN_ADVERT_SOLICIT_CLK_DURATION	PA solicit trickle timer duration in ms (this parameter is available only for the sensor)
CONFIG_PAN_CONFIG_SOLICIT_CLK_DURATION	PAN configuration solicit trickle timer duration in ms (this parameter is available only for the sensor)
CONFIG_FH_START_POLL_DATA_RAND_WINDOW	FH poll/sensor message start time randomization window (this parameter is available only for the sensor)
CONFIG_POLLING_INTERVAL	Polling interval in ms (this parameter is available only for the sensor)
CONFIG_FH_MAX_ASSOCIATION_ATTEMPTS	Maximum number of attempts for association in FH mode after reception of a PAN configuration frame (this parameter is available only for the sensor)
CONFIG_SCAN_BACKOFF_INTERVAL	Scan back-off interval in ms (this parameter is available only for the sensor)
CONFIG_RX_ON_IDLE	Used to indicate if a device is sleepy or nonsleepy: FALSE for sleepy, and TRUE for nonsleepy (this parameter is available only for the sensor).
CONFIG_ORPHAN_BACKOFF_INTERVAL	Delay between orphan notifications

6.5 Coprocessor

The coprocessor project is used to build a MAC coprocessor device that works with a host processor in a 2-chip scenario. The coprocessor provides an interface to the TI 15.4-MAC protocol stack, full-function MAC capability over serial interface to the application running on the host. This device, programmed with the coprocessor application and the TI 15.4-MAC protocol stack, allows the addition of TI 15.4-MAC wireless functionality to systems that are not suited to single-chip solutions. A prebuilt hex file for the coprocessor is provided in the SDK. If changes are needed, such as an addition of a custom API command, the coprocessor project can be used to generate a new hex file.

6.6 Linux Example Applications

A brief description of the Linux example applications follows. For more detail, refer to the documentation included with the TI 15.4-MAC Linux SDK installer at http://www.ti.com/tool/SIMPLELINK-CC13X0-SDK.

6.6.1 Linux Collector and Gateway Application

These two example applications are provided inside the TI 15.4-MAC Linux SDK installer, a component of the TI 15.4-MAC installation. The Linux collector example application interfaces with the CC13x0 running the coprocessor and stack image through a UART. The Linux collector example application provides the same functionality as the embedded collector application, while also providing a socket server interface to the Linux gateway application. The Linux gateway application implemented within the Node.js framework connects as a client to the socket server created by the Linux collector example application, and establishes a local web server to which the user can connect through a web browser (in the local network), and monitor and control the network devices. The collector and gateway applications can be great starting points for creating IOT applications. For more details, refer to the *Linux Developers Guide* included with the TI 15.4-MAC Linux SDK installer.

6.6.2 Linux Serial Bootloader Application

This example application is included inside the TI 15.4-MAC Linux SDK installer. This application provides the capability to upgrade the firmware of the CC13x0 MCU through the CC13x0 ROM bootloader. For more details, refer to the *Linux Developers Guide* included with the TI 15.4-MAC Linux SDK installer.

Packet Sniffer

A packet sniffer can be created using the CC1200 mounted over a TRxEB evaluation board. This feature enables easier development and debugging for those developing products with the TI 15.4-Stack. This section provides details on the required software, where to get it, and how to set it up to sniff the over-the-air (OTA) traffic. Wireshark™ is the recommended packet sniffer.

The CC13x0 SimpleLink SDK installs the essential software tools required to set up the packet sniffer. The TI 15.4-Stack installs the TiWsPc, which uses TI hardware to capture OTA data before sending the packets to Wireshark or a PCAP file, and provides .dll files to dissect packets that follow the TI 802.15.4ge protocol to Wireshark. Figure 7-1 is an example of TI 15.4-Stack-based application OTA traffic being presented as a Wireshark capture.

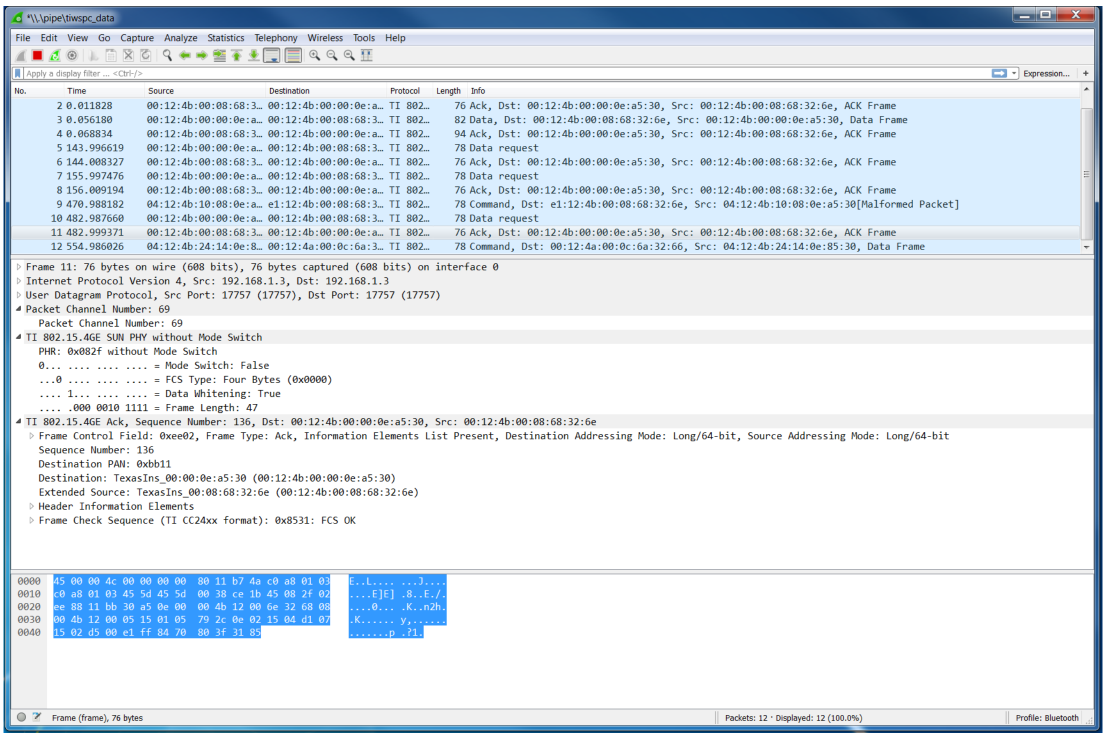


Figure 7-1. OTA Traffic

Choose a packet to get detailed information on the data in that packet. The installed .dll file lets Wireshark dissect the information in a TI 802.15.4GE packet for easy debugging.

Topic **Page**

7.1 Setting Up the Sniffer

7.1.1 Install the Required Software

1. Install the CC13x0 SimpleLink SDK. This SDK installs:
   - TiWsPc at *C:/Program Files (x86)/Texas Instruments/TiWsPc-1.12.15*
   - .DLL files at *C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\tools\ti154stack\tiwsds\plugins*
2. Install the **2.0.x** stable Wireshark release from https://www.wireshark.org/#download. The architecture version downloaded (64-bit vs 32-bit) effects which plug-in to install.

NOTE: The latest Wireshark version is not compatible, only use v2.0.x

3. Run the Wireshark installer as administrator. If this step is not done and a previous Wireshark version is installed, the installer can fail with the message:

   Error opening the file for writing:
   C:\Program Files\Wireshark\uninstall.exe

7.1.2 Hardware Setup

7.1.2.1 Required Hardware

- CC1200EM
- TRxEB

7.1.2.2 Setup

1. Mount the CC1200EM on the TRxEB and connect the board to the PC.
2. Start the SmartRF flash programmer.
3. Select the Program Evaluation Board option.
4. If the Update EB Firmware option is available, update the firmware, as shown in Figure 7-2.

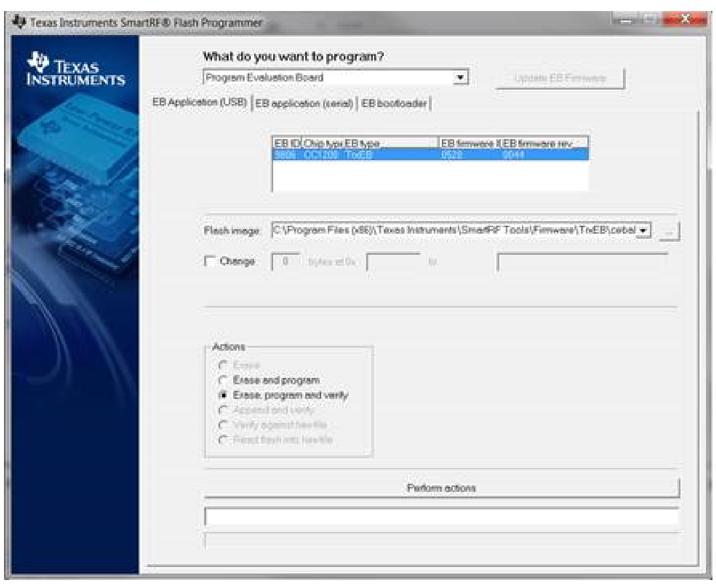


Figure 7-2. Update EB Firmware

TrxEB firmware rev. 0044 is the latest version at the time of this writing, and has been tested with this sniffer setup.

7.1.3 Software Setup

7.1.3.1 Texas Instruments Wireshark Packet Converter Setup

The following are ways to transfer data from Texas Instruments Wireshark Packet Converter (TiWsPc) to Wireshark:

- Pipe – (recommended): data is sent to Wireshark on the local machine. (Vista/Windows 7 or higher only)
- Socket – (stand-alone mode): data is sent to the Microsoft Loopback Adapter with Wireshark running on the local machine.
- Socket – (remote mode): data is sent to Wireshark on another machine or the local machine using the network adapter.
- File – data is sent to a PCAP file that can be opened in Wireshark.

The following guide demonstrates how to use the pipe solution with Windows 7. More advanced users might want to try a socket; for more details, consult the TiWsPc README for instructions.

1. Run TiWsPc.
2. When the TiWsPc opens and prompts to select a device family, select *TIMAC/TI 802.15.4ge*.
3. Select Data → Data Out, check Use Pipe, and click Ok as shown in Figure 7-3.

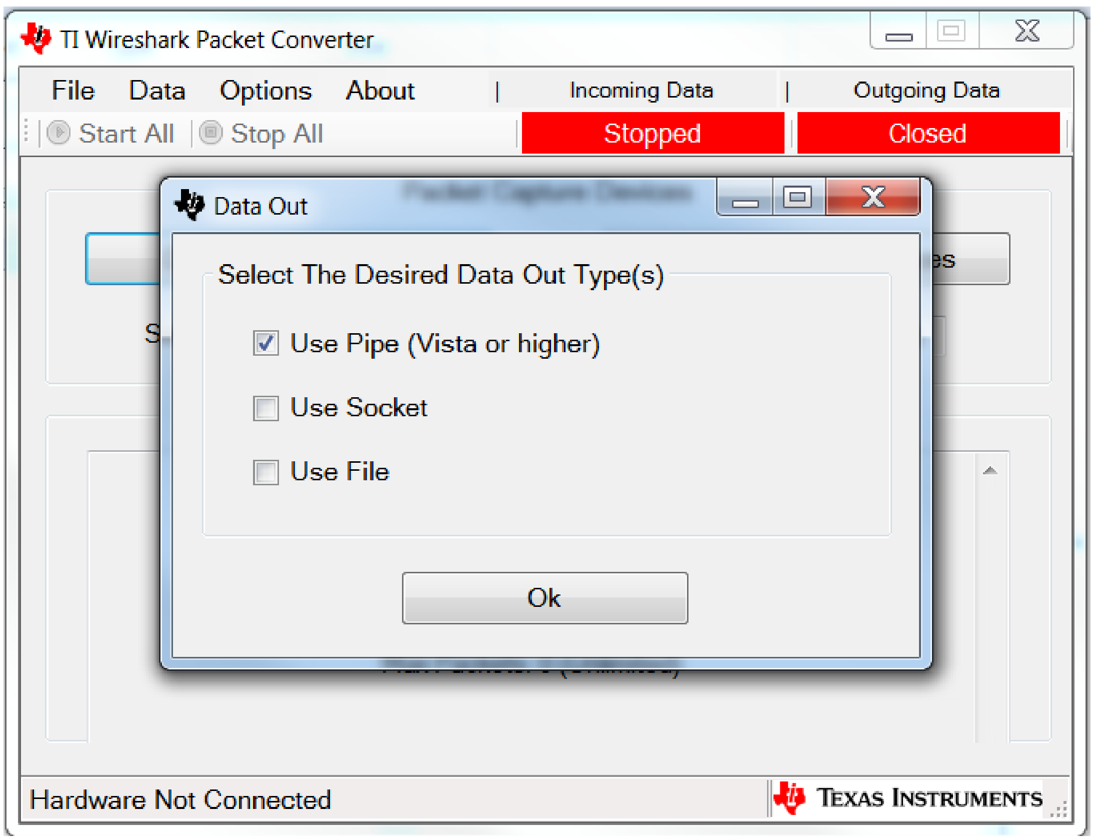


Figure 7-3. Use Pipe

4. Press the Device Configuration button, select a sniffer device and channel to use, then press Done.

   Various prebuilt *.prs files for 915-MHz band and 868-MHz band are provided with the CC13x0 SimpleLink SDK. The files are at *C:\ti\simplelink_cc13x0_sdk_1_00_00_xx\tools\ti154stack\tiwspc\PRS*.

   There are two folders at this location:

   - phy1 – for the US 915-MHz band, 50-kbps data rate, 2FSK modulation scheme.
   - phy3 – for the ETSI 868-MHz band, 50-kbps data rate, 2FSK modulation scheme.

5. Select the desired .phy and .prs file for the required channel number.
6. Press Start All; incoming data goes green, and outgoing turns blue. The TiWsPc icon is blue.
7. Create a new Wireshark desktop shortcut, modifying the target by adding *-i\\.\pipe\tiwspc_data -k* to the end, as shown in Figure 7-4.

   Example target entry: "C:\<path>\wireshark.exe" -i\\.pipe\tiwspc_data -k

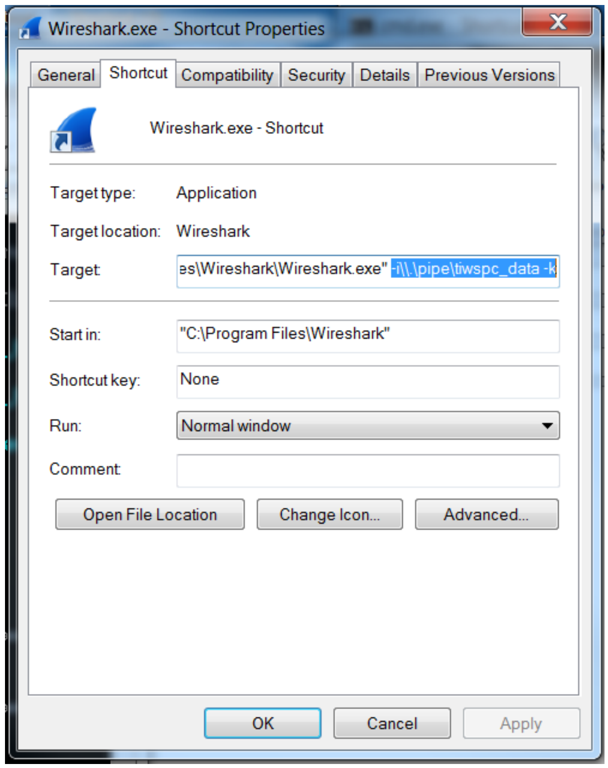


Figure 7-4. Shortcut Properties

8. Run Wireshark from the new shortcut, which opens the other end of the pipe.

Wireshark now shows captured data (packets sent to UDP address 17757 indicate TI 802.15.4GE packets, now set up the dissector to enable detailed dissection of this protocol), and the TiWsPc turns green.

7.1.3.2 Wireshark Dissector Setup

1. Check which architecture version (32-bit or 64-bit) of Wireshark was downloaded. Follow Step 2 according to that choice before going to Step 3.
2. **For 32-bit**: Copy ti802154ge-x86-2x.dll
   From: *c:/Program Files (x86)/Texas Instruments/TI 802.15.4ge Wireshark Plugin-<Version>/Plugins/ti802154ge-x86-2x.dll*
   To: *c:/Program Files (x86)/Wireshark/plugins/2.0.x* (x can be any number)
   For 64-bit: Copy ti802154ge-x64-2x.dll
   From: *c:/Program Files (x86)/Texas Instruments/TI 802.15.4ge Wireshark Plugin-<Version>/Plugins/ti802154ge-x64-2x.dll*
   To: *c:/Program Files/Wireshark/plugins/2.0.x* (x can be any number)

3. Open Wireshark, and check that the plug-in is installed by going to Help->About Wireshark and clicking the Plugins tab. The ti802154ge-x(32/64)-2x.dll file is listed, as shown in Figure 7-5.

   If so, the plugin is installed and receives packets from TiWsPc. If not, see the following for troubleshooting.

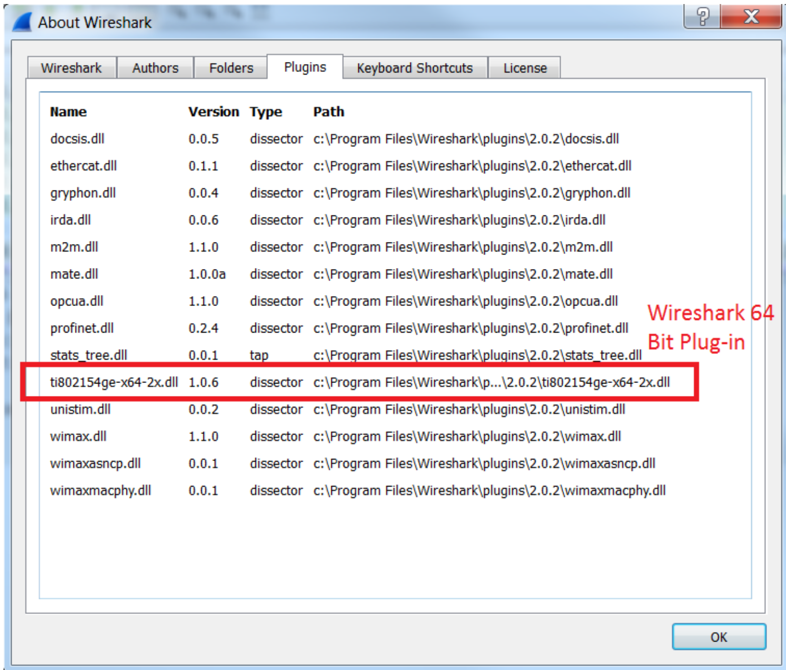


Figure 7-5. Wireshark Plugin

4. If using TiWsPc, navigate to Edit → Preferences and select Protocols → TI 802.15.4GE under the left-hand menu. The first two checkboxes must be checked, as shown in Figure 7-6.

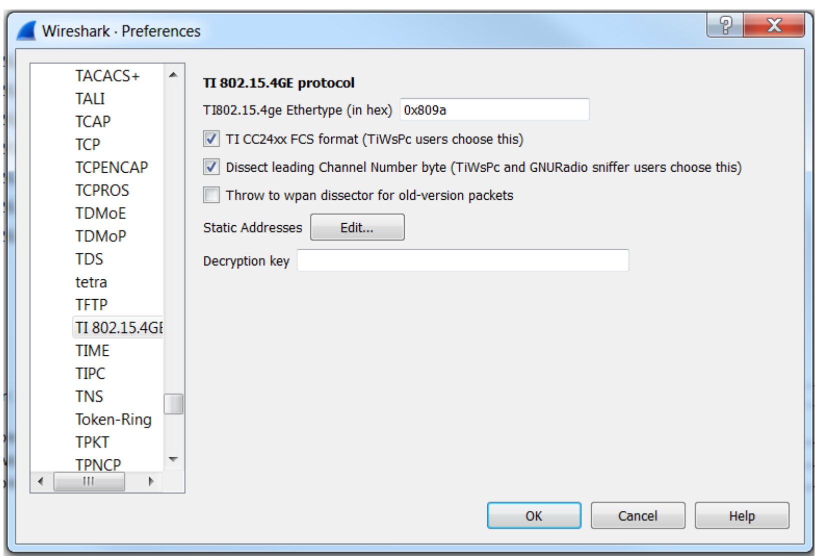


Figure 7-6. Wireshark Preferences

Additionally, to use secured packets, add a decryption key and static address pairings (for pairing short address and PAN-IDs with long addresses for decryption).

7.2 Using Wireshark

1. To filter a certain packet attribute, right-click on the selected packet attribute.
2. Choose Apply as Filter, and then Selected, as shown in Figure 7-7.

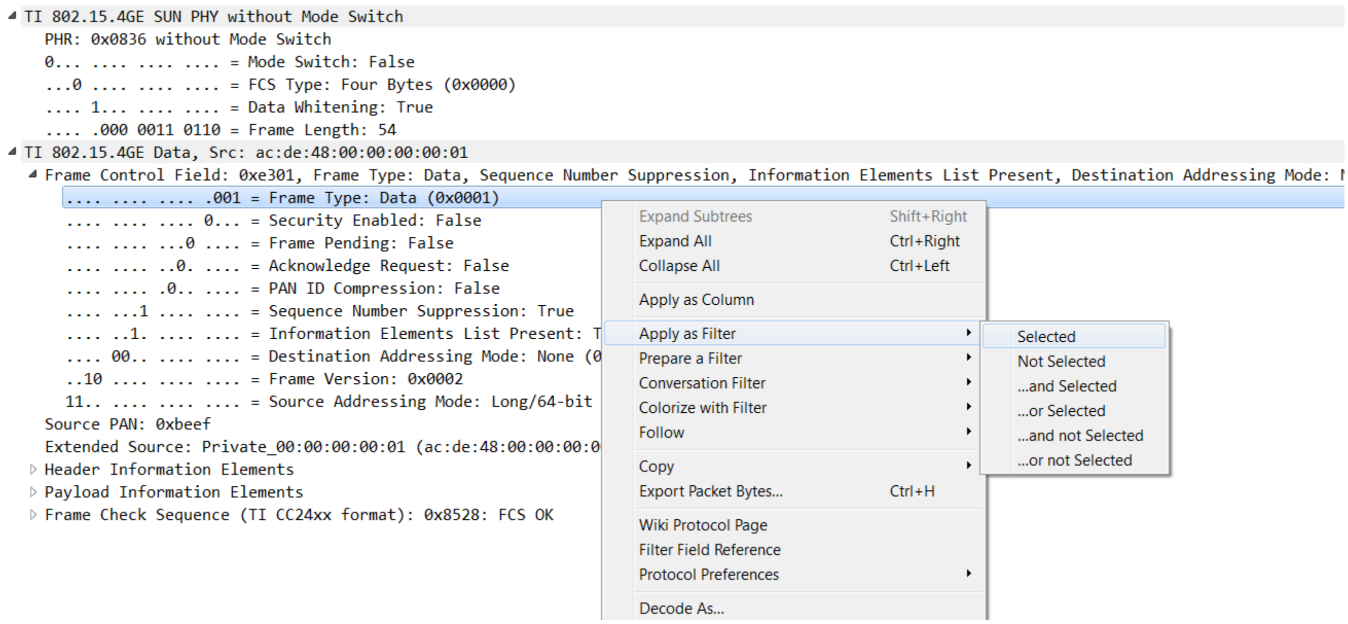


Figure 7-7. Apply Filter

3. In the filter textbox, select a filter of the form ti802.15.4ge.<attribute>==0x<XXXX>. Figure 7-8 shows how to filter the capture to display only TI 802.15.4GE data packets.

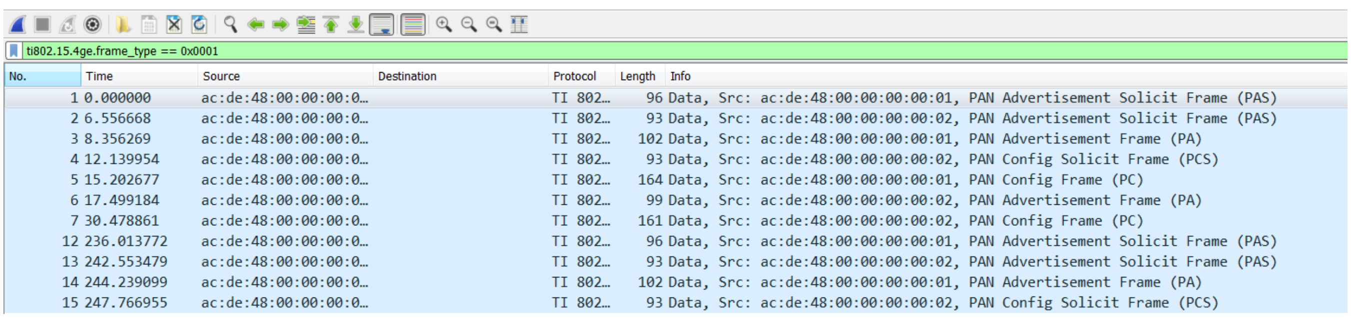


Figure 7-8. Filter Selection

4. Get the attribute name of any field in a packet, as well as a description based on the specification, by looking to the bottom of the screen, underneath the raw packet data viewer, as shown in Figure 7-9.

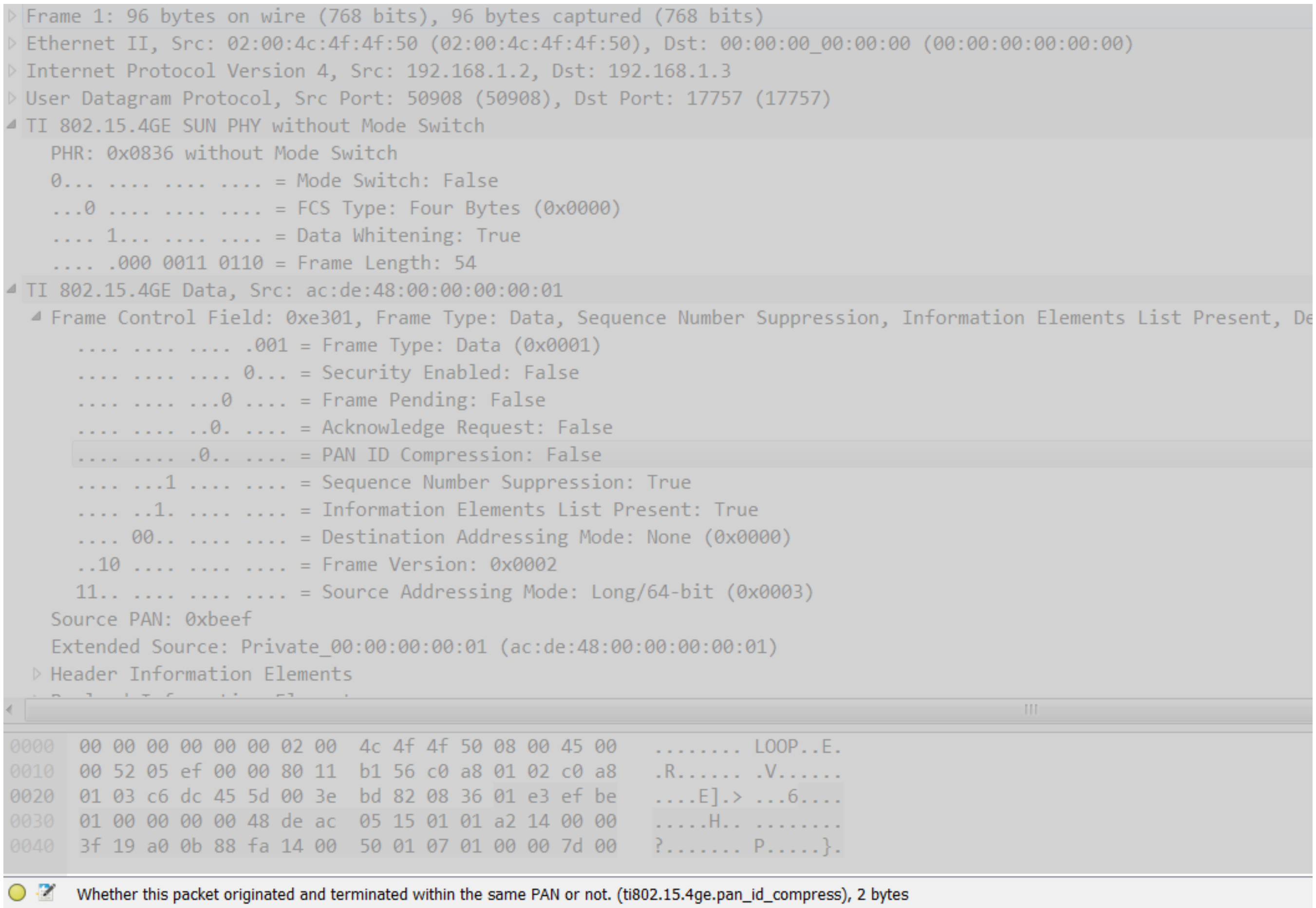


Figure 7-9. Get Attribute Name

7.3 Troubleshooting

7.3.1 TiWsPc Troubleshooting

- If a Communication error occurs when a device is started, try power-cycling the sniffer hardware to correct the issue.
- If a Data Buffer Overflow occurs, the TiWsPc program cannot get the data fast enough from the device. Try any or all of the following: reduce CPU load, network traffic, and disk load from other programs, or reduce the number of capturing devices.
- If Wireshark reports corrupted memory or throws an assertion and exits, this is a Wireshark issue; TiWsPc can deliver more messages in a short period of time than Wireshark can handle. Try reducing the number of sniffer device options in use, to reduce the flow to Wireshark using the file data out. Alternatively, configure the TiWsPc packet limit option for the selected data output method. When this limit is reached, TiWsPc automatically stops the current data capture.

7.3.2 Wireshark Dissector Troubleshooting

- If after installing Wireshark, the error shown in Figure 7-10 appears, a 64-bit plugin is installed, but you are using 32-bit Wireshark. To debug, repeat Steps 1 and 2 from Section 7.1.3.2.

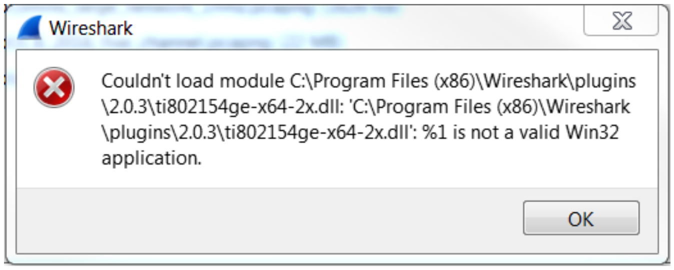


Figure 7-10. Wireshark Plugin Error

- If after starting Wireshark, the error shown in Figure 7-11 appears, delete one of the two TI 802.15.4ge plug-ins in the Wireshark plug-ins folder.

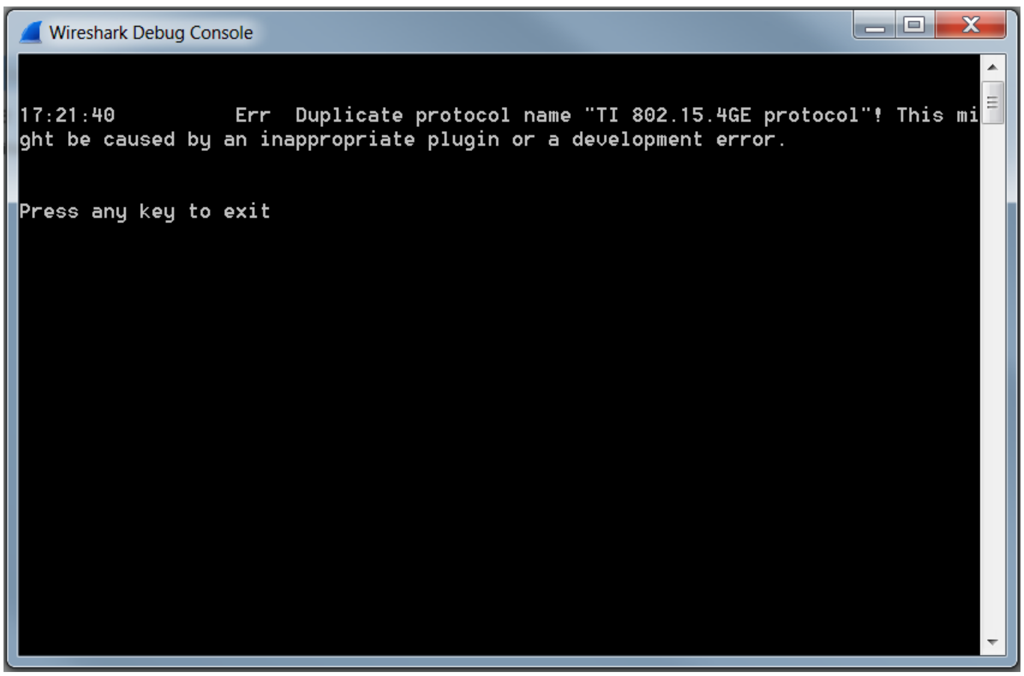


Figure 7-11. Wireshark Debug Error

- When opening Wireshark, you may get an error that opens a command prompt from the Wireshark Debug Console and reads *Err Field (abbrev='Frame Length') does not have a name*, and *Press any key to exit*. Alternatively, you may get a message that reads *The procedure entry point ep_alloc could not be located in the dynamic link library libwireshark.dll*. These start-up errors indicate that the installed plug-in is for an incompatible version of Wireshark. Check this by going to Help → About Wireshark, and check that the version number is 2.0.x. If the version number is not 2.0.x., download and install a 2.0.x release, because the plug-in is not backwards-compatible.
- For any other questions or problems, consult the README at *C:\Program Files (x86)\Texas Instruments\TI 802.15.4ge Wireshark Plugin-<version>\README.txt*.

Peripherals and Drivers

The TI-RTOS provides a suite of CC13x0 and CC26xx peripheral drivers that can be added to an application. The drivers provide a mechanism for the application to interface to the CC13x0 and CC26xx onboard peripherals, and communicate with external devices.

8.1 Adding a Driver

The TI-RTOS drivers (DRV_PACKAGE) and corresponding DriverLib (CC13XXWARE) are provided in source and precompiled library form. By default, the TI 15.4-Stack project configuration links to the prebuilt library in the Project Properties → Resource → Linked Resources, Path Variables tab:

- CC13XXWARE_LOC: ${COM_TI_SIMPLELINK_CC13XX_CC26XX_SDK_INSTALL_DIR}\source\ti\devices\cc13x0
- DRIVER_LOC: ${COM_TI_SIMPLELINK_CC13XX_CC26XX_SDK_INSTALL_DIR}\source

To use a precompiled driver, include the respective driver C include file in the application files where the driver APIs are referenced.

For example, to add the PIN driver for reading or controlling an output I/O pin:

```
#include <ti/drivers/pin/PIN.h>
```

To override a specific prebuilt version of a driver, include the respective C source and include files to the project within the IDE. The IDE uses the source versions included in the project in lieu of the respective prebuilt library version. This override option is useful in cases where the prebuilt drivers are used for other drivers, but source-level debugging is available within the IDE for specific drivers.

For a description of available features and driver APIs, see the TI-RTOS Driver documentation.

8.2 Board File

The board file is used to set fixed driver configuration parameters for a specific board configuration, such as configuring the GPIO table for the PIN driver or defining which pins are allocated to the I^2C, SPI, or UART driver.

The board files for the LaunchPad are in the following path:

${COM_TI_SIMPLELINK_CC13XX_CC26XX_SDK_INSTALL_DIR}\source\ti\boards

Available CC13x0 options include CC1310_LAUNCHXL and CC1350_LAUNCHXL. The TI 15.4-Stack uses a tailored board file from TI-RTOS release. The board type source and include file is in the Project Explorer → sensor_cc13x0lp → launchpad folder, as shown in Figure 8-1.

In CCS, open the launchpad folder. Edit the file as needed.

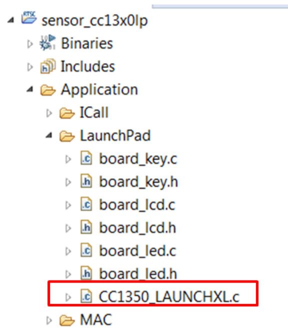


Figure 8-1. LaunchPad Folder

At a minimum, the board file contains a PIN_Config structure that places all configured and unused pins in a default, safe state and defines the state when the pin is used:

```
/*
 *  ======================== IO driver initialization ========================
 *  From main, PIN_init(BoardGpioInitTable) should be called to setup safe
 *  settings for this board.
 *  When a pin is allocated and then de-allocated, it will revert to the state
 *  configured in this table.
*/
/* Place into subsections to allow the TI linker to remove items properly */
#if defined(__TI_COMPILER_VERSION__)
#pragma DATA_SECTION(BoardGpioInitTable, ".const:BoardGpioInitTable")
#pragma DATA_SECTION(PINCC26XX_hwAttrs, ".const:PINCC26XX_hwAttrs")
#endif

const PIN_Config BoardGpioInitTable[] = {

    Board_RLED   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,         /*
LED initially off             */
    Board_GLED   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,         /*
LED initially off             */
    Board_BTN1   | PIN_INPUT_EN | PIN_PULLUP | PIN_IRQ_BOTHEDGES | PIN_HYSTERESIS,          /*
Button is active low          */
    Board_BTN2   | PIN_INPUT_EN | PIN_PULLUP | PIN_IRQ_BOTHEDGES | PIN_HYSTERESIS,          /*
Button is active low          */
    Board_SPI_FLASH_CS | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH | PIN_PUSHPULL | PIN_DRVSTR_MIN,  /*
External flash chip select    */
    Board_UART_RX | PIN_INPUT_EN | PIN_PULLDOWN,                                            /*
UART RX via debugger back channel */
    Board_UART_TX | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH | PIN_PUSHPULL,                      /*
UART TX via debugger back channel */
    Board_DIO1_RFSW | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH | PIN_PUSHPULL | PIN_DRVSTR_MAX,     /*
(compatibility with CC1350LP) */
    Board_SPI0_MOSI | PIN_INPUT_EN | PIN_PULLDOWN,                                          /*
SPI master out - slave in */
    Board_SPI0_MISO | PIN_INPUT_EN | PIN_PULLDOWN,                                          /*
SPI master in - slave out */
    Board_SPI0_CLK | PIN_INPUT_EN | PIN_PULLDOWN,                                           /*
SPI clock */

    PIN_TERMINATE
};
```

This structure is then used to initialize the pins in main(), as seen in Section 5.1 and here:

```
PIN_init(BoardGpioInitTable);
```

8.3 Available Drivers

This section describes each available driver, and provides a basic example of adding the driver to the sensor_cc13x0lp project. For more detailed information on each driver, see the *TI-RTOS API Reference*.

8.3.1 PIN Driver

The PIN driver allows control of the I/O pins for software-controlled general-purpose I/O (GPIO) or connections to hardware peripherals. Example projects that use the PIN driver are collector_cc13x0lp or sensor_cc13x0lp.

As stated in Section 8.2, the pins should first be initialized to a safe state in main(). After this occurs, any module can use the PIN driver to configure a set of pins for use as desired. The following is an example of configuring the sensor_cc13x0lp task to use one pin as an interrupt and another as an output to an LED.

IOID_x pin numbers directly map to DIO pin numbers, as referenced in the *CC1310 Technical Reference Manual*. The pins used are stated in Table 8-1, as well as their mapping on the LaunchPad board.

Table 8-1. DIO Pin Mapping

Signal Name	Pin ID	LaunchPad Mapping
Board_LED0	IOID_6	Board_RLED
Board_BUTTON0	IDIO_13	Board_BTN1

The following sensor-specific functions, ssf.c, and code modifications are required:

1. Include PIN driver files:

```
#include <ti/drivers/pin/PIN.h>
```

2. Declare the pin configuration table and pin state and handle variables to be used by the sensor_cc13x0lp task:

```
static PIN_Config SSF_configTable[] =
{
  Board_LED0   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
  Board_BUTTON0  | PIN_INPUT_EN  | PIN_PULLUP | PIN_HYSTERESIS,
  PIN_TERMINATE
};

static PIN_State ssfPins;
static PIN_Handle hSsfPins;
```

3. Declare the ISR to be performed in the hwi context. This sets an event in the application task and wakes it up, to minimize processing in the hwi context.

```
static void buttonHwiFxn (PIN_Handle hPin, PIN_Id pinId)
{
  // set event in SSF task to process outside of hwi context
  events |= SSF_BTN_EVT;

  // Wake up the application.
  Semaphore_post(sem);
}
```

4. Define the event and related processing (in Sensor_process()) to handle the event from Step 3.

```
#define SSF_BTN_EVT                                  0x0001

    if (events & SSF_BTN_EVT)
    {
      events &= ~SSF_BTN_EVT; //clear event

      //toggle LED0
      if (LED_value)
      {
        PIN_setOutputValue(hSsfPins, Board_LED0, LED_value--);
      }
      else
      {
        PIN_setOutputValue(hSsfPins, Board_LED0, LED_value++);
      }
    }
```

5. In Sensor_init(), open the pins for use and configure the interrupt:

```
// Open pin structure for use
  hSsfPins = PIN_open(&ssfPins, SSF_configTable);
  // Register ISR
  PIN_registerIntCb(hSsfPins, buttonHwiFxn);
  // Configure interrupt
  PIN_setConfig(hSsfPins, PIN_BM_IRQ, Board_BUTTON0 | PIN_IRQ_NEGEDGE);
  // Enable wakeup
  PIN_setConfig(hSsfPins, PINCC26XX_BM_WAKEUP, Board_BUTTON0|PINCC26XX_WAKEUP_NEGEDGE);
```

6. Compile, download, and run. Pushing the Up button on the LaunchPad toggles the red LED. There is no debouncing implemented here.

8.3.2 UART

There are many possible ways to configure the UART driver. See the *TI-RTOS API Reference* for more information. An example project that uses the UART driver is coprocessor_cc13x0lp. The coprocessor_cc13x0lp project includes, in addition to a UART driver, additional GPIOs for power management, a packet parser, and other items that are out of the scope of this documentation. In this section, an example is provided for using the UART driver with the default settings from UART_Params_init(): blocking mode, baud rate 115200, and so forth.

This example uses the UART peripheral already defined in the board file, as listed in Table 8-2.

Table 8-2. UART Pin Mapping

Signal Name	Pin ID	LaunchPad Mapping
Board_UART_RX	IOID_2	J1.4 (RXD)
Board_UART_TX	IDIO_3	J1.3 (TXD)

1. Include the UART driver:

```
#include <ti/drivers/UART.h>
```

2. Declare the UART handle and parameter structures as local variables:

```
static UART_Handle uartHandle;
static UART_Params params;
```

3. Initialize the UART driver in NPITLUART_initializeTransport():

```
UART_Params_init(&params);
uartHandle = UART_open(Board_UART, &params);
```

4. Perform a sample 5-byte UART write where desired:

```
uint8 txbuf[] = {0,1,2,3,4};
  UART_write(uartHandle, txbuf, 5);
```

Sensor Controller

The Sensor Controller Engine (SCE) is an autonomous processor within the CC13x0. The SCE can control the peripherals in the sensor controller independently of the main CPU. Thus, the main CPU does not have to wake up to execute an ADC sample or poll a digital sensor over SPI, and saves both current and wake-up time that would otherwise be wasted. A PC tool enables the user to configure the sensor controller and choose what peripherals are controlled and what conditions will wake up the main CPU.

The Sensor Controller Studio (SCS) is a stand-alone IDE used to develop and compile microcode for execution on the SCE. Refer to the SCS webpage (http://www.ti.com/tool/sensor-controller-studio) for more details on the SCS, including documentation embedded within the SCS IDE.

Startup Sequence

For a complete description of the CC13x0 reset sequence, see the *CC1310 Technical Reference Manual*.

10.1 Programming Internal Flash With the ROM Bootloader

The CC13x0 internal flash memory can be programmed using the bootloader located in device ROM. Both UART and SPI protocols are supported. See chapter 8 of the *CC1310 Technical Reference Manual* for more details on the programming protocol and requirements.

NOTE: Because the ROM bootloader uses predefined DIO pins for internal flash programming, allocate these pins in the board layout. The *CC1310 Technical Reference Manual* has more details on the pins allocated to the bootloader based on the chip package type.

10.2 Resets

Reset the device using only hard resets. From the software, this reset can be accomplished using:

```
HAL_SYSTEM_RESET();
```

In CCS, select Board Reset (automatic) from the reset menu (see Figure 10-1).

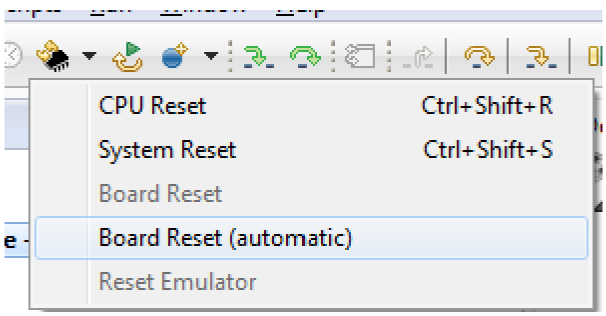


Figure 10-1. Board Reset

Development and Debugging

11.1 Debug Interfaces

The CC13x0 platform supports the cJTAG (2-wire) and JTAG (4-wire) interfaces. Any debuggers that support cJTAG, such as the TI XDS100v3, XDS110, and XDS200, will work natively. Others, such as the IAR I-Jet and Segger J-Link, can only be used in JTAG mode, but their drivers can inject a cJTAG sequence to enable JTAG mode when connecting.

The hardware resources included on the devices that can be used for debugging follow. Not all debugging functionality is available in all combinations of debugger and IDE.

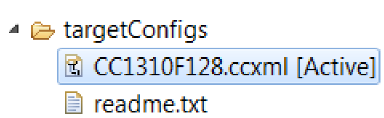
- Breakpoint unit (FBP) – Six instruction comparators, two literal comparators
- Data watchpoint unit (DWT) – Four watchpoints on memory access
- Instrumentation trace module (ITM) – 32 × 32-bit stimulus registers
- Trace port interface unit (TPIU) – Serialization and time stamping of DWT and ITM events

The LaunchPad board contains an XDS110 debug probe, which the debugger used by default in the sample projects.

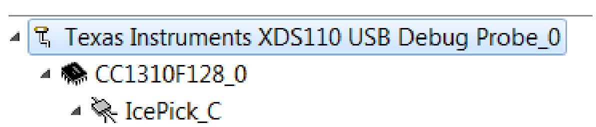
11.1.1 Connecting to the XDS Debugger

If only one debugger is attached, the IDE automatically uses this debugger. If multiple debuggers are connected, the individual debugger must be chosen manually. The following steps detail how to select a debugger in CCS.

1. Open the target configuration file and open the Advanced Setup pane.



2. Choose the top-level debugger entry.



3. Choose select by serial number, and then enter in the serial number.

To find the serial number for XDS100v3 debuggers, open a command prompt and run *C:\ti\ccsv6\ccs_base\common\uscif\xds100serial.exe* to get a list of connected debugger serial numbers.

11.1.2 Load Debug Symbols

The sensor_cc13x0lp output file can be flash downloaded to the target by clicking RUN → Debug (F11), as shown in Figure 11-1.

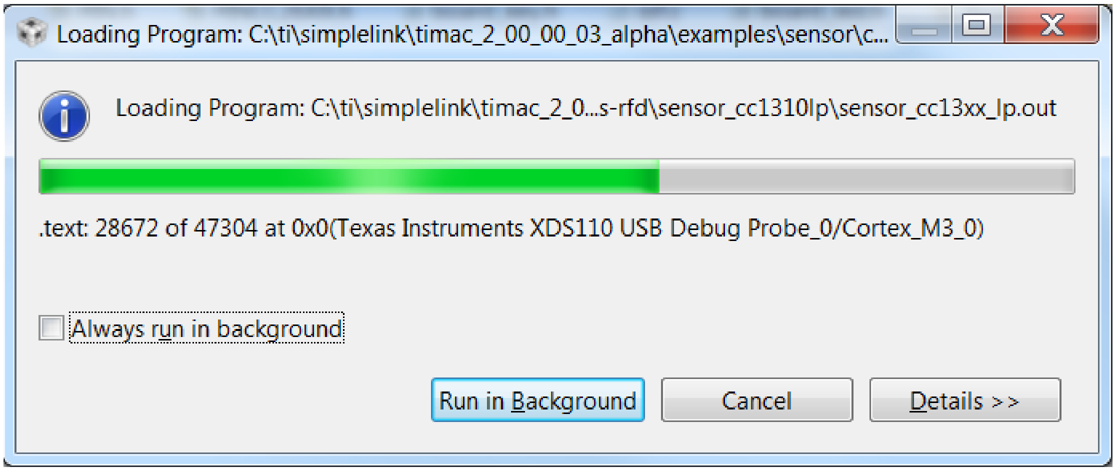


Figure 11-1. Debug Output File

11.2 Breakpoints

CCS reserves one of the instruction comparators, leaving five hardware breakpoints available for debugging. This section describes setting the breakpoints in CCS.

To toggle a breakpoint, either:

- Double-click the area to the left of the line number.
- Press Ctrl+Shift+B.
- Right-click on the line and select Breakpoint → Hardware Breakpoint.

For example, a breakpoint set on line 247 looks like Figure 11-2.

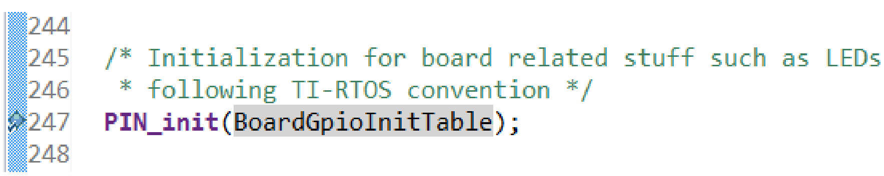


Figure 11-2. Breakpoint Set Example

To get an overview of the active and inactive breakpoints, click on View → Breakpoints, as shown in Figure 11-3.

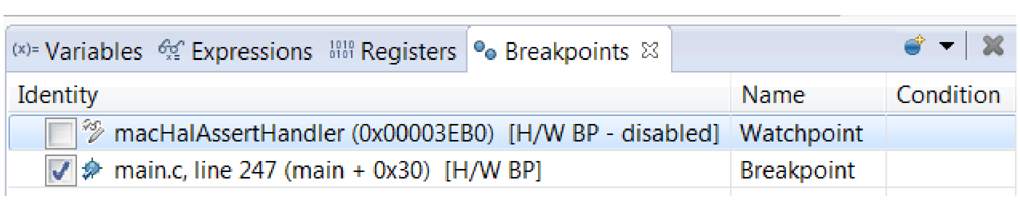


Figure 11-3. View Breakpoints

To set a conditional break, right-click the breakpoint in the overview, and choose Breakpoint Properties, as shown in Figure 11-4.

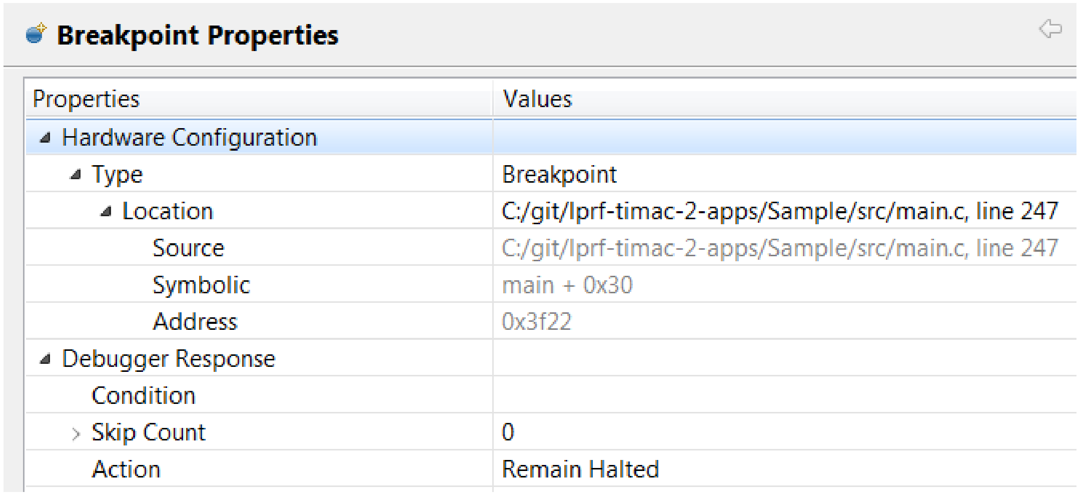


Figure 11-4. Breakpoint Properties

Skip Count and Condition can used when debugging to skip a number of breaks, or only break if a variable is a certain value.

NOTE: Conditional breaks require a debugger response and, although unlikely, may halt the processor long enough to break a beacon-enabled network, even if the condition is false or the skip count has not been reached.

11.2.1 Considerations When Using Breakpoints With Frequency Hopping or a Beacon-Enabled Network

As the frequency-hopping and IEEE802.15.4g protocols are timing sensitive, any breakpoints are likely to break the execution long enough that network timing is lost and the link breaks. Therefore, it is necessary to place breakpoints as close as possible to where the relevant debug information can be read or offending code can be stepped through. Consider experimenting on breakpoint placements by restarting debugging and repeating the conditions that lead to hitting the breakpoint.

11.2.2 Considerations on Breakpoints and Compiler Optimization

When the compiler is optimizing code, toggling a breakpoint on a line of C-code may not result in the expected behavior. Some examples include:

- Code is removed or not compiled in: toggling a breakpoint in the IDE results in a breakpoint on some other unintended place, not on the selected line. Some IDEs can disable breakpoints on nonexisting code.
- Code block is part of a common subexpression: toggling a breakpoint works, but code also breaks on an execution path other than the intended one.
- If-clause is represented by a conditional branch in assembly: a breakpoint inside an if-clause always breaks on the conditional statement, even if not executed.

Because of this, TI recommends selecting as low an optimization level as possible when debugging. See Section 11.7 for information on modifying optimization levels.

11.3 Watching Variables and Registers

CCS provides several ways to view the state of a halted program. Global variables are statically placed during link-time, and can end up anywhere in the RAM available to the project, or potentially in flash if they are declared as a constant value. These variables can be accessed at any time through the Watch and Expression windows. Unless removed due to optimization, global variables are always available in these views. Local variables, or variables that are valid only inside a limited scope, are placed on the active task stack. Such variables can be viewed with the Watch or Expression views, and can also be automatically displayed when breaking or stepping through code.

11.3.1 Variables in CCS

Global variables can be viewed by selecting View → Expressions as shown in Figure 11-5, or by selecting a variable name in code, right-clicking, and selecting Add Watch Expression.

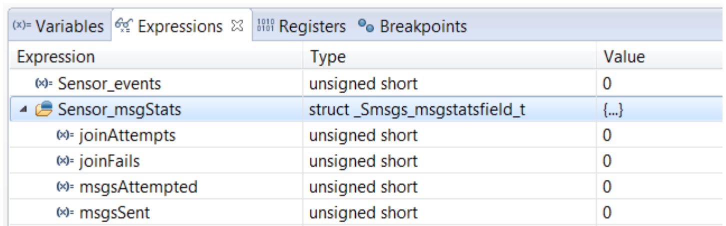


Figure 11-5. View Expressions

Local variables can be automatically viewed by selecting View → Variables, as shown in Figure 11-6.

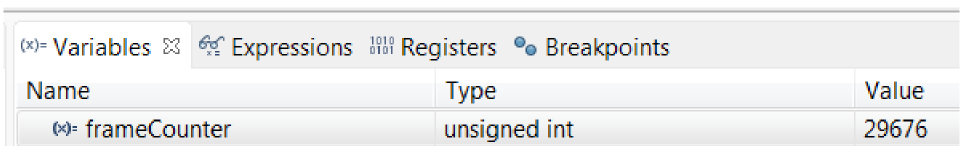


Figure 11-6. View Variables

11.3.2 Considerations When Viewing Variables

Local variables are often placed in CPU registers and not on the stack. Local variables also typically have a very limited lifetime, even within the scope in which they are valid, depending on the optimization performed. Therefore, CCS may struggle to show a particularly interesting variable. The solution when debugging is to:

- Move the variable to global scope, so that it is always accessible in RAM.
- Make the variable volatile, so that the compiler does not use a limited scope.
- Alternatively make a shadow copy of the variable that is both global and volatile.

11.4 Memory Watchpoints

As mentioned in Section 11.1, the DWT module contains four memory watchpoints which allow breakpoints on memory access. The hardware match functionality only examines the address, so if this is intended for use on a variable, the variable must be global.

NOTE: If using a data watchpoint with a value match, two of the four watchpoints are used.

11.4.1 Watchpoints in CCS

Right-click on a global variable and select Breakpoint → Hardware Watchpoint to add it to the breakpoint overview, as shown in Figure 11-7.

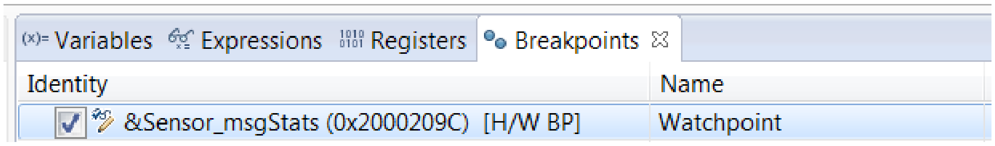


Figure 11-7. Hardware Watchpoint

Similar to code breakpoints, right-click and edit the Breakpoint Properties to configure the watchpoint, as shown in Figure 11-8.

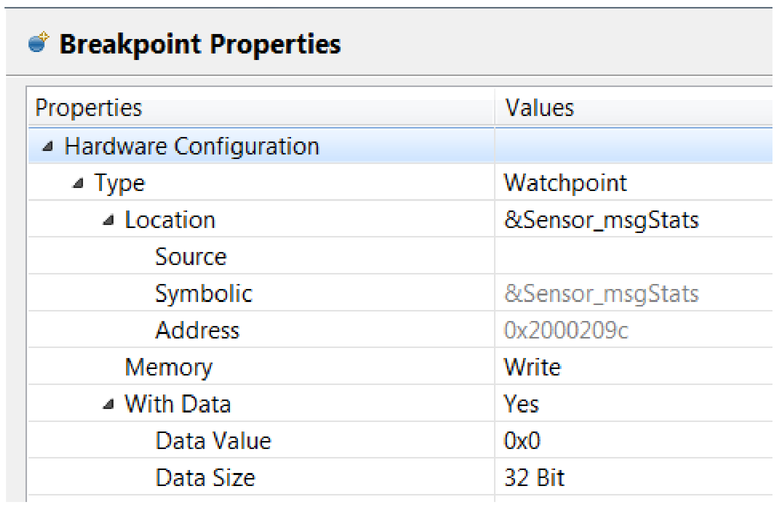


Figure 11-8. Breakpoint Properties

This example configuration ensures that if 0x0 is written to the memory location for Sensor_msgStats in the sensor_cc13x0lp example project, the device halts execution.

11.5 TI-RTOS Object Viewer

CCS includes the RTOS Object Viewer (ROV) plug-in, which provides insight into the current state of TI-RTOS, including task states, stacks, and so forth. CCS has a similar interface, so the following examples primarily discuss CCS. To access the ROV in CCS, click the Tools menu, then RTOS Object View. This section discusses some ROV views useful for debugging and profiling.

11.5.1 Scanning the BIOS for Errors

The BIOS → Scan for errors view (see Figure 11-9) sweeps through the available ROV modules and reports any errors found. This feature can be a good starting point if anything has gone wrong for unknown reasons. This scan only shows errors related to TI-RTOS modules, and only errors that it can detect.

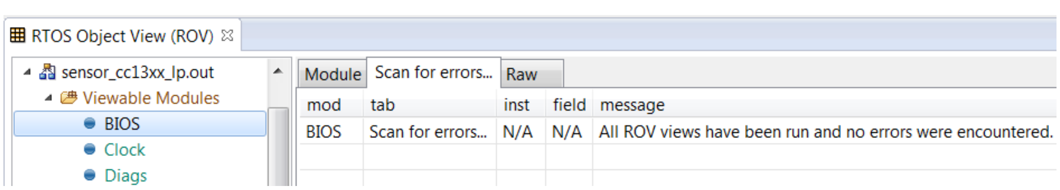
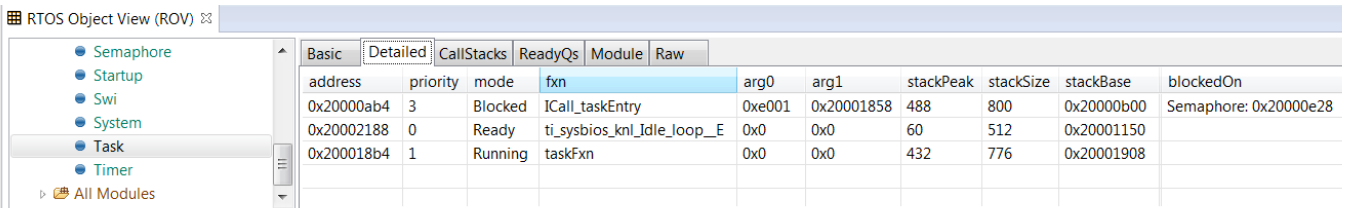


Figure 11-9. Scan for Errors

11.5.2 Viewing the State of Each Task

The Task → Detailed view is useful for viewing the state of each task and its related runtime stack usage. This example shows the state the first time the user-thread is called. The image shows the sensor application task, the idle task, and the stack task, represented by its ICall_taskEntry, as shown in Figure 11-10.



address	priority	mode	fxn	arg0	arg1	stackPeak	stackSize	stackBase	blockedOn
0x20000ab4	3	Blocked	ICall_taskEntry	0xe001	0x20001858	488	800	0x20000b00	Semaphore: 0x20000e28
0x20002188	0	Ready	ti_sysbios_knl_Idle_loop__E	0x0	0x0	60	512	0x20001150	
0x200018b4	1	Running	taskFxn	0x0	0x0	432	776	0x20001908	

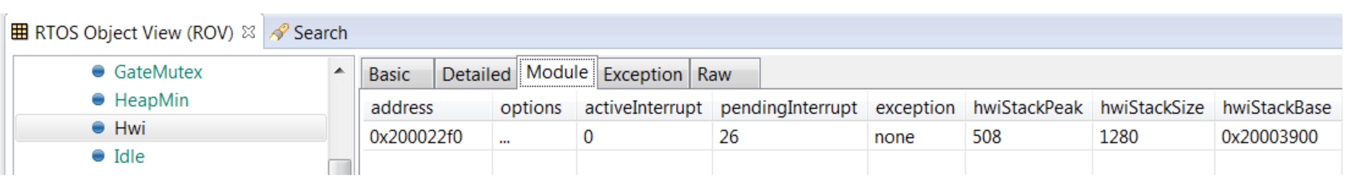
Figure 11-10. Detailed View

The columns are explained here (see Section 3.2 for more information on the various runtime stacks):

- address: Memory location of the Task_Struct instance for each task.
- priority: The TI-RTOS priority for the task
- mode: The current state of the task
- fxn: The name of the task entry function
- arg0, arg1: Arbitrary values that can be given to the entry function of the task. In the image, the ICall_taskEntry is given 0xe001, which is the flash location of the RF stack image entry function, and 0x20001858, which is the location of mscUserCfg_t user0Cfg, defined in main().
- stackPeak: Maximum runtime stack memory used based on watermark in RAM, where the stacks are prefilled with 0xBE and there is a sentinel word at the end of the runtime stack. Function calls may push the stack pointer out of the runtime stack, but not actually write to the entire area. Therefore, a stack peak near stackSize but not exceeding it may still indicate stack overflow.
- stackSize: The size of the runtime stack, configured when instantiating a task.
- stackBase: Logical top of the task runtime stack. Usage starts at stackBase and stackSize, and grows down to this address.
- blockedOn: Type and address of the synchronization object the thread is blocked on, if available. For semaphores, the addresses are listed under Semaphore → Basic.

11.5.3 Viewing the System Stack

The HWI → Module view (see Figure 11-11) allows profiling of the system stack used during boot, for main(), HWI execution, and SWI execution. See Section 3.11.1 for more information on the system stack.



address	options	activeInterrupt	pendingInterrupt	exception	hwiStackPeak	hwiStackSize	hwiStackBase
0x200022f0	...	0	26	none	508	1280	0x20003900

Figure 11-11. HWI Module View

The hwiStackPeak, hwiStackSize, and hwiStackBase can be used to check for system stack overflow.

11.5.4 Power Manager Information

See the *TI-RTOS Power Management Guide* for more information.

11.6 Profiling the ICall Heap Manager (heapmgr.h)

As described in Section 5.3.2, the ICall heap manager and its heap is used to allocate messages between the TI 15.4-MAC Stack task and the application task, as well as dynamic memory allocations in the various tasks.

Profiling functionality is provided for the ICall heap, but is not enabled by default. Therefore, it must be compiled in by adding HEAPMGR_METRICS to the defined preprocessor symbols. This functionality is useful both for finding potential sources for unexplained behavior, and to optimize the size of the heap. When HEAPMGR_METRICS is defined, the variables and functions that follow become available:

Global variables:

- HEAPMGR_BLKMAX: maximum amount of simultaneous allocated blocks
- HEAPMGR_BLKCNT: current amount of allocated blocks
- HEAPMGR_BLKFREE: current amount of free blocks
- HEAPMGR_MEMALO: current total memory allocated in bytes
- HEAPMGR_MEMMAX: maximum amount of simultaneous allocated memory in blocks

NOTE: This amount of memory must not exceed the size of the heap.

- HEAPMGR_MEMUB: the furthest memory location of an allocated block, measured as an offset from the start of the heap
- HEAPMGR_MEMFAIL: amount of memory allocation failure (instances where ICall_malloc() has returned NULL)

Functions:

- void HEAPMGR_GETMETRICS(hmU16_t *pBlkMax, hmU16_t *pBlkCnt, hmU16_t *pBlkFree, hmU16_t *pMemAlo, hmU16_t *pMemMax, hmU16_t *pMemUb)

  Returns the preceding variables in the pointers passed in as parameters
- int HEAPMGR_SANITY_CHECK(void)

  Returns 0 if the heap is ok, nonzero otherwise (such as when an array access has overwritten a header in the heap)

11.7 Optimizations

During debugging, it is sometimes useful to turn off or lower optimizations to ease single-stepping through code. This is possible at the following levels.

11.7.1 Project-Wide Optimizations

Figure 11-12 shows the menu view for project-wide optimizations. There may not be enough available flash to accomplish this.

In CCS: Project Properties → CCS Build → ARM Compiler → Optimization

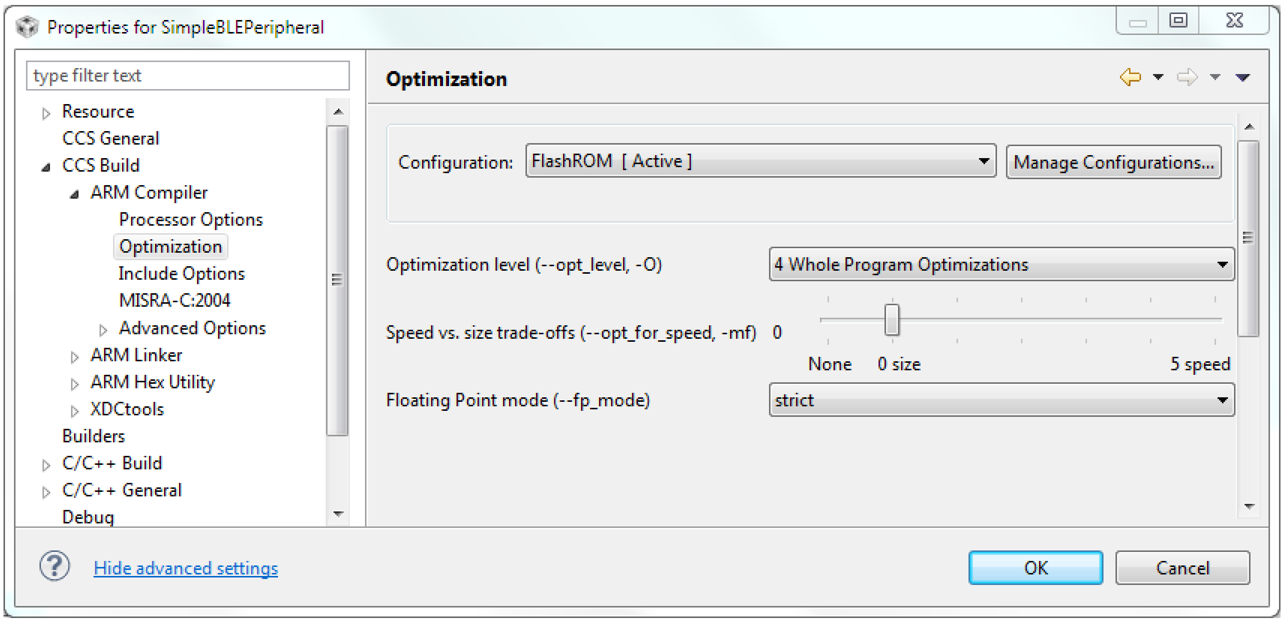


Figure 11-12. Project-Wide Optimization Menu

11.7.2 Single-File Optimizations

In CCS: Right-click on the file in the Workspace pane, and choose Properties. Change the file optimization level using the same menu, as shown in Figure 11-12.

11.8 Deciphering CPU Exceptions

There are several possible exception causes: if an exception is caught, an exception handler function can be called. Depending on the project settings, this may be a default handler in the ROM, which is just an infinite loop, or a custom function called from this default handler instead of a loop. When an exception occurs, depending on the debugger, it may be caught immediately and the execution halted in debug mode; or, if the exception is halted manually later through a debugger break, the execution is then stopped within the exception handler loop.

11.8.1 Exception Cause

With the default setup using TI-RTOS, the exception cause can be found in the system control space register group (CPU_SCS) in the configurable fault status register (CFSR). This register is described in detail in the *ARM Cortex-M3 Devices Generic User's Guide*. Most exceptions causes fall into three categories:

- Stack overflow or corruption leads to arbitrary code execution: almost any exception is possible
- A NULL pointer has been dereferenced and written to: typically IMPRECISERR exceptions
- A peripheral module (such as UART or Timer) is accessed without being powered: typically IMPRECISERR exceptions

The CFSR is available in View → Registers in CCS.

Normally when an access violation occurs, the exception type is IMPRECISERR because writes to flash and peripheral memory regions are mostly buffered writes.

Tips:

- If the CFSR:BFARVALID flag is set when the exception occurs, which is typical for PRECISERR, the BFAR register in CPU_SCS can be read out to determine which memory address caused the exception.
- If the exception is IMPRECISERR, PRECISERR can be forced by manually disabling buffered writes. Set [CPU_SCS:ACTRL:DISDEFWBUF] to 1, either by manually setting the bit in the register view in CCS, or by including <inc/hw_cpu_scs.h> from Driverlib and calling:

```
HWREG(CPU_SCS_BASE + CPU_SCS_O_ACTLR) = CPU_SCS_ACTLR_DISDEFWBUF;
```

This will negatively affect performance.

11.8.2 Using a Custom Exception Handler

A custom exception handler can be used instead of the default exception handler from ROM. In the sample projects, this is configured in app.cfg, through the M3Hwi.excHandlerFunc property, as shown in Figure 11-13.

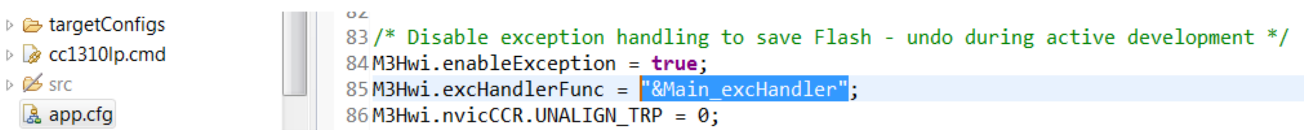


Figure 11-13. M3Hwi.excHandlerFunc Property

When this function is called, the Core-M3 has already pushed the core registers R0-3, R12, PC, LR, and xPSR on the active task run-time stack (when the exception was registered), and the TI-RTOS exception handler has pushed R4-11 onto the runtime stack.

11.8.3 Parsing the Exception Frame

The custom exception handler must be of the type:

```
xdc_Void Main_excHandler(UInt *excStack, UInt lr){..}
```

where lr is the LR value set by the Core-M3, and excStack points to the following structure which describes the CPU state (core registers) at the time the exception happened:

```
struct execptionFrame
{
  unsigned int _r4;
  unsigned int _r5;
  unsigned int _r6;
  unsigned int _r7;
  unsigned int _r8;
  unsigned int _r9;
  unsigned int _r10;
  unsigned int _r11;
  unsigned int _r0;
  unsigned int _r1;
  unsigned int _r2;
  unsigned int _r3;
  unsigned int _r12;
  unsigned int _lr;
  unsigned int _pc;
  unsigned int _xpsr;
};
```

Due to optimization, these variables are often not shown properly in the IDE watch windows. The Main_excHandler() implementation is shown here:

```
xdc_Void Main_excHandler(UInt *excStack, UInt lr)
{
    /* User defined function */
    Main_assertHandler(MAIN_ASSERT_HWI_TIRTOS);
}
```

11.9 Debugging HAL Assert

The HAL assert also calls the user-defined assert handler function:

```
void halAssertHandler(void)
{
    /* User defined function */
    Main_assertHandler(MAIN_ASSERT_ICALL);
}
```

This action is likely because the ICall_abort() function was called, which can be caused, among other things, by:

- Calling an ICALL function from a stack callback
- Misconfiguring additional ICALL tasks and entities
- Registering incorrect ICALL tasks

A breakpoint can be set in the ICall_abort function to locate the origin of this error.

11.10 Debugging MAC Assert

The MAC assert also calls the user-defined assert handler function:

```
void macHalAssertHandler(void)
{
    /* User defined function */
    Main_assertHandler(MAIN_ASSERT_MAC);
}
```

This action is likely because the MAC exception is generated in the TI 15.4-MAC Stack, which can be caused by:

- An internal MAC error
- A MAC function called with parameters out-of-range

11.11 Debugging Memory Problems

This section describes how to debug a situation in which the program runs out of memory, either on the heap or on the runtime stack for the individual thread contexts. Also, exceeding array bounds or dynamically allocating too little memory for a structure can corrupt the memory. If an exception such as INVPC, INVSTATE, or IBUSERR appears in the CFSR register, this error is a likely cause.

11.11.1 Task and System Stack Overflow

If there is an overflow in a task runtime stack or the system stack (found using the ROV plug-in as described in Section 11.5.2 and Section 11.5.3), perform the following steps:

1. Note the current size of each task runtime stack, and increase it by a 100 bytes as described in Section 3.2.1 and Section 3.11.1.
2. Check the stackPeaks using the ROV, as described in Section 11.5.2 and Section 11.5.3. If the peak is higher than the previous runtime stack size, the issue has been found.
3. If desired, reduce the runtime stack sizes so that they are still larger than their respective stackPeaks, to save memory.

11.11.2 Dynamic Allocation Errors

Using the ICALL heap profiling functionality described in Section 11.6, perform the following steps:

1. Check if memAlo or memMax approach the preprocessor-defined HEAPMGR_SIZE.
2. Check memFail to see if allocation failures have occurred.
3. Call the sanity check function.

If the heap is sane, but there are allocation errors, try to increase HEAPMGR_SIZE and see if the problem goes away. Alternatively, find the failing allocation by setting a breakpoint in heapmgr.h in HEAPMGR_MALLOC() on the line hdr = NULL;

11.12 Preprocessor Options

Preprocessor symbols are used to configure system behavior, features, and resource usage at compile time. Some symbols are required as part of the TI 15.4-MAC system, while others are configurable. See for details on accessing preprocessor symbols within the IDE. Symbols defined in a particular project are defined in all files within the project.

11.12.1 Modifying

To disable a symbol, put an x in front of the name. To enable a symbol, remove the x in front of the name. For example, to enable assert LEDs, change xASSERT_LEDS to ASSERT_LEDS.

11.12.2 Options

Table 11-1 lists the preprocessor symbols used by the application in the sensor_cc13x0lp project. Symbols that must never be modified are marked with an N in the Modify column, while modifiable or configurable symbols are marked with a Y.

Table 11-1. Application Preprocessor Symbols

Preprocessor Symbol	Description	Modify
TEMP_SENSOR	Required to enable temperature sensor on the LaunchPad board	Y
ASSERT_LEDS	Allows the LEDs to blink when Main_assertHandler() is called. The flag is turned off by default by placing an x in front of the name.	Y
CC1310_LAUNCHXL	This flag should be defined for the CC1310 LaunchPad board.	N
TI_DRIVERS_LCD_INCLUDED	Includes SmartRF06 LCD driver. This define is required to use the LCD on the CC2650EM 7x7 evaluation module. The SPI DMA driver is required to use the LCD driver.	Y
BOARD_DISPLAY_EXCLUDE_UART	Allows the LCD to display information, but excludes the UART from sending the same information. Disabling this flag uses more RAM.	Y
USE_ICALL	Required to use ICALL TI15.4 MAC and primitive services.	N
HEAPMGR_SIZE=0	Defines the size in bytes of the ICALL heap. Memory is allocated in .bss section. This is automatically generated and should not be modified.	N
FEATURE_MAC_SECURITY	Required for MAC security	N
FEATURE_GREEN_POWER	Required for Green Power feature.	N
FEATURE_BEACON_MODE	Required for IEEE802.15.4g beacon-enabled network	N
FEATURE_ENHANCED_ACK	Required for IEEE802.15.4g enhanced ACK support	N
ICALL_HOOK_ABORT_FUNC =halAssertHandler	Maps the ICall abort function to halAssertHandler()	Y
xdc_runtime_Assert_DISABLE_ALL	Disables XDC runtime assert	N
xdc_runtime_Log_DISABLE_ALL	Disables XDC runtime logging	N
MODULE_CC13XX_7X7	Required for CC1310 7 × 7 package	Y
NV_RESTORE	Allows the sensor application to restore user configuration from nonvolatile memory.	Y

Table 11-2 lists the only stack preprocessor options that may be modified.

Table 11-2. Stack Preprocessor Symbols

Preprocessor Symbol	Description	Modify
MAX_DEVICE_TABLE_ENTRIES=50	Defines the maximum number of secured devices	Y
NO_OSAL_SNV	Excludes OSAL Simple NV from the build	Y
RCN_APP_ASSERT	Allows the application to register an assert handler to be called from stack	Y
FEATURE_MAC_SECURITY	Required for MAC security	N
FEATURE_GREEN_POWER	Required for Green Power feature	N
FEATURE_BEACON_MODE	Required for IEEE802.15.4g beacon-enabled network	N
FEATURE_ENHANCED_BEACON	Required for IEEE802.15.4g enhanced beacon support	N
FEATURE_ENHANCED_ACK	Required for IEEE802.15.4g enhanced ACK support	N
DRIVERLIB_NOROM	Defines this flag in the project to use flash versions as default. See the CC13xx Driver Library document.	Y
USE_ICALL	Required to use ICall TI 15.4-MAC and primitive services	N
HAL_ASSERT_SPIN	Maps halAssertHandler() to a spinlock with interrupt disabled	Y
HALNODEBUG	Defines the HALNODEBUG to disable all assert functions. This flag works with RCN_APP_ASSERT, EXT_HAL_ASSERT, ICALL_HAL_ASSERT, and LEGACY_HAL_ASSERT flags.	Y
FEATURE_SYSTEM_STATS	Allows TI 15.4-MAC to collect statistics	Y
FH_DH1CF	Required for DH1CH channel hopping algorithm	N

11.13 Check System Flash and RAM Usage With a Map File

Both application and Stack projects produce a map file that can be used to compute the combined flash and RAM system memory usage. Both projects have their own memory space, therefore both map files must be analyzed to determine the total system memory usage. The map file is in the Output folder for each respective project. In CCS, the map file of the respective project gives a summary of flash and RAM usage. To determine the remaining available memory for each project, see Section 3.10 and Section 3.11.

NOTE: Due to section placement and alignment requirements, some remaining memory may not be available. The map file memory usage is valid only if the project builds and links successfully.

Creating Custom Applications

TI 15.4-MAC-based system designers must have a firm grasp on the general system architecture, application, and TI 15.4-MAC Stack framework to implement a custom application. This section provides indications on where and how to start writing a custom application, and to decide which role and purpose the custom application should have. If an application is to start the network and be the central node in the network, begin with the Collector Example Application. If the application is to join the network and be a node in the network that communicates with the central node, begin with the Sensor Example Application.

12.1 Adding a Board File

After selecting the reference application and preprocessor symbol, add a board file that matches the custom board layout. The following steps provide guidance on adding a custom board file to the project.

1. Create a custom board file (TI recommends using the Launchpad module board file CC1310_LAUNCHXL.c as a starting reference).
2. Modify the PIN structure.
3. Add peripheral driver initialization objects, according to the board design.
4. Include files from the folder of the start-up application.
5. Add the custom board file to the application project.
6. Update the C compiler search path of the IDE to point to the header file of the new board file.
7. Define an identifier for the new board file.
8. Rebuild the application project.

12.2 Configuring Parameters for Custom Hardware

1. Set the parameters, such as the sleep clock accuracy of the 32.768-kHz crystal.
2. Define the CCFG parameters.

For a description of the CCFG configuration parameters, see the *TI CC13xx Technical Reference Manual*.

12.3 Creating Additional Tasks

Many implementations can use the RTOS environment to operate in the application task framework. However, if the system design requires an additional RTOS task, see Section 3.2.1 for guidance on adding a task.

12.4 Configuring TI 15.4-MAC Stack

Configure the TI 15.4-MAC Stack with parameters and features. Section 4.1, Section 4.2, and Section 4.3 describe the operation and configuration parameters for the stack project for beacon-mode, nonbeacon mode, and the frequency-hopping configuration mode of the network, respectively.

TI 15.4-Stack API

13.1 TIMAC 2.0 API

The following is the application programming interface (API) for the Texas Instruments 802.15.4 MAC software. This API provides an interface to the management and data services of the 802.15.4 stack.

13.1.1 Callback Functions

These functions must be implemented by the application, and are used to pass events and data from the MAC to the application. Data accessed through callback function parameters (such as a pointer to data) are only valid for the execution of the function, and should not be considered valid when the function returns. These functions execute in the context of the MAC. The callback function implementation should avoid using critical sections and CPU-intensive operations. The callback table structure should be set up by the application, then ApiMac_registerCallbacks() should be called to register the table.

13.1.2 Common Constants and Structures

- Address type – The common address type used by the MAC is the ApiMac_sAddr_t.
- Status values – The common MAC status type is ApiMac_status_t.
- MAC security level – The security level (ApiMac_secLevel_t) defines the encryption or authentication methods used on the message frame.
- Key identifier modes – The key identifier mode (ApiMac_keyIdMode_t) defines how the key is determined from the key index.
- Security type – MAC security structure (ApiMac_sec_t).

13.1.3 Initialization and Task Interfaces

- ApiMac_init()
- ApiMac_registerCallbacks()
- ApiMac_processIncoming()

13.1.4 Data Interfaces

- ApiMac_mcpsDataReq()
- ApiMac_mcpsPurgeReq()

13.1.5 Management Interfaces

- ApiMac_mlmeAssociateReq()
- ApiMac_mlmeAssociateRsp()
- ApiMac_mlmeDisassociateReq()
- ApiMac_mlmeOrphanRsp()
- ApiMac_mlmePollReq()
- ApiMac_mlmeResetReq()
- ApiMac_mlmeScanReq()
- ApiMac_mlmeStartReq()
- ApiMac_mlmeSyncReq()
- ApiMac_mlmeWSAsyncReq()

13.1.6 Management Attribute Interfaces

The MAC attributes can be read and written to by using the following Get and Set functions, organized by the attributes data type:

- ApiMac_mlmeGetReqBool()
- ApiMac_mlmeGetReqUint8()
- ApiMac_mlmeGetReqUint16()
- ApiMac_mlmeGetReqUint32()
- ApiMac_mlmeGetReqArray()
- ApiMac_mlmeGetFhReqUint8()
- ApiMac_mlmeGetFhReqUint16()
- ApiMac_mlmeGetFhReqUint32()
- ApiMac_mlmeGetFhReqArray()
- ApiMac_mlmeGetSecurityReqUint8()
- ApiMac_mlmeGetSecurityReqUint16()
- ApiMac_mlmeGetSecurityReqArray()
- ApiMac_mlmeGetSecurityReqStruct()
- ApiMac_mlmeSetReqBool()
- ApiMac_mlmeSetReqUint8()
- ApiMac_mlmeSetReqUint16()
- ApiMac_mlmeSetReqUint32()
- ApiMac_mlmeSetReqArray()
- ApiMac_mlmeSetFhReqUint8()
- ApiMac_mlmeSetFhReqUint16()
- ApiMac_mlmeSetFhReqUint32()
- ApiMac_mlmeSetFhReqArray()
- ApiMac_mlmeSetSecurityReqUint8()
- ApiMac_mlmeSetSecurityReqUint16()
- ApiMac_mlmeSetSecurityReqArray()
- ApiMac_mlmeSetSecurityReqStruct()

13.1.7 Simplified Security Interfaces

- ApiMac_secAddDevice()
- ApiMac_secDeleteDevice()
- ApiMac_secDeleteKeyAndAssocDevices()
- ApiMac_secDeleteAllDevices()
- ApiMac_secGetDefaultSourceKey()
- ApiMac_secAddKeyInitFrameCounter()

13.1.8 Extension Interfaces

- ApiMac_randomByte()
- ApiMac_updatePanId()
- ApiMac_startFH()
- ApiMac_enableFH()
- ApiMac_parsePayloadGroupIEs()
- ApiMac_parsePayloadSubIEs()
- ApiMac_freeIEList()

- ApiMac_convertCapabilityInfo()
- ApiMac_buildMsgCapInfo()

13.2 File Documentation – api_mac.h File Reference

13.2.1 Data Structures

- struct ApiMac_sAddr_t
- struct ApiMac_sData_t
- struct ApiMac_MRFSKPHYDesc_t
- struct ApiMac_sec_t
- struct ApiMac_keyIdLookupDescriptor_t
- struct ApiMac_keyDeviceDescriptor_t
- struct ApiMac_keyUsageDescriptor_t
- struct ApiMac_keyDescriptor_t
- struct ApiMac_deviceDescriptor_t
- struct ApiMac_securityLevelDescriptor_t
- struct ApiMac_securityDeviceDescriptor_t
- struct ApiMac_securityKeyEntry_t
- struct ApiMac_securityPibKeyIdLookupEntry_t
- struct ApiMac_securityPibKeyDeviceEntry_t
- struct ApiMac_securityPibKeyUsageEntry_t
- struct ApiMac_securityPibKeyEntry_t
- struct ApiMac_securityPibDeviceEntry_t
- struct ApiMac_securityPibSecurityLevelEntry_t
- struct ApiMac_capabilityInfo_t
- struct ApiMac_txOptions_t
- struct ApiMac_mcpsDataReq_t
- struct ApiMac_payloadIeItem_t
- struct ApiMac_payloadIeRec_t
- struct ApiMac_mcpsDataInd_t
- struct ApiMac_mcpsDataCnf_t
- struct ApiMac_mcpsPurgeCnf_t
- struct ApiMac_panDesc_t
- struct ApiMac_mlmeAssociateReq_t
- struct ApiMac_mlmeAssociateRsp_t
- struct ApiMac_mlmeDisassociateReq_t
- struct ApiMac_mlmeOrphanRsp_t struct ApiMac_mlmePollReq_t
- struct ApiMac_mlmeScanReq_t
- struct ApiMac_mpmParams_t
- struct ApiMac_mlmeStartReq_t
- struct ApiMac_mlmeSyncReq_t
- struct ApiMac_mlmeWSAsyncReq_t
- struct ApiMac_secAddDevice_t
- struct ApiMac_secAddKeyInitFrameCounter_t
- struct ApiMac_mlmeAssociateInd_t
- struct ApiMac_mlmeAssociateCnf_t

- struct ApiMac_mlmeDisassociateInd_t
- struct ApiMac_mlmeDisassociateCnf_t
- struct ApiMac_beaconData_t
- struct ApiMac_coexist_t
- struct ApiMac_eBeaconData_t
- struct ApiMac_mlmeBeaconNotifyInd_t
- struct ApiMac_mlmeOrphanInd_t
- struct ApiMac_mlmeScanCnf_t
- struct ApiMac_mlmeStartCnf_t
- struct ApiMac_mlmeSyncLossInd_t
- struct ApiMac_mlmePollCnf_t
- struct ApiMac_mlmeCommStatusInd_t
- struct ApiMac_mlmePollInd_t
- struct ApiMac_mlmeWsAsyncCnf_t
- struct ApiMac_callbacks_t
- union ApiMac_sAddr_t.addr
- union ApiMac_mlmeBeaconNotifyInd_t.beaconData
- union ApiMac_mlmeScanCnf_t.result

13.2.2 Macros

- #define APIMAC_KEY_MAX_LEN 16
- #define APIMAC_SADDR_EXT_LEN 8
- #define APIMAC_MAX_KEY_TABLE_ENTRIES 2
- #define APIMAC_KEYID_IMPLICIT_LEN 0
- #define APIMAC_KEYID_MODE1_LEN 1
- #define APIMAC_KEYID_MODE4_LEN 5
- #define APIMAC_KEYID_MODE8_LEN 9
- #define APIMAC_KEY_SOURCE_MAX_LEN 8
- #define APIMAC_KEY_INDEX_LEN 1
- #define APIMAC_FRAME_COUNTER_LEN 4
- #define APIMAC_KEY_LOOKUP_SHORT_LEN 5
- #define APIMAC_KEY_LOOKUP_LONG_LEN 9
- #define APIMAC_MAX_KEY_LOOKUP_LEN APIMAC_KEY_LOOKUP_LONG_LEN
- #define APIMAC_DATA_OFFSET 24
- #define APIMAC_MAX_BEACON_PAYLOAD 16
- #define APIMAC_MIC_32_LEN 4
- #define APIMAC_MIC_64_LEN 8
- #define APIMAC_MIC_128_LEN 16
- #define APIMAC_MHR_LEN 37
- #define APIMAC_CHANNEL_PAGE_9 9
- #define APIMAC_CHANNEL_PAGE_10 10
- #define APIMAC_STANDARD_PHY_DESCRIPTOR_ENTRIES 3
- #define APIMAC_GENERIC_PHY_DESCRIPTOR_ENTRIES 3
- #define APIMAC_STD_US_915_PHY_1 1
- #define APIMAC_STD_US_915_PHY_2 2
- #define APIMAC_STD_ETSI_863_PHY_3 3

- #define APIMAC_MRFSK_GENERIC_PHY_ID_BEGIN 128
- #define APIMAC_MRFSK_GENERIC_PHY_ID_END 143
- #define APIMAC_MRFSK_STD_PHY_ID_BEGIN APIMAC_STD_US_915_PHY_1
- #define APIMAC_MRFSK_STD_PHY_ID_END APIMAC_STD_ETSI_863_PHY_3
- #define APIMAC_PHY_DESCRIPTOR 0x01
- #define APIMAC_ADDR_USE_EXT 0xFFFE
- #define APIMAC_SHORT_ADDR_BROADCAST 0xFFFF
- #define APIMAC_SHORT_ADDR_NONE 0xFFFF
- #define APIMAC_RANDOM_SEED_LEN 32
- #define APIMAC_FH_UTT_IE 0x00000002
- #define APIMAC_FH_BT_IE 0x00000008
- #define APIMAC_FH_US_IE 0x00010000
- #define APIMAC_FH_BS_IE 0x00020000
- #define APIMAC_FH_HEADER_IE_MASK 0x000000FF
- #define APIMAC_FH_PROTO_DISPATCH_NONE 0x00
- #define APIMAC_FH_PROTO_DISPATCH_MHD_PDU 0x01
- #define APIMAC_FH_PROTO_DISPATCH_6LOWPAN 0x02
- #define APIMAC_154G_MAX_NUM_CHANNEL 129
- #define APIMAC_154G_CHANNEL_BITMAP_SIZ ((APIMAC_154G_MAX_NUM_CHANNEL + 7) / 8)
- #define APIMAC_HEADER_IE_MAX 2
- #define APIMAC_PAYLOAD_IE_MAX 2
- #define APIMAC_PAYLOAD_SUB_IE_MAX 4
- #define APIMAC_SFS_BEACON_ORDER(s) ((s) & 0x0F)
- #define APIMAC_SFS_SUPERFRAME_ORDER(s) (((s) >> 4) & 0x0F)
- #define APIMAC_SFS_FINAL_CAP_SLOT(s) (((s) >> 8) & 0x0F)
- #define APIMAC_SFS_BLE(s) (((s) >> 12) & 0x01)
- #define APIMAC_SFS_PAN_COORDINATOR(s) (((s) >> 14) & 0x01)
- #define APIMAC_SFS_ASSOCIATION_PERMIT(s) (((s) >> 15) & 0x01)
- #define APIMAC_FH_MAX_BIT_MAP_SIZE 32
- #define APIMAC_FH_NET_NAME_SIZE_MAX 32
- #define APIMAC_FH_GTK_HASH_SIZE 8

13.2.3 Typedefs

- typedef uint8_t ApiMac_sAddrExt_t[APIMAC_SADDR_EXT_LEN]
- typedef ApiMac_mcpsDataInd_t ApiMac_mlmeWsAsyncInd_t
- typedef void(* ApiMac_associateIndFp_t) (ApiMac_mlmeAssociateInd_t *pAssocInd)
- typedef void(* ApiMac_associateCnfFp_t) (ApiMac_mlmeAssociateCnf_t *pAssocCnf)
- typedef void(* ApiMac_disassociateIndFp_t) (ApiMac_mlmeDisassociateInd_t *pDisassociateInd)
- typedef void(* ApiMac_disassociateCnfFp_t) (ApiMac_mlmeDisassociateCnf_t *pDisassociateCnf)
- typedef void(* ApiMac_beaconNotifyIndFp_t) (ApiMac_mlmeBeaconNotifyInd_t *pBeaconNotifyInd)
- typedef void(* ApiMac_orphanIndFp_t) (ApiMac_mlmeOrphanInd_t *pOrphanInd)
- typedef void(* ApiMac_scanCnfFp_t) (ApiMac_mlmeScanCnf_t *pScanCnf)
- typedef void(* ApiMac_startCnfFp_t) (ApiMac_mlmeStartCnf_t *pStartCnf)
- typedef void(* ApiMac_syncLossIndFp_t) (ApiMac_mlmeSyncLossInd_t *pSyncLossInd)
- typedef void(* ApiMac_pollCnfFp_t) (ApiMac_mlmePollCnf_t *pPollCnf)
- typedef void(* ApiMac_commStatusIndFp_t) (ApiMac_mlmeCommStatusInd_t *pCommStatus)

- typedef void(* ApiMac_pollIndFp_t) (ApiMac_mlmePollInd_t *pPollInd)
- typedef void(* ApiMac_dataCnfFp_t) (ApiMac_mcpsDataCnf_t *pDataCnf)
- typedef void(* ApiMac_dataIndFp_t) (ApiMac_mcpsDataInd_t *pDataInd)
- typedef void(* ApiMac_purgeCnfFp_t) (ApiMac_mcpsPurgeCnf_t *pPurgeCnf)
- typedef void(* ApiMac_wsAsyncIndFp_t) (ApiMac_mlmeWsAsyncInd_t *pWsAsyncInd)
- typedef void(* ApiMac_wsAsyncCnfFp_t) (ApiMac_mlmeWsAsyncCnf_t *pWsAsyncCnf)
- typedef void(* ApiMac_unprocessedFp_t) (uint16_t param1, uint16_t param2, void *pMsg)

13.2.4 Enumerations

- enum ApiMac_assocStatus_t { ApiMac_assocStatus_success = 0, ApiMac_assocStatus_panAtCapacity = 1, ApiMac_assocStatus_panAccessDenied = 2 }
- enum ApiMac_addrType_t { ApiMac_addrType_none = 0, ApiMac_addrType_short = 2, ApiMac_addrType_extended = 3 }
- enum ApiMac_beaconType_t { ApiMac_beaconType_normal = 0, ApiMac_beaconType_enhanced = 1 }
- enum ApiMac_disassocateReason_t { ApiMac_disassocateReason_coord = 1, ApiMac_disassocateReason_device = 2 }
- enum ApiMac_commStatusReason_t { ApiMac_commStatusReason_assocRsp = 0, ApiMac_commStatusReason_orphanRsp = 1, ApiMac_commStatusReason_rxSecure = 2 }
- enum ApiMac_status_t { ApiMac_status_success = 0, ApiMac_status_subSystemError = 0x25, ApiMac_status_commandIDError = 0x26, ApiMac_status_lengthError = 0x27, ApiMac_status_unsupportedType = 0x28, ApiMac_status_autoAckPendingAllOn = 0xFE, ApiMac_status_autoAckPendingAllOff = 0xFF, ApiMac_status_beaconLoss = 0xE0, ApiMac_status_channelAccessFailure = 0xE1, ApiMac_status_counterError = 0xDB, ApiMac_status_denied = 0xE2, ApiMac_status_disabledTrxFailure = 0xE3, ApiMac_status_frameTooLong = 0xE5, ApiMac_status_improperKeyType = 0xDC, ApiMac_status_improperSecurityLevel = 0xDD, ApiMac_status_invalidAddress = 0xF5, ApiMac_status_invalidGts = 0xE6, ApiMac_status_invalidHandle = 0xE7, ApiMac_status_invalidIndex = 0xF9, ApiMac_status_invalidParameter = 0xE8, ApiMac_status_limitReached = 0xFA, ApiMac_status_noAck = 0xE9, ApiMac_status_noBeacon = 0xEA, ApiMac_status_noData = 0xEB, ApiMac_status_noShortAddress = 0xEC, ApiMac_status_onTimeTooLong = 0xF6, ApiMac_status_outOfCap = 0xED, ApiMac_status_panIdConflict = 0xEE, ApiMac_status_pastTime = 0xF7, ApiMac_status_readOnly = 0xFB, ApiMac_status_realignment = 0xEF, ApiMac_status_scanInProgress = 0xFC, ApiMac_status_securityError = 0xE4, ApiMac_status_superframeOverlap = 0xFD, ApiMac_status_trackingOff = 0xF8, ApiMac_status_transactionExpired = 0xF0, ApiMac_status_transactionOverflow = 0xF1, ApiMac_status_txActive = 0xF2, ApiMac_status_unavailableKey = 0xF3, ApiMac_status_unsupportedAttribute = 0xF4, ApiMac_status_unsupportedLegacy = 0xDE, ApiMac_status_unsupportedSecurity = 0xDF, ApiMac_status_unsupported = 0x18, ApiMac_status_badState = 0x19, ApiMac_status_noResources = 0x1A, ApiMac_status_ackPending = 0x1B, ApiMac_status_noTime = 0x1C, ApiMac_status_txAborted = 0x1D, ApiMac_status_duplicateEntry = 0x1E, ApiMac_status_fhError = 0x61, ApiMac_status_fhIeNotSupported = 0x62, ApiMac_status_fhNotInAsync = 0x63, ApiMac_status_fhNotInNeighborTable = 0x64, ApiMac_status_fhOutSlot = 0x65, ApiMac_status_fhInvalidAddress = 0x66, ApiMac_status_fhIeFormatInvalid = 0x67, ApiMac_status_fhPibNotSupported = 0x68, ApiMac_status_fhPibReadOnly = 0x69, ApiMac_status_fhPibInvalidParameter = 0x6A, ApiMac_status_fhInvalidFrameType = 0x6B, ApiMac_status_fhExpiredNode = 0x6C }
- enum ApiMac_secLevel_t { ApiMac_secLevel_none = 0, ApiMac_secLevel_mic32 = 1, ApiMac_secLevel_mic64 = 2, ApiMac_secLevel_mic128 = 3, ApiMac_secLevel_enc = 4, ApiMac_secLevel_encMic32 = 5, ApiMac_secLevel_encMic64 = 6, ApiMac_secLevel_encMic128 = 7 }
- enum ApiMac_keyIdMode_t { ApiMac_keyIdMode_implicit = 0, ApiMac_keyIdMode_1 = 1, ApiMac_keyIdMode_4 = 2, ApiMac_keyIdMode_8 = 3 }
- enum ApiMac_attribute_bool_t { ApiMac_attribute_associatePermit = 0x41, ApiMac_attribute_autoRequest = 0x42, ApiMac_attribute_battLifeExt = 0x43,

ApiMac_attribute_gtsPermit = 0x4D, ApiMac_attribute_promiscuousMode = 0x51, ApiMac_attribute_RxOnWhenIdle = 0x52, ApiMac_attribute_associatedPanCoord = 0x56, ApiMac_attribute_timestampSupported = 0x5C, ApiMac_attribute_securityEnabled = 0x5D, ApiMac_attribute_includeMPMIE = 0x62, ApiMac_attribute_fcsType = 0xE9 }

- enum ApiMac_attribute_uint8_t { ApiMac_attribute_ackWaitDuration = 0x40, ApiMac_attribute_battLifeExtPeriods = 0x44, ApiMac_attribute_beaconPayloadLength = 0x46, ApiMac_attribute_beaconOrder = 0x47, ApiMac_attribute_bsn = 0x49, ApiMac_attribute_dsn = 0x4C, ApiMac_attribute_maxCsmaBackoffs = 0x4E, ApiMac_attribute_backoffExponent = 0x4F, ApiMac_attribute_superframeOrder = 0x54, ApiMac_attribute_maxBackoffExponent = 0x57, ApiMac_attribute_maxFrameRetries = 0x59, ApiMac_attribute_responseWaitTime = 0x5A, ApiMac_attribute_syncSymbolOffset = 0x5B, ApiMac_attribute_eBeaconSequenceNumber = 0x5E, ApiMac_attribute_eBeaconOrder = 0x5F, ApiMac_attribute_offsetTimeslot = 0x61, ApiMac_attribute_phyTransmitPowerSigned = 0xE0, ApiMac_attribute_logicalChannel = 0xE1, ApiMac_attribute_altBackoffExponent = 0xE3, ApiMac_attribute_deviceBeaconOrder = 0xE4, ApiMac_attribute_rf4cePowerSavings = 0xE5, ApiMac_attribute_frameVersionSupport = 0xE6, ApiMac_attribute_channelPage = 0xE7, ApiMac_attribute_phyCurrentDescriptorId = 0xE8 }
- enum ApiMac_attribute_uint16_t { ApiMac_attribute_coordShortAddress = 0x4B, ApiMac_attribute_panId = 0x50, ApiMac_attribute_shortAddress = 0x53, ApiMac_attribute_transactionPersistenceTime = 0x55, ApiMac_attribute_maxFrameTotalWaitTime = 0x58, ApiMac_attribute_eBeaconOrderNBPAN = 0x60 }
- enum ApiMac_attribute_uint32_t { ApiMac_attribute_beaconTxTime = 0x48, ApiMac_attribute_diagRxCrcPass = 0xEA, ApiMac_attribute_diagRxCrcFail = 0xEB, ApiMac_attribute_diagRxBroadcast = 0xEC, ApiMac_attribute_diagTxBroadcast = 0xED, ApiMac_attribute_diagRxUnicast = 0xEE, ApiMac_attribute_diagTxUnicast = 0xEF, ApiMac_attribute_diagTxUnicastRetry = 0xF0, ApiMac_attribute_diagTxUnicastFail = 0xF1, ApiMac_attribute_diagRxSecureFail = 0xF2, ApiMac_attribute_diagTxSecureFail = 0xF3 }
- enum ApiMac_attribute_array_t { ApiMac_attribute_beaconPayload = 0x45, ApiMac_attribute_coordExtendedAddress = 0x4A, ApiMac_attribute_extendedAddress = 0xE2 }
- enum ApiMac_securityAttribute_uint8_t { ApiMac_securityAttribute_keyTableEntries = 0x81, ApiMac_securityAttribute_deviceTableEntries = 0x82, ApiMac_securityAttribute_securityLevelTableEntries = 0x83, ApiMac_securityAttribute_autoRequestSecurityLevel = 0x85, ApiMac_securityAttribute_autoRequestKeyIdMode = 0x86, ApiMac_securityAttribute_autoRequestKeyIndex = 0x88 }
- enum ApiMac_securityAttribute_uint16_t { ApiMac_securityAttribute_panCoordShortAddress = 0x8B }
- enum ApiMac_securityAttribute_array_t { ApiMac_securityAttribute_autoRequestKeySource = 0x87, ApiMac_securityAttribute_defaultKeySource = 0x89, ApiMac_securityAttribute_panCoordExtendedAddress = 0x8A }
- enum ApiMac_securityAttribute_struct_t { ApiMac_securityAttribute_keyTable = 0x71, ApiMac_securityAttribute_keyIdLookupEntry = 0xD0, ApiMac_securityAttribute_keyDeviceEntry = 0xD1, ApiMac_securityAttribute_keyUsageEntry = 0xD2, ApiMac_securityAttribute_keyEntry = 0xD3, ApiMac_securityAttribute_deviceEntry = 0xD4, ApiMac_securityAttribute_securityLevelEntry = 0xD5 }
- enum ApiMac_FHAttribute_uint8_t { ApiMac_FHAttribute_unicastDwellInterval = 0x2004, ApiMac_FHAttribute_broadcastDwellInterval = 0x2005, ApiMac_FHAttribute_clockDrift = 0x2006, ApiMac_FHAttribute_timingAccuracy = 0x2007, ApiMac_FHAttribute_unicastChannelFunction = 0x2008, ApiMac_FHAttribute_broadcastChannelFunction = 0x2009, ApiMac_FHAttribute_useParentBSIE = 0x200A, ApiMac_FHAttribute_routingCost = 0x200F, ApiMac_FHAttribute_routingMethod = 0x2010, ApiMac_FHAttribute_eapolReady = 0x2011, ApiMac_FHAttribute_fanTPSVersion = 0x2012, ApiMac_FHAttribute_numNonSleepDevice = 0x201b, ApiMac_FHAttribute_numSleepDevice = 0x201c }
- enum ApiMac_FHAttribute_uint16_t { ApiMac_FHAttribute_broadcastSchedId = 0x200B, ApiMac_FHAttribute_unicastFixedChannel = 0x200C, ApiMac_FHAttribute_broadcastFixedChannel = 0x200D, ApiMac_FHAttribute_panSize = 0x200E, ApiMac_FHAttribute_panVersion = 0x2014, ApiMac_FHAttribute_neighborValidTime = 0x2019 }
- enum ApiMac_FHAttribute_uint32_t { ApiMac_FHAttribute_BCInterval = 0x2001 }

- enum ApiMac_FHAttribute_array_t { ApiMac_FHAttribute_trackParentEUI = 0x2000, ApiMac_FHAttribute_unicastExcludedChannels = 0x2002, ApiMac_FHAttribute_broadcastExcludedChannels = 0x2003, ApiMac_FHAttribute_netName = 0x2013, ApiMac_FHAttribute_gtk0Hash = 0x2015, ApiMac_FHAttribute_gtk1Hash = 0x2016, ApiMac_FHAttribute_gtk2Hash = 0x2017, ApiMac_FHAttribute_gtk3Hash = 0x2018 }
- enum ApiMac_fhFrameType_t { ApiMac_fhFrameType_panAdvert = 0x00, ApiMac_fhFrameType_panAdvertSolicit = 0x01, ApiMac_fhFrameType_config = 0x02, ApiMac_fhFrameType_configSolicit = 0x03, ApiMac_fhFrameType_data = 0x04, ApiMac_fhFrameType_ack = 0x05, ApiMac_fhFrameType_eapol = 0x06, ApiMac_fhFrameType_invalid = 0xFF }
- enum ApiMac_payloadIEGroup_t { ApiMac_payloadIEGroup_ESDU = 0x00, ApiMac_payloadIEGroup_MLME = 0x01, ApiMac_payloadIEGroup_WiSUN = 0x04, ApiMac_payloadIEGroup_term = 0x0F }
- enum ApiMac_MLMESubIE_t { ApiMac_MLMESubIE_coexist = 0x21, ApiMac_MLMESubIE_sunDevCap = 0x22, ApiMac_MLMESubIE_sunFSKGenPhy = 0x23 } • enum ApiMac_wisunSubIE_t { ApiMac_wisunSubIE_USIE = 1, ApiMac_wisunSubIE_BSIE = 2, ApiMac_wisunSubIE_PANIE = 4, ApiMac_wisunSubIE_netNameIE = 5, ApiMac_wisunSubIE_PANVersionIE = 6, ApiMac_wisunSubIE_GTKHashIE = 7 }
- enum ApiMac_scantype_t { ApiMac_scantype_energyDetect = 0, ApiMac_scantype_active = 1, ApiMac_scantype_passive = 2, ApiMac_scantype_orphan = 3, ApiMac_scantype_activeEnhanced = 5 }
- enum ApiMac_wisunAsycnOperation_t { ApiMac_wisunAsycnOperation_start = 0, ApiMac_wisunAsycnOperation_stop = 1 } • enum ApiMac_wisunAsyncFrame_t { ApiMac_wisunAsyncFrame_advertisement = 0, ApiMac_wisunAsyncFrame_advertisementSolicit = 1, ApiMac_wisunAsyncFrame_config = 2, ApiMac_wisunAsyncFrame_configSolicit = 3 }
- enum ApiMac_fhDispatchType_t { ApiMac_fhDispatchType_none = 0, ApiMac_fhDispatchType_MHD_PDU = 1, ApiMac_fhDispatchType_6LowPAN = 2 }

13.2.5 Functions

- void * ApiMac_init (bool enableFH)

  Initialize this module.
- void ApiMac_registerCallbacks (ApiMac_callbacks_t *pCallbacks)

  Register for MAC callbacks.
- void ApiMac_processIncoming (void)

  Process incoming messages from the MAC stack.
- ApiMac_status_t ApiMac_mcpsDataReq (ApiMac_mcpsDataReq_t *pData)

  This function sends application data to the MAC for transmission in a MAC data frame. The MAC can only buffer a certain number of data request frames. When the MAC is congested and cannot accept the data request, it initiates a callback (ApiMac_dataCnfFp_t) with an overflow status (ApiMac_status_transactionOverflow) . Eventually the MAC will become uncongested and initiate the callback (ApiMac_dataCnfFp_t) for a buffered request. At this point, the application can attempt another data request. Using this scheme, the application can send data at any time, but it must queue data to be resent if it receives an overflow status.
- ApiMac_status_t ApiMac_mcpsPurgeReq (uint8_t msduHandle)

  This function purges and discards a data request from the MAC data queue. When the operation is complete, the MAC sends a MCPS Purge Confirm to initiate a callback (ApiMac_purgeCnfFp_t).
- ApiMac_status_t ApiMac_mlmeAssociateReq (ApiMac_mlmeAssociateReq_t *pData)

  This function sends an associate request to a coordinator device. The application tries to associate only with a PAN that is currently allowing association, as indicated in the results of the scanning procedure. In a beacon-enabled PAN, the beacon order must be set by using ApiMac_mlmeSetReq() before making the call to ApiMac_mlmeAssociateReq(). When the associate request is complete, the application receives the ApiMac_associateCnfFp_t callback.

- ApiMac_status_t ApiMac_mlmeAssociateRsp (ApiMac_mlmeAssociateRsp_t *pData)
  This function sends an associate response to a device requesting to associate. This function must be called after the ApiMac_associateIndFp_t callback. When the associate response is complete, the callback ApiMac_commStatusIndFp_t is called to indicate the success or failure of the operation.
- ApiMac_status_t ApiMac_mlmeDisassociateReq (ApiMac_mlmeDisassociateReq_t *pData)
  This function is used by an associated device to notify the coordinator of its intent to leave the PAN. This function is also used by the coordinator to instruct an associated device to leave the PAN. When the disassociate procedure is complete, the application callback ApiMac_disassociateCnfFp_t is called.
- ApiMac_status_t ApiMac_mlmeGetReqBool (ApiMac_attribute_bool_t pibAttribute, bool *pValue)
  This direct execute function retrieves an attribute value from the MAC PIB.
- ApiMac_status_t ApiMac_mlmeGetReqUint8 (ApiMac_attribute_uint8_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC PIB.
- ApiMac_status_t ApiMac_mlmeGetReqUint16 (ApiMac_attribute_uint16_t pibAttribute, uint16_t *pValue)
  This direct execute function retrieves an attribute value from the MAC PIB.
- ApiMac_status_t ApiMac_mlmeGetReqUint32 (ApiMac_attribute_uint32_t pibAttribute, uint32_t *pValue)
  This direct execute function retrieves an attribute value from the MAC PIB.
- ApiMac_status_t ApiMac_mlmeGetReqArray (ApiMac_attribute_array_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC PIB.
- ApiMac_status_t ApiMac_mlmeGetFhReqUint8 (ApiMac_FHAttribute_uint8_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeGetFhReqUint16 (ApiMac_FHAttribute_uint16_t pibAttribute, uint16_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeGetFhReqUint32 (ApiMac_FHAttribute_uint32_t pibAttribute, uint32_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeGetFhReqArray (ApiMac_FHAttribute_array_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeGetSecurityReqUint8 (ApiMac_securityAttribute_uint8_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Security PIB.
- ApiMac_status_t ApiMac_mlmeGetSecurityReqUint16 (ApiMac_securityAttribute_uint16_t pibAttribute, uint16_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Security PIB.
- ApiMac_status_t ApiMac_mlmeGetSecurityReqArray (ApiMac_securityAttribute_array_t pibAttribute, uint8_t *pValue)
  This direct execute function retrieves an attribute value from the MAC Security PIB.
- ApiMac_status_t ApiMac_mlmeGetSecurityReqStruct (ApiMac_securityAttribute_struct_t pibAttribute, void *pValue)
  This direct execute function retrieves an attribute value from the MAC Security PIB.
- ApiMac_status_t ApiMac_mlmeOrphanRsp (ApiMac_mlmeOrphanRsp_t *pData)
  This function is called in response to an orphan notification from a peer device. This function must be called after receiving an Orphan Indication Callback. When the orphan response is complete, the Comm Status Indication Callback is called to indicate the success or failure of the operation.

- ApiMac_status_t ApiMac_mlmePollReq (ApiMac_mlmePollReq_t *pData)

  This function is used to request pending data from the coordinator. When the poll request is complete, the Poll Confirm Callback is called. If a data frame of nonzero length is received from the coordinator, the Poll Confirm Callback has a status ApiMac_status_success, and calls the Data Indication Callback for the received data.
- ApiMac_status_t ApiMac_mlmeResetReq (bool setDefaultPib)

  This direct execute function resets the MAC. This function must be called once at system startup before any other function in the management API is called.
- ApiMac_status_t ApiMac_mlmeScanReq (ApiMac_mlmeScanReq_t *pData)

  *This function initiates an energy detect, active, passive, or orphan scan on one or more channels. An energy detect scan measures the peak energy on each requested channel. An active scan sends a beacon request on each channel, then listens for beacons. A passive scan is a receive-only operation that listens for beacons on each channel. An orphan scan is used to locate the coordinator with which the scanning device had previously associated. When a scan operation is complete, the Scan Confirm callback is called. For active or passive scans, the application sets the maxResults parameter the maximum number of PAN descriptors to return. If maxResults is greater than zero, the application must also set result.panDescriptor to point to a buffer of size maxResults * sizeof(ApiMac_panDesc_t) to store the results of the scan. The application must not access or deallocate this buffer until the Scan Confirm Callback is called. The MAC stores up to maxResults PAN descriptors, and ignores duplicate beacons. An alternative way to get results for an active or passive scan is to set maxResults to zero, or set PIB attribute ApiMac_attribute_autoRequest to FALSE. Then the MAC will not store results, but rather call the Beacon Notify Indication Callback for each beacon received. The application does not need to supply any memory to store the scan results, but the MAC does not filter out duplicate beacons. For energy detect scans, the application must set result.energyDetect to point to a buffer of size 18 bytes to store the results of the scan. The application must not access or deallocate this buffer until the Scan Confirm Callback is called. An energy detect, active, or passive scan may be performed at any time if a scan is not already in progress. However, a device cannot perform any other MAC management operation or send or receive MAC data until the scan is complete.*
- ApiMac_status_t ApiMac_mlmeSetReqBool (ApiMac_attribute_bool_t pibAttribute, bool value)

  This direct execute function sets an attribute value in the MAC PIB.
- ApiMac_status_t ApiMac_mlmeSetReqUint8 (ApiMac_attribute_uint8_t pibAttribute, uint8_t value)

  This direct execute function sets an attribute value in the MAC PIB.
- ApiMac_status_t ApiMac_mlmeSetReqUint16 (ApiMac_attribute_uint16_t pibAttribute, uint16_t value)

  This direct execute function sets an attribute value in the MAC PIB.
- ApiMac_status_t ApiMac_mlmeSetReqUint32 (ApiMac_attribute_uint32_t pibAttribute, uint32_t value)

  This direct execute function sets an attribute value in the MAC PIB.
- ApiMac_status_t ApiMac_mlmeSetReqArray (ApiMac_attribute_array_t pibAttribute, uint8_t *pValue)

  This direct execute function sets an attribute value in the MAC PIB.
- ApiMac_status_t ApiMac_mlmeSetFhReqUint8 (ApiMac_FHAttribute_uint8_t pibAttribute, uint8_t value)

  This direct execute function sets an attribute value in the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeSetFhReqUint16 (ApiMac_FHAttribute_uint16_t pibAttribute, uint16_t value)

  This direct execute function sets an attribute value in the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeSetFhReqUint32 (ApiMac_FHAttribute_uint32_t pibAttribute, uint32_t value)

  This direct execute function sets an attribute value in the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeSetFhReqArray (ApiMac_FHAttribute_array_t pibAttribute, uint8_t *pValue)

  This direct execute function sets an attribute value in the MAC Frequency Hopping PIB.
- ApiMac_status_t ApiMac_mlmeSetSecurityReqUint8 (ApiMac_securityAttribute_uint8_t pibAttribute, uint8_t value)

This direct execute function sets an attribute value in the MAC Security PIB.

- ApiMac_status_t ApiMac_mlmeSetSecurityReqUint16 (ApiMac_securityAttribute_uint16_t pibAttribute, uint16_t value)

  This direct execute function sets an attribute value in the MAC Security PIB.

- ApiMac_status_t ApiMac_mlmeSetSecurityReqArray (ApiMac_securityAttribute_array_t pibAttribute, uint8_t *pValue)

  This direct execute function sets an attribute value in the MAC Security PIB.

- ApiMac_status_t ApiMac_mlmeSetSecurityReqStruct (ApiMac_securityAttribute_struct_t pibAttribute, void *pValue)

  This direct execute function sets an attribute value in the MAC Security PIB.

- ApiMac_status_t ApiMac_mlmeStartReq (ApiMac_mlmeStartReq_t *pData)

  This function is called by a coordinator or PAN coordinator to start or reconfigure a network. Before starting a network, the device must have set its short address. A PAN coordinator sets the short address by setting the attribute ApiMac_attribute_shortAddress. A coordinator sets the short address through association. When parameter panCoordinator is TRUE, the MAC automatically sets attributes ApiMac_attribute_panID and ApiMac_attribute_logicalChannel to the panId and logicalChannel parameters. If panCoordinator is FALSE, these parameters are ignored (they would already be set through association). The parameter beaconOrder controls whether the network is beacon-enabled or non beacon-enabled. For a beacon-enabled network, this parameter also controls the beacon transmission interval. When the operation is complete, the Start Confirm Callback is called.

- ApiMac_status_t ApiMac_mlmeSyncReq (ApiMac_mlmeSyncReq_t *pData)

  This function requests the MAC to synchronize with the coordinator by acquiring and optionally tracking its beacons. Synchronizing with the coordinator is recommended before associating in a beacon-enabled network. If the beacon could not be located on its initial search or during tracking, the MAC calls the Sync Loss Indication Callback with ApiMac_status_beaconLoss as the reason. Before calling this function, the application must set PIB attributes ApiMac_attribute_beaconOrder, ApiMac_attribute_panId, and either ApiMac_attribute_coordShortAddress or ApiMac_attribute_coordExtendedAddress to the address of the coordinator with which to synchronize. The application may wish to set PIB attribute ApiMac_attribute_autoRequest to FALSE before calling this function. Then, when the MAC successfully synchronizes with the coordinator, it will call the Beacon Notify Indication Callback. After receiving the callback, the application may set ApiMac_attribute_autoRequest to TRUE to stop receiving beacon notifications. This function is only applicable to beacon-enabled networks.

- uint8_t ApiMac_randomByte (void)

  This function returns a random byte from the MAC random number generator.

- ApiMac_status_t ApiMac_updatePanId (uint16_t panId)

  This function updates the Device Table entry and PIB with new Pan ID.

- ApiMac_status_t ApiMac_mlmeWSAsyncReq (ApiMac_mlmeWSAsyncReq_t *pData)

  This functions handles a WiSUN async request. The possible operation is Async Start or Async Stop. For the async start operation, the caller of this function can indicate which WiSUN async frame type to be sent on the specified channels.

- ApiMac_status_t ApiMac_startFH (void)

  This function starts the frequency hopping. Frequency hopping operation should have been enabled using ApiMac_enableFH() before calling this API. This API does not need to be called if called ApiMac_mlmeStartReq() has been called with the startFH field set to true.

- ApiMac_status_t ApiMac_parsePayloadGroupIEs (uint8_t *pPayload, uint16_t payloadLen, ApiMac_payloadIeRec_t **pList)

  Parses the Group payload information element. This function creates a linked list (plist) from the Payload IE (pPayload). Each item in the linked list is a seperate Group IE with its own content. If no IEs are found, pList is set to NULL. The caller is responsible for releasing the memory for the linked list by calling ApiMac_freeIEList(). Call this function to create the list of Group IEs, then call ApiMac_parsePayloadSubIEs() to parse each of the group IE's content into sub IEs.

- ApiMac_status_t ApiMac_parsePayloadSubIEs (uint8_t *pContent, uint16_t contentLen, ApiMac_payloadIeRec_t **pList)

Parses the payload subinformation element. This function creates a linked list (pList) of sub IEs from the Group IE content (pContent). Each item in the linked list is a seperate sub IE with its own content. If no IEs are found, pList is set to NULL. The caller is responsible for releasing the memory for the linked list by calling ApiMac_freeIEList(). Call this function after calling ApiMac_parsePayloadGroupIEs().

- void ApiMac_freeIEList (ApiMac_payloadIeRec_t *pList)

  Frees the linked list allocated by ApiMac_parsePayloadGroupIEs() or ApiMac_parsePayloadSubIEs().
- ApiMac_status_t ApiMac_enableFH (void)

  Enables the Frequency hopping operation. Call this function before setting any FH parameters, or before calling ApiMac_mlmeStartReq() or ApiMac_startFH(), if using FH.
- uint8_t ApiMac_convertCapabilityInfo (ApiMac_capabilityInfo_t *pMsgcapInfo)

  Converts ApiMac_capabilityInfo_t data type to uint8 capInfo.
- void ApiMac_buildMsgCapInfo (uint8_t cInfo, ApiMac_capabilityInfo_t *pPBcapInfo)

  Converts from bitmask byte to API MAC capInfo.
- ApiMac_status_t ApiMac_secAddDevice (ApiMac_secAddDevice_t *pAddDevice)

  Adds a new MAC device table entry.
- ApiMac_status_t ApiMac_secDeleteDevice (ApiMac_sAddrExt_t *pExtAddr)

  Removes MAC device table entries.
- ApiMac_status_t ApiMac_secDeleteKeyAndAssocDevices (uint8_t keyIndex)

  Removes the key at the specified key Index and removes all MAC device table enteries associated with this key. Also removes (initializes) the key lookup list associated with this key.
- ApiMac_status_t ApiMac_secDeleteAllDevices (void)

  Removes all MAC device table entries.
- ApiMac_status_t ApiMac_secGetDefaultSourceKey (uint8_t keyId, uint32_t *pFrameCounter)

  Reads the frame counter value associated with a MAC security key indexed by the designated key identifier and the default key source.
- ApiMac_status_t ApiMac_secAddKeyInitFrameCounter (ApiMac_secAddKeyInitFrameCounter_t *pInfo)

  Adds the MAC security key, adds the associated lookup list for the key, and initializes the frame counter to the value provided. The function also duplicates the device table entries (associated with the previous key if any) if available, based on the flag dupDevFlag value and associates the device descriptor with this key.

13.3 Data Structure Documentation

struct ApiMac_sAddr_t *MAC address type field structure*

Data Fields

union ApiMac_sAddr_t	addr	The address can be either a long address or a short address depending the addrMode field.
ApiMac_addrType_t	addrMode	Address type/mode

struct ApiMac_sData_t *Data buffer structure*

Data Fields

uint8_t *	p	pointer to the data buffer
uint16_t	len	length of the data buffer

struct ApiMac_MRFSKPHYDesc_t *Generic PHY descriptor*

Data Fields

uint32_t	firstChCentrFreq	First Channel Center frequency
uint16_t	numChannels	Number of channels defined for the particular PHY mode
uint32_t	channelSpacing	Distance between adjacent center channel frequencies
uint8_t	fskModScheme	2-FSK/2-GFSK/4-FSK/4-GFSK
uint8_t	symbolRate	Symbol rate selection
uint8_t	fskModIndex	Modulation index as a value encoded in MR-FSK Generic PHY Descriptor IE (IEEE802.15.4g section 5.2.4.20c). 2FSK MI = 0.25 + Modulation Index * 0.05 4FSK MI is a third of 2FSK MI
uint8_t	ccaType	Channel clearance algorithm selection

struct ApiMac_sec_t *Common security type*

Data Fields

uint8_t	keySource[APIMAC_KEY_SOURCE_MAX_LEN]	Key source
uint8_t	securityLevel	Security Level
uint8_t	keyIdMode	Key identifier mode
uint8_t	keyIndex	Key index

struct ApiMac_keyIdLookupDescriptor_t *Key ID lookup descriptor*

Data Fields

uint8_t	lookupData[APIMAC_MAX_KEY_LOOKUP_LEN]	Data used to identify the key
uint8_t	lookupDataSize	0x00 indicates 5 octets; 0x01 indicates 9 octets

struct ApiMac_keyDeviceDescriptor_t *Key device descriptor*

Data Fields

uint8_t	deviceDescriptorHandle	Handle to the DeviceDescriptor
bool	uniqueDevice	True if the device is unique
bool	blackListed	This key exhausted the frame counter.

struct ApiMac_keyUsageDescriptor_t *Key usage descriptor*

Data Fields

uint8_t	frameType	Frame type
uint8_t	cmdFrameId	Command frame identifier

struct ApiMac_keyDescriptor_t *Key descriptor*

Data Fields

ApiMac_keyIdLookupDescriptor_t *	keyIdLookupList	A list identifying this KeyDescriptor
uint8_t	keyIdLookupEntries	The number of entries in KeyIdLookupList
ApiMac_keyDeviceDescriptor_t *	keyDeviceList	A list indicating which devices are currently using this key, including their blacklist status.
uint8_t	keyDeviceListEntries	The number of entries in KeyDeviceList
ApiMac_keyUsageDescriptor_t *	keyUsageList	A list indicating which frame types this key may be used with.
uint8_t	keyUsageListEntries	The number of entries in KeyUsageList
uint8_t	key[APIMAC_KEY_MAX_LEN]	The actual value of the key
uint32_t	frameCounter	PIB frame counter in 802.15.4 is universal across the key, but should associate a frame counter with a key.

struct ApiMac_deviceDescriptor_t *Device descriptor*

Data Fields

uint16_t	panID	The 16-bit PAN identifier of the device
uint16_t	shortAddress	The 16-bit short address of the device
ApiMac_sAddrExt_t	extAddress	The 64-bit IEEE extended address of the device. This element is also used in unsecuring operations on incoming frames.

struct ApiMac_securityLevelDescriptor_t *Security level descriptor*

Data Fields

uint8_t	frameType	Frame Type
uint8_t	commandFrameIdentifier	Command Frame ID
uint8_t	securityMinimum	The minimal required or expected security level for incoming MAC frames.
bool	securityOverrideSecurityMinimum	Indication of whether originating devices for which the Exempt flag is set may override the minimum security level indicated by the Security Minimum element. If TRUE, this indicates that for originating devices with Exempt status, the incoming security level zero is acceptable.

struct ApiMac_securityDeviceDescriptor_t *Security device descriptor*

Data Fields

ApiMac_deviceDescriptor_t	devInfo	Device information
uint32_t	frameCounter[APIMAC_MAX_KEY_TABLE_ENTRIES]	The incoming frame counter of the device. This value is used to ensure sequential freshness of frames.
bool	exempt	Device may override the minimum security level settings.

struct ApiMac_securityKeyEntry_t *MAC key entry structure*

Data Fields

uint8_t	keyEntry[APIMAC_KEY_MAX_LEN]	The 128-bit key
uint8_t	keyIndex	The unique key index
uint32_t	frameCounter	The key frame counter

struct ApiMac_securityPibKeyIdLookupEntry_t *Security PIB Key ID lookup entry for a Get/Set ApiMac_securityAttribute_keyIdLookupEntry*

Data Fields

uint8_t	keyIndex	Index into the macKeyIdLookupList
uint8_t	keyIdLookupIndex	Index into macKeyIdLookupList[keyIndex]
ApiMac_keyIdLookupDescriptor_t	lookupEntry	Place to put the requested data

struct ApiMac_securityPibKeyDeviceEntry_t *Security PIB Key ID device entry for a Get/Set ApiMac_securityAttribute_keyDeviceEntry*

Data Fields

uint8_t	keyIndex	Index into the macKeyDeviceList
uint8_t	keyDeviceIndex	Index into macKeyDeviceList[keyIndex]
ApiMac_keyDeviceDescriptor_t	deviceEntry	Place to put the requested data

struct ApiMac_securityPibKeyUsageEntry_t *Security PIB Key ID usage entry for a Get/Set ApiMac_securityAttribute_keyUsageEntry*

Data Fields

uint8_t	keyIndex	Index into the macKeyUsageList
uint8_t	keyUsageIndex	Index into macKeyUsageList[keyIndex]
ApiMac_keyUsageDescriptor_t	usageEntry	Place to put the requested data

struct ApiMac_securityPibKeyEntry_t *Security PIB Key entry for a Get/Set ApiMac_securityAttribute_keyEntry*

Data Fields

uint8_t	keyIndex	Index into the macKeyTable
uint8_t	keyEntry[APIMAC_KEY_MAX_LEN]	Key entry
uint32_t	frameCounter	Frame counter

struct ApiMac_securityPibDeviceEntry_t *Security PIB device entry for a Get/Set ApiMac_securityAttribute_deviceEntry*

Data Fields

uint8_t	deviceIndex	Index into the macDeviceTable
ApiMac_securityDeviceDescriptor_t	deviceEntry	Place to put the requested data

struct ApiMac_securityPibSecurityLevelEntry_t *Security PIB level entry for a Get/Set ApiMac_securityAttribute_securityLevelEntry*

Data Fields

uint8_t	levelIndex	Index into the macSecurityLevelTable
ApiMac_securityLevelDescriptor_t	levelEntry	Place to put the requested data

struct ApiMac_capabilityInfo_t *Structure defines the Capabilities Information bit field.*

Data Fields

bool	panCoord	True if the device is a PAN coordinator
bool	ffd	True if the device is a full function device (FFD)
bool	mainsPower	True if the device is mains powered
bool	rxOnWhenIdle	True if the device RX is on when the device is idle
bool	security	True if the device is capable of sending and receiving secured frames
bool	allocAddr	True if allocation of a short address in the associate procedure is needed.

struct ApiMac_txOptions_t *Data request transmit options*

Data Fields

bool	ack	Acknowledged transmission. The MAC attempts to retransmit the frame until it is acknowledged.
bool	indirect	Indirect transmission. The MAC queues the data and waits for the destination device to poll for it. This can only be used by a coordinator device.
bool	pendingBit	This proprietary option forces the pending bit set for direct transmission.
bool	noRetransmits	This proprietary option prevents the frame from being retransmitted.
bool	noConfirm	This proprietary option prevents a MAC_MCPS_DATA_CNF event from being sent for this frame.
bool	useAltBE	Use PIB value MAC_ALT_BE for the minimum backoff exponent
bool	usePowerAndChannel	Use the power and channel values in macDataReq_t instead of the PIB values

struct ApiMac_mcpsDataReq_t *MCPS data request type*

Data Fields

ApiMac_sAddr_t	dstAddr	The address of the destination device
uint16_t	dstPanId	The PAN ID of the destination device
ApiMac_addrType_t	srcAddrMode	The source address mode
uint8_t	msduHandle	Application-defined handle value associated with this data request
ApiMac_txOptions_t	txOptions	TX options bit mask
uint8_t	channel	Transmit the data frame on this channel
uint8_t	power	Transmit the data frame at this power level
uint8_t *	pIEList	Pointer to the payload IE list, excluding termination IEs
uint16_t	payloadIELen	Length of the payload IE
ApiMac_fhDispatchType_t	fhProtoDispatch	Frequency-hopping protocol dispatch - RESERVED for future use, should be cleared.
uint32_t	includeFhIEs	Bitmap indicates which FH IEs must be included
ApiMac_sData_t	msdu	Data buffer
ApiMac_sec_t	sec	Security parameters

struct ApiMac_payloadIeItem_t *Structure a payload information item*

Data Fields

bool	ieTypeLong	True if payload IE type is long
uint8_t	ieId	IE ID
uint16_t	ieContentLen	IE Content Length – maximum size of 2047 bytes
uint8_t *	pIEContent	Pointer to the IE content

struct ApiMac_payloadIeRec_t *A Payload IE link list record*

Data Fields

void *	pNext	Pointer to the next element in the linked list, NULL if no more
ApiMac_payloadIeItem_t	item	Payload IE information item

struct ApiMac_mcpsDataInd_t *MCPS data indication type*

Data Fields

ApiMac_sAddr_t	srcAddr	The address of the sending device
ApiMac_sAddr_t	dstAddr	The address of the destination device
uint32_t	timestamp	The time, in backoffs, at which the data were received
uint16_t	timestamp2	The time, in internal MAC timer units, at which the data were received
uint16_t	srcPanId	The PAN ID of the sending device

uint16_t	dstPanId	The PAN ID of the destination device
uint8_t	mpduLinkQuality	The link quality of the received data frame
uint8_t	correlation	The raw correlation value of the received data frame
int8_t	rssi	The received RF power in units dBm
uint8_t	dsn	The data sequence number of the received frame
uint16_t	payloadIeLen	Length of the payload IE buffer (pPayloadIE)
uint8_t *	pPayloadIE	Pointer to the start of payload IEs
ApiMac_fhFrameType_t	fhFrameType	Frequency-hopping frame type
ApiMac_fhDispatchType_t	fhProtoDispatch	Frequency-hopping protocol dispatch. RESERVED for future use.
uint32_t	frameCntr	Frame counter value of the received data frame (if used)
ApiMac_sec_t	sec	Security parameters
ApiMac_sData_t	msdu	Data buffer

struct ApiMac_mcpsDataCnf_t *MCPS data confirm type*

Data Fields

ApiMac_status_t	status	Contains the status of the data request operation
uint8_t	msduHandle	Application-defined handle value associated with the data request
uint32_t	timestamp	The time, in backoffs, at which the frame was transmitted
uint16_t	timestamp2	The time, in internal MAC timer units, at which the frame was transmitted
uint8_t	retries	The number of retries required to transmit the data frame
uint8_t	mpduLinkQuality	The link quality of the received ack frame
uint8_t	correlation	The raw correlation value of the received ack frame
int8_t	rssi	The RF power of the received ack frame in units dBm
uint32_t	frameCntr	Frame counter value used (if any) for the transmitted frame

struct ApiMac_mcpsPurgeCnf_t *MCPS purge confirm type*

Data Fields

ApiMac_status_t	status	The status of the purge request operation
uint8_t	msduHandle	Application-defined handle value associated with the data request

struct ApiMac_panDesc_t *PAN descriptor type*

Data Fields

ApiMac_sAddr_t	coordAddress	The address of the coordinator sending the beacon
uint16_t	coordPanId	The PAN ID of the network
uint16_t	superframeSpec	The superframe specification of the network, this field contains the beacon order, superframe order, final CAP slot, battery life extension, PAN coordinator bit, and association permit flag. Use the following macros to parse this field: APIMAC_SFS_BEACON_ORDER(), APIMAC_SFS_SUPERFRAME_ORDER(), APIMAC_SFS_FINAL_CAP_SLOT() APIMAC_SFS_BLE(), APIMAC_SFS_PAN_COORDINATOR(), and APIMAC_SFS_ASSOCIATION_PERMIT() .
uint8_t	logicalChannel	The logical channel of the network
uint8_t	channelPage	The current channel page occupied by the network
bool	gtsPermit	TRUE if coordinator accepts GTS requests. This field is not used for enhanced beacons.
uint8_t	linkQuality	The link quality of the received beacon
uint32_t	timestamp	The time at which the beacon was received, in backoffs
bool	securityFailure	TRUE if there was an error in the security processing
ApiMac_sec_t	sec	The security parameters for the received beacon frame

struct ApiMac_mlmeAssociateReq_t *MLME associate request type*

Data Fields

ApiMac_sec_t	sec	The security parameters for this message
uint8_t	logicalChannel	The channel on which to attempt association
uint8_t	channelPage	The channel page on which to attempt association
uint8_t	phyID	Identifier for the PHY descriptor
ApiMac_sAddr_t	coordAddress	Address of the coordinator with which to associate
uint16_t	coordPanId	The identifier of the PAN with which to associate
ApiMac_capabilityInfo_t	capabilityInformation	The operational capabilities of this device

struct ApiMac_mlmeAssociateRsp_t *MLME associate response type*

Data Fields

ApiMac_sec_t	sec	The security parameters for this message
ApiMac_sAddrExt_t	deviceAddress	The address of the device requesting association
uint16_t	assocShortAddress	The short address allocated to the device
ApiMac_assocStatus_t	status	The status of the association attempt

struct ApiMac_mlmeDisassociateReq_t *MLME disassociate request type*

Data Fields

ApiMac_sec_t	sec	The security parameters for this message
ApiMac_sAddr_t	deviceAddress	The address of the device with which to disassociate
uint16_t	devicePanId	The PAN ID of the device
ApiMac_disassocateReason_t	disassociateReason	The disassociate reason
bool	txIndirect	Transmit Indirect

struct ApiMac_mlmeOrphanRsp_t *MLME orphan response type*

Data Fields

ApiMac_sec_t	sec	The security parameters for this message
ApiMac_sAddrExt_t	orphanAddress	The extended address of the device sending the orphan notification
uint16_t	shortAddress	The short address of the orphaned device
bool	associatedMember	TRUE if the orphaned device is associated with this coordinator

struct ApiMac_mlmePollReq_t *MLME poll request type*

Data Fields

ApiMac_sAddr_t	coordAddress	The address of the coordinator device to poll
uint16_t	coordPanId	The PAN ID of the coordinator
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmeScanReq_t *MLME scan request type*

Data Fields

uint8_t	scanChannels[APIMAC_154G_CHANNEL_BITMAP_SIZ]	Bit mask indicating which channels to scan
ApiMac_scantype_t	scanType	The type of scan
uint8_t	scanDuration	The exponent used in the scan duration calculation
uint8_t	channelPage	The channel page on which to perform the scan
uint8_t	phyID	PHY ID corresponding to the PHY descriptor to use
uint8_t	maxResults	The maximum number of PAN descriptor results; these results are returned in the scan confirm.

bool	permitJoining	Only devices with permit joining enabled respond to the enhanced beacon request
uint8_t	linkQuality	The device responds to the enhanced beacon request if mpduLinkQuality is equal or higher than this value.
uint8_t	percentFilter	The device randomly determines if it is to respond to the enhanced beacon request, based on meeting this probability (0 to 100%).
ApiMac_sec_t	sec	The security parameters for this message
bool	MPMScan	When TRUE, scanDuration is ignored. When FALSE, scan duration is set to scanDuration; MPMScanDuration is ignored
uint8_t	MPMScanType	BPAN or NBPAN
uint16_t	MPMScanDuration	If MPMScanType is BPAN, MPMScanDuration values are 0-14. It is used in determining the maximum time spent scanning for an EB in a beacon-enabled PAN on the channel. [aBaseSuperframeDuration * 2^n symbols], where n is the MPMScanDuration. If MPMScanType is NBPAN, valid values are 1 to 16383. It is used in determining the maximum time spent scanning for an EB in nonbeacon-enabled PAN on the channel. [aBaseSlotDuration * n] symbols, where n is MPMScanDuration.

struct ApiMac_mpmParams_t *MPM (Multi-PHY layer management) parameters*

Data Fields

uint8_t	eBeaconOrder	The exponent used to calculate the enhanced beacon interval. A value of 15 indicates no EB in a beacon-enabled PAN.
uint8_t	offsetTimeSlot	Indicates the time diff between the EB and the preceeding periodic beacon. The valid range for this field is 10 to 15.
uint16_t	NBPANEBeaconOrder	Indicates how often the EB to tx in a nonbeacon-enabled PAN. A value of 16383 indicates no EB in a nonbeacon-enabled PAN.
uint8_t *	pIEIDs	Pointer to the buffer containing the information element IDs which must be sent in an enhanced beacon. This field is reserved for future use and should be set to NULL.
uint8_t	numIEs	The number of information elements in the buffer (size of buffer at pIEIDs). This field is reserved for future use and should be set to 0.

struct ApiMac_mlmeStartReq_t *MLME start request type*

Data Fields

uint32_t	startTime	The time to begin transmitting beacons relative to the received beacon
uint16_t	panId	The PAN ID to use. This parameter is ignored if panCoordinator is FALSE.
uint8_t	logicalChannel	The logical channel to use. This parameter is ignored if panCoordinator is FALSE.
uint8_t	channelPage	The channel page to use. This parameter is ignored if panCoordinator is FALSE.
uint8_t	phyID	PHY ID corresponding to the PHY descriptor to use
uint8_t	beaconOrder	The exponent used to calculate the beacon interval
uint8_t	superframeOrder	The exponent used to calculate the superframe duration
bool	panCoordinator	Set to TRUE to start a network as PAN coordinator
bool	batteryLifeExt	If this value is TRUE, the receiver is disabled after MAC_BATT_LIFE_EXT_PERIODS full backoff periods following the interframe spacing period of the beacon frame.
bool	coordRealignment	Set to TRUE to transmit a coordinator realignment prior to changing the superframe configuration
ApiMac_sec_t	realignSec	Security parameters for the coordinator realignment frame
ApiMac_sec_t	beaconSec	Security parameters for the beacon frame
ApiMac_mpmParams_t	mpmParams	MPM (multi-PHY layer management) parameters
bool	startFH	Indicates whether frequency hopping must be enabled

struct ApiMac_mlmeSyncReq_t *MAC_MlmeSyncReq type*

Data Fields

uint8_t	logicalChannel	The logical channel to use
uint8_t	channelPage	The channel page to use
uint8_t	phyID	PHY ID corresponding to the PHY descriptor to use
uint8_t	trackBeacon	Set to TRUE to continue tracking beacons after synchronizing with the first beacon. Set to FALSE to only synchronize with the first beacon.

struct ApiMac_mlmeWSAsyncReq_t *MLME WiSUN Async request type*

Data Fields

ApiMac_wisunAsycnOperation_t	operation	Start or Stop Async operation
ApiMac_wisunAsyncFrame_t	frameType	Async frame type
uint8_t	channels[APIMAC_154G_CHANNEL_BITMAP_SIZ]	Bit mask indicating which channels to send the Async frames for the start operation
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_secAddDevice_t *Structure to pass information to the ApiMac_secAddDevice().*

Data Fields

uint16_t	panID	PAN ID of the new device
uint16_t	shortAddr	Short address of the new device
ApiMac_sAddrExt_t	extAddr	Extended address of the new device
bool	exempt	Device descriptor exempt field value (true or false); setting this field to true means that this device can override the minimum security level setting.
uint8_t	keyIdLookupDataSize	Key ID lookup data size as it is stored in PIB, (that is, 0 for 5 bytes, 1 for 9 bytes).
uint8_t	keyIdLookupData[APIMAC_MAX_KEY_LOOKUP_LEN]	Key ID lookup data, to look for the key table entry and create proper key device descriptor for this device.
uint32_t	frameCounter	Frame counter
bool	uniqueDevice	Key device descriptor uniqueDevice field value (true or false)
bool	duplicateDevFlag	A flag (true or false) to indicate whether the device entry should be duplicated even for the keys that do not match the key ID lookup data. The device descriptors that are pointed by the key device descriptors that do not match the key ID lookup data do not update the frame counter based on the frameCounter argument to this function, or set the frame counter to zero when the entry is newly created.

struct ApiMac_secAddKeyInitFrameCounter_t *Structure to pass information to the ApiMac_secAddKeyInitFrameCounter().*

Data Fields

uint8_t	key[APIMAC_KEY_MAX_LEN]	Key
uint32_t	frameCounter	Frame counter
uint8_t	replaceKeyIndex	Key index of the mac security key table where the key must be written
bool	newKeyFlag	If set to true, the function duplicates the device table entries associated with the previous key, and associates it with the key. If set to false, the function does not alter the device table entries associated with whatever key that was stored in the key table location, as designated by replaceKeyIndex.
uint8_t	lookupDataSize	Key ID lookup data size as it is stored in PIB, that is, 0 for 5 bytes, 1 for 9 bytes.
uint8_t	lookupData[APIMAC_MAX_KEY_LOOKUP_LEN]	Key ID lookup data, to look for the key table entry and create a proper key device descriptor for this device.

struct ApiMac_mlmeAssociateInd_t *MAC_MLME_ASSOCIATE_IND type*

Data Fields

ApiMac_sAddrExt_t	deviceAddress	The address of the device requesting association
ApiMac_capabilityInfo_t	capabilityInformation	The operational capabilities of the device requesting association
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmeAssociateCnf_t *MAC_MLME_ASSOCIATE_CNF type*

Data Fields

ApiMac_assocStatus_t	status	Status of associate attempt
uint16_t	assocShortAddress	If successful, the short address allocated to this device
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmeDisassociateInd_t *MAC_MLME_DISASSOCIATE_IND type*

Data Fields

ApiMac_sAddrExt_t	deviceAddress	The address of the device sending the disassociate command
ApiMac_disassocateReason_t	disassociateReason	The disassociate reason
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmeDisassociateCnf_t *MAC_MLME_DISASSOCIATE_CNF type*

Data Fields

ApiMac_status_t	status	Status of the disassociate attempt
ApiMac_sAddr_t	deviceAddress	The address of the device that has either requested disassociation or been instructed to disassociate by its coordinator.
uint16_t	panId	The pan ID of the device that has either requested disassociation or been instructed to disassociate by its coordinator.

struct ApiMac_beaconData_t *MAC Beacon data type*

Data Fields

uint8_t	numPendShortAddr	The number of pending short addresses
uint16_t *	pShortAddrList	The list of device short addresses for which the sender of the beacon has data
uint8_t	numPendExtAddr	The number of pending extended addresses
uint8_t *	pExtAddrList	The list of device short addresses for which the sender of the beacon has data
uint8_t	sduLength	The number of bytes in the beacon payload of the beacon frame
uint8_t *	pSdu	The beacon payload

struct ApiMac_coexist_t *Coexistence information element content type*

Data Fields

uint8_t	beaconOrder	Specifies the transmission interval of the beacon
uint8_t	superFrameOrder	Specifies the length of time during which the superframe is active (that is, receiver-enabled), including the beacon frame transmission time.
uint8_t	finalCapSlot	Final CAP slot
uint8_t	eBeaconOrder	Specifies the transmission interval of the enhanced beacon frames in a beacon-enabled network
uint8_t	offsetTimeSlot	Time offset between periodic beacon and the enhanced beacon.
uint8_t	capBackOff	Actual slot position in which the enhanced beacon frame is transmitted due to the backoff procedure in the CAP
uint16_t	eBeaconOrderNBPAN	Specifies the transmission interval between consecutive enhanced beacon frames in the nonbeacon-enabled mode

struct ApiMac_eBeaconData_t *MAC enhanced beacon data type*

Data Fields

ApiMac_coexist_t	coexist	Beacon coexist data

struct ApiMac_mlmeBeaconNotifyInd_t *MAC_MLME_BEACON_NOTIFY_IND type*

Data Fields

ApiMac_beaconType_t	beaconType	Indicates the beacon type: beacon or enhanced beacon
uint8_t	bsn	The beacon sequence number or enhanced beacon sequence number
ApiMac_panDesc_t	panDesc	The PAN descriptor for the received beacon
union ApiMac_mlmeBeaconNotifyInd_t	beaconData	Beacon data union depending on beaconType, select beaconData or or eBeaconData.

struct ApiMac_mlmeOrphanInd_t *MAC_MLME_ORPHAN_IND type*

Data Fields

ApiMac_sAddrExt_t	orphanAddress	The address of the orphaned device
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmeScanCnf_t *MAC_MLME_SCAN_CNF type*

Data Fields

ApiMac_status_t	status	Status of the scan request
ApiMac_scantype_t	scanType	The type of scan requested
uint8_t	channelPage	The channel page of the scan
uint8_t	phyId	PHY ID corresponding to the PHY descriptor used during scan
uint8_t	unscannedChannels[APIMAC_154G_CHANNEL_BITMAP_SIZ]	Bit mask of channels that were not scanned
uint8_t	resultListSize	The number of PAN descriptors returned in the results list
union ApiMac_mlmeScanCnf_t	result	Depending on the scanType the results are in this union

struct ApiMac_mlmeStartCnf_t *MAC_MLME_START_CNF type*

Data Fields

ApiMac_status_t	status	Status of the start request

struct ApiMac_mlmeSyncLossInd_t *MAC_MLME_SYNC_LOSS_IND type*

Data Fields

ApiMac_status_t	reason	Reason that the synchronization was lost
uint16_t	panId	The PAN ID of the realignment
uint8_t	logicalChannel	The logical channel of the realignment
uint8_t	channelPage	The channel page of the realignment
uint8_t	phyID	PHY ID corresponding to the PHY descriptor of the realignment
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmePollCnf_t *MAC_MLME_POLL_CNF type*

Data Fields

ApiMac_status_t	status	Status of the poll request
uint8_t	framePending	Set if framePending bit in data packet is set

struct ApiMac_mlmeCommStatusInd_t *MAC_MLME_COMM_STATUS_IND type*

Data Fields

ApiMac_status_t	status	Status of the event
ApiMac_sAddr_t	srcAddr	The source address associated with the event
ApiMac_sAddr_t	dstAddr	The destination address associated with the event
uint16_t	panId	The PAN ID associated with the event
ApiMac_commStatusReason_t	reason	The reason the event was generated
ApiMac_sec_t	sec	The security parameters for this message

struct ApiMac_mlmePollInd_t *MAC_MLME_POLL_IND type*

Data Fields

ApiMac_sAddr_t	srcAddr	Address of the device sending the data request
uint16_t	srcPanId	Pan ID of the device sending the data request
bool	noRsp	indication that no MAC_McpsDataReq() is required. It is set when MAC_MLME_POLL_IND is generated, to indicate that a received data request frame was acked with the pending bit cleared.

struct ApiMac_mlmeWsAsyncCnf_t *MAC_MLME_WS_ASYNC_FRAME_CNF type*

Data Fields

ApiMac_status_t	status	Status of the Async request

struct ApiMac_callbacks_t *Structure containing all the MAC callbacks (indications). To receive the confirmation or indication, fill in the associated callback with a pointer to the function that handles that callback. To ignore a callback, set that function pointer to NULL.*

Data Fields

ApiMac_associateIndFp_t	pAssocIndCb	Associate indicated callback
ApiMac_associateCnfFp_t	pAssocCnfCb	Associate confirmation callback
ApiMac_disassociateIndFp_t	pDisassociateIndCb	Disassociate indication callback
ApiMac_disassociateCnfFp_t	pDisassociateCnfCb	Disassociate confirmation callback
ApiMac_beaconNotifyIndFp_t	pBeaconNotifyIndCb	Beacon notify indication callback
ApiMac_orphanIndFp_t	pOrphanIndCb	Orphan indication callback
ApiMac_scanCnfFp_t	pScanCnfCb	Scan confirmation callback
ApiMac_startCnfFp_t	pStartCnfCb	Start confirmation callback
ApiMac_syncLossIndFp_t	pSyncLossIndCb	Sync loss indication callback
ApiMac_pollCnfFp_t	pPollCnfCb	Poll confirm callback
ApiMac_commStatusIndFp_t	pCommStatusCb	Comm status indication callback
ApiMac_pollIndFp_t	pPollIndCb	Poll indication callback
ApiMac_dataCnfFp_t	pDataCnfCb	Data confirmation callback
ApiMac_dataIndFp_t	pDataIndCb	Data indication callback
ApiMac_purgeCnfFp_t	pPurgeCnfCb	Purge confirm callback
ApiMac_wsAsyncIndFp_t	pWsAsyncIndCb	WiSUN Async indication callback
ApiMac_wsAsyncCnfFp_t	pWsAsyncCnfCb	WiSUN Async confirmation callback
ApiMac_unprocessedFp_t	pUnprocessedCb	Unprocessed message callback

union ApiMac_sAddr_t.addr *The address can be either a long address or a short address, depending the addrMode field.*

Data Fields

uint16_t	shortAddr	16-bit address
ApiMac_sAddrExt_t	extAddr	Extended address

union ApiMac_mlmeBeaconNotifyInd_t.beaconData *Beacon data union; depending on beaconType, select beaconData or or eBeaconData.*

Data Fields

ApiMac_beaconData_t	beacon	Beacon data
ApiMac_eBeaconData_t	eBeacon	Enhanced beacon data

union ApiMac_mlmeScanCnf_t.result *Depending on the scanType, the results are in this union.*

Data Fields

uint8_t *	pEnergyDetect	The list of energy measurements, one for each channel scanned
ApiMac_panDesc_t *	pPanDescriptor	The list of PAN descriptors, one for each beacon found

13.4 Macro Definition Documentation

- #define APIMAC_KEY_MAX_LEN 16
  Key Length
- #define APIMAC_SADDR_EXT_LEN 8 IEEE
  Address Length
- #define APIMAC_MAX_KEY_TABLE_ENTRIES 2
  Maximum number of key table entries
- #define APIMAC_KEYID_IMPLICIT_LEN 0
  Key identifier field length – Implicit mode
- #define APIMAC_KEYID_MODE1_LEN 1
  Key identifier field length – mode 1
- #define APIMAC_KEYID_MODE4_LEN 5
  Key Identifier field length – mode 4
- #define APIMAC_KEYID_MODE8_LEN 9
  Key Identifier field length – mode 8
- #define APIMAC_KEY_SOURCE_MAX_LEN 8
  Key source maximum length in bytes
- #define APIMAC_KEY_INDEX_LEN 1
  Key index length in bytes
- #define APIMAC_FRAME_COUNTER_LEN 4
  Frame counter length in bytes
- #define APIMAC_KEY_LOOKUP_SHORT_LEN 5
  Key lookup data length in bytes – short length

- #define APIMAC_KEY_LOOKUP_LONG_LEN 9
  Key lookup data length in bytes – long length
- #define APIMAC_MAX_KEY_LOOKUP_LEN APIMAC_KEY_LOOKUP_LONG_LEN
  Key lookup data length in bytes – lookup length
- #define APIMAC_DATA_OFFSET 24
  Bytes required for MAC header in data frame
- #define APIMAC_MAX_BEACON_PAYLOAD 16
  Maximum length allowed for the beacon payload
- #define APIMAC_MIC_32_LEN 4
  Length required for MIC-32 authentication
- #define APIMAC_MIC_64_LEN 8
  Length required for MIC-64 authentication
- #define APIMAC_MIC_128_LEN 16
  Length required for MIC-128 authentication
- #define APIMAC_MHR_LEN 37
  MHR length for received frame
  FCF (2) + Seq (1) + Addr Fields (20) + Security HDR (14)
- #define APIMAC_CHANNEL_PAGE_9 9
  Channel Page – standard-defined SUN PHY operating modes
- #define APIMAC_CHANNEL_PAGE_10 10
  Channel Page – MR-FSK Generic-PHY-defined PHY modes
- #define APIMAC_STANDARD_PHY_DESCRIPTOR_ENTRIES 3
  Maximum number of standard PHY descriptor entries
- #define APIMAC_GENERIC_PHY_DESCRIPTOR_ENTRIES 3
  Maximum number of generic PHY descriptor entries
- #define APIMAC_STD_US_915_PHY_1 1
  PHY IDs – 915-MHz US frequency band operating mode # 1
- #define APIMAC_STD_US_915_PHY_2 2
  PHY IDs – 915-MHz US frequency band operating mode # 2
- #define APIMAC_STD_ETSI_863_PHY_3 3
  863-MHz ETSI frequency band operating mode #1
- #define APIMAC_MRFSK_GENERIC_PHY_ID_BEGIN 128
  PHY IDs – MRFSK generic Phy ID start
- #define APIMAC_MRFSK_GENERIC_PHY_ID_END 143
  PHY IDs – MRFSK generic Phy ID end
- #define APIMAC_MRFSK_STD_PHY_ID_BEGIN APIMAC_STD_US_915_PHY_1
  PHY IDs – MRFSK standard Phy ID start
- #define APIMAC_MRFSK_STD_PHY_ID_END APIMAC_STD_ETSI_863_PHY_3
  PHY IDs – MRFSK standard Phy ID end
- #define APIMAC_PHY_DESCRIPTOR 0x01
  PHY descriptor table entry
- #define APIMAC_ADDR_USE_EXT 0xFFFE
  Special address value – Short address value indicating extended address is used
- #define APIMAC_SHORT_ADDR_BROADCAST 0xFFFF
  Special address value – Broadcast short address

- #define APIMAC_SHORT_ADDR_NONE 0xFFFF
  Special address value – Short address when there is no short address
- #define APIMAC_RANDOM_SEED_LEN 32
  The length of the random seed is set for maximum requirement, which is 32
- #define APIMAC_FH_UTT_IE 0x00000002
  Frequency-hopping UTT IE selection bit
- #define APIMAC_FH_BT_IE 0x00000008
  Frequency-hopping BT IE selection bit
- #define APIMAC_FH_US_IE 0x00010000
  Frequency-hopping US IE selection bit
- #define APIMAC_FH_BS_IE 0x00020000
  Frequency-hopping BS IE selection bit
- #define APIMAC_FH_HEADER_IE_MASK 0x000000FF
  Frequency-hopping header IE mask
- #define APIMAC_FH_PROTO_DISPATCH_NONE 0x00
  Frequency-hopping protocol dispatch values – Protocol dispatch none
- #define APIMAC_FH_PROTO_DISPATCH_MHD_PDU 0x01
  Frequency-hopping protocol dispatch values – Protocol dispatch MHD-PDU
- #define APIMAC_FH_PROTO_DISPATCH_6LOWPAN 0x02
  Frequency-hopping protocol dispatch values – Protocol dispatch 6LOWPAN
- #define APIMAC_154G_MAX_NUM_CHANNEL 129
  Maximum number of channels
- #define APIMAC_154G_CHANNEL_BITMAP_SIZ ((APIMAC_154G_MAX_NUM_CHANNEL + 7) / 8)
  Bitmap size to hold the channel list
- #define APIMAC_HEADER_IE_MAX 2
  Maximum number of header IEs
- #define APIMAC_PAYLOAD_IE_MAX 2
  Maximum number of payload-IEs
- #define APIMAC_PAYLOAD_SUB_IE_MAX 4
  Maximum number of sub-IEs
- #define APIMAC_SFS_BEACON_ORDER(s) ((s) & 0x0F)
  MACRO that returns the beacon order from the superframe specification
- #define APIMAC_SFS_SUPERFRAME_ORDER(s) (((s) >> 4) & 0x0F)
  MACRO that returns the superframe order from the superframe specification
- #define APIMAC_SFS_FINAL_CAP_SLOT(s) (((s) >> 8) & 0x0F)
  MACRO that returns the final CAP slot from the superframe specification
- #define APIMAC_SFS_BLE(s) (((s) >> 12) & 0x01)
  MACRO that returns the battery life extension bit from the superframe specification
- #define APIMAC_SFS_PAN_COORDINATOR(s) (((s) >> 14) & 0x01)
  MACRO that returns the PAN coordinator bit from the superframe specification
- #define APIMAC_SFS_ASSOCIATION_PERMIT(s) (((s) >> 15) & 0x01)
  MACRO that returns the Associate Permit bit from the superframe specification
- #define APIMAC_FH_MAX_BIT_MAP_SIZE 32
  Maximum size of the frequency-hopping channel-map size
- #define APIMAC_FH_NET_NAME_SIZE_MAX 32

Maximum size of the frequency-hopping network name
- #define APIMAC_FH_GTK_HASH_SIZE 8
  Size of the frequency-hopping GTK hash size

13.5 Typedef Documentation

- typedef uint8_t ApiMac_sAddrExt_t[APIMAC_SADDR_EXT_LEN]
  Extended address
- typedef ApiMac_mcpsDataInd_t ApiMac_mlmeWsAsyncInd_t
  MAC_MLME_WS_ASYNC_FRAME_IND type
- typedef void(* ApiMac_associateIndFp_t) (ApiMac_mlmeAssociateInd_t *pAssocInd)
  Associate Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_associateCnfFp_t) (ApiMac_mlmeAssociateCnf_t *pAssocCnf)
  Assocate Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_disassociateIndFp_t) (ApiMac_mlmeDisassociateInd_t *pDisassociateInd)
  Disassociate Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_disassociateCnfFp_t) (ApiMac_mlmeDisassociateCnf_t *pDisassociateCnf)
  Disassociate Confirm Callback function pointer prototype for the callback table
- typedef void(* ApiMac_beaconNotifyIndFp_t) (ApiMac_mlmeBeaconNotifyInd_t *pBeaconNotifyInd)
  Beacon Notify Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_orphanIndFp_t) (ApiMac_mlmeOrphanInd_t *pOrphanInd)
  Orphan Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_scanCnfFp_t) (ApiMac_mlmeScanCnf_t *pScanCnf)
  Scan Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_startCnfFp_t) (ApiMac_mlmeStartCnf_t *pStartCnf)
  Start Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_syncLossIndFp_t) (ApiMac_mlmeSyncLossInd_t *pSyncLossInd)
  Sync Loss Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_pollCnfFp_t) (ApiMac_mlmePollCnf_t *pPollCnf)
  Poll Confirm Callback function pointer prototype for the callback table
- typedef void(* ApiMac_commStatusIndFp_t) (ApiMac_mlmeCommStatusInd_t *pCommStatus)
  Comm Status Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_pollIndFp_t) (ApiMac_mlmePollInd_t *pPollInd)
  Poll Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_dataCnfFp_t) (ApiMac_mcpsDataCnf_t *pDataCnf)
  Data Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_dataIndFp_t) (ApiMac_mcpsDataInd_t *pDataInd)
  Data Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_purgeCnfFp_t) (ApiMac_mcpsPurgeCnf_t *pPurgeCnf)
  Purge Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_wsAsyncIndFp_t) (ApiMac_mlmeWsAsyncInd_t *pWsAsyncInd)
  WiSUN Async Indication Callback function pointer prototype for the callback table
- typedef void(* ApiMac_wsAsyncCnfFp_t) (ApiMac_mlmeWsAsyncCnf_t *pWsAsyncCnf)
  WiSUN Async Confirmation Callback function pointer prototype for the callback table
- typedef void(* ApiMac_unprocessedFp_t) (uint16_t param1, uint16_t param2, void *pMsg)
  Unprocessed Message Callback function pointer prototype for the callback table. This function is called when an unrecognized message is received.

13.6 Enumeration Type Documentation

- enum ApiMac_assocStatus_t

  Associate Response status types

  Enumerator

  – *ApiMac_assocStatus_success*: Success, join allowed
  – *ApiMac_assocStatus_panAtCapacity*: PAN at capacity
  – *ApiMac_assocStatus_panAccessDenied*: PAN access denied

- enum ApiMac_addrType_t

  Address types – used to set addrMode field of the ApiMac_sAddr_t structure.

  Enumerator

  – *ApiMac_addrType_none*: Address not present
  – *ApiMac_addrType_short*: Short address (16 bits)
  – *ApiMac_addrType_extended*: Extended address (64 bits)

- enum ApiMac_beaconType_t

  Beacon types in the ApiMac_mlmeBeaconNotifyInd_t structure.

  Enumerator

  – *ApiMac_beaconType_normal*: Normal beacon type
  – *ApiMac_beaconType_enhanced*: Enhanced beacon type

- enum ApiMac_disassocateReason_t

  Disassociate reasons

  Enumerator

  – *ApiMac_disassocateReason_coord*: The coordinator wishes the device to disassociate.
  – *ApiMac_disassocateReason_device*: The device wishes to disassociate.

- enum ApiMac_commStatusReason_t

  Comm status indication reasons

  Enumerator

  – *ApiMac_commStatusReason_assocRsp*: Reason for comm status indication was in response to an associate response
  – *ApiMac_commStatusReason_orphanRsp*: Reason for comm status indication was in response to an orphan response
  – *ApiMac_commStatusReason_rxSecure*: Reason for comm status indication was result of receiving a secure frame

- enum ApiMac_status_t

  General MAC status values

  Enumerator

  – *ApiMac_status_success*: Operation successful
  – *ApiMac_status_subSystemError*: MAC co-processor only – subsystem error
  – *ApiMac_status_commandIDError*: MAC co-processor only – command ID error
  – *ApiMac_status_lengthError*: MAC co-processor only – length error
  – *ApiMac_status_unsupportedType*: MAC co-processor only – unsupported extended type
  – *ApiMac_status_autoAckPendingAllOn*: The AUTOPEND pending all is turned on.
  – *ApiMac_status_autoAckPendingAllOff*: The AUTOPEND pending all is turned off.
  – *ApiMac_status_beaconLoss*: The beacon was lost following a synchronization request.
  – *ApiMac_status_channelAccessFailure*: The operation or data request failed because of activity on the channel.
  – *ApiMac_status_counterError*: The frame counter applied by the originator of the received frame is invalid.

- *ApiMac_status_denied*: The MAC was not able to enter low-power mode.
- *ApiMac_status_disabledTrxFailure*: Unused
- *ApiMac_status_frameTooLong*: The received frame or frame resulting from an operation or data request is too long to be processed by the MAC.
- *ApiMac_status_improperKeyType*: The key applied by the originator of the received frame is not allowed.
- *ApiMac_status_improperSecurityLevel*: The security level applied by the originator of the received frame does not meet the minimum security level.
- *ApiMac_status_invalidAddress*: The data request failed because neither the source address nor the destination address parameters were present.
- *ApiMac_status_invalidGts*: Unused
- *ApiMac_status_invalidHandle*: The purge request contained an invalid handle.
- *ApiMac_status_invalidIndex*: Unused
- *ApiMac_status_invalidParameter*: The API function parameter is out of range.
- *ApiMac_status_limitReached*: The scan terminated because the PAN descriptor storage limit was reached.
- *ApiMac_status_noAck*: The operation or data request failed because no acknowledgement was received.
- *ApiMac_status_noBeacon*: The scan request failed because no beacons were received, or the orphan scan failed because no coordinator realignment was received.
- *ApiMac_status_noData*: The associate request failed because no associate response was received, or the poll request did not return any data.
- *ApiMac_status_noShortAddress*: The short address parameter of the start request was invalid.
- *ApiMac_status_onTimeTooLong*: Unused
- *ApiMac_status_outOfCap*: Unused
- *ApiMac_status_panIdConflict*: A PAN identifier conflict was detected and communicated to the PAN coordinator.
- *ApiMac_status_pastTime*: Unused
- *ApiMac_status_readOnly*: A set request was issued with a read-only identifier.
- *ApiMac_status_realignment*: A coordinator realignment command was received.
- *ApiMac_status_scanInProgress*: The scan request failed because a scan is already in progress.
- *ApiMac_status_securityError*: Cryptographic processing of the received secure frame failed.
- *ApiMac_status_superframeOverlap*: The beacon start time overlapped the coordinator transmission time.
- *ApiMac_status_trackingOff*: The start request failed because the device is not tracking the beacon of its coordinator.
- *ApiMac_status_transactionExpired*: The associate response, disassociate request, or indirect data transmission failed because the peer device did not respond before the transaction expired or was purged.
- *ApiMac_status_transactionOverflow*: The request failed because the MAC data buffers are full.
- *ApiMac_status_txActive*: Unused
- *ApiMac_status_unavailableKey*: The operation or data request failed because the security key is not available.
- *ApiMac_status_unsupportedAttribute*: The set or get request failed because the attribute is not supported.
- *ApiMac_status_unsupportedLegacy*: The received frame was secured with legacy security which is not supported.
- *ApiMac_status_unsupportedSecurity*: The security of the received frame is not supported.
- *ApiMac_status_unsupported*: The operation is not supported in the current configuration.

  - *ApiMac_status_badState*: The operation could not be performed in the current state.
  - *ApiMac_status_noResources*: The operation could not be completed because no memory resources were available.
  - *ApiMac_status_ackPending*: For internal use only
  - *ApiMac_status_noTime*: For internal use only
  - *ApiMac_status_txAborted*: For internal use only
  - *ApiMac_status_duplicateEntry*: For internal use only – a duplicated entry is added to the source matching table.
  - *ApiMac_status_fhError*: Frequency hopping – general error
  - *ApiMac_status_fhIeNotSupported*: Frequency hopping – IE is not supported
  - *ApiMac_status_fhNotInAsync*: Frequency hopping – there is no ASYNC message in the MAC TX queue.
  - *ApiMac_status_fhNotInNeighborTable*: Frequency hopping – the destination address is not in the neighbor table.
  - *ApiMac_status_fhOutSlot*: Frequency hopping – not in UC or BC dwell time slot
  - *ApiMac_status_fhInvalidAddress*: Frequency hopping – invalid address
  - *ApiMac_status_fhIeFormatInvalid*: Frequency hopping – IE format is wrong
  - *ApiMac_status_fhPibNotSupported*: Frequency hopping – PIB is not supported
  - *ApiMac_status_fhPibReadOnly*: Frequency hopping – PIB is read only
  - *ApiMac_status_fhPibInvalidParameter*: Frequency hopping – PIB API invalid parameter
  - *ApiMac_status_fhInvalidFrameType*: Frequency hopping – invalid frame type
  - *ApiMac_status_fhExpiredNode*: Frequency hopping – expired node
- enum ApiMac_secLevel_t MAC

  Security levels

  Enumerator
  - *ApiMac_secLevel_none*: No security is used
  - *ApiMac_secLevel_mic32*: MIC-32 authentication is used
  - *ApiMac_secLevel_mic64*: MIC-64 authentication is used
  - *ApiMac_secLevel_mic128*: MIC-128 authentication is used
  - *ApiMac_secLevel_enc*: AES encryption is used
  - *ApiMac_secLevel_encMic32*: AES encryption and MIC-32 authentication are used
  - *ApiMac_secLevel_encMic64*: AES encryption and MIC-64 authentication are used
  - *ApiMac_secLevel_encMic128*: AES encryption and MIC-128 authentication are used
- enum ApiMac_keyIdMode_t

  Key identifier mode

  Enumerator
  - *ApiMac_keyIdMode_implicit*: Key is determined implicitly
  - *ApiMac_keyIdMode_1*: Key is determined from the 1-byte key index
  - *ApiMac_keyIdMode_4*: Key is determined from the 4-byte key index
  - *ApiMac_keyIdMode_8*: Key is determined from the 8-byte key index
- enum ApiMac_attribute_bool_t

  Standard PIB Get and Set attributes – size bool

  Enumerator
  - *ApiMac_attribute_associatePermit*: TRUE if a coordinator is currently allowing association
  - *ApiMac_attribute_autoRequest*: TRUE if a device automatically sends a data request if its address is listed in the beacon frame
  - *ApiMac_attribute_battLifeExt*: TRUE if battery life extension is enabled

- *ApiMac_attribute_gtsPermit*: TRUE if the PAN coordinator accepts GTS requests
- *ApiMac_attribute_promiscuousMode*: TRUE if the MAC is in promiscuous mode
- *ApiMac_attribute_RxOnWhenIdle*: TRUE if the MAC enables its receiver during idle periods
- *ApiMac_attribute_associatedPanCoord*: TRUE if the device is associated to the PAN coordinator
- *ApiMac_attribute_timestampSupported*: TRUE if the MAC supports RX and TX timestamps
- *ApiMac_attribute_securityEnabled*: TRUE if security is enabled
- *ApiMac_attribute_includeMPMIE*: TRUE if MPM IE must be included
- *ApiMac_attribute_fcsType*: FCS type

• enum ApiMac_attribute_uint8_t

Standard PIB Get and Set attributes – size uint8_t

Enumerator

- *ApiMac_attribute_ackWaitDuration*: The maximum number of symbols to wait for an acknowledgment frame
- *ApiMac_attribute_battLifeExtPeriods*: The number of backoff periods during which the receiver is enabled following a beacon in battery-life extension mode
- *ApiMac_attribute_beaconPayloadLength*: The length in bytes of the beacon payload; the maximum value for this parameters is APIMAC_MAX_BEACON_PAYLOAD.
- *ApiMac_attribute_beaconOrder*: How often the coordinator transmits a beacon
- *ApiMac_attribute_bsn*: The beacon sequence number
- *ApiMac_attribute_dsn*: The data or MAC command frame sequence number
- *ApiMac_attribute_maxCsmaBackoffs*: The maximum number of backoffs the CSMA-CA algorithm attempts before declaring a channel failure
- *ApiMac_attribute_backoffExponent*: The minimum value of the backoff exponent in the CSMA-CA algorithm. If this value is set to 0, collision avoidance is disabled during the first iteration of the algorithm. Also, for the slotted version of the CSMA-CA algorithm with the battery life extension enabled, the minimum value of the backoff exponent will be at least 2.
- *ApiMac_attribute_superframeOrder*: This specifies the length of the active portion of the superframe
- *ApiMac_attribute_maxBackoffExponent*: The maximum value of the backoff exponent in the CSMA-CA algorithm
- *ApiMac_attribute_maxFrameRetries*: The maximum number of retries allowed after a transmission failure
- *ApiMac_attribute_responseWaitTime*: The maximum number of symbols a device waits for a response command to be available following a request command in multiples of aBaseSuperframeDuration
- *ApiMac_attribute_syncSymbolOffset*: The timestamp offset from SFD in symbols
- *ApiMac_attribute_eBeaconSequenceNumber*: Enhanced beacon sequence number
- *ApiMac_attribute_eBeaconOrder*: Enhanced beacon order in a beacon-enabled network
- *ApiMac_attribute_offsetTimeslot*: Offset time slot from the beacon
- *ApiMac_attribute_phyTransmitPowerSigned*: Duplicate transmit power attribute in signed (2's complement) dBm unit
- *ApiMac_attribute_logicalChannel*: The logical channel
- *ApiMac_attribute_altBackoffExponent*: Alternate minimum backoff exponent
- *ApiMac_attribute_deviceBeaconOrder*: Device beacon order
- *ApiMac_attribute_rf4cePowerSavings*: Valid values are true and false
- *ApiMac_attribute_frameVersionSupport*: Currently supports 0 and 1. If 0, frame version is always 0 and set to 1 only for secure frames. If 1, frame version is set to 1 only if packet len > 102 or for secure frames.
- *ApiMac_attribute_channelPage*: Channel page
- *ApiMac_attribute_phyCurrentDescriptorId*: PHY descriptor ID, used to support the channel page

number and index into the descriptor table.

- enum ApiMac_attribute_uint16_t

  Standard PIB Get and Set attributes – size uint16_t

  Enumerator

  – *ApiMac_attribute_coordShortAddress*: The short address assigned to the coordinator with which the device is associated. A value of MAC_ADDR_USE_EXT indicates that the coordinator is using its extended address.

  – *ApiMac_attribute_panId*: The PAN identifier. If this value is 0xffff, the device is not associated.

  – *ApiMac_attribute_shortAddress*: The short address that the device uses to communicate in the PAN. If the device is a PAN coordinator, this value is set before calling MAC_StartReq(). Otherwise, the value is allocated during association. Value MAC_ADDR_USE_EXT indicates that the device is associated but not using a short address.

  – *ApiMac_attribute_transactionPersistenceTime*: The maximum time in beacon intervals that a transaction is stored by a coordinator and indicated in the beacon.

  – *ApiMac_attribute_maxFrameTotalWaitTime*: The maximum number of CAP symbols in a beacon-enabled PAN, or symbols in a non beacon-enabled PAN, to wait for a frame intended as a response to a data request frame.

  – *ApiMac_attribute_eBeaconOrderNBPAN*: Enhanced beacon order in a nonbeacon-enabled network

- enum ApiMac_attribute_uint32_t

  Standard PIB Get and Set attributes – size uint32_t

  Enumerator

  – *ApiMac_attribute_beaconTxTime*: The time the device transmitted its last beacon frame, in backoff period units.

  – *ApiMac_attribute_diagRxCrcPass*: Diagnostics PIB – Received CRC pass counter

  – *ApiMac_attribute_diagRxCrcFail*: Diagnostics PIB – Received CRC fail counter

  – *ApiMac_attribute_diagRxBroadcast*: Diagnostics PIB – Received broadcast counter

  – *ApiMac_attribute_diagTxBroadcast*: Diagnostics PIB – Transmitted broadcast counter

  – *ApiMac_attribute_diagRxUnicast*: Diagnostics PIB – Received unicast counter

  – *ApiMac_attribute_diagTxUnicast*: Diagnostics PIB – Transmitted unicast counter

  – *ApiMac_attribute_diagTxUnicastRetry*: Diagnostics PIB – Transmitted unicast retry counter

  – *ApiMac_attribute_diagTxUnicastFail*: Diagnostics PIB – Transmitted unicast fail counter

  – *ApiMac_attribute_diagRxSecureFail*: Diagnostics PIB – Received Security fail counter

  – *ApiMac_attribute_diagTxSecureFail*: Diagnostics PIB – Transmit Security fail counter

- enum ApiMac_attribute_array_t

  Standard PIB Get and Set attributes – these attributes are an array of bytes

  Enumerator

  – *ApiMac_attribute_beaconPayload*: The contents of the beacon payload

  – *ApiMac_attribute_coordExtendedAddress*: The extended address of the coordinator with which the device is associated

  – *ApiMac_attribute_extendedAddress*: The extended address of the device

- enum ApiMac_securityAttribute_uint8_t

  Security PIB Get and Set attributes – size uint8_t

  Enumerator

  – *ApiMac_securityAttribute_keyTableEntries*: The number of entries in macKeyTable

  – *ApiMac_securityAttribute_deviceTableEntries*: The number of entries in macDeviceTable

  – *ApiMac_securityAttribute_securityLevelTableEntries*: The number of entries in macSecurityLevelTable

  – *ApiMac_securityAttribute_autoRequestSecurityLevel*: The security level used for automatic data requests

- *ApiMac_securityAttribute_autoRequestKeyIdMode*: The key identifier mode used for automatic data requests
- *ApiMac_securityAttribute_autoRequestKeyIndex*: The index of the key used for automatic data requests

- enum ApiMac_securityAttribute_uint16_t

  Security PIB Get and Set attributes – size uint16_t

  Enumerator

  - *ApiMac_securityAttribute_panCoordShortAddress*: The 16-bit short address assigned to the PAN coordinator

- enum ApiMac_securityAttribute_array_t

  Security PIB Get and Set attributes – array of bytes

  Enumerator

  - *ApiMac_securityAttribute_autoRequestKeySource*: The originator of the key used for automatic data requests
  - *ApiMac_securityAttribute_defaultKeySource*: The originator of the default key used for key ID mode 0x01
  - *ApiMac_securityAttribute_panCoordExtendedAddress*: The 64-bit address of the PAN coordinator

- enum ApiMac_securityAttribute_struct_t

  Security PIB Get and Set attributes – these attributes are structures

  Enumerator

  - *ApiMac_securityAttribute_keyTable*: A table of KeyDescriptor entries, each containing keys and related information required for secured communications. This is a SET-only attribute. Call ApiMac_mlmeSetSecurityReqStruct() with pValue set to NULL, for the MAC to build the table.
  - *ApiMac_securityAttribute_keyIdLookupEntry*: The key lookup table entry, and part of an entry of the key table. To GET or SET to this attribute, setup the keyIndex and keyIdLookupIndex fields of ApiMac_securityPibKeyIdLookupEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibKeyIdLookupEntry_t structure. For the GET, the lookupEntry field contains the required data.
  - *ApiMac_securityAttribute_keyDeviceEntry*: The key device entry, and part of an entry of the key table. To GET or SET to this attribute, setup the keyIndex and keyDeviceIndex fields of ApiMac_securityPibKeyDeviceEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibKeyDeviceEntry_t structure. For the GET, the deviceEntry field contains the required data.
  - *ApiMac_securityAttribute_keyUsageEntry*: The key usage entry, and part of an entry of the key table. To GET or SET to this attribute, setup the keyIndex and keyUsageIndex fields of ApiMac_securityPibKeyUsageEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibKeyUsageEntry_t structure. For the GET, the usageEntry field contains the required data.
  - *ApiMac_securityAttribute_keyEntry*: The MAC key entry, and an entry of the key table. To GET or SET to this attribute, setup the keyIndex field of ApiMac_securityPibKeyEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibKeyEntry_t structure. For the GET, the rest of the fields contain the required data.
  - *ApiMac_securityAttribute_deviceEntry*: The MAC device entry, and an entry of the device table. To GET or SET to this attribute, setup the deviceIndex field of ApiMac_securityPibDeviceEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibDeviceEntry_t structure. For the GET, the deviceEntry field contains the required data.
  - *ApiMac_securityAttribute_securityLevelEntry*: The MAC security level entry, and an entry of the security level table. To GET or SET to this attribute, setup the levelIndex field of ApiMac_securityPibSecurityLevelEntry_t, call ApiMac_mlmeGetSecurityReqStruct() or ApiMac_mlmeSetSecurityReqStruct() with a pointer to the ApiMac_securityPibSecurityLevelEntry_t

structure. For the GET, the levelEntry field contains the required data.

- enum ApiMac_FHAttribute_uint8_t

  Frequency-hopping PIB Get and Set attributes – size uint8_t

  Enumerator

  – *ApiMac_FHAttribute_unicastDwellInterval*: Duration of the unicast slot of the node (in ms) – uint8_t
  – *ApiMac_FHAttribute_broadcastDwellInterval*: Duration of the broadcast slot of the node (in ms) – uint8_t
  – *ApiMac_FHAttribute_clockDrift*: Clock drift in PPM – uint8_t
  – *ApiMac_FHAttribute_timingAccuracy*: Timing accuracy in 10-µs resolution – uint8_t
  – *ApiMac_FHAttribute_unicastChannelFunction*: Unicast channel-hopping function – uint8_t
  – *ApiMac_FHAttribute_broadcastChannelFunction*: Broadcast channel-hopping function – uint8_t
  – *ApiMac_FHAttribute_useParentBSIE*: Node is propagating parent BS-IE – uint8_t
  – *ApiMac_FHAttribute_routingCost*: Estimate of routing path ETX to the PAN coordinator – uint8_t
  – *ApiMac_FHAttribute_routingMethod*: RPL(1), MHDS(0) – uint8_t
  – *ApiMac_FHAttribute_eapolReady*: Node can accept EAPOL message – uint8_t
  – *ApiMac_FHAttribute_fanTPSVersion*: Wi-SUN FAN version – uint8_t
  – *ApiMac_FHAttribute_numNonSleepDevice*: Number of non-sleepy device – uint8_t
  – *ApiMac_FHAttribute_numSleepDevice*: Number of sleepy device – uint8_t

- enum ApiMac_FHAttribute_uint16_t

  Frequency-hopping PIB Get and Set attributes – size uint16_t

  Enumerator

  – *ApiMac_FHAttribute_broadcastSchedId*: Broadcast schedule ID for broadcast channel-hopping sequence – uint16_t
  – *ApiMac_FHAttribute_unicastFixedChannel*: Unicast channel number when no hopping – uint16_t
  – *ApiMac_FHAttribute_broadcastFixedChannel*: Broadcast channel number when no hopping – uint16_t
  – *ApiMac_FHAttribute_panSize*: Number of nodes in the PAN – uint16_t
  – *ApiMac_FHAttribute_panVersion*: PAN version to notify PAN configuration changes – uint16_t
  – *ApiMac_FHAttribute_neighborValidTime*: Time in minutes during which the node info considered as valid – uint16_t

- enum ApiMac_FHAttribute_uint32_t

  Frequency-hopping PIB Get and Set attributes – size uint32_t

  Enumerator

  – *ApiMac_FHAttribute_BCInterval*: Time between the start of two broadcast slots (in ms) – uint32_t

- enum ApiMac_FHAttribute_array_t

  Frequency-hopping PIB Get and Set attributes – array of bytes

  Enumerator

  – *ApiMac_FHAttribute_trackParentEUI*: The parent EUI address – ApiMac_sAddrExt_t
  – *ApiMac_FHAttribute_unicastExcludedChannels*: Unicast excluded channels – APIMAC_FH_MAX_BIT_MAP_SIZE
  – *ApiMac_FHAttribute_broadcastExcludedChannels*: Broadcast excluded channels – APIMAC_FH_MAX_BIT_MAP_SIZE
  – *ApiMac_FHAttribute_netName*: Network name – APIMAC_FH_NET_NAME_SIZE_MAX uint8_t
  – *ApiMac_FHAttribute_gtk0Hash*: Low order 64 bits of SHA256 hash of GTK
  – *ApiMac_FHAttribute_gtk1Hash*: Next low order 64 bits of SHA256 hash of GTK
  – *ApiMac_FHAttribute_gtk2Hash*: Next low order 64 bits of SHA256 hash of GTK
  – *ApiMac_FHAttribute_gtk3Hash*: Next low order 64 bits of SHA256 hash of GTK

- enum ApiMac_fhFrameType_t
  FH frame types
  Enumerator
  – *ApiMac_fhFrameType_panAdvert*: WiSUN PAN advertisement
  – *ApiMac_fhFrameType_panAdvertSolicit*: WiSUN PAN advertisement solicit
  – *ApiMac_fhFrameType_config*: WiSUN PAN config
  – *ApiMac_fhFrameType_configSolicit*: WiSUN PAN config solicit
  – *ApiMac_fhFrameType_data*: WiSUN Data frame
  – *ApiMac_fhFrameType_ack*: WiSUN Ack frame
  – *ApiMac_fhFrameType_eapol*: WiSUN Ack frame
  – *ApiMac_fhFrameType_invalid*: Internal – WiSUN Invalid frame
- enum ApiMac_payloadIEGroup_t
  Payload IE group IDs
  Enumerator
  – *ApiMac_payloadIEGroup_ESDU*: Payload ESDU IE Group ID
  – *ApiMac_payloadIEGroup_MLME*: Payload MLME IE Group ID
  – *ApiMac_payloadIEGroup_WiSUN*: Payload WiSUN IE Group ID
  – *ApiMac_payloadIEGroup_term*: Payload Termination IE Group ID
- enum ApiMac_MLMESubIE_t
  MLME Sub IEs
  Enumerator
  – *ApiMac_MLMESubIE_coexist*: MLME Sub IEs – short format – coexistence IE
  – *ApiMac_MLMESubIE_sunDevCap*: MLME Sub IEs – short format – SUN device capabilities IE
  – *ApiMac_MLMESubIE_sunFSKGenPhy*: MLME Sub IEs – short format – SUN FSK generic PHY IE
- enum ApiMac_wisunSubIE_t
  WiSUN Sub IEs
  Enumerator
  – *ApiMac_wisunSubIE_USIE*: WiSUN Sub IE – long format – unicast schedule IE
  – *ApiMac_wisunSubIE_BSIE*: WiSUN Sub IE – long format – broadcast schedule IE
  – *ApiMac_wisunSubIE_PANIE*: WiSUN Sub IE – short format – PAN IE
  – *ApiMac_wisunSubIE_netNameIE*: WiSUN Sub IE – short format – network name IE
  – *ApiMac_wisunSubIE_PANVersionIE*: WiSUN Sub IE – short format – PAN version IE
  – *ApiMac_wisunSubIE_GTKHashIE*: WiSUN Sub IE – short format – GTK hash IE
- enum ApiMac_scantype_t
  Scan types
  Enumerator
  – *ApiMac_scantype_energyDetect*: Energy detect scan. The device tunes to each channel and performs an energy measurement. The list of channels and their associated measurements is returned at the end of the scan.
  – *ApiMac_scantype_active*: Active scan. The device tunes to each channel, sends a beacon request, and listens for beacons. The PAN descriptors are returned at the end of the scan.
  – *ApiMac_scantype_passive*: Passive scan. The device tunes to each channel and listens for beacons. The PAN descriptors are returned at the end of the scan.
  – *ApiMac_scantype_orphan*: Orphan scan. The device tunes to each channel and sends an orphan notification to try and find its coordinator. The status is returned at the end of the scan.
  – *ApiMac_scantype_activeEnhanced*: Enhanced active scan. In addition to active scan, this command is also used by a device to locate a subset of all coordinators within its POS during an

active scan.

- enum ApiMac_wisunAsycnOperation_t

  WiSUN Async operations

  Enumerator

  – *ApiMac_wisunAsycnOperation_start*: Start Async
  – *ApiMac_wisunAsycnOperation_stop*: Stop Async

- enum ApiMac_wisunAsyncFrame_t

  WiSUN Async frame types

  Enumerator

  – *ApiMac_wisunAsyncFrame_advertisement*: WiSUN Async PAN advertisement frame type
  – *ApiMac_wisunAsyncFrame_advertisementSolicit*: WiSUN Async PAN advertisement solicitation frame type
  – *ApiMac_wisunAsyncFrame_config*: WiSUN Async PAN configuration frame type
  – *ApiMac_wisunAsyncFrame_configSolicit*: WiSUN Async PAN configuration solicitation frame type

- enum ApiMac_fhDispatchType_t

  Frequency-hopping dispatch values

  Enumerator

  – *ApiMac_fhDispatchType_none*: No protocol dispatch
  – *ApiMac_fhDispatchType_MHD_PDU*: MHD-PDU protocol dispatch
  – *ApiMac_fhDispatchType_6LowPAN*: 6LowPAN protocol dispatch

13.7 Function Documentation

void* ApiMac_init (bool enableFH) *Initialize this module.*

Parameters enableFH – True to enable frequency hopping, false to not.

Returns Pointer to a wakeup variable (semaphore in some systems)

void ApiMac_registerCallbacks (ApiMac_callbacks_t * pCallbacks) *Register for MAC callbacks.*

Parameters pCallbacks – Pointer to callback structure

void ApiMac_processIncoming (void) *Process incoming messages from the MAC stack.*

Parameters TBD

ApiMac_status_t ApiMac_mcpsDataReq (ApiMac_mcpsDataReq_t * pData) *This function sends application data to the MAC for transmission in a MAC data frame. The MAC can only buffer a certain number of data request frames. When the MAC is congested and cannot accept the data request, it initiates a callback (ApiMac_dataCnfFp_t) with an overflow status (ApiMac_status_transactionOverflow) . Eventually the MAC will become uncongested and initiate the callback (ApiMac_dataCnfFp_t) for a buffered request. At this point, the application can attempt another data request. Using this scheme, the application can send data at any time, but it must queue data to be resent if it receives an overflow status.*

Parameters pData – Pointer to parameter structure

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mcpsPurgeReq (uint8_t msduHandle) *This function purges and discards a data request from the MAC data queue. When the operation is complete, the MAC sends a MCPS Purge Confirm, which initiates a callback (ApiMac_purgeCnfFp_t).*

Parameters msduHandle – The application-defined handle value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeAssociateReq (ApiMac_mlmeAssociateReq_t * pData) *This function sends an associate request to a coordinator device. The application tries to associate only with a PAN that is currently allowing association, as indicated in the results of the scanning procedure. In a beacon-enabled PAN, the beacon order must be set by using ApiMac_mlmeSetReq() before making the call to ApiMac_mlmeAssociateReq(). When the associate request is complete, the application receives the ApiMac_associateCnfFp_t callback.*

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeAssociateRsp (ApiMac_mlmeAssociateRsp_t * pData) *This function sends an associate response to a device requesting to associate. This function must be called after the ApiMac_associateIndFp_t callback. When the associate response is complete, the callback ApiMac_commStatusIndFp_t is called to indicate the success or failure of the operation.*

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeDisassociateReq (ApiMac_mlmeDisassociateReq_t * pData) *This function is used by an associated device to notify the coordinator of its intent to leave the PAN. It is also used by the coordinator to instruct an associated device to leave the PAN. When the disassociate procedure is complete, the applications callback ApiMac_disassociateCnfFp_t is called.*

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeGetReqBool (ApiMac_attribute_bool_t pibAttribute, bool * pValue) *This direct execute function retrieves an attribute value from the MAC PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeGetReqUint8 (ApiMac_attribute_uint8_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeGetReqUint16 (ApiMac_attribute_uint16_t pibAttribute, uint16_t * pValue) *This direct execute function retrieves an attribute value from the MAC PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeGetReqUint32 (ApiMac_attribute_uint32_t pibAttribute, uint32_t * pValue) *This direct execute function retrieves an attribute value from the MAC PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeGetReqArray (ApiMac_attribute_array_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeGetFhReqUint8 (ApiMac_FHAttribute_uint8_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC frequency-hopping PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetFhReqUint16 (ApiMac_FHAttribute_uint16_t pibAttribute, uint16_t * pValue) *This direct execute function retrieves an attribute value from the MAC frequency-hopping PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetFhReqUint32 (ApiMac_FHAttribute_uint32_t pibAttribute, uint32_t * pValue) *This direct execute function retrieves an attribute value from the MAC frequency-hopping PIB.*

Parameters pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetFhReqArray (ApiMac_FHAttribute_array_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC frequency-hopping PIB.*

Parameters

pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns

The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetSecurityReqUint8 (ApiMac_securityAttribute_uint8_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC security PIB.*

Parameters

pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns

The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetSecurityReqUint16 (ApiMac_securityAttribute_uint16_t pibAttribute, uint16_t * pValue) *This direct execute function retrieves an attribute value from the MAC security PIB.*

Parameters

pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns

The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetSecurityReqArray (ApiMac_securityAttribute_array_t pibAttribute, uint8_t * pValue) *This direct execute function retrieves an attribute value from the MAC security PIB.*

Parameters

pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns

The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeGetSecurityReqStruct (ApiMac_securityAttribute_struct_t pibAttribute, void * pValue) *This direct execute function retrieves an attribute value from the MAC security PIB.*

Parameters

pibAttribute – The attribute identifier

pValue – Pointer to the attribute value

Returns

The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeOrphanRsp (ApiMac_mlmeOrphanRsp_t * pData) ***This function is called in response to an orphan notification from a peer device. This function must be called after receiving an Orphan Indication Callback. When the orphan response is complete, the Comm Status Indication Callback is called to indicate the success or failure of the operation.***

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmePollReq (ApiMac_mlmePollReq_t * pData) ***This function is used to request pending data from the coordinator. When the poll request is complete, the Poll Confirm Callback is called. If a data frame of nonzero length is received from the coordinator, the Poll Confirm Callback has a status ApiMac_status_success, then calls the Data Indication Callback for the received data.***

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeResetReq (bool setDefaultPib) ***This direct execute function resets the MAC. This function must be called once at system startup before any other function in the management API is called.***

Parameters setDefaultPib – Set to TRUE to reset the MAC PIB to its default values.

Returns Always ApiMac_status_success

ApiMac_status_t ApiMac_mlmeScanReq (ApiMac_mlmeScanReq_t * pData) ***This function initiates an energy detect, active, passive, or orphan scan on one or more channels. An energy detect scan measures the peak energy on each requested channel. An active scan sends a beacon request on each channel and then listening for beacons. A passive scan is a receive-only operation that listens for beacons on each channel. An orphan scan is used to locate the coordinator with which the scanning device had previously associated. When a scan operation is complete, the Scan Confirm callback is called.***

For active or passive scans, the application sets the maxResults parameter the maximum number of PAN descriptors to return. If maxResults is greater than zero, the application must also set result.panDescriptor to point to a buffer of size maxResults * sizeof(ApiMac_panDesc_t) to store the results of the scan. The application must not access or deallocate this buffer until the Scan Confirm Callback is called. The MAC stores up to maxResults PAN descriptors and ignores duplicate beacons.

An alternative way to get results for an active or passive scan is to set maxResults to zero or set PIB attribute ApiMac_attribute_autoRequest to FALSE. Then the MAC will not store results, but rather call the Beacon Notify Indication Callback" for each beacon received. The application does not need to supply any memory to store the scan results, but the MAC does not filter out duplicate beacons.

For energy detect scans, the application must set result.energyDetect to point to a buffer of size 18 bytes to store the results of the scan. The application must not access or deallocate this buffer until the Scan Confirm Callback is called.

An energy detect, active, or passive scan may be performed at any time if a scan is not already in progress. However, a device cannot perform any other MAC management operation or send or receive MAC data until the scan is complete.

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_scanInProgress – Already scanning
- ApiMac_status_noResources – Memory allocation error

ApiMac_status_t ApiMac_mlmeSetReqBool (ApiMac_attribute_bool_t pibAttribute, bool value) ***This direct execute function sets an attribute value in the MAC PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeSetReqUint8 (ApiMac_attribute_uint8_t pibAttribute, uint8_t value) ***This direct execute function sets an attribute value in the MAC PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeSetReqUint16 (ApiMac_attribute_uint16_t pibAttribute, uint16_t value) ***This direct execute function sets an attribute value in the MAC PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeSetReqUint32 (ApiMac_attribute_uint32_t pibAttribute, uint32_t value) ***This direct execute function sets an attribute value in the MAC PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeSetReqArray (ApiMac_attribute_array_t pibAttribute, uint8_t * pValue) ***This direct execute function sets an attribute value in the MAC PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request

ApiMac_status_t ApiMac_mlmeSetFhReqUint8 (ApiMac_FHAttribute_uint8_t pibAttribute, uint8_t value) ***This direct execute function sets an attribute value in the MAC frequency-hopping PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetFhReqUint16 (ApiMac_FHAttribute_uint16_t pibAttribute, uint16_t value) ***This direct execute function sets an attribute value in the MAC frequency-hopping PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetFhReqUint32 (ApiMac_FHAttribute_uint32_t pibAttribute, uint32_t value) ***This direct execute function sets an attribute value in the MAC frequency-hopping PIB.***

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetFhReqArray (ApiMac_FHAttribute_array_t pibAttribute, uint8_t * pValue) *This direct execute function sets an attribute value in the MAC frequency-hopping PIB.*

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetSecurityReqUint8 (ApiMac_securityAttribute_uint8_t pibAttribute, uint8_t value) *This direct execute function sets an attribute value in the MAC security PIB.*

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetSecurityReqUint16 (ApiMac_securityAttribute_uint16_t pibAttribute, uint16_t value) *This direct execute function sets an attribute value in the MAC security PIB.*

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetSecurityReqArray (ApiMac_securityAttribute_array_t pibAttribute, uint8_t * pValue) *This direct execute function sets an attribute value in the MAC security PIB.*

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupportedAttribute – Attribute not found

ApiMac_status_t ApiMac_mlmeSetSecurityReqStruct (ApiMac_securityAttribute_struct_t pibAttribute, void * pValue) *This direct execute function sets an attribute value in the MAC security PIB.*

Parameters pibAttribute – The attribute identifier

value – The attribute value

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeStartReq (ApiMac_mlmeStartReq_t * pData) ***This function is called by a coordinator or PAN coordinator to start or reconfigure a network. Before starting a network, the device must have set its short address. A PAN coordinator sets the short address by setting the attribute ApiMac_attribute_shortAddress. A coordinator sets the short address through association.***

When parameter panCoordinator is TRUE, the MAC automatically sets attributes ApiMac_attribute_panID and ApiMac_attribute_logicalChannel to the panId and logicalChannel parameters. If panCoordinator is FALSE, these parameters are ignored (they would already be set through association).

The parameter beaconOrder controls whether the network is beacon-enabled or nonbeacon-enabled. For a beacon-enabled network, this parameter also controls the beacon transmission interval. When the operation is complete, the Start Confirm Callback is called.

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeSyncReq (ApiMac_mlmeSyncReq_t * pData) ***This function requests the MAC to synchronize with the coordinator by acquiring and optionally tracking its beacons. Synchronizing with the coordinator is recommended before associating in a beacon-enabled network. If the beacon could not be located on its initial search or during tracking, the MAC calls the Sync Loss Indication Callback with ApiMac_status_beaconLoss as the reason.***

Before calling this function, the application must set PIB attributes ApiMac_attribute_beaconOrder, ApiMac_attribute_panId, and either ApiMac_attribute_coordShortAddress or ApiMac_attribute_coordExtendedAddress to the address of the coordinator with which to synchronize.

The application may wish to set PIB attribute ApiMac_attribute_autoRequest to FALSE before calling this function. Then, when the MAC successfully synchronizes with the coordinator, it calls the Beacon Notify Indication Callback. After receiving the callback, the application may set ApiMac_attribute_autoRequest to TRUE to stop receiving beacon notifications. This function is only applicable to beacon-enabled networks.

Parameters pData – Pointer to parameters structure

Returns The status of the request, as follows:

- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

uint8_t ApiMac_randomByte (void) ***This function returns a random byte from the MAC random number generator.***

Returns A random byte

ApiMac_status_t ApiMac_updatePanId (uint16_t panId) ***Updates device table entry and PIB with new PAN ID.***

Parameters panID – The new PAN ID

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_mlmeWSAsyncReq (ApiMac_mlmeWSAsyncReq_t * pData) ***This functions handles a WiSUN async request. The possible operation is Async Start or Async Stop. For the async start operation, the caller of this function can indicate the WiSUN async frame type to be sent on the specified channels.***

Parameters pData – Pointer to the asynchronous parameters structure

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_startFH (void) ***This function starts the frequency hopping. Frequency-hopping operations should have been enabled using ApiMac_enableFH() before calling this API. Do not call this API if ApiMac_mlmeStartReq() has been called with the startFH field set to true.***

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noResources – Resources not available

ApiMac_status_t ApiMac_parsePayloadGroupIEs (uint8_t * pPayload, uint16_t payloadLen, ApiMac_payloadIeRec_t ** pList) ***Parses the Group payload information element. This function creates a linked list (plist) from the Payload IE (pPayload). Each item in the linked list is a separate Group IE with its own content.***

If no IEs are found, pList is set to NULL.

The caller is responsible to release the memory for the linked list by calling ApiMac_freeIEList(). Call this function to create the list of Group IEs, then call ApiMac_parsePayloadSubIEs() to parse each of the group IE's content into sub IEs.

Parameters pPayload – Pointer to the buffer with the payload IEs

payloadLen – Length of the buffer with the payload IEs

pList – Pointer to link list pointer

Returns The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noData – pPayload or payloadLen is NULL

- ApiMac_status_unsupported – Invalid field found
- ApiMac_status_noResources – If memory allocation fails

ApiMac_status_t ApiMac_parsePayloadSubIEs (uint8_t * pContent, uint16_t contentLen, ApiMac_payloadIeRec_t ** pList) *Parses the payload sub information element. This function creates a linked list (pList) of sub IEs from the Group IE content (pContent). Each item in the linked list is a separate sub IE with its own content.*

If no IEs are found pList will be set to NULL.

The caller is responsible to release the memory for the linked list by calling ApiMac_freeIEList(). Call this function after calling ApiMac_parsePayloadGroupIEs().

Parameters

pContent – Pointer to the buffer with the sub IEs

contentLen – Length of the buffer with the sub IEs

pList – Pointer to link list pointer

Returns

The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_noData – pContent or contentLen is NULL
- ApiMac_status_unsupported – Invalid field found
- ApiMac_status_noResources – If memory allocation fails

void ApiMac_freeIEList (ApiMac_payloadIeRec_t * pList) *Frees the linked list allocated by ApiMac_parsePayloadGroupIEs() or ApiMac_parsePayloadSubIEs().*

Parameters

pList – Pointer to the linked list

ApiMac_status_t ApiMac_enableFH (void) *Enables the frequency-hopping operation. Call this function before setting any FH parameters, or before calling ApiMac_mlmeStartReq() or ApiMac_startFH(), if using FH.*

Returns

The status of the request, as follows:
- ApiMac_status_success – Operation successful
- ApiMac_status_unsupported – Feature not available.

uint8_t ApiMac_convertCapabilityInfo (ApiMac_capabilityInfo_t * pMsgcapInfo) *Converts ApiMac_capabilityInfo_t data type to uint8 capInfo.*

Parameters

pMsgcapInfo – CapabilityInfo pointer

Returns

capInfo bit mask byte

void ApiMac_buildMsgCapInfo (uint8_t cInfo, ApiMac_capabilityInfo_t * pPBcapInfo) *Converts from bitmask byte to API MAC capInfo.*

Parameters

cInfo – Source

pPBcapInfo – Destination

ApiMac_status_t ApiMac_secAddDevice (ApiMac_secAddDevice_t * pAddDevice) *Adds a new MAC device table entry.*

Parameters pAddDevice – Add device information

Returns ApiMac_status_success if successful, other status value if not.

ApiMac_status_t ApiMac_secDeleteDevice (ApiMac_sAddrExt_t * pExtAddr) ***Removes a MAC device table entry.***

Parameters pExtAddr – Extended address of the device table entries to be removed

Returns ApiMac_status_success if successful, other status value if not.

ApiMac_status_t ApiMac_secDeleteKeyAndAssocDevices (uint8_t keyIndex) ***Removes the key at the specified key index, and removes all MAC device table entries associated with this key. Also removes(initializes) the key lookup list associated with this key.***

Parameters keyIndex – MAC security key table index of the key to be removed.

Returns ApiMac_status_success if successful, other status value if not.

ApiMac_status_t ApiMac_secDeleteAllDevices (void) ***Removes all MAC device table entries.***

Returns ApiMac_status_success if successful, other status value if not.

ApiMac_status_t ApiMac_secGetDefaultSourceKey (uint8_t keyId, uint32_t * pFrameCounter) ***Reads the frame counter value associated with a MAC security key indexed by the designated key identifier and the default key source.***

Parameters keyID – Key ID

pFrameCounter – Pointer to a buffer to store the outgoing frame counter of the key.

Returns ApiMac_status_success if successful, other status value if not.

ApiMac_status_t ApiMac_secAddKeyInitFrameCounter (ApiMac_secAddKeyInitFrameCounter_t * pInfo) ***Adds the MAC security key, adds the associated lookup list for the key, and initializes the frame counter to the value provided. It also duplicates the device table entries (associated with the previous key if any) if available, based on the flag dupDevFlag value, and associates the device descriptor with this key.***

Parameters pInfo – Structure needed to perform this function

Returns ApiMac_status_success if successful, other status value if not.

ICALL API

14.1 Commands

The ICall commands used for application tasks are defined in Section 5.3.

14.2 Error Codes

Table 14-1 lists the error codes associated with ICall failures. These codes can be returned from any function defined in icall.h.

Table 14-1. Error Codes

Value	Error Name	Description
0x04	MSG_BUFFER_NOT_AVAIL	Allocation of ICall message failed
0xFF	ICALL_ERRNO_INVALID_SERVICE	The service corresponding to a passed service ID is not registered
0xFE	ICALL_ERRNO_INVALID_FUNCTION	The function ID is unknown to the registered handler of the service
0xFD	ICALL_ERRNO_INVALID_PARAMETER	Invalid parameter value
0xFC	ICALL_ERRNO_NO_RESOURCE	Not available entities, tasks, or other ICall resources
0xFB	ICALL_ERRNO_UNKNOWN_THREAD	The task is not a registered task of the entity; ID is not a registered entity
0xFA	ICALL_ERRNO_CORRUPT_MSG	Corrupt message error
0xF9	ICALL_ERRNO_OVERFLOW	Counter overflow
0xF8	ICALL_ERRNO_UNDERFLOW	Counter underflow

References

- **[1]** TI SYS/BIOS API Guide
  C:\ti\ tirtos_cc13xx_cc26xx_2_16_01_14\products\ bios_6_45_02_31 \docs\Bios_APIs.html
- **[2]** TI-RTOS Power Management
  C:\ti\ tirtos_cc13xx_cc26xx_2_16_01_14 \docs\Power_Management.pdf
- **[3]** TI-15.4MAC Wiki
- **[4]** TI-RTOS API Reference
  C:/ti/tirtos_cc13xx_cc26xx_2_16_01_14/docs/Documentation_Overview_cc13xx_cc26xx.html
- **[5]** Sensor Controller Studio
- **[6]** ARM Cortex-M3 Devices Generic User's Guide
  http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf
- **[7]** TI-RTOS SYS/BIOS Kernel User's Guide
  C:\ti\tirtos_cc13xx_cc26xx_2_16_01_14\products\bios_6_45_02_31\docs\Bios_User_Guide.pdf
- **[8]** Wi-SUN FAN Specification
  Wi-SUN FAN Specification, version 1v00, May 2016, Wi-SUN Alliance

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (September 2016) to A Revision **Page**

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265