Atmel AVR2050: Atmel BitCloud

Developer Guide

8199H-MCU Wireless-04/12

Table of contents

Section 1
Introduction

The Atmel® BitCloud® full-featured, next-generation embedded software stack provides a firmware development platform for reliable, scalable, and secure wireless applications running on Atmel hardware kits. The BitCloud stack is designed to support a broad ecosystem of user designed applications addressing diverse requirements and enabling a full spectrum of software customization. Primary application domains include home automation, commercial building automation, automated meter reading, asset tracking, and industrial automation.

The BitCloud stack is fully compliant with ZigBee® PRO and ZigBee standards for wireless sensing and control. It provides an augmented set of APIs which, while maintaining compliance with the standard, offer extended functionality designed with developers' convenience and ease-of-use in mind. As seasoned ZigBee technology experts, Atmel engineers created the BitCloud stack to dramatically lower the developer learning curve, factor out the unnecessary complexity, and expose as much power of the underlying hardware platform as possible.

The BitCloud target audience is system designers, embedded programmers, and hardware engineers evaluating, prototyping, and deploying wireless solutions and products. The BitCloud stack is delivered as a software development kit, which includes:

1. Extensive documentation.
2. A standard set of libraries comprising multiple components of the stack.
3. Sample user applications in source code.
4. A complete set of peripheral drivers for the supported platforms.

1.1 Intended audience and purpose

The purpose of this document is to familiarize the audience of embedded software developers and system designers with the BitCloud SDK. The document covers the following topics:

1. BitCloud stack architecture.
2. User application programming model.
3. Memory and resource allocation.
4. System design considerations.
5. Walkthrough of key APIs with code samples and commentary.

The audience is assumed to be familiar with the C programming language. Some knowledge of embedded systems is recommended, but not required. The audience is assumed to be familiar with key aspects of the ZigBee and ZigBee PRO standards for low-power wireless networking [3]. You may refer to online tutorials at http://www.zigbee.org/LearnMore/WhitePapers.aspx for more information on ZigBee fundamentals.

1.2 Key features

- Full ZigBee PRO and ZigBee compliance

- Easy-to-use C API and serial AT commands available
- Ultimate in data reliability with true mesh routing
- Large network support (hundreds of devices)
- Optimized for ultra-low power consumption (5 to 15 years battery life)
- Extensive security API
- Software update capability
- Flexible and easy-to-use developer tools

1.3 Supported platforms

Different releases of the Atmel BitCloud SDK support a wide range of hardware platforms. Detailed information on each platform can be found in the Atmel AVR2052: Atmel BitCloud Quick Start Guide application note.

1.4 Related documents

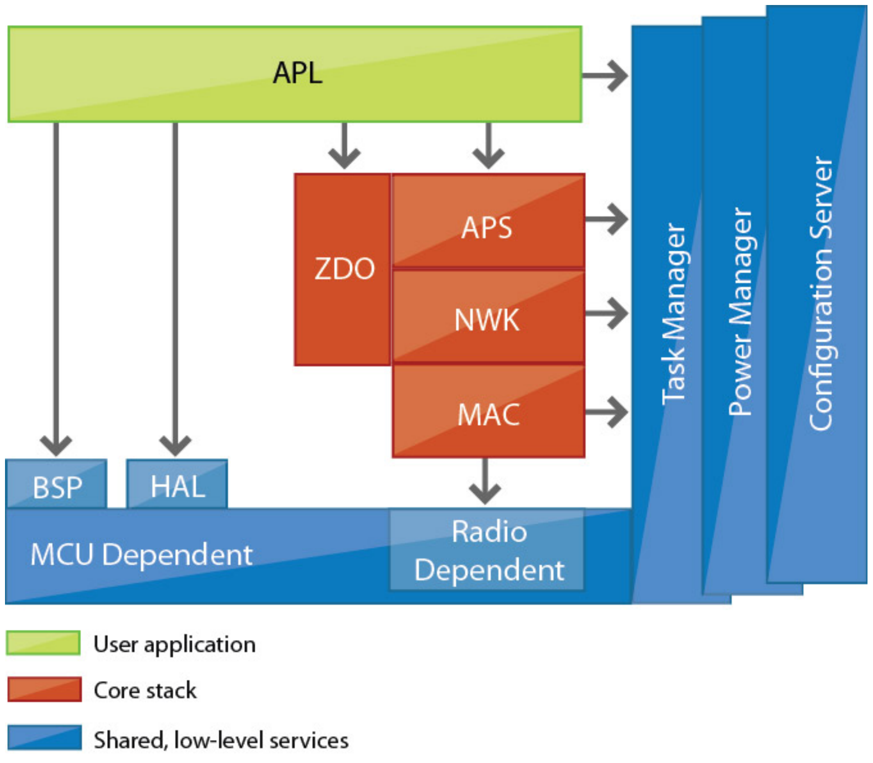
1. GNU 'make.' http://www.gnu.org/software/make/manual/make.html.
2. make (software). http://en.wikipedia.org/wiki/Make_(software).
3. 053474r17ZB_TSC-ZigBee stack profile Pro Specification.
4. Atmel AVR2052: Atmel BitCloud Quick Start Guide.
5. BitCloud Stack API Reference.
6. AVR2059: BitCloud Porting Guide.
7. AVR2058: BitCloud OTAU User Guide.
8. AVR2061: BitCloud ZLL Developer Guide

Section 2
BitCloud architecture

2.1 Architecture highlights

The Atmel BitCloud internal architecture follows the suggested separation of the network stack into logical layers, as found in IEEE® 802.15.4 and ZigBee. Besides the core stack containing protocol implementation, the BitCloud architecture contains additional layers implementing shared services (for example, task manager, security, and power manager) and hardware abstractions (for example, hardware abstraction layer (HAL) and board support package (BSP)). The APIs contributed by these layers are outside the scope of core stack functionality. However, these essential additions to the set of APIs significantly help reduce application complexity and simplify integration. The BitCloud API Reference manual provides detailed information on all public APIs and their use [5].

Figure 2-1. BitCloud **s**oftware stack architecture.



The topmost of the core stack layers, APS, provides the highest level of networking-related API visible to the application. ZDO provides a set of fully compliant ZigBee Device Object APIs, which enable main network management functionality (start, reset, formation, join). ZDO also implements ZigBee Device Profile commands, including Device Discovery and Service Discovery.

There are three vertical service components responsible for configuration management, task management, and power down control. These services are available to the user application, and may also be utilized by lower stack layers.

Configuration server (CS) is used to manage the configuration parameters provided with the Atmel BitCloud stack.

Task manager is the stack scheduler that mediates the use of the MCU among internal stack components and the user application. The task manager implements a priority based cooperative scheduler specifically tuned for the multi-layer stack environment and demands of time-critical network protocols. Section 3.2 describes the task scheduler and its interface in more detail.

Power management routines are responsible for gracefully shutting down all stack components and saving system state when preparing to sleep and restoring system state when waking up.

Hardware abstraction layer (HAL) includes a complete set of APIs for using on-module hardware resources (EEPROM, sleep, and watchdog timers) as well as the reference drivers for rapid designing and smooth integration with a range of external peripherals (IRQ, TWI, SPI, USART, and 1-wire).

Board support package (BSP) includes a complete set of drivers for managing standard peripherals (sensors, UID chip, sliders, and buttons) placed on a development board.

2.2 Naming conventions

Due to a high number of API functions exposed to the user, a simple naming convention is employed to simplify the task of getting around and understanding user applications. Here are the basic rules:

1. Each API function name is prefixed by the name of the layer where the function resides. For example, the ZDO_GetLqiRssi API function is contributed by the ZDO layer of the stack.
2. Each function name prefix is followed by an underscore character separating the prefix from the descriptive function name.
3. The descriptive function name may have a *Get* or *Set* prefix, indicating that some parameter is returned or set in the underlying layer, respectively (for example, HAL_GetSystemTime).
4. The descriptive function name may have a *Req*, *Request*, *Ind*, or *Conf* suffix, indicating the following:
   - *Req* and *Request* correspond to the asynchronous (see Section 3.1.1) requests from the user application to the stack (for example, APS_DataReq)
   - *Ind* corresponds to the asynchronous indication of events propagated to the user application from the stack (for example, ZDO_NetworkLostInd)
   - By convention, function names ending in *Conf* are user-defined callbacks for the asynchronous requests, which confirm the request's execution
5. Each structure and type name carries a *_t* suffix, standing for type.
6. Enumeration and macro variable names are in capital letters.

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

2.3 File system layout

The file system layout mirrors the stack architecture, with extra folders constituting sublayers added. Those components distributed in binary form include 'lib' and 'include' directories, which describe the component interface header files and contain the library image, respectively. Those components distributed in source code include 'objs' and 'src' subdirectories, and a makefile containing the build script that builds that component ([1], [2]).

The applications are always provided as header and source files, the makefile build script, as well as project files for supported IDEs. The main action of the application makefile is to compile the application using the header files found under the respective `Component` directories and link the resulting object file with library images (also found under `Components` directory). The resulting binary image is a cross compiled application, which can then be programmed and debugged on the target platform. Further information on setting up the developer build environment and tool chain can be found in [4].

Table 2-1. Atmel BitCloud SDK file system layout.

Folder	Source code	Description
BitCloud		Stack root
Components		
APS	no	Application support sublayer
BSP	yes	Board support package (reference drivers for supported evaluation and development boards)
ConfigServer	yes	Generic parameter storage sublayer
HAL	yes	Hardware abstraction layer (reference drivers for supported platforms)
MAC_PHY	no	Media access control and physical layers
NWK	no	Network layer
PersistDataServer	yes	EEPROM access and persist parameters management
Security	no	Security services
SystemEnvironment	no	Main function and task manager
ZDO	no	ZigBee Device Object sublayer
WSNDemo	yes	Full-featured WSN application demonstrating data acquisition, security, and power management

Section 3
User application programming

The goal of this section is to provide clear introduction into application programming with the Atmel BitCloud stack. Every application built on top of the BitCloud stack might be considered as an additional layer or component to the components already present in the stack. This implies that the application shall be designed to conform to the architecture pattern used in BitCloud components and to communicate with other components in the same manner as they do inside the stack. That is why the whole section focuses on two main topics: describing request/confirm communications between the application layer and the underlying stack layers as well as application flow organization.

Most of the functions provided by BitCloud components are executed asynchronously. By calling a function, the application sends a corresponding request that specifies, among other parameters, a confirmation callback function to be executed at the request completion. The stack also informs the application about events that occurred in the network or within the device with the help of special indication functions. Event-driven programming, described in Section 3.1, is the basis for this type of communication between the application and the stack. Further details on how to apply it in the application to enable interaction with the stack components are given in Section 3.1.1.

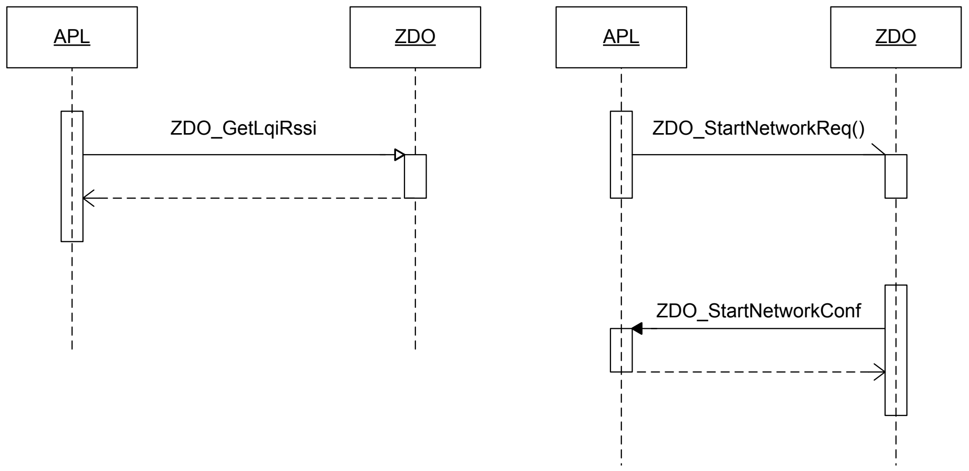
The life cycle of the entire BitCloud environment, including BitCloud components and the application, is maintained by a task scheduler, which is a core mechanism for controlling application flow. The task scheduler determines the sequence of task execution on different components. Actual task processing is performed by an appropriate task handler, which is a function present in each component, including the user application, that identifies actual component behavior. Section 3.2 provides the details. Additional features concerned with breaking normal application flow, such as hardware interrupts and application timers, are outlined in corresponding subsections 3.2.3 and 3.2.4.

In order to simplify application programming on the basis of described concepts, Section 3.3 illustrates the structure of a typical application developed on top of the BitCloud stack with source code examples. Full source code examples are provided as sample applications within the BitCloud SDK. They are intended to demonstrate all major BitCloud functionalities and serve as starting points in developing custom applications, dramatically reducing the time for creating application prototypes. Nevertheless, an application developer should understand the concepts behind existing source code to recognize the logic quickly and be able to extend it correctly.

3.1 Event-driven programming

Event-driven systems are a common programming paradigm for small footprint, embedded systems with significant memory constraints and little room for the overhead of a full operating system. Event-driven or event-based programming refers to programming style and architectural organization that pair each invocation of an API function with an asynchronous notification (and result of the operation) of the function completion. The notification is delivered through a callback associated with the initial request. Programmatically, the user application provides the underlying layers with a function pointer, which the layers below call after the request is serviced.

Figure 3-1. Synchronous vs. asynchronous.



Event-driven programming may also be familiar to embedded developers with GUI programming experience. GUI frameworks resort to the same callback mechanism when user-defined code needs to execute on some external system event (for example, a keyboard event or mouse click) providing the natural asynchrony. The similarity is superficial, however, as most user code interacting with a GUI framework is still synchronous; that is, function calls block, and the return value is typically retrieved immediately. In a fully event-driven system, all user code executes in a callback either a priori known to the system or registered with the stack by the user application. Thus, the user application runs entirely in stack-invoked callbacks.

3.1.1 Request/confirm and indication mechanism

All applications based on the Atmel BitCloud SDK require in an event-driven or event-based programming style. In fact, all internal stack interfaces are also defined in terms of forward calls and corresponding callbacks. Each layer defines a number of callbacks for the lower layers to invoke, and, in turn, invokes callback functions defined by higher levels. Each layer is also equipped with a dedicated callback called the task handler, which is responsible for executing main layer-specific logic. APL_TaskHandler() is the reserved callback name known by the stack as the application task handler. Invocation of APL_TaskHandler() is discussed in Section 3.2.1.

The request/confirm mechanism is a particular instance of an event-driven programming model. Simply put, a request is an asynchronous call to the underlying stack to perform some action on behalf of the user application; a confirm is the callback that executes when that action has completed and the result of that action is available.

For example, consider a ZDO_StartNetworkReq(ZDO_StartNetworkReq_t *req) function call, which requests the ZDO layer to start the network. The argument is a pointer to the structure of type ZDO_StartNetworkReq_t defined in zdo.h as follows:

```
typedef struct
{
  ZDO_StartNetworkConf_t      confParams;
  void (*ZDO_StartNetworkConf)(ZDO_StartNetworkConf_t *conf);
} ZDO_StartNetworkReq_t;
```

The first field (confParams) is a structure used to store the stack's response (in this case, actual network parameters) to the request. The last field (ZDO_StartNetworkConf) is a pointer to a callback function. This is how most of the other request parameters structures are designed. The ZDO_StartNetworkReq()

request is paired up with a user-defined callback function with the following signature:

```
//Callback for the start network request,the body of the function
//shall be also present in the application
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confirmInfo);
```

The actual callback name can be different from ZDO_StartNetworkConf, although it follows the naming convention used by the stack and is a good practice to follow as well. An actual request is preceded by an assignment to the callback pointer as follows:

```
static ZDO_StartNetworkReq_t networkParams; //global variable
...
//When network start should be requested
networkParams.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
ZDO_StartNetworkReq(&networkParams);
...
```

The example illustrates a particular instance of using a request/confirm mechanism (see WSNDemoApp.c in the WSNDemo folder for more detail), but all uses follow the same general setup.

Note, that a request function receives as an argument a pointer to a parameter structure, which should be defined before the function call. The argument received with the confirmation callback points exactly to the confirmation parameters contained inside the request parameters (networkParams.confParams in the example above). Therefore, a variable for request parameters must be defined globally. Another important requirement is that the confirmation function may not use the global variable for request parameters. Instead, it must operate with the pointer given as an argument (confirmInfo).

Whether or not an operation completed successfully can be observed through the status field contained in the confirmation parameters. If the operation fails, the status indicates the reason for it. The type of the status field is generally the same for all confirmation functions in a given stack component. Consider the following example with ZDO_StartNetworkConf(), provided appState is a global variable used to store application state:

```
//Implementation of the callback for the start network request
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confirmInfo)
{
  if (ZDO_SUCCESS_STATUS == confirmInfo->status)
  {
    appState = APP_IN_NETWORK_STATE;
    ...
  }
}
```

For some operations, request execution takes a considerable amount of time and a confirmation function can be called minutes after the request was issued. During this time, the application may need to issue another request of the same type. But this should be done carefully because the application cannot use the same global variable for request parameters since it is going to be used by the first request for confirmation. Instead, the application needs to allocate another structure for request parameters. Otherwise, it has to postpone the second request until the first request is completed.

Note that the need to decouple the request from the answer is especially important when the request can take an unspecified amount of time. For instance, when requesting the stack to start the network, the underlying layers may perform an energy detecting scan, which takes significantly longer than we are willing to block for. Sections 3.2.3 and 3.3 outline the reasons why prolonged blocking calls are not acceptable. Some system calls, especially those with predictable execution times, are synchronous. Calls from one user-defined function to another are synchronous.

Apart from request/confirm pairs, there are cases when the application needs to be notified of an external event that is not a reply to any specific request. For this, there are a number of user-defined callbacks with fixed names that are invoked by the stack asynchronously. These include events indicating loss of network or readiness of the underlying stack to sleep, or notifying that the system is now awake. For the list of indication callbacks that shall be implemented on the application level, refer to Section 3.3.

System rule 1: All user applications are organized as a set of callbacks executing on completion of requests to the underlying layer.

System rule 2: The user application is responsible for declaring callbacks to handle unsolicited system events of interest.

3.2 Task manager, priorities, and preemption

A major aspect of application development is managing the control flow and ensuring that different parts of the application do not interfere with each other's execution. In non-embedded applications, mediation of access to system resources is typically managed by the operating system, which ensures, for instance, that every application receives its fair share of system resources. Because multiple concurrent applications can coexist in the same system (also known as multitasking), commodity operating system cores such as Windows® are typically very complex in comparison to single-task systems like the Atmel BitCloud stack. In BitCloud context, there is a single application running on top of the stack, and thus most of the contention for system resources happens not among concurrent applications but between the single application and the underlying stack. Both the stack and the application must execute their code on the same MCU.

In contrast to preemptive operating systems, which are better suited to handle multiple applications, but require significant overhead themselves, cooperative multitasking systems are low in overhead, but require, not surprisingly, cooperation of the application and the stack. Preemptive operating systems time slice among different applications (multiplexing them transparently on one CPU) so that application developers can have the illusion that their application has the exclusive control of the CPU. An application running in a cooperative multitasking system must be actively be aware of the need to yield the resources that it's using (primarily, the processor) to the underlying stack. Another benefit of the cooperative system is that only one stack is used, which saves a considerable amount of data memory.

Returning to the example with user callbacks, if the ZDO_StartNetworkConf() callback takes too long to execute, the rest of the stack will be effectively blocked waiting for the callback to return control to the underlying layer. Note that callbacks run under the priority of the invoking layer, and so ZDO_StartNetworkConf() runs at the ZDO priority level. Users should exercise caution when executing long sequences of instructions, including instructions in nested function calls, in the scope of a callback invoked from another layer.

System rule 3: All user callbacks should execute in 10ms or less.

The maximum execution time specified in System rule 3 refers to the actual execution time of the callback code, rather than the waiting time for the response. For some requests, a response may be issued minutes after the request is sent.

System rule 4: Callbacks run at the priority level of the invoking layer.

The strategy for callbacks executing longer than 10ms is to defer execution. Deferred execution is a strategy for breaking up the execution context between the callback and the layer's task handler by using the task manager API. The way deferred execution is achieved is by preserving the current application state and posting a task to the task queue as follows:

```
SYS_PostTask(APL_TASK_ID);
```

Task posting operation is synchronous, and the effect of the call is to notify the scheduler that the posting layer has a deferred task to execute. For the user application, the posting layer is always identified by APL_TASK_ID. Posting a task results in a deferred call to the application task handler, APL_TaskHandler(), which, unlike other callbacks, runs at the application's priority level. In other words, the application task handler runs only after all higher priority tasks have completed. This permits longer execution time in a task handler context versus a callback context.

System rule 5: The application task handler runs only after all tasks of higher priority have completed.

System rule 6: The application task handler should execute in 50ms or less.

Additional task IDs of interest are HAL_TASK_ID and BSP_TASK_ID, which refer to tasks belonging to the hardware abstraction layer or board support package, respectively. When a user application involves modifications to the HAL or BSP layers, the deferred execution of HAL and BSP callbacks should utilize those layers' task IDs when posting.

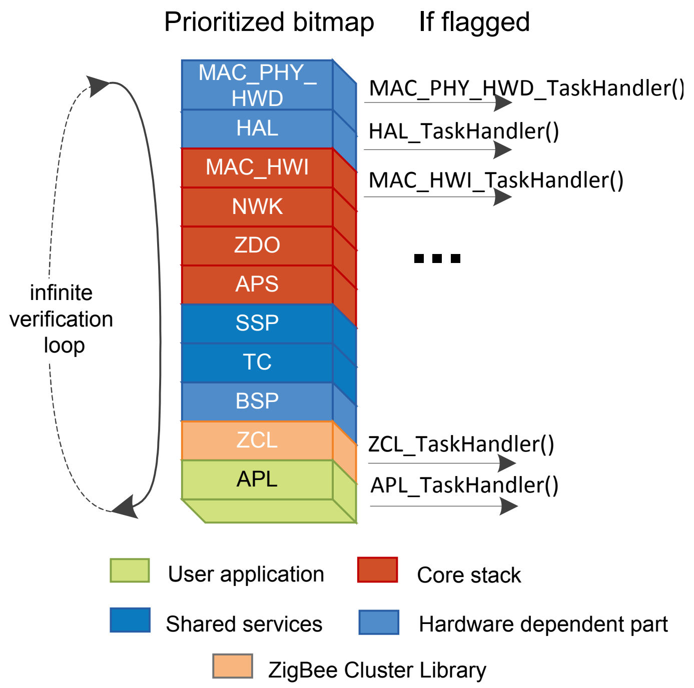
3.2.1 Task manager implementation

As it was described above, the stack executes in a single thread, and the execution context switches between separate actions or tasks. A special internal mechanism, usually referred to as the task manager or simply the scheduler, controls the execution flow and chooses the next task handler to be called. A task handler operates in the scope of a particular stack component performing tasks associated with this component. At the code level, each task handler is represented by a special function named LAYER_TaskHandler() where LAYER represents the name of a layer (for example, APL, APS, NWK, HAL, etc.). Note that the source code for some task handlers is not available in the Atmel BitCloud SDK. In any case, the changing of non-application task handlers should be done with extreme care and with full awareness of why it is needed.

While processing, the task manager determines which layers have tasks that need to be processed and calls the layers' task handlers one after another. The existence of tasks is indicated to the scheduler by special bits, which are set by calling the SYS_PostTask() function with a corresponding layer's task id as an argument. For example, to request execution of a HAL task handler, call SYS_PostTask(HAL_TASK_ID). The task manager chooses the next task handler according to the layer's priority. The highest priority is owned by a combined MAC-PHY-HWD layer provided with a single task handler. An application layer has the lowest priority, and so APL_TaskHandler() executes only when there are no uncompleted tasks on the other layers. Note that APL_TaskHandler() must be implemented in the application code. Figure 3-2 illustrates the task management process.

The operation of the task manager is maintained by an infinite loop residing in the main() function's body. Since BitCloud version 1.11.0, the main() function has been opened, and so an application developer can change its logic as needed, but the structure must remain unchanged: main() starts with a calling to the SYS_Init() function, which performs device initialization, and then enters an infinite loop where each iteration includes a call to the SYS_RunTask() function, thus telling the task manager to continue priority-based task execution.

Figure 3-2. Control flow inside the stack.



3.2.2 User application as a state machine

Developers are recommended to implement their application as a state machine. Following this abstraction, the application's life cycle is divided into a finite number of states. Each state implies specific application logic, which is inserted by the application developer. The representation of this concept in code is given by a set of global variables that constitute the state and the APL_TaskHandler() function, which switches on the state's value to a specific part of the code. Typically, the application starts with some kind of initial state, which is changed upon the completion of certain requests issued by the application to the stack.

Following the state machine architecture helps conform to certain restrictions aimed at achieving better application performance, such as the rule that all callbacks should finish in 10ms or less. In a Atmel BitCloud application, most of the substantial actions include requests to the underlying stack layers. Completion of the request is indicated by calling the confirmation callback, which may in some applications require executing a significant portion of the application code and consume much more than 10ms. A common practice to cope with this is to defer execution of a callback. In this case, all that it does is assign the state with a value describing what happened and post a task via the SYS_PostTask(APL_TASK_ID) function. This results eventually in calling the application task handler when its turn comes, which performs all the actions associated with the deferred callback.

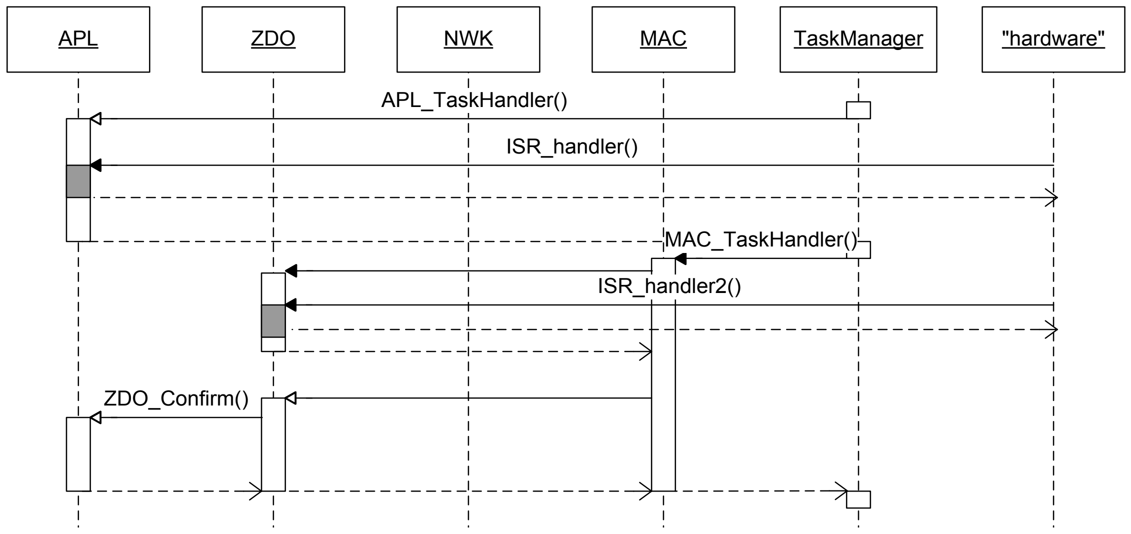
Organization of all sample applications provided with the SDK follows the described concept, and so their behavior can be easily recognized in the source code. Custom user applications created on top of the BitCloud API should follow the same approach.

3.2.3 Concurrency and interrupts

Concurrency refers to several independent threads of control executing at the same time. In preemptive systems with time slicing and multiple threads of control, the execution of one function may be interrupted by the system scheduler at an arbitrary point, giving control to another thread of execution that could potentially execute a different function of the same application. Because of the unpredictability of interruptions and the fact that the two functions may share data, the application developer must ensure atomic access to all shared data.

As discussed previously, a single thread of control is shared between the application and the stack in Atmel BitCloud applications. By running a task in a given layer of the stack, the thread acquires a certain priority, but its identity does not change; it simply executes a sequence of non-interleaved functions from different layers of the stack and the application. Thus the application control flow may be in no more than one user-defined callback at any given time. In the general case, the execution of multiple callbacks cannot be interleaved; each callback executes as an atomic code block.

Figure 3-3. Normal and interrupt control flow in the stack.



Even though time slicing is not an issue, there is a special condition where another concurrent thread of control may emerge. This happens due to hardware interrupts, which can interrupt execution at an arbitrary point in time (the main thread of control may be either in the stack code or the application code when this happens) to handle a physical event from an MCU's peripheral device (for example, USART or SPI channel). This is analogous to handling hardware interrupts in any other system.

Figure 3-3 illustrates an example interaction between the application, the stack, the task manager, and the hardware interrupts. Initially, the task handler processes an application task by invoking APL_TaskHandler. While the application-level task handler is running, it is interrupted by a hardware event (shown in gray). A function that executes on a hardware interrupt is called an interrupt service routine or interrupt handler. After the interrupt handler completes, control is returned to the application task handler. Once the task handler finishes, control is returned to the task manager, which selects a MAC layer task to run next. While the MAC_TaskHandler is running, it invokes a confirm callback in the ZDO layer, and this callback is, in turn, interrupted by another hardware interrupt. Note also that the MAC task handler invokes another ZDO callback, which invokes another callback registered by the application. Thus, the application callback executes as if it had the priority of the MAC layer or MAC_TASK_ID.

A BitCloud application may register an interrupt service routine that will be executed upon a hardware interrupt. Typically this is done by handling additional peripheral devices whose hardware drivers are not part of the standard SDK delivery and are instead added by the user. The call to add a user-defined interrupt handler is:

```
HAL_RegisterIrq(uint8_t irqNumber, HAL_irqMode_t irqMode, void(*)(void) f);
```

irqNumber is an identifier corresponding to one of the available hardware IRQ lines, irqMode specifies when the hardware interrupts are to be triggered, and f is a user-defined callback function which is the interrupt handler. Naturally, the execution of an interrupt handler may be arbitrarily interleaved with the execution of another application callback. If the interrupt handler accesses global state data also accessed by any of the application callbacks, then access to that shared state data must be made atomic. Failure to provide atomic access can lead to data races; that is, non-deterministic code interleaving, which will surely result in incorrect application behavior.

Atomic access is ensured by framing each individual access site with atomic macros ATOMIC_SECTION_ENTER and ATOMIC_SECTION_LEAVE. These macros start and end what is called a critical section, a block of code that is uninterruptible. The macros operate by turning off the hardware interrupts. The critical sections must be kept as short as possible to reduce the amount of time hardware interrupts are masked. On an Atmel AVR® microcontroller, flags are used, and so interrupts arriving during the masking will be saved.

System rule 7: Critical sections should not exceed 50ms in duration.

3.2.4 Application timers

Thus far, the three ways that control flow enters application code are:

1. Through the task handler, following a SYS_PostTask() invocation.
2. Through confirm callbacks invoked by underlying stack layers on request completion.
3. Through asynchronous event notifications invoked by the stack.

Note that none of the three has a time boundary for when the invocation is to be expected. One way to ensure execution of a user-defined callback at a specific time in the future is to use a timer.

The stack provides a high-level application timer interface, which uses a low-level hardware timer. The timer interface is part of the HAL, and its use is illustrated in many sample applications (see BlinkApp.c in the Blink folder for more detail). The general idea is that the HAL_AppTimer_t structure defines the timer interval (in milliseconds), whether it is a one-shot timer or a timer firing continuously, and the callback function to invoke when the timer fires. The structure can then be passed to HAL_StartAppTimer() and HAL_StopAppTimer() to start and stop the timer, respectively.

3.3 Typical application structure

An Atmel BitCloud application differs significantly in its organization from a typical non-embedded C program. The following sections describe the most important code blocks that constitute a typical BitCloud application, with source code examples for each part. The complete source code can be found in the sample application projects provided with the SDK.

3.3.1 The main function

Every application contains a main function, which is, as usual, the starting point of the application. Typical main function is presented below.

```
int main(void)
{
  SYS_SysInit();
  for (;;) {
    SYS_RunTask();
  }
}
```

A developer can add any additional code into the body of the function, but the main function should always fit the structure provided: first, the `SYS_SysInit()` function is invoked to initialize the microcontroller, then the `SYS_RunTask()` function is called in the infinite loop to pass control to the task manager. The task manager begins invoking layers' task handlers in order of priority (from highest to lowest), eventually invoking the application task handler. Following the initial call to the application task handler, the control flow passes between the stack and the callbacks, as shown in Figure 3-2.

3.3.2 Application state

Every application maintains global state data shared between the callbacks and the task handler. In real world applications, the state is represented by a number of variables for device role-dependent sub-states, various network parameters, sensor state, etc. For example, consider that the global state is represented by a single variable, `appState`, of type `AppState_t`, which is defined as arbitrary in the application code, and has a set of predefined values for different application states. The definition of the state variable might be given in the global scope as follows:

```
AppState_t appState = APP_INITING_STATE;
```

The task handler utilizes the state variable to switch to the code that has to be executed for the particular state.

3.3.2.1 Application state values

`AppState_t` can be a custom enumerated type, taking such values as `APP_IN_NETWORK_STATE`, `APP_INITING_STATE`, etc. We can assume that `APP_IN_NETWORK_STATE` means that the node is joined to network, `APP_INITING_STATE` means the node has initial state (this is actually a point to which control passes during the first call of the application task handler), and `APP_STARTING_NETWORK_STATE` indicates that the node is trying to join network. A typical task handler switches on a state variable to execute the code for each particular state. This is a programming template shared by all sample applications provided with the SDK.

3.3.3 The application task handler

As discussed previously, every application defines a single task handler, which contains the bulk of the application's code in its scope (including code accessible through nested function calls). The name of this function should be `APL_TaskHandler()`. The code for it might look like this:

```
void APL_TaskHandler()
{
  switch (appState)
  {
    case APP_IN_NETWORK_STATE:
      ...
      break;
    case APP_INITING_STATE: //node has initial state
      ...
      break;
    case APP_STARTING_NETWORK_STATE:
      ...
      break;
  }
}
```

3.3.4 Indication callbacks

Every application defines a number of indication callbacks with known names that execute when an event is invoked by the stack. A number of such callbacks with predefined names must be present in every application. These callbacks are:

```
void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *nwkParams)
void ZDO_WakeUpInd(void)
void ZDO_BindIndication(ZDO_BindInd_t *bindInd)
void ZDO_UnbindIndication(ZDO_UnbindInd_t *unbindInd)
```

The last two callbacks need to be implemented only if binding is activated (controlled by the `_BINDING_` flag). If the callback does not have to carry any logic, its body can be left empty. For example, consider the following code for `ZDO_WakeUpInd()`:

```
void ZDO_WakeUpInd(void)
{
  ...
  if (APP_IN_NETWORK_STATE == appState)
  {
    appState = APP_ WOKE_UP_STATE;
    ...
    SYS_PostTask(APL_TASK_ID);
  }
}
```

3.3.5 Confirmation callbacks

Every application defines a number of confirmation callback functions contributing code that executes when an asynchronous request to the underlying layer is serviced. For example, a typical application might request a network start. To make such request, the application first defines a global variable to hold the request data:

```
ZDO_StartNetworkReq_t  startNetworkReq; // global variable
```

After that, in an arbitrary point of the application, there should be the following code:

```
startNetworkReq.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
ZDO_StartNetworkReq(&startNetworkReq);
```

Here, `ZDO_StartNetworkConf` is the name of the callback function that is to be invoked when the request completes. Its implementation might look like this:

```
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confirmInfo)
{
  ...
  if (ZDO_SUCCESS_STATUS == confirmInfo->status) {
    appState = APP_IN_NETWORK_STATE;
    ...
    SYS_PostTask(APL_TASK_ID);
  }
}
```

Note the use of the `SYS_PostTask()` scheduler function. This might happen when, for a particular reason, the application defers processing of a network join to the task handler, which deals with it fully. The `ZDO_StartNetworkConf` callback simply changes the global state and returns to the ZDO layer that invoked it. This style of programming is consistent with cooperative multitasking system setup, and it permits the stack to handle higher priority tasks before returning to the deferred action.

Section 4
ConfigServer interface

The Atmel BitCloud stack provides an extensive set of configuration parameters, which determine different aspects of network and node behavior. These parameters are accessible to the application via the configuration server interface (ConfigServer, or CS for short). This section gives only a brief introduction to the CS interface, and a complete list and description of CS parameters can be found in the BitCloud Stack API Reference [5].

All CS parameters can be divided into two categories: persistent and non-persistent. Persistent parameters are stored in power-independent EEPROM, and their values are accessible to applications and the stack after node hardware reset. Non-persistent parameters are stored in RAM, and upon hardware reset are reinitialized with their default values. Whether a certain CS parameter is persistent or not can also be found in [5].

4.1 Reading/writing CS parameters

Definitions of all CS parameters and their default values are contained in the `csDefaults.h` header file inside the stack. Changing this file directly is not recommended. Instead, a developer is able to overwrite the default parameter values in the application header file, `configuration.h`. This is the simplest method for assigning new values to CS parameters. For example, the following line in the configuration file of any sample application sets the RF output power to 3dBm:

```
#define CS_RF_TX_POWER 3
```

Although this method is fairly simple, it allows parameter configuration only at compile time and does not support runtime modifications. Management of CS parameters at run time is performed with the CS read/write functions, described below.

4.1.1 CS read/write functions

For the reading and writing of CS parameters at run time, ConfigServer provides two corresponding API functions: CS_ReadParameter() and CS_WriteParameter(). Both functions require the parameter ID and a pointer to the parameter's value as arguments. The parameter ID identifies the CS parameter the function is applied to and is constructed by adding "_ID" at the end of the CS parameter's name.

The code below provides an example of how the application can read and write the RF output power value (given by the CS_RF_TX_POWER parameter):

```
int8_t new_txPwr = 3; // variable for writing new value
int8_t curr_txPwr; // variable for reading current value
CS_ReadParameter(CS_RF_TX_POWER_ID, &curr_txPwr);
CS_WriteParameter(CS_RF_TX_POWER_ID, &new_txPwr)
```

Section 5
Network management

As already described in Section 2.1, the Atmel BitCloud architecture follows the structure of the ZigBee PRO specification, which allows applications to interact directly only with APS and ZDO components of the core stack. Such an approach significantly simplifies application development and guarantees that the application has no impact on the networking protocol, and, hence, the application always behaves in compliance with the ZigBee PRO specification.

This section presents basic network-related concepts and introduces major stack functions concerning network structure and organization. Special subsections briefly describe algorithms behind common operations such as network formation, join, leave, etc., and provide source code examples to illustrate interactions between the user application and the BitCloud stack. The BitCloud stack abstracts the low-level network operations, providing the high-level API that allows the application to perform any major network action with a single request to ZDO or APS.

5.1 Networking overview

A network consists of devices that can communicate with each other even if some devices may not have a direct wireless link with all the other devices. Frame packets sent by a device are routed through other devices to the destination address, allowing transferring data over comparatively large distances. A device as a part of a network is usually called a node. ZigBee networks contain nodes of several types with different network capabilities, and all nodes present in the network can perform specific application-defined actions.

To join an existing network, a device chooses a certain other device already attached to the network and establishes connection with it, thus making it its own parent node. Since each node present in a network except one has a single parent, we can view the network as a tree. While data transmission can be processed between a child and a parent, additional transmitting links are also possible. This topic as well as other important networking concepts such as node types, node addressing, network topology, and network parameters are covered in detail in the following subsections. Implementation of basic network operations is covered by Section 5.2.

Note that all ConfigServer parameters mentioned in Section 5.1 should not be changed if a device is already joined to a network. However, most of these parameters can be modified after the node leaves the network. To learn when a given parameter can be safely changed, refer to [5].

5.1.1 Device type

ZigBee specification [3] differentiates among three device types that can be present in a network: coordinator, router, and end device.

- The main responsibility of a **coordinator (C)** is to form a network with desired characteristics. After network formation, other nodes may be allowed to join the network via the coordinator or routers already present in the network. The coordinator node is also able to execute data routing functionality. Due to the static short address (0x0000) associated with the coordinator, only one node of such device type is allowed in the network
- A **router (R)** is able to provide transparent forwarding of data originated on other nodes to the destination address. A router can also serve as an originator of data. In the same way as the

coordinator node, routers are able to act as network entry points for other devices, and can serve as direct parents for end device nodes

- An **end device (ED)** has the least networking capabilities. It can only send and receive data frames (even broadcast and multicast frames) that are always forwarded to/from the destination via the parent node the end device is currently associated with. However, among all nodes, only end devices are able to sleep

5.1.1.1 ConfigServer parameters affecting the device type

In Atmel BitCloud applications, the device type is defined by the CS_DEVICE_TYPE parameter of the DeviceType_t type. It can be set to one of the following values: DEVICE_TYPE_COORDINATOR (or 0x00), DEVICE_TYPE_ROUTER (or 0x01) and DEVICE_TYPE_END_DEVICE (or 0x02) for coordinator, router, and end device, respectively.

Additionally, a Boolean parameter, CS_RX_ON_WHEN_IDLE, must be set to true on the coordinator and router nodes, while it must be set to false on sleeping end devices to enable indirect frame reception via poll requests, as described in Section 6.4. If this parameter is set to true on an end device, polling will not be applied.

5.1.1.2 Runtime configuration of the device type

Since the device type is determined by ConfigServer parameters, it can be specified either at compile time or during application execution. The latter case is restricted; the application can specify the device type only when the device is out of the network. Below is a code example that configures a node as an end device:

```
DeviceType_t deviceType = DEVICE_TYPE_END_DEVICE;
bool rxOnWhenIdle = false;

CS_WriteParameter(CS_DEVICE_TYPE_ID,&deviceType);
CS_WriteParameter(CS_RX_ON_WHEN_IDLE_ID, &rxOnWhenIdle);
```

From now on, the single term “network start” will be used instead of “network formation” if a particular node is a coordinator and instead of “network join” if a node is a router or end device. The reason for using a single term is that both the network formation and network join procedures are initiated in the application code by the same network start request, but when processing a request, the stack determines a particular action according to the device type making the request.

5.1.2 Device addressing

5.1.2.1 IEEE/extended address

In accordance with the ZigBee standard [3], each device shall be uniquely identified by an extended (64-bit) address, often also called a MAC or IEEE address. In the BitCloud stack, the extended address is determined by the CS_UID parameter, and, as with any other ConfigServer parameter, it can be set as described in Section 4.1. Before the network start procedure, the application shall ensure that the extended address is assigned uniquely on every device.

Note that on certain hardware platforms (for example, Atmel ZigBit®, RCB, Atmel ATmega128RFA1), the stack will attempt to read the extended address value from the UID chip or external EEPROM located on the board (using the HAL_ReadUid() function) if CS_UID is set to 0 at compile time. In case of a success, the value obtained is assigned to the extended address, thus ensuring uniqueness among devices on the same development platform. Otherwise, the extended address stays unchanged and equals 0.

5.1.2.2 *Short/network address*

When joined to a ZigBee network, each node is identified by the so-called short (16-bit) address, sometimes also referred to as the network (NWK) address, which is inserted into frame headers instead of the extended address during data exchange to reduce overhead.

In an Atmel BitCloud application, a short address can be either selected randomly by the stack (stochastic addressing) during entering the network or assigned to by the user application to a desired value (static addressing) prior to the network start procedure. It is critical to ensure that all nodes in the network use the same addressing scheme; that is, either stochastic or static addressing.

5.1.2.3 *Stochastic vs. static addressing mode*

The boolean CS_NWK_UNIQUE_ADDR parameter specifies whether stochastic addressing (if set to 0) or static addressing (if set to 1) is applied on the node. In the latter case, a desired short address value should be assigned to the CS_NWK_ADDR parameter. If the static addressing scheme is used, the application shall always assign the coordinator short address to 0x0000. In the stochastic addressing scheme, this value is set on the coordinator automatically by the stack. After the network start, a device can read its own short address from the .shortAddr field in the argument of the ZDO_StartNetworkConf() callback function registered for the network start request, as described in Section 5.2.1.1.

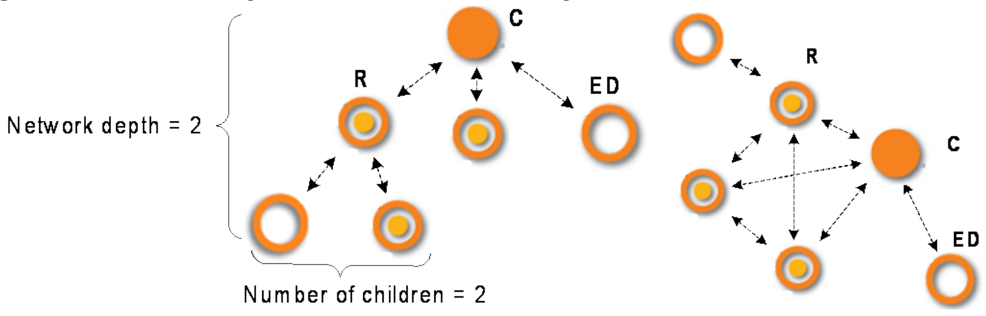
5.1.2.4 *Address conflict resolution*

In the stochastic addressing scheme, a special address conflict resolution mechanism automatically detects and resolves situations where a randomly chosen short address appears to not be unique; that is, another node with the same address is already present in the network. After the short address is updated, the application on the corresponding node is informed about this via the ZDO_MgmtNwkUpdateNotf() function with status ZDO_NWK_UPDATE_STATUS (0x8E) and new short address value in the argument. In a network with the static addressing scheme, the application is responsible for resolving such conflicts. But the stack provides some assistance by calling the ZDO_MgmtNwkUpdateNotf() function with status ZDO_STATIC_ADDRESS_CONFLICT_STATUS (0x95) on the node that has detected the conflict (which can differ from the node with a conflicting address).

5.1.3 Network topology

As was mentioned earlier, network topology follows a tree structure. Network construction is always started by a coordinator node, which performs network the formation procedure and sets the desired network parameters. When the formation procedure is completed, the coordinator is able to establish connections with routers and end devices. Each node joining the network connects to a certain parent node, which can be either a router or the coordinator.

Figure 5-1. Network organization vs. data exchange.



a. Network organization b. Available transmission links

Since a router can send data to any other router or the coordinator that it can reach, network organization (topology) can be very different from the actual communication links used to route data across the network. This is illustrated by Figure 5-1. Figure 5-1b shows the direct transmission links available in the network organized according to Figure 5-1a, assuming all nodes are located within signal range of each other.

It is essential that in addition to the node parameters described in sections 5.1.1 and 5.1.2 a node shall also configure topology-related parameters prior to network join. Proper node and network configuration will help achieve the desired network behavior and enhance performance.

5.1.3.1 Limits on the number of children

Router and coordinator nodes can limit the number of direct children that can join to them by configuring CS_MAX_CHILDREN_AMOUNT, which defines the total maximum number of direct child nodes (routers plus end devices). The CS_MAX_CHILDREN_ROUTER_AMOUNT parameter defines the maximum number of routers among the child nodes. Thus, the difference between both parameters specifies the maximum number of end devices that can be connected to this node simultaneously, namely:

```
CS_MAX_CHILDREN_AMOUNT - CS_MAX_CHILDREN_ROUTER_AMOUNT
```

Note that the following should hold true on router and coordinator nodes:

```
CS_NEIB_TABLE_SIZE > CS_MAX_CHILDREN_AMOUNT >= CS_MAX_CHILDREN_ROUTER_AMOUNT
```

5.1.3.2 Neighbor table size

For an end device node, an essential role is played by the CS_NEIB_TABLE_SIZE parameter during the network join in determining the maximum number of potential parents the current end device will be able to choose from. CS_NEIB_TABLE_SIZE must be set during compilation, as its value is used for allocating memory for stack tables. Among the detected nodes capable of accommodating another child end device, the one with the best link quality is chosen. Since routers and the coordinator continue to actively use their neighbor tables to store information about children and neighbors after the network start, its size should larger than on an end device.

5.1.3.3 Maximum network depth

The CS_MAX_NETWORK_DEPTH parameter specifies the maximum network depth, which is the maximum possible number of edges in a network tree from a node to the coordinator. If maximum network depth is reached on a certain router, the router will not be able to have any children, even if other parameters allow it. Additionally, CS_MAX_NETWORK_DEPTH has impact on several timeout intervals used inside the stack (for example, the broadcast delivery timeout). Besides, CS_MAX_NETWORK_DEPTH multiplied by two bounds equals the maximum number of hops that can be made during data routing. For correct network operation, this parameter should be assigned the same value on all the nodes in the network.

5.1.3.4 Changes in losing parents between routers and end devices

If an end device loses connection to the parent, it cannot continue functioning as a normal network node. From this moment on it is out of the network, and so to be able to exchange data it must search for the network again. On the other hand, if a router’s parent becomes inaccessible, as would happen if the coordinator to which the router has connected switches off, the router still operates as a normal network device, receiving and routing data frames. Moreover, a router is not considered a child by its parent after the parent receives the first link status frame from the router. And so from this point of view, a node of the router type can be considered as not having a parent. The device to which it connects on the network serves only as an entry point that provides the necessary network information. More information on the subject of parent loss and network leave can be found in Section 5.2.2.

5.1.4 Target network parameters

Prior to initiating a network start procedure, the node is responsible for setting parameters that characterize either the network it wishes to form (for a coordinator) or the network it wishes to join (for routers and end devices). These parameters are:

1. Supported modulation scheme, the so called channel page (CS_CHANNEL_PAGE).
2. Supported frequency channels, specified via a 32-bit channel mask (CS_CHANNEL_MASK).
3. 64-bit extended PAN ID (CS_EXT_PANID).
4. Security parameters (see Section 8).

In parenthesis are shown the parameter names in the ConfigServer (CS) component that are used by the Atmel BitCloud application in order to assign desired values to corresponding network parameters, as described in Section 4.

5.1.4.1 Channel page and channel mask

CS_CHANNEL_PAGE defines the modulation type to be used by the device. This parameter is ignored for 2.4GHz frequencies, while for 868/915MHz bands, only values 0 and 2 are accepted. For the 780MHz band, the channel page parameter shall be always set to 5.

CS_CHANNEL_MASK is a 32-bit field that determines the frequency channels supported by the node. The five most-significant bits (b_{31},..., b_{27}) of channel mask should be set to 0. The remaining 27 bits (b_{26}, b_{25},...b_0) indicate availability status for each of the 27 valid channels (1=supported, 0=unsupported).

Table 5-1 shows channel distribution among IEEE 802.15.4 frequency bands, as well as the data rates on the physical level for different channel pages.

Table 5-1. Characteristics of IEEE 802.15.4 channel pages and frequency bands.

Channel page (decimal)	Frequency band	Channel numbers (decimal)	Modulation scheme	Data rate, kbps
0	868MHz	0	BPSK	20
	915MHz	1 – 10	BPSK	40
	2.4GHz	11 – 26	O-QPSK	250
2	868MHz	0	O-QPSK	100
	915MHz	1 – 10	O-QPSK	250
5	780MHz	0 - 3	O-QPSK	250

Example 1: A transceiver operates in the 2.4GHz band, and channel 17 (decimal) should be enabled for operation.

a) Since it operates at 2.4GHz, there is no need to set CS_CHANNEL_PAGE.

b) The channel mask, CS_CHANNEL_MASK, in this case is set as follows:

$0_{31}\,0_{30}\,0_{29}\,0_{28}\,0_{27}\,0_{26}\,0_{25}\,0_{24}\,0_{23}\,0_{22}\,0_{21}\,0_{20}\,0_{19}\,0_{18}\,1_{17}\,0_{16}\,0_{15}\,0_{14}\,0_{13}\,0_{12}\,0_{11}\,0_{10}\,0_{9}\,0_{8}$
$0_{7}\,0_{6}\,0_{5}\,0_{4}\,0_{3}\,0_{2}\,0_{1}\,0_{0}$

which is equivalent to 0x00020000.

Example 2: A transceiver operates in the 915MHz band with 250kbps data rate. Channels 3, 4, and 9 shall be enabled for operation.

a) CS_CHANNEL_PAGE = 2

b) CS_CHANNEL_MASK = $0_{31}0_{30}0_{29}0_{28}0_{27}0_{26}0_{25}0_{24}0_{23}0_{22}0_{21}0_{20}0_{19}0_{18}0_{17}0_{16}0_{15}$ $0_{14}0_{13}0_{12}0_{11}0_{10}1_{9}0_{8}0_{7}0_{6}0_{5}1_{4}1_{3}0_{2}0_{1}0_{0}$

= 0x000000218

Example 3: A transceiver operates in the 783MHz band while only channels 0 and 1 shall be enabled for operation.

a) CS_CHANNEL_PAGE = 5

b) CS_CHANNEL_MASK = $0_{31}0_{30}0_{29}0_{28}0_{27}0_{26}0_{25}0_{24}0_{23}0_{22}0_{21}0_{20}0_{19}0_{18}0_{17}0_{16}0_{15}$ $0_{14}0_{13}0_{12}0_{11}0_{10}0_{9}0_{8}0_{7}0_{6}0_{5}0_{4}0_{3}0_{2}1_{1}1_{0}$

= 0x000000003

5.1.4.2 *Network identifiers*

CS_EXT_PANID is the 64-bit (extended) identifier of the network, which is verified during the network association procedure. Hence, devices that wish to join a particular network must configure their extended PAN ID to equal the one on the network coordinator. If CS_EXT_PANID is set to 0x0 on a router or on an end device, then it will attempt to join the first network detected on the air.

During network formation, the coordinator selects another network identifier, the 16-bit short PANID (also called the network PANID), which is used in frame headers during data exchange instead of the heavy, 64-bit extended PANID. By default, the network PANID is generated randomly. If the coordinator detects another network with the same extended PANID during network formation, it will automatically select a different network PANID to avoid conflicts during data transmissions.

This mechanism, however, may sometimes lead to undesired behavior. If the coordinator node is reset and initiates the network start again with the same extended PANID and on the same channel, it might find routers from the previous network present and, hence, will form a new network with a different network PANID. But often it is required that the coordinator rejoin the same network as before to participate in data exchange. In order to force the coordinator to remain on this network, the network PANID should be predefined at the application level prior to network start, as shown in the example below:

```
bool predefPANID = true;
uint16_t nwkPANID = 0x1111;

CS_WriteParameter(CS_NWK_PREDEFINED_PANID_ID,&predefPANID);
CS_WriteParameter(CS_NWK_PANID_ID, &nwkPANID);
```

If the network PANID is predefined on a non-coordinator node, it will be possible for the node to enter only that network with same extended and network PANIDs as configured on the node.

There are network parameters that do not require being specially set before network start, either because they are maintained automatically by the stack throughout the network life cycle or because a typical application is well served with the predefined values. One such parameter is a network manager short address. A network manager is a dedicated node in the network that is able to influence network configuration by playing two roles: selecting a new short PANID to resolve a PANID conflict (which happens when two different networks have the same short PANID value) and changing working channel. Note, that a new short PANID is selected automatically only if it is not predefined. If a static scheme is used, the application is fully responsible for resolving such conflicts on its own. A network manager typically, but not necessarily, is the coordinator.

5.2 Basic network operations

This section explains how the user application can perform typical network operations, such as network formation, join, and leave, and generally describes what stands behind these common actions. Within the Atmel BitCloud architecture, most network management related requests are carried out by the ZDO component.

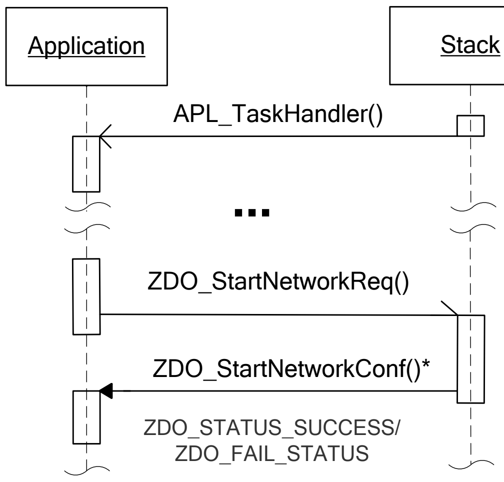
5.2.1 Network start

The BitCloud application is fully responsible for initiating a network start procedure. Before this, all essential node and network parameters must be set by the application, as described in previous sections. The network start procedure is performed with a single request, but it is still important for the user application developer to understand what actions are performed by the BitCloud stack on behalf of the application in order to be able to diagnose any runtime errors that may occur.

5.2.1.1 Network start request and confirm

Figure 5-2 shows the sequence diagram for the network start procedure, which is the same for all types of devices. Once the network parameters described in previous sections are configured properly, the application can initiate the network start procedure by executing an asynchronous call to ZDO_StartNetworkReq(). If the node is configured as a coordinator, this will perform a network formation. If the node is configured as a router or end device, this will perform a network join. After finishing the network formation/join procedure, the ZDO component informs the application about the result, invoking the registered callback function with an argument of the ZDO_StartNetworkConf_t type. This structure contains the status of the performed operation, as well as information about the started network, namely the network address for the node. A ZDO_SUCCESS_STATUS status is received if the procedure has executed successfully, while any other status means that the network start has failed for the indicated failure reason. Detailed information on ZDO_StartNetworkConf_t can be found in [5].

Figure 5-2. Network start sequence diagram.



* Function registered as callback for argument of ZDO_StartNetworkReq()

After an end device or router joins a network, its parent receives a notification via the ZDO_MgmtNwkUpdateNotf() function with an argument having a status field set to ZDO_CHILD_JOINED_STATUS (0x92). Child device type and extended and network addresses are returned as argument fields, too.

5.2.1.2 Network formation algorithm

As it should be clear from previous sections, network formation can only be initiated by a coordinator. Forming a new network is a typical way for a coordinator to become connected to a ZigBee network, because any previously formed network should already have a coordinator. However, there are additional cases where the coordinator may need to restore its connection to an existing network, in which case the short PANID value shall be predefined (see Section 5.1.4.2).

The network formation procedure consists of several steps. The coordinator starts by scanning the channels allowed by CS_CHANNEL_MASK and ignoring those channels with too much noise. On the sufficiently clear channels, the coordinator searches for ZigBee networks by sending a beacon request frame. Nodes located in a direct range respond with the beacon frame, which contains network configuration options and their own parameters. The coordinator collects the responses and stores information about the detected nodes in its neighbor table. The number of nodes remembered is limited by the CS_NEIB_TABLE_SIZE parameter.

After the information about existing networks has been collected, the coordinator distinguishes the channels with the least number of networks detected. From among these, it chooses one channel, preferring one which does not overlap with the WiFi frequency range (namely, logical channels 15, 16 and 21, 22 in the 2.4GHz ISM band).

The last action the coordinator performs is choosing the short PANID value, which is used in frame headers instead of the full PANID to reduce overhead. Just as with the short network address, there are two methods of short PANID assignment. The most common is stochastic, which is ZigBee compliant and chooses a value randomly that is different from all the values stored in the neighbors table (see the source code example in Section 5.1.4.2).

When the network configuration is completed, the coordinator is ready to establish connections with other devices.

5.2.1.3 Network join algorithm

Network join procedure is similar to the one for network formation, although it has fewer steps. A node starts by sending a beacon request to collect information about neighbor devices. Incoming beacon responses are filtered according to several conditions. If CS_EXT_PANID does not equal 0, the node can accept responses from only those devices that provide the proper PANID value (since the node is likely to rejoin a network already known to it). Otherwise, it joins the first suitable network (this is also called *associating*). Moreover, if the node is preconfigured with a short PANID, it declines responses that do not have a matching short PANID value. A responding node also sets two special bits to indicate whether it is able to accept an additional child end device or router, which is also taken into consideration.

Information about nodes that conform to these requirements is saved in the neighbor table. This again implies that the maximum number of candidates for attempting to connect cannot be larger than the value in the CS_NEIB_TABLE_SIZE parameter. The application should choose the value for this parameter carefully, because if the table size is too small, the node might fail to connect to any of the nodes in its neighbor list while overlooking other devices that can still accept new nodes.

Once the detection of neighbors and filtering is finished, the node applies a special algorithm to choose one potential parent and tries to attach to it. In addition to some extra filtering (considering the network depth of the potential parent and some other parameters), the algorithm ensures that one of the neighbors with the best LQI/RSSI is chosen and distributes the choice randomly among all suitable nodes. This helps avoid overloading one particular router having the best link quality with multiple, simultaneous join requests from many end devices. If the node fails to connect to the chosen neighbor, it repeats the procedure two more times, and returns an error to the application if the connection is not established. Otherwise, the application confirmation callback receives ZDO_SUCCESS_STATUS, which indicates that the node has joined to the network and can start sending data.

5.2.1.4 Link status frame

After entering the network, a router and a coordinator start sending link status frames every 15 seconds. The frame contains up-to-date information about the sending node, and is delivered only to the neighbors in the originator's network neighborhood.

5.2.2 Parent loss and network leave

Nodes present in a ZigBee network can leave it for two reasons:

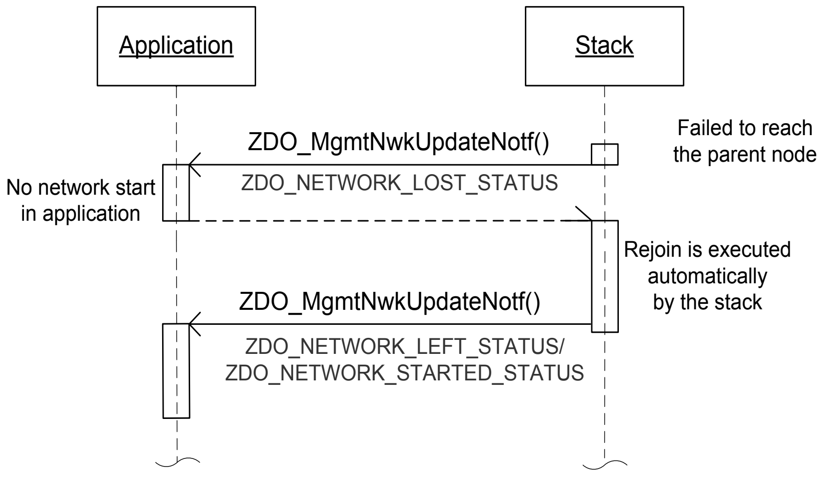
1. Parent loss. Because routers do not have dedicated parent nodes in a mesh topology, such a scenario is valid only for end devices.
2. The node is requested to leave the network. Such a request can be issued in one of two ways:
   a) By the application running on the node.
   b) By a remote node.

   Reason 2a) works for all types of devices, while 2b) can be applied to routers and end devices only.

5.2.2.1 Parent loss by an end device

Figure 5-3 shows what notification messages the stack on an end device issues if the connection to the parent node is lost. First, a notification with the ZDO_NETWORK_LOST_STATUS status is issued via the ZDO_MgmtNetworkUpdateNotf() function when the end device cannot reach its current parent node. After receiving this message, the application can perform some actions, but may not initiate a network rejoin procedure. This will be done automatically once control is passed back to the stack. The stack tries to find a new parent for the node to enter the network again. The result is reported to the application via the ZDO_MgmtNwkUpdateNotf() function. The ZDO_NETWORK_STARTED_STATUS status means that the node has successfully rejoined the network with network parameters indicated in argument to ZDO_MgmtNwkUpdateNotf(). If the network rejoin procedure has failed, the application receives notification with the ZDO_NETWORK_LEFT_STATUS status. After this, the application is responsible for changing network parameters if necessary and initiating a new network start procedure, as described in Section 5.2.1.

Figure 5-3. Parent loss sequence diagram.


Application
Stack
ZDO_MgmtNwkUpdateNotf()
Failed to reach the parent node
No network start in application
ZDO_NETWORK_LOST_STATUS
Rejoin is executed automatically by the stack
ZDO_MgmtNwkUpdateNotf()
ZDO_NETWORK_LEFT_STATUS/
ZDO_NETWORK_STARTED_STATUS

5.2.2.2 *Child loss notification*

It is often important for a parent node to be able to register when a child is lost; that is, is out of the network. As mentioned above, because normally only end devices can be child nodes, such notification will not be triggered on a router node if another router is turned off or is out of signal reach. A router is considered a child node by its parent only before the parent receives the first link status frame from the router (a router sends link status frames every 15 seconds). During this starting period, a router behaves as a normal child, and so in the case of losing connection to such a router, a child loss notification is issued in the same way as for end devices.

The main challenge in tracking child loss events is the fact that end devices are very likely to have sleep periods and hence there is often no data exchange performed over extensive time intervals, even though end devices are actually in the network and would be ready to send data after wake up.

So to ensure that a child node is out of the network and not just in a sleep mode, the parent node should know the length of end device's sleep period (that is, have the same CS_END_DEVICE_SLEEP_PERIOD as child nodes). Every parent node expects that its child will periodically wake up and issue a poll request (see Section 6.4 for detail). However, if during the time interval equal to:

```
CS_NWK_END_DEVICE_MAX_FAILURES * (CS_END_DEVICE_SLEEP_PERIOD +
CS_INDIRECT_POLL_RATE),
```

where all parameters are taken from the parent node, a child doesn't deliver any poll frames, the parent node will assume that the child node has left and call the ZDO_MgmtNwkUpdateNotf() function with the status equal to ZDO_CHILD_REMOVED_STATUS and the argument indicating the extended address of the child node.

5.2.2.3 *Network leave*

In many scenarios, it is desirable for a node to leave the network upon certain events (or even to force a certain device to disassociate itself from the network). The Atmel BitCloud stack allows the application to initiate such a procedure on a node of any type using the ZDO_ZdpReq() function, executed as shown below:

```
static ZDO_ZdpReq_t zdpLeaveReq; //globally defined variable
…
//set corresponding cluster ID
```

```
zdpLeaveReq.reqCluster = MGMT_LEAVE_CLID;
zdpLeaveReq.dstAddrMode = EXT_ADDR_MODE;
zdpLeaveReq.dstExtAddr = 0; // for own node address shall be 0
zdpLeaveReq.ZDO_ZdpResp = ZDO_ZdpLeaveResp; // callback

//for own node address shall be 0
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.deviceAddr = 0;
//specify whether to force children leave or not
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.removeChildren = 0;

//specify whether to perform rejoin procedure after network leave
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.rejoin = 1;
ZDO_ZdpReq(&zdpLeaveReq); // request network leave
```

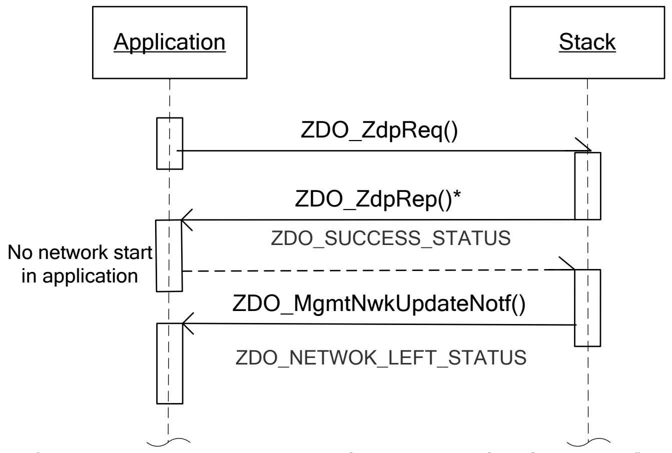
If in the code above the destination address is set to the address of a remote node, then after calling ZDO_ZdpReq(&zdpLeaveReq), the node will transmit a network command frame to the destination node, forcing it to leave the network. Introduction to ZDP requests is given later, in Section 5.3.2.

Implementation of the callback function may look like this:

```
void ZDO_ZdpLeaveResp(ZDO_ZdpResp_t* zdpResp)
{
    uint8_t reqStatus = zdpResp->respPayload.status;
    if(reqStatus == ZDO_SUCCESS_STATUS)
    {
       //make any required actions
       ...
    } else {
       //handling failed operation
    }
}
```

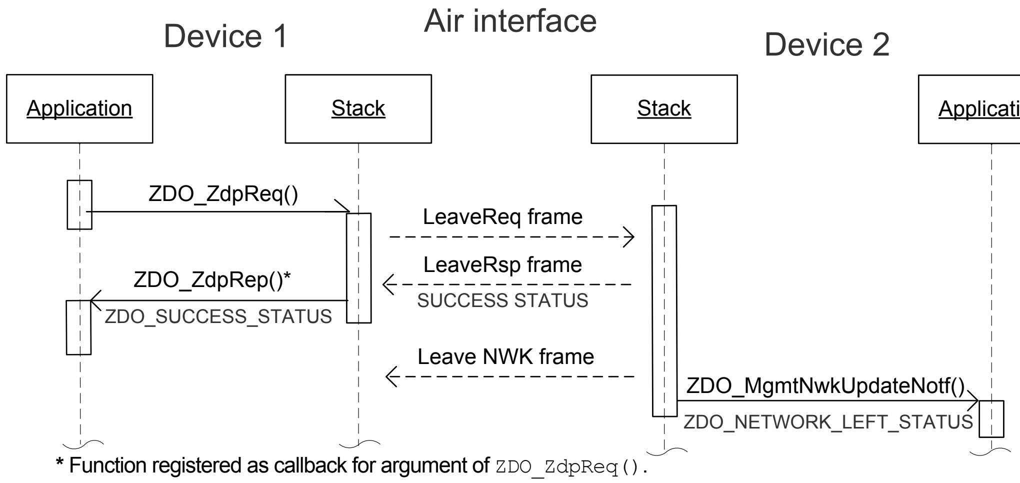
The sequence diagrams for a network leave procedure are shown in Figure 5-4 (node leaves network on its own request) and Figure 5-5 (node leaves network upon remote request).

Figure 5-4. Network leave sequence diagram (local call).



* Function registered as callback for argument of ZDO_ZdpReq().

Figure 5-5. Network leave sequence diagram. Device 1 requests Device 2 to leave the network.



* Function registered as callback for argument of `ZDO_ZdpReq()`.

The application issues a ZDP request of type MGMT_LEAVE_CLID. The stack starts processing the request and calls the registered callback function. Callback execution does not mean that the node has left the network; it simply indicates that request processing has started. Instead, when the node finally leaves, the network issues notification via the ZDO_MgmtNwkUpdateNotf() function with ZDO_NETWORK_LEFT_STATUS. If the destination address does not equal zero (that is, a remote node is to leave the network), the stack replies with a callback as soon as it sends the leave request frame and receives the leave response frame with a success status, which means that the remote node has started the network leave procedure. If this procedure executes successfully, the application on the remote node, rather than the node that initiated the process, is notified with ZDO_NETWORK_LEFT_STATUS.

5.3 Controlling the network

In addition to general commands used to enter or leave the network (see Section 5.2.1), the Atmel BitCloud stack provides a set of functions to help applications track the network state, extract information about a node's neighbors, retrieve network parameters, etc. These functions, all part of the ZDO component of the stack, are described briefly in the next section, while detailed information with usage examples can be found in the BitCloud API Reference [5].

Another important piece of functionality is provided by the ZDO component as ZigBee Device Profile requests (ZDP requests). See Section 5.3.2 for details.

5.3.1 Obtaining network information

Most of the network-controlling operations described in this section employ the neighbor table. This essential internal object has already been mentioned in reference to the network start procedure (see Section 5.2.1.2). The stack fills it during the network start procedure and maintains it afterwards, updating stored information upon certain network events, such as receiving link status frames, accepting a new child, losing a child, etc.

Table 5-2 summarizes ZDO functions used to determine the current network state at a particular node. Since the neighbor table on the node is always kept up to date, calling these functions does not generate any over-the-air messages.

Table 5-2. Network controlling functions.

Function	Description
`ZDO_GetNwkStatus()`	Checks whether the node is connected to network.
`ZDO_GetParentAddr()`	Gets parent's short and extended addresses.
`ZDO_GetChildrenAddr()`	Gets children's short and extended addresses.
`ZDO_GetNeibAmount()`	Gets the number of neighbor routers and end devices.
`ZDO_GetNeibTable()`	Retrieves the neighbor table contents.
`ZDO_GetLqiRssi()`	Determines LQI and RSSI values of a remote node. Information about the remote node must be present in the neighbor table. Otherwise, the function returns zero values.

Except for ZDO_GetNwkStatus(), which simply returns either the ZDO_IN_NETWORK_STATUS value or the ZDO_OUT_NETWORK_STATUS value, all functions listed in Table 5-2 write the requested information to a dedicated structure specified by the pointer in the argument. If the structure occupies a considerable amount of memory, then the variable for the structure must be declared with a static keyword in the global scope. To call a function from the above list, do the following:

1. Define a variable of an appropriate type. Consider using a static keyword.
2. Call the function providing the pointer to the defined variable as an argument.
3. Manipulate the extracted information through the variable defined in step 1.

For example, to retrieve the neighbor table, the following code might be used:

```
static ZDO_Neib_t neighborTable[CS_NEIB_TABLE_SIZE]; //Buffer
...
ZDO_GetNeibTable(neighborTable); //Call to ZDO
```

Note that if a structure contains a pointer to memory that should be used by a function, then the memory must be allocated separately. Consider the following definition of variables to be used by ZDO_GetChildrenAddr()

```
static NodeAddr_t childAddrTable[CS_MAX_CHILDREN_AMOUNT -
CS_MAX_CHILDREN_ROUTER_AMOUNT];
static ZDO_GetChildrenAddr_t children =
{
    .childrenTable = childAddrTable,
};
```

5.3.2 Using ZDP requests to control the network

The ZigBee Device Profile (ZDP) is a set of commands defined in the ZigBee specification to enable a range of ZigBee network related functionality. ZDP requests are managed through the ZigBee Device Object, which is implemented by the stack and resides at endpoint 0 on the same level as the Application Framework (see Section 6.1.1). This section focuses on the practical use of ZDP requests.

ZDP functionality may be classified as follows:

- Device and service discovery, including obtaining descriptors (simple, complex, and user) from a remote node corresponding to a specified endpoint, discovering nodes by a given address, etc.
- Binding, unbinding, and related commands (explained in detail in Section 6.5)
- Network control, including commands for network leave (see code example in Section 5.2.2.3), network update, obtaining the neighbor table from a remote node, etc.

This section focuses on general questions related to ZDP requests, and describes commands concerned with network controlling. Some other commands are presented in Section 6.5.4, while a complete list of ZDP command types can be found in [5].

To issue a ZDP request, the application calls the ZDO_ZdpReq() function with a pointer to an appropriately filled request parameters object of type ZDO_ZdpReq_t as an argument. The specific command type is given by a reqCluster field, which takes a value from a range of predefined command IDs. A word cluster is used in ZigBee to name set of operations or commands implemented by clients and servers over a set of attributes or parameters. A predefined collection of clusters constitutes a certain application profile (profiles and clusters are described in Section 6.1.1). However, in this context, cluster simply means a command. Table 5-3 provides a list of ZDP commands for network control and device discovery. For a complete list of ZDP commands, refer to the Atmel BitCloud API Reference.

Table 5-3. Basic ZDP commands.

Command	**Description**	**Type of request parameters**
`MGMT_LQI_CLID`	Obtains the list of a remote device's neighbors, along with corresponding LQI values.	`ZDO_MgmtLqiReq_t`
`MGMT_RTG_CLID`	Retrieves the contents of the routing table from a remote device.	`n/a`
`MGMT_BIND_CLID`	Retrieves the contents of the binding table from a remote device.	`n/a`
`MGMT_LEAVE_CLID`	Requests network leave either for a current device or a remote node.	`ZDO_MgmtLeaveReq_t`
`MGMT_NWK_DISC_CLID`	Requests a remote node to scan a network in its vicinity and report on the scan results.	`n/a`
`MGMT_PERMIT_JOINING_CLID`	Allows a remote node to join the network via association (joining the network for the first time) for a given number of secinds, disallows it if the time is set to 0x00, and allows it permanently if the time is set to 0xff (except for the ZigBee Light Link profile; see [8]). Rejoining is not constrained by this command.	`ZDO_MgmtPermitJoiningReq_t`
`MGMT_NWK_UPDATE_CLID`	Sends or requests network update information. A general command that can be used for many purposes.	`ZDO_MgmtNwkUpdateReq_t`
`NWK_ADDR_CLID`	Requests the short (network) address of a remote node with a given IEEE (extended) address.	`ZDO_NwkAddrReq_t`
`IEEE_ADDR_CLID`	Requests the IEEE (extended) address of a remote node with a given short (network) address.	`ZDO_IeeeAddrReq_t`

In addition to command type, the user must also specify the address mode (`dstAddrMode`) used to address a node to which the request is sent, the destination address and the response callback. Depending on the selected address mode, either the short address (`dstNwkAddr`) or the extended address (`dstExtAddr`) shall be specified. For most of the commands, the user also needs to set request parameters via the `req.reqPayload` field of type `ZDO_ZdpReqFrame_t`. Each command type that needs additional parameters is supplied with a corresponding field, all sharing the same memory as packed in a single C union.

Note: All ZDP requests can be sent using broadcast or unicast addressing. Moreover, some requests allow specifying a destination address inside the request parameters. In such cases, the outer destination address given by `dstNwkAddr` is ignored.

The code below illustrates issuing and processing the result of an LQI request. First, define a global object with the static keyword to hold request data as follows:

```
static ZDO_ZdpReq_t zdpRequest;
```

In the following code the request is initialized and sent to the node with the short address equal to 0x0001:

```
zdpRequest.reqCluster = MGMT_LQI_CLID; //Request type
zdpRequest.dstAddrMode = SHORT_ADDR_MODE; //Addressing mode
zdpRequest.dstNwkAddr = 0x0001; //Address of the destination
zdpRequest.req.reqPayload.mgmtLqiReq.startIndex = 0;
zdpRequest.ZDO_ZdpResp = ZDO_ZdpResp; //Confirm callback
ZDO_ZdpReq(&zdpRequest);
```

The confirmation callback function that is called upon completion of the request might look like this:

```
uint8_t uartMsg[400];

void ZDO_ZdpResp(ZDO_ZdpResp_t* zdpResp)
{
    uint8_t reqStatus = zdpResp->respPayload.status;
    if(reqStatus == ZDO_SUCCESS_STATUS)
    {
        ZDO_MgmtLqiResp_t* lqiResp = &zdpResp->respPayload.mgmtLqiResp;
        sprintf((char*)uartMsg, "\r\n\r\nTotal neighbors: %u\r\nStart index:
%u\r\nNeighbors received: %u\r\n",
              lqiResp->neighborTableEntries,
              lqiResp->startIndex,
              lqiResp->neighborTableListCount);
        //More code goes here like sending a composed uartMsg and changing
        //the application state
    }
}
```

In the code above and in the case of successful request execution, a part of the received information obtained from the `respPayload.mgmLqiResp` field is formatted and written to a buffer, which can be sent to a UART.

5.3.2.1 ZDP response timeout

The maximum execution time for a ZDP request is limited by the CS_ZDP_RESPONSE_TIMEOUT parameter. The stack waits for responses for a ZDP request during the time specified in this parameter. Reponses received after this time exceeds are ignore and do not reach the application.

The parameter is set, by default, to an automatically calculated value, which takes into account only active (not-sleeping) devices. Devices that are sleeping when the request is sent are not expected to send the response. See sections 6.4 and 7.1 for detail on sleeping devices and data exchange with them.

5.3.3 Updating network information

The stack is responsible for updating network information stored in internal structures (the neighbor table, the routing table, etc.) upon receiving various events. However, the application is able to update some network parameters using dedicated ZDP requests.

When a node receives certain network events, the stack on the node notifies the application by calling the ZDO_MgmtNwkUpdateNotf() function, which must be implemented by the application. The event type is indicated by the status field of the argument. Switching among possible status values, the application can choose execution logic corresponding to the handled event. Some situations in which the notification is issued have been already mentioned. The application is not intended to process all the possible network notifications. Moreover, the body of ZDO_MgmtNwkUpdateNotf() can be left empty if the application is not going to treat any notifications, though this is not likely to happen. Table 5-4 provides descriptions and usage notes for the most common events. To find a complete list of statuses, refer to the Atmel BitCloud API Reference.

Table 5-4. Types of network management notifications.

Status	Description
`ZDO_NETWORK_STARTED_STATUS`	Received when the stack performs a network start by itself, not initiated on the application level (for example, when the node automatically rejoins the network after it has lost connection to the parent).
`ZDO_NETWORK_LOST_STATUS`	Indicates parent loss. After issuing this notification, the stack automatically attempts to rejoin the network. This status can be received only on end devices.
`ZDO_NETWORK_LEFT_STATUS`	Received either when the node loses its parent and fails to rejoin the network, or when the node leaves the network by itself in response a ZDP command.
`ZDO_NWK_UPDATE_STATUS`	Indicates that one or more of network parameters has been changed. Parameters include PANID, channel mask, and node short address.
`ZDO_CHILD_JOINED_STATUS`	Indicates that a new child has successfully joined to the current node. In secured networks the indication is received before child's authentication (which may fail); see Section 5.3.4. This status is not valid for end devices.
`ZDO_CHILD_REMOVED_STATUS`	Indicates that a child has left the network.
`ZDO_STATIC_ADDRESS_CONFLICT_STATUS`	Indicates that a short address set statically has resulted in an address conflict. The event is raised on the node that discovered the conflict. The application is responsible for resolving the conflict, typically by choosing a different short address and updating its value on the node via a ZDP request.

As mentioned in the beginning of this section, the application is able to manipulate certain network parameters, as well as influence the network state of single nodes with the help of ZDP requests. The ZDP command of type MGMT_LEAVE_CLID, which has been discussed previously (see Section 5.2.2.3), makes it possible to force the current node, a single remote node, or a node with its children to leave the network. Another ZDP command with type MGMT_NWK_UPDATE_CLID is designed to cope with multiple tasks, such as energy detection scan and changing a working channel. The need to change a working channel arises when the current channel has too much noise on it. This is usually done by a network manager, which performs the energy detection scan to determine the noise level on a specific node. If a node other than the network manager issues a request to change a working channel on a remote node, the request will be declined.

5.3.4 Joining notifications in secured networks

In a secured network the current node is notified about child's joining via ZDO_MgmtNwkUpdateNotf() function invoked with the ZDO_CHILD_JOINED_STATUS code. This indicates that the child node has joined the network, but has not yet passed authentication. The application shall not rely on this indication to start sending data to the child, because the child may not pass authentication.

When authentication is completed the child sends a device announcement frame. The application may subscribe to the device announcement event to track the end of child's authentication. This is done by having the application call the SYS_SubscribeToEvent() function. The first argument shall be set to the device announcement event's ID, which is BC_EVENT_DEVICE_ANNCE. The second argument shall point to an instance of SYS_EventReceiver_t type, having the func field set to the pointer to a callback function, which will be called when the specified event happens.

See the following example:

```
//The callback function
static void deviceAnnceCallback(SYS_EventId_t eventId, SYS_EventData_t data)
{
  ...
}
//Globally defined variable
SYS_EventReceiver_t deviceAnnceEventReceiver = { .func = deviceAnnceCallback };
...
SYS_SubscribeToEvent(BC_EVENT_DEVICE_ANNCE, &deviceAnnceEventReceiver);
```

The callback's `data` argument will contain the extended address of the device that has sent the device announcement frame. Since it has the eventId argument, the same callback function may be used to proceed several events, switching logic according to the event ID received.

Section 6
Data exchange

Obviously, the purpose of establishing a ZigBee network is to perform data exchange between remote nodes as required by application functionality. This section provides an extensive guide into all major aspects concerning data exchange.

Data exchange occurs between two endpoints located on different nodes in the network. An endpoint is a registered object that presents one of the application objects constituting the application framework. The overview section below describes endpoints and other important ZigBee concepts, and explains generally how data exchange is handled in the Atmel BitCloud stack. A practical guide to packing and sending a data request is provided in Section 6.2. The two subsections there are indented to give a basic introduction to the subject, allowing the developer to start using data exchange.

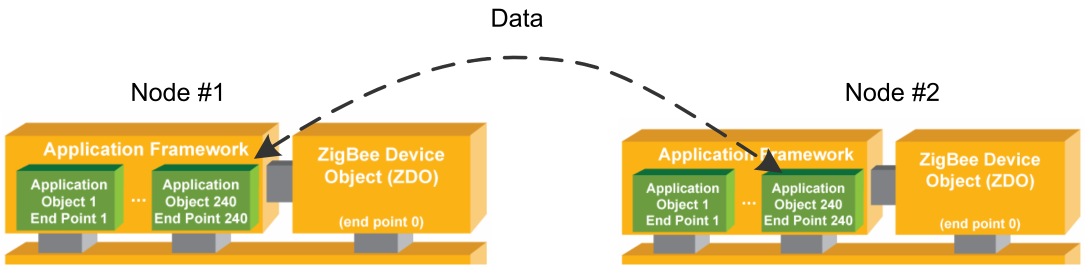
Further sections cover more advanced topics. Understanding these issues is essential to reveal and benefit from all the major BitCloud features. Multicasting and broadcasting allow sending data simultaneously to a group or to every node in the network. The polling mechanism enables delivering data to sleeping end devices by delaying the receipt and storing the message on a parent until the destination child wakes up. Binding enables establishing virtual connections between endpoints on different nodes that are not required to know each other's network addresses. Several other topics are also discussed.

6.1 Overview

On the application level, data is exchanged between application endpoints. A node does not participate in network communications until it registers at least one endpoint. If a node wants to deliver data to a remote node, it is not sufficient to know the remote node's network address. The node must specify a destination endpoint, too. Figure 6-1 illustrates this concept.

Endpoints correspond to application objects, which are logically embraced into an application framework. Each endpoint is identified with an endpoint ID, which can take values in a range from 1 to 240. Endpoint 0 is reserved for ZigBee Device Object (ZDO). Besides the endpoint ID, each application endpoint is characterized by a callback function, which is triggered upon data reception, and a set of parameters, such as the application profile ID, supported clusters, etc., which describes the behavior of the application object and constitute the so-called simple descriptor.

Figure 6-1. Data exchange between two endpoints.


Data
Node #1
Application Framework
Application Object 1 End Point 1
...
Application Object 240 End Point 240
ZigBee Device Object (ZDO)
(end point 0)
Node #2
Application Framework
Application Object 1 End Point 1
...
Application Object 240 End Point 240
ZigBee Device Object (ZDO)
(end point 0)

6.2 Clusters and profiles

At this point, some additional important ZigBee concepts shall be introduced. A cluster is a collection of commands and attributes that define an interface to specific functionality. A node supporting a particular cluster can be a server, a client, or both. A server is responsible for storing attributes, while a client manipulates the attributes, issuing commands to the server.

An application profile is a specification that determines application behavior. A profile defines several device types and a collection of supported clusters specifying which clusters shall be supported by each device type. Data exchange requires that communicating endpoints are assigned to the same profile. An application profile identifier is a mandatory field in a simple descriptor of every endpoint.

6.2.1 Endpoint registration

An endpoint can be registered at any time using the APS_RegisterEndpointReq() function, which must be provided with a simple descriptor of the endpoint and a callback function to be called upon data reception. The application is also free to unregister an endpoint using APS_UnregisterEndpointReq(). A simple descriptor specifies endpoint identifier, application profile, device identifier (within a given profile), and lists of supported input and output clusters. For input clusters, the node can perform the server role in cluster-specific interactions, while for output clusters the node can be a client. There is no restriction to add a cluster to both input and output lists.

The stack uses a simple descriptor to filter incoming frames, declining messages that specify a wrong profile ID. The indication of data reception is sent to the destination endpoint only when the message fits the profile and cluster information contained in the simple descriptor. A simple descriptor is also employed in service discovery procedures, which allow a node to search for devices with particular profile and cluster configurations.

An example of endpoint registration is given in Section 6.2.1.

6.2.2 Specifying a destination

When an endpoint is registered, the application can use it to send data to endpoints on remote nodes with the same application profile using the APS_DataReq() function. The application shall properly configure a data frame packet, clarify a destination, and issue a request to the APS layer. After that, the stack is fully responsible for delivering the data frame to the specified destination.

During data request configuration, a destination for a data frame can be set using different address modes. The simplest way is to apply short addresses. Another available option is sending data to a group of devices. Additionally, the application is able to deliver data without knowing the destination short address. To achieve this, service discovery and the binding feature are used. Binding is described further in Section 6.5. If the application has already established correspondence between a network device's short and extended addresses (with the help of ZDP request, see Section 6.5.1.2), it is also possible to send data to this device by specifying its extended address.

The application can send unicast, broadcast, or multicast messages. A unicast message is delivered to a single node identified by its short address or extended address, or as a remote node to which the current node is connected through binding. Broadcasting means that a message is sent to all nodes in the network or to all nodes except for end devices. To broadcast a destination short address in a request, parameters shall be set to a dedicated value such as 0xFFFF to spread a message over the whole network. Multicasting means addressing a group of nodes. For details concerning broadcasting and multicasting, refer to Section 6.3.

6.2.3 Routing

One of the main purposes of forming a network is to be able to send data to devices that cannot be reached by a signal directly. Nodes joined to the network can transmit a data frame to each other until it reaches the destination, thus allowing the message to cross large distances. Routing means the process of building an optimal path through several nodes in the network to the destination node. Without applying routing, a node would not be able to send data over more than a single hop; that is, only to devices that can directly "hear" the signal sent by the node.

In Atmel BitCloud applications, the stack is fully responsible for establishing and managing routes. These procedures are absolutely transparent to the application. Information obtained during route discovery is stored in internal structures such as the route table and route discovery table, and reused whenever possible.

The length of a route can be limited by assigning the radius field of data request parameters to a non-zero value. The radius field specifies the maximum number of transmission hops allowed for the message. If a message does not reach the destination within the specified number of hops, it will be dropped on its route and the confirmation callback on the data request originator will indicate failure. If the radius field is set to 0, then the maximum number of hops equals 2*CS_MAX_NETWORK_DEPTH.

6.2.4 Capturing data response and acknowledgement

When a unicast message is sent, the application can request an acknowledgement. If an acknowledgement is not requested, the stack calls the confirmation callback as soon as it receives a response from the device one hop away. Therefore, the success status in the callback only confirms that data propagation has started. If acknowledgement is requested, the stack defers calling the callback until it receives a response from the destination node. It should be noted that the response indicates delivery to the destination endpoint, not simply to the device. It also implies that the data frame has not been declined due to improper profile or cluster information, and that the indication function has been called on the destination endpoint.

If a node sends a message to more than one node, as with a broadcast message, then acknowledgement is not possible. In this case, the success status only indicates that the device managed to transmit data over the air to its neighborhood.

The stack may report the failure of a data request for various reasons. The specific reason can be determined with the help of the status value. Note that failure can occur not only on the APS layer, but on the underlying layers, such as NWK or MAC, as well.

In the case of a successful data delivery, the data indication callback associated with the destination node is executed, allowing the application on the destination node to perform whatever is required.

6.3 Implementing data exchange

This section provides a practical guide to implementing data exchange with the BitCloud stack. Data request and endpoint registration functions are provided by the APS component of the stack. The following sections explain how to register endpoints, configure data messages, and send and receive data, and provide many code examples to illustrate the usage of BitCloud interfaces.

6.3.1 Application endpoint registration

As described above, each node shall register at least one endpoint using the APS_ReqisterEndpointReq() function with an argument of the APS_RegisterEndpointReq_t type to enable communication on the application level. The argument specifies the endpoint descriptor (simpleDescriptor field), which includes such parameters as endpoint ID (a number from 1 to 240), application profile ID, and number and list of supported input and output clusters. In addition, the APS_DataInd field specifies the indication callback function to be called upon data reception destined for this endpoint.

The code snippet below provides an example of how to define and register application endpoint 1 with profile ID equal to 1 and no limitation regarding supported clusters.

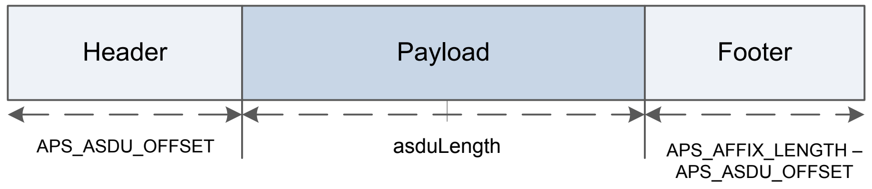
```
//Specify endpoint descriptor
static SimpleDescriptor_t simpleDescriptor = {1, 1, 1, 1, 0, 0, NULL, 0, NULL};
//variable for registering endpoint
static APS_RegisterEndpointReq_t endpointParams;…
//Set application endpoint properties
…
endpointParams.simpleDescriptor = &simpleDescriptor;
endpointParams.APS_DataInd = APS_DataIndication;
//Register endpoint
APS_RegisterEndpointReq(&endpointParams);
```

6.3.2 Data frame configuration

In order to perform data transmission itself, the application first needs to create a data transmission request of APS_DataReq_t type that specifies the APS service data unit (ASDU) payload (asdu and asduLength fields), sets various transmission parameters, and defines the callback function (APS_DataConf field) to be executed to inform the application about transmission result.

Because the stack requires the ASDU as a contiguous block in RAM with specific characteristics, the application must construct such a structure, as shown in Figure 6-2. See the ZigBee Specification [3] for more details on ASDU.

Figure 6-2. ASDU format.



The maximum allowed application payload size depends on the security mode enabled, as shown in Table 6-1. The more secure the transmission, the smaller the size of the payload.

Table 6-1. Maximum payload size for different stack configurations.

Option	Maximum ASDU size
Default (no security)	95
Standard security	77
Standard security with link keys	59

The code extract below provides an example of how to create a data request with a correctly structured

ASDU:

```
//Definitions in a global scope
//Application message buffer descriptor
BEGIN_PACK
typedef struct
{
  uint8_t header[APS_ASDU_OFFSET]; // Header
  uint8_t data[APP_ASDU_SIZE]; // Application data
  uint8_t footer[APS_AFFIX_LENGTH - APS_ASDU_OFFSET]; //Footer
} PACK AppMessageBuffer_t;
END_PACK
static AppMessageBuffer_t appMessageBuffer; // Message buffer
APS_DataReq_t dataReq; // Data transmission request
...
//Assigning payload memory for the APS data request
dataReq.asdu = appMessageBuffer.data;
dataReq.asduLength = sizeof(appMessageBuffer.data);
```

The application should allocate enough memory for the entire ASDU, including frame headers and footers required by the ZigBee protocol, as shown in the example (`AppMessageBuffer_t`). To specify header, data, and footer lengths, certain predefined APS constants shall be used. Although the asdu field in the data request parameters is a pointer to a message buffer data section only (`appMessageBuffer.data`), the stack calculates indents to employ header and footer sections of the buffer to write other request parameters and security related information.

Note the use of the BEGIN_PACK and END_PACK macros embracing the AppMessageBuffer_t structure definition. These macros disable the data structure alignment applied on MCUs with a word length different from 8 bits, thus preventing the insertion of unused bytes into the structure. Since a data frame structure requires a continuous block of memory without unnecessary gaps, the BEGIN_PACK/PACK/END_PACK macros should be used for all data frame structures that are to be sent over the air. At the same time, these macros must be used with care, because some compilers may not process them correctly and cause unexpected MCU failures at runtime.

After the payload for the data request is correctly configured, the developer is likely to choose an addressing mode by assigning the data request dstAddrMode field to one of the enumerated values. Depending on the addressing mode, different sets of parameters are set to identify the destination of the message. If a direct addressing scheme is used (any scheme except for APS_NO_ADDRESS), then the source node shall have full knowledge about the destination; that is, node address, application profile ID, application endpoint, and input cluster (specify dstAddress, dstEndpoint, and clusterid). In the indirect addressing scheme (APS_NO_ADDRESS), these parameters are set during the binding procedure and are not required for a data request. Binding is described in Section 6.5.

In the following example, which continues the previous one, a destination node is set to direct addressing, all other necessary parameters are assigned, and the application requests data transmission to the specified destination:

```
dataReq.profileId = APP_PROFILE_ID;
dataReq.dstAddrMode = APS_SHORT_ADDRESS;
dataReq.dstAddress.shortAddress = CPU_TO_LE16(0);
dataReq.dstEndpoint = 1;
dataReq.clusterId = CPU_TO_LE16(1);
dataReq.srcEndpoint = APP_ENDPOINT_ID;
dataReq.txOptions.acknowledgedTransmission = 1;
dataReq.radius = 0x0;
dataReq.APS_DataConf = APS_DataConf;
APS_DataReq(&dataReq);
```

The example assumes that APP_PROFILE_ID and APP_ENDPOINT_ID are application-defined constants describing the source endpoint. The data frame will be sent to the coordinator, since it is the only node with a short address equal to 0. It is assumed that the coordinator has an endpoint under the number 1, which supports an input cluster with ID equal to 1. The CPU_TO_LE16 macro converts a 16-bit value to a 16-bit little endian value. Frame transmission is requested to be acknowledged, and the number of hops is not limited, as the radius is set to 0. The stack will report the request result by calling a callback named APS_DataConf(), which is a user-defined function with the following signature:

```
static void APS_DataConf(APS_DataConf_t *confInfo)
```

6.3.3 Tx options

A data request is additionally configured through the txOptions field of the data request parameters. By default, all txOptions fields are set to 0. To turn on an option the corresponding txOptions field should be set to 1. All txOptions fields are described in Table 6-2. Besides, following sections add comments on the use of tx options where these tx options are applicable.

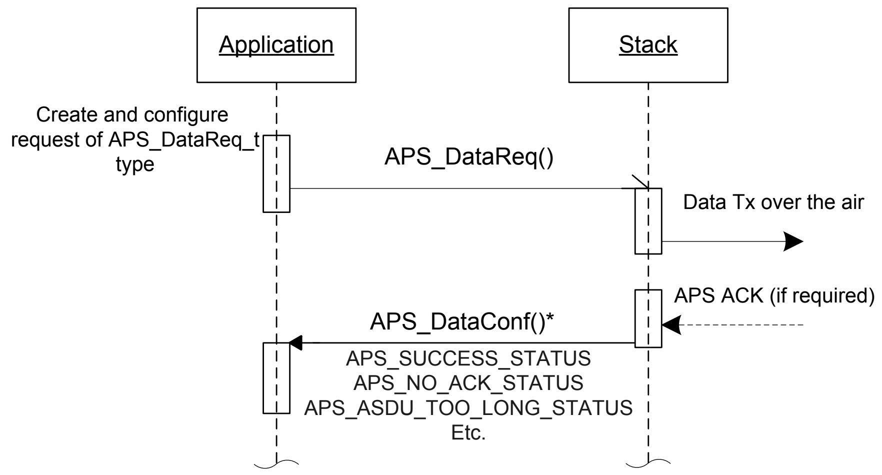
Table 6-2. Data frame tx options.

Option	Description
`securityEnabledTransmission`	Indicates if the application payload is encrypted; see Section 8.3.2
`useNwkKey`	Use network key (instead of a link key) to encrypt the application payload
`acknowledgedTransmission`	Request for acknowledgement of successful delivery
`fragmentationPermitted`	Enable fragmentation; see Section 6.6
`includeExtendedNonce`	Include extended address of the sender into the auxiliary header of a data frame; this allows a recipient to select a proper link key to decrypt the application payload even if the correspondence between short and extended addresses is not established
`doNotDecrypt`	The stack encrypts a data frame in place, in the memory provided by the user, and decrypts it after the frame has been sent. Decryption may be turned off to speed up processing.
`indicateBroadcasts`	Send a broadcast frame to yourself; see Section 6.3.2

6.3.4 Data frame transmission

After a data request is created and all parameters are set as required, the application can transfer it to the APS layer to perform the actual over-the-air transmission using the APS_DataReq() function. Figure 6-3 illustrates sequence diagram on the source node during data transmission.

Figure 6-3. Data transmission.



* Function registered as callback for argument of `APS_DataReq()`.

When the application sends data, APS_DataReq() automatically performs a route request and finds the most reliable path to the desired destination. Moreover, the stack keeps that path and updates it when the network topology or link qualities change. Thus, the next time the application sends data to the same destination, route discovery consumes no time because the best route is already known. The application can set the maximum number of hops allowed for the transmission of the frame (and hence limit the latency) by setting the corresponding number in the radius field of the data transmission request (APS_DataReq_t).

Since the ZigBee protocol allows bidirectional communication, applications can request acknowledgement of the data frame reception on the application level (so-called APS ACK) by setting acknowledgedTransmission to 1 in the txOptions field of the data request. In this case, the application is notified about successful frame delivery (APS_SUCCESS_STATUS) via the registered confirmation callback function only after an acknowledgement frame for the corresponding data frame is received. If during the CS_ACK_TIMEOUT interval no acknowledgement is received, the callback function with APS_NO_ACK_STATUS is issued.

Transmission with APS ACK turned off provides higher data throughput, but is not reliable because frame delivery cannot be confirmed. In this case, if all parameters are set correctly, confirmation callback with APS_SUCCESS_STATUS is called after the transceiver has sent the frame over the air.

As for other asynchronous requests, an instance of request parameters, that is, of the APS_DataReq_t type, and the memory allocated to ASDU can be reused only after the corresponding confirmation callback is executed.

6.3.5 Data frame reception

As described in Section 6.2.1, in order to enable data exchange on the application level, the node must register at least one application endpoint with an indication callback function for the data reception procedure.

After a frame destined for the node is received by the transceiver, the stack verifies whether the destination endpoint indicated in the frame header has a corresponding match among endpoints registered on the node. When such an endpoint exists, the corresponding indication function, as specified in the APS_DataInd field of the endpoint registration request (APS_RegisterEndpointReq_t), will be executed with an argument of the APS_DataInd_t type. This argument contains the application message (asdu field) as well as information about the source and destination addresses, endpoints, profile, etc.

A data frame received by the stack is written to the memory allocated for stack use during compilation. If the application is going to use the received data later, it should prepare a buffer and write the data to the buffer in the indication callback.

The following example shows how an indication callback can be implemented:

```
void APS_DataInd (APS_DataInd_t* indData)
{
  //Assuming that the payload contains a structure of a particular type,
  //convert asdu to that type
  AppMessage_t *appMessage = (AppMessage_t *) indData->asdu;

  //Suppose the application processes not types of messages
  if (appMessage->messageId == appMessageForUsart)
  {
    //Save data to a buffer for further use
    saveData(appMessage->data, indData->asduLength - 1);
  }
}
```

It is assumed that the application defines the AppMessage_t type to hold the application-specific information that is exchanged with other devices in the network. During request parameter configuration, an instance of AppMessage_t can be written to the data field of the message buffer.

6.4 Broadcast and multicast transmission

In addition to unicast (node-to-node) transmissions, Atmel BitCloud applications can send data to more than one node in the network at a time by applying either broadcasting or multicasting. Broadcast data frames are destined for all network nodes or all nodes with specific properties. Broadcasting is useful when the application needs to notify every device in the network with a single request, but since the delivery of all messages is not guaranteed, it cannot be applied for issuing messages of critical importance.

Multicasting is very similar to broadcasting except that messages are transmitted to an arbitrary group of nodes. Nodes can add themselves to any groups. Moreover, members of a particular group are not aware of other members of the same group. The application could implement multicasting with the use of broadcast messages and additional checking in the data indication function. In fact, on the network level, multicasting is partly based on broadcasting. The reasons for using multicast transmission as implemented in the Atmel BitCloud stack are convenience and the reduced number of messages sent to the air during message delivery when compared with broadcasting.

6.4.1 Sending a broadcast message

To send a broadcast data frame, the APS_SHORT_ADDRESS addressing mode shall be specified in the request parameters, and the destination address shall be assigned to one of the predefined values. The following enumerators can be used as the destination address (corresponding hexadecimal values may be set instead):

- All nodes in the network: BROADCAST_ADDR_ALL (or 0xFFFF)
- All nodes with rxOnWhenIdle equal to 1: BROADCAST_ADDR_RX_ON_WHEN_IDLE (or 0xFFFD)
- All router nodes: BROADCAST_ADDR_ROUTERS (or 0xFFFC)

Broadcast frames cannot be acknowledged and must be transmitted with acknowledgedTransmission set to 0 in the txOptions field of the data request.

By default, a device that sends a broadcast frame does not receives the frame itself; that is, the data indication callback is not called. But this behavior is changed if the indicateBroadcasts field of txOptions is set to 1. In this case data indication callback on the destination endpoint is called on the device that initiated the broadcast transmission as well.

In addition to broadcasting a frame over the network, the application can configure the transmission so that the frame is delivered to all endpoints registered on the destination nodes. This can be done by setting the dstEndpoint field in the data request to APS_BROADCAST_ENDPOINT (equal to 0xFF). Such a broadcast on the node level can be performed for unicast transmissions, too.

As with a unicast frame, a broadcast message can be sent out with a limited number of hops (as configured via the radius field). If the radius is set to 0, all nodes in the network that correspond to the destination type will be covered by the transmission.

6.4.2 Broadcast implementation

On the network level, the broadcast procedure is performed as follows: after the APS_DataReq() function with a broadcast data request is called, the node sends the frame up to three times over the air. Each node, after receiving a copy of this frame (only one copy is accepted, others are ignored), decreases the transmission radius by one, and if it is greater than zero, broadcasts the message three times to its neighbors. Such a procedure repeats on the other nodes until the transmission radius is exhausted.

If a node, after re-transmitting a broadcast frame, receives the same frame from all its neighbors within 500ms it stops re-transmitting the broadcast frame. So a node will re-transmit a broadcast frame only once, if it discovers that all its neighbors received the frame. This feature significantly reduces the amount of data exchanged during a broadcast transmission, since a node normally receives replies from all of its neighbors in a few milliseconds and, thus, only a single re-transmission of a broadcast frame is typically needed.

6.4.2.1 Broadcast transmission table

To be able to recognize a broadcast message that has been already processed, each node maintains a broadcast transmission table. Information about a message that has been initiated by the node or propagated to its neighbors is written to the table and kept there for a certain period of time. The duration for keeping the frame depends on the network depth; the greater the depth, the longer a message is kept. While the information about the message is present in the table, the stack declines all identical messages sent by other nodes and the indication callback is not called. Thus, the message is processed by the application only once. After the time for keeping the message is exceeded, information about it is dropped. As with other internal tables, the size of the broadcast transmission table is given by a ConfigServer parameter, CS_NWK_BTT_SIZE.

Note that sending broadcast data frames too often may overload a working channel considerably, preventing the delivery of important unicast messages. Besides, it is not recommended to broadcast messages of critical importance, because successful delivery to all nodes in the network cannot be verified.

6.4.3 Multicast transmission

Multicasting allows data to be transmitted to an arbitrary group of nodes with a single data request. Groups are not registered or controlled in any way. In addition, as outlined earlier, a member of a particular group is not aware of other members of the same group. Joining groups just allows a node to accept or decline messages sent to those groups.

6.4.3.1 Adding a device to and removing it from a group

Each node can add itself to a group with a specified group address by calling the APS_AddGroupReq() function. The function is synchronous. The user shall provide two parameters: a 16-bit group address, which identifies the group on the network level, and an endpoint number to use for communication. It is possible to call the function several times with the same group address and different endpoints specified, and vice versa. When a node receives a group message, it executes indication functions for all endpoints associated with the given group address. The number of groups a node can be registered with depends on the CS_GROUP_TABLE_SIZE parameter, which determines the size of the group table. The table contains an entry for each group and endpoint pair that has been registered.

To unsubscribe a node from a particular group or from all groups, the application on the node calls APS_RemoveGroupReq() or APS_RemoveAllGroupsReq(), respectively.

6.4.3.2 Sending a data frame to a group

Issuing a message to a group is done by calling APS_DataReq() with the addressing mode set to APS_GROUP_ADDRESS and the destination address field, groupAddress, assigned the group address (specified by the members of the group in APS_AddGroupReq() parameters). The destination endpoint shall be set to 0xFF. As with other types of data requests, the radius limits the maximum number of hops between the source node and nodes that can be reached by the request.

6.4.3.3 Multicast implementation

Multicast implementation on the network layer is more complicated than broadcasting, and consists of several steps. First, route discovery is performed to find a group member with the least route cost so that sending data frame to it will take minimal network resources. When that group member is found, the stack sends it a unicast message. Upon receiving this message, the group member starts retransmitting it in a broadcast manner with the group address included. Group members that receive the frame retransmit it and execute indication functions. Other nodes also pass the frame further as well as decrease a special non-member radius field if its value is greater than 0. This non-member radius field is different from the radius provided in the initial request. If the value of the non-member radius equals 0, the frame propagation stops. This means that if a cluster of group members is located beyond the non-member radius of other group members, then this cluster may not receive the message destined to the group.

Apparently, multicasting is useful and may reduce considerably the amount of data sent to the air if group members are located in a bounded area close to each other. If the group members are spread over the whole network, however, addressing them via multicasting may be not very efficient, because not all members may be reached.

6.5 Polling mechanism

In ZigBee networks, a polling mechanism is used to deliver data to sleeping end devices over the last hop (that is, from a parent node to its child node). If polling is applied for an end device, the parent does not transfer data destined to the end device child directly, but buffers it instead, because the end device may be asleep and not receive the signal. When the end device wakes up, it sends a poll request to its parent. If the buffer contains messages for the child, the parent then delivers the buffered data. While the end device is awake, it sends polling requests to its parent periodically to retrieve buffered data, if any. The time interval between two polling requests equals CS_INDIRECT_POLL_RATE milliseconds.

6.5.1 Polling configuration on end devices

An end device's parent buffers not only unicast messages with the destination address of its child and broadcast frames with non-exhausted transmission radius, but also acknowledgements for the frame data requests issued by the end device.

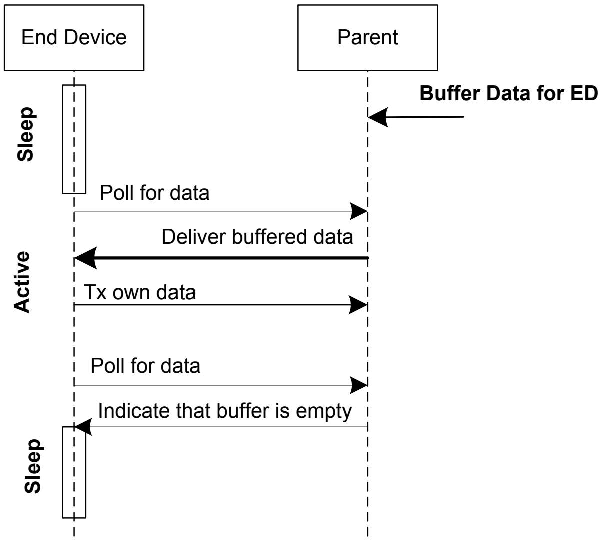
An end device sends its own data frames to other nodes in the network only via its parent. Data is sent immediately when such transmission is requested by the application. However, as described in the previous paragraph, retrieving the APS acknowledgement is possible only after a data poll. Since data polls occur periodically as set by CS_INDIRECT_POLL_RATE, reception of the APS acknowledgement cannot be expected in less than this interval after a data frame is sent by the end device. To reduce this interval, the application can send poll requests manually, as described below.

Polling is handled by the lower stack layers and is transparent to the application, which can only control it with the help of certain ConfigServer parameters and requests to the stack. To use polling, an end device shall have CS_RX_ON_WHEN_IDLE set to false (see the example in Section 5.1.1.2), indicating that the radio is turned on only for short periods of time when the device is actually sending data. After transmitting a frame, the device waits for a certain period of time and then turns the radio off when that time has elapsed. After an end device enters a network, it cannot change CS_RX_ON_WHEN_IDLE as a usual parameter via the ConfigServer interface, because if it could change the parameter, the parent would not be able to recognize the change and would continue buffering messages. However, an end device is able to control the sending of poll requests. The ZDO_StopSyncReq() and ZDO_StartSyncReq() synchronous requests stop and restart the sending of poll requests to a parent, respectively. After it has stopped the automatic sending of poll requests, the application can issue them manually with the NWK_SyncReq() function.

Note that ZDP requests do not reach sleeping end devices; see Section 6.4.1.1 for detail.

Figure 6-4 illustrates polling mechanism processing.

Figure 6-4. End device polling mechanism.



6.5.1.1 ZDP requests and sleeping end devices

ZDP requests do not reach sleeping end devices. A sleeping end device's parent does not store and re-transmits when polled frames with ZDP commands addressed to its sleeping child. Instead, the parent answers to ZDP commands itself, if it has enough information about the device for that. Thus, it can respond with the extended or short address of its child.

If the parent is not able to answer the request it reports the failure status to the originator of the request. For example, a match descriptor request cannot be executed (see Table 5-3 and Table 6-3 that list typical ZDP commands). A request to leave the network is remembered by the parent, which sends the success status to the originator and eventually forces the child to leave the network when the child wakes up.

6.5.2 Polling configuration on fully functional devices

Special care should be taken when configuring fully functional devices, specifically routers and the coordinator, that communicate with sleeping end devices. The CS_MAC_TRANSACTION_TIME parameter determines how much time a cached frame is being stored on a sleeping end device's parent. Upon receiving a frame for a sleeping child, a parent calculates the time when the child is expected to wake up by adding the CS_END_DEVICE_SLEEP_PERIOD parameter value to the time when the last poll request was received from the child. If the time till the calculated moment is greater than CS_MAC_TRANSACTION_TIME, the frame is not stored and the parent responds to the originator with an error, since the end device is not likely to wake up during the timeout. Otherwise, the frame is saved to the buffer and is transferred to the destination upon receiving a poll request from it. If a poll request is not received within CS_MAC_TRANSACTION_TIME after frame arrival, then the buffered frame is dropped.

Note that if the frame is stored for a long time, the originator may cease waiting for a response, since its timeouts have exceeded, and consider the delivery as failed even though the frame may successfully reach the destination. Thus, the application developer should track a frame storing timeout on functional devices with care. Note also that a parent is not informed about an end device sleep period automatically. Instead, it uses the CS_END_DEVICE_SLEEP_PERIOD value, the same for all children.

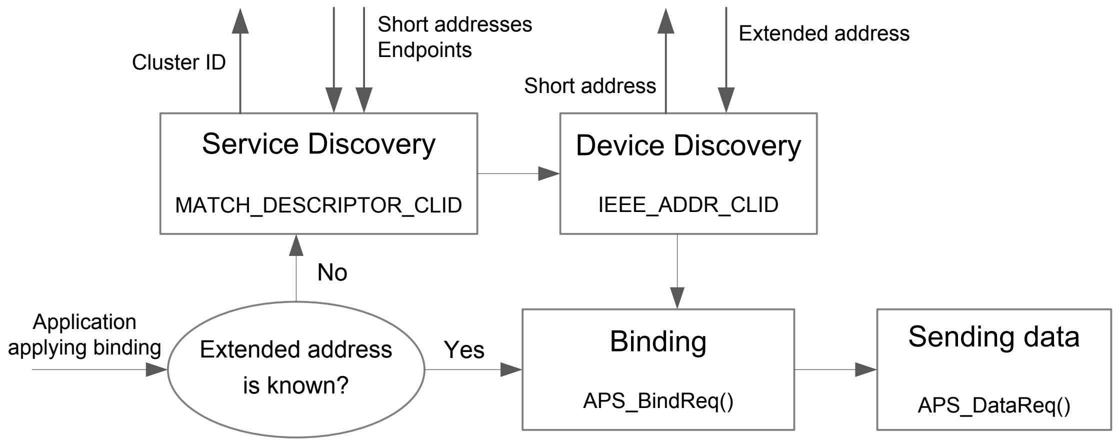
This implies that if an end device has an actual sleeping period that is longer than the parameter value, then messages for such a child may be dropped.

6.6 Binding

Binding is an important feature defined in the ZigBee specification. Binging enables the establishment of virtual connections between endpoints on different nodes so that, during data transmission between the connected endpoints, the destination is identified by a specific cluster, rather than by the destination address.

The stack keeps entries for each binding in the binding table. To establish a new connection, the application inserts an entry into the binding table. If the remote node is going to reply, it may be useful to add an entry into the binding table on the remote node as well, which is achieved with a dedicated ZDP request. When the entry to the binding table has been added, binding is considered established and data can be sent to the remote node with usual data requests.

Figure 6-5. Applying binding.



To bind to a device, the application must know the extended address of the device and the destination endpoint. If it is unknown to the application, additional preparation must be done. The application can collect information about network devices that support a required cluster with the help of the service discovery procedure. This procedure returns short addresses of discovered devices, as well as other parameters. But since binding requires extended addresses, the application should perform one more device discovery operations to set up a correspondence between extended and short addresses. Both service and device discovery are carried out with the help of ZDP requests. Once the information has been collected, the application can bind to the devices by inserting corresponding entries into the binding table. It is also possible to request a remote device to add an entry to its binding table via a ZDP request. After all preparation is completed, the data can be transmitted with a simple APS data request to all nodes that support a specific cluster. Figure 6-5 illustrates the described process.

It is also possible for a node to bind to an entire group. For this purpose, a node must know a group address. Binding is established by adding an entry to the local binding table with a group address inside and issuing a binding ZDP request, which is delivered to all group members. Details on group binding are provided in Section 6.5.3.

To better understand when binding is required, consider the following example. Suppose a network contains more than one metering device and a controlling device. The controller, which in terms of Smart Energy profiles is called the Energy Service Portal, has to discover all the meters present in the network and collect information from them. Meters support a simple metering cluster for which meters are servers because they store attributes with metered information. The controller does not know anything about meters except that they support the metering cluster, which it also supports acting as a client. And so the controller has to run a service discovery operation to learn the meter addresses, bind to them, and then request a reading of the attributes with a single data request.

The following sections provide details on obtaining information about nodes and how to apply binding. Section 6.5.4 is dedicated to service and device discovery.

6.6.1 Obtaining information about devices in the network

6.6.1.1 Using service discovery

If a node wishes to communicate with devices that support a specific cluster, but does not possess any information about them, it can use service discovery to find out whether such devices are present in the network and obtain their short addresses and the endpoints that support the cluster.

To perform service discovery, the application issues a ZDP request with the cluster type equal to MATCH_DESCRIPTOR_CLID. It is possible to indicate in the request payload either a specific short address for sending a unicast request or a broadcast address to search for devices across the whole network. The application can specify lists of input and output clusters, although it is rarely useful to point out more than one cluster. A node that receives a match descriptor request responds with a list of all its endpoints that support at least one of the input or output clusters specified in the request. Thus, if the request contained more than one cluster in two lists, the application is not able to distinguish received endpoints according to clusters they support. Each matching device replies with its short address and a list of matching endpoints. For each reply, the stack on the originator node calls a confirmation callback. The stack waits for replies until a special timeout expires. After that, it calls the callback function for the last time, reporting the ZDO_CMD_COMPLETED_STATUS status value.

The following code example shows how to send a service discovery request:

```
//define global variables
static ZDO_ZdpReq_t zdpReq;
...
ZDO_MatchDescReq_t *matchDescReq = &zdpReq.req.reqPayload.matchDescReq;
zdpReq.ZDO_ZdpResp = zdpMatchDescResp; //confirmation callback
zdpReq.reqCluster = MATCH_DESCRIPTOR_CLID; //set request type
matchDescReq->nwkAddrOfInterest = CPU_TO_LE16(BROADCAST_SHORT_ADDRESS);
matchDescReq->profileId = CPU_TO_LE16(APP_PROFILE); //set profile ID
matchDescReq->numInClusters = 1; //number of clusters
matchDescReq->inClusterList[0] = APP_CLUSTER; //set cluster ID
ZDO_ZdpReq(&zdpReq); //send the request
```

In the code above, it is assumed that APP_PROFILE and APP_CLUSTER are constants defined by the application and that they designate the profile and cluster identifiers of interest. The implementation of the confirmation callback might be as follows:

```
static void zdpMatchDescResp(ZDO_ZdpResp_t *resp)
{
  ZDO_MatchDescResp_t *matchResp = &resp->respPayload.matchDescResp;
  if (ZDO_CMD_COMPLETED_STATUS == resp->respPayload.status) {
    //timeout has expired; this is the last time the callback is called
    //for the current request
  }
  else if (ZDO_SUCCESS_STATUS == resp->respPayload.status) {
    //process another response from the network
  }
  else {
    //process failure statuses
  }
}
```

In the example above, the matchResp->nwkAddrOfInterest field contains the short address of the responding device. The matchResp->matchList field is an array containing discovered endpoint IDs. The number of matching endpoints received with the response is given by matchResp->matchLength. For example, if it equals 1, the only endpoint is held in matchResp->matchList[0].

6.6.1.2 *Obtaining the extended address by a short address*

Since binding requires knowledge of an extended address of a remote node, the application is expected to discover it with the use of the corresponding ZDP request of IEEE_ADDR_CLID type, which is one of the requests that enable device discovery. The following code illustrates how to issue the request:

```
ZDO_IeeeAddrReq_t *ieeeAddrReq = &zdpReq.req.reqPayload.ieeeAddrReq;
zdpReq.ZDO_ZdpResp = zdpIeeeAddrResp; //confirmation callback
zdpReq.reqCluster = IEEE_ADDR_CLID; //type of request
ieeeAddrReq->nwkAddrOfInterest = nwkAddr; //short address received in the
                                          //previous example
ieeeAddrReq->reqType = SINGLE_RESPONSE_REQUESTTYPE;
ieeeAddrReq->startIndex = 0;
ZDO_ZdpReq(&zdpReq); //send the request
```

Since the network address is already known, the destination device is identified within the network, and the request requires just a single, unicast frame transmission. Upon receiving the response with the extended address enclosed, the application may insert an entry to the binding table immediately using the APS_BindReq() function. This topic is covered in detail in the next section. Consider the following example of callback implementation:

```
static void zdpIeeeAddrResp(ZDO_ZdpResp_t *resp)
{
  ZDO_IeeeAddrResp_t *ieeeAddrResp =
                     (ZDO_IeeeAddrResp_t*)&resp->respPayload.ieeeAddrResp;
  if (ZDO_SUCCESS_STATUS == resp->respPayload.status)
  {
    //Obtain desired extended address from the ieeeAddrResp->ieeeAddrRemote field.
  }
}
```

Once a pair of short and extended addresses of the same device is established, it is saved by stack in the address map table. The application can get the extended address by a short address from this table, using the NWK_GetExtByShortAddress() function, and do the opposite task via the NWK_GetShortByExtAddress() function. These functions may be also used to check if the mapping between a short or extended address and its counterpart is already known to the stack: the functions return zero if the requested mapping is not found.

6.6.2 Implementing binding

6.6.2.1 Binding to a specific device

If the application is aware of the destination extended address of the node, it is ready to bind to that node. Binding is established by adding an entry to the binding table via the APS_BindReq() function.

Calling to the APS_BindReq() function is synchronous because no request has to be sent to the network. The function must be supplied with information about the originator, including extended address, source endpoint, and cluster identifier, as well as the destination address and endpoint. The address mode shall be set to APS_EXT_ADDRESS. Group address mode is also permitted if the application is going to bind to a group (see Section 6.5.3). Note that the application can learn the local extended address by reading the CS_UID parameter via the CS_ReadParameter() function.

Consider the following example, which illustrates the use of APS_BindReq():

```
 APS_BindReq_t apsReq; //binding request parameters need not to be given by a

                      //global variable since function call is synchronous
CS_ReadParameter(CS_UID_ID, &apsReq.srcAddr); //read own extended address
apsReq.srcEndpoint = APP_ENDPOINT; //assign to application-defined constant
apsReq.clusterId = APP_CLUSTER; //assign to application-defined constant
apsReq.dstAddrMode = APS_EXT_ADDRESS;
apsReq.dst.unicast.extAddr = dstExtAddr; //assign to the extended address of the
                                         //destination node
apsReq.dst.unicast.endpoint = dstEndpoint; //assign to the destination endpoint
APS_BindReq(&apsReq); //synchronous call to APS
```

Binding will fail if the binding table capacity is exceeded. The size of the binding table is given by CS_APS_BINDING_TABLE_SIZE, which like all other table sizes can be set only at compile time, as it affects the amount of memory that has to be allocated. Note that the default value is set to 1 to reduce the memory required by the stack. If the application is likely to bind to more than one device, a proper value should be considered for the binding table size.

To unbind from a remote node, that is, to remove the corresponding entry from the binding table, invoke the APS_UnbindReq() function. Two more APS functions will help control binding links, namely APS_DeactivateBindRecords() and APS_ActivateBindRecords(), which respectively switch off and on the entries in the binding table with the specified destination extended address.

6.6.2.2 Sending data to a bound device

After binding has been established, the application can start sending data to the destination device. Data transmission is performed with the APS_DataReq() function, as usual. In addition to confirmation callback, the request parameters should identify the source endpoint and the cluster identifier. The address mode shall be set to APS_NO_ADDRESS. If a data request is a command to a remote device as a piece of cluster functionality, then the command itself may be written to the data payload.

Note that requesting a data delivery acknowledgement is impossible since the number of nodes that will receive the message is unknown to the application and could be greater than one.

Reading entries from the binding table is not as straightforward as adding entries, and should be used with care. The user can obtain a pointer to the memory allocated for the binding table by applying the CS_GetMemory() function, as follows:

```
APS_BindReq_t* bindingEntry;
uint8_t bindingTableSize;
//Obtain a pointer to the binding table memory
CS_GetMemory(CS_APS_BINDING_TABLE_ID, (void *)&bindingEntry);
//Read the size of the binding table
CS_ReadParameter(CS_APS_BINDING_TABLE_SIZE_ID, (void *)&bindingTableSize);
```

Since the binding table size can be easily read with CS_ReadParameter(), the application can then iterate through the table, skipping to the next entry by adding 1 to bindingEntry. It is strictly required to ensure that the pointer to the current table entry does not leave the table memory.

The application is also able to request a remote node to insert an entry into its own binding table. This is not mandatory, because adding information to the binding table is sufficient to send data frames to specified nodes, but in some scenarios this option may be useful. To fulfill the task, the application should issue a ZDP request of the BIND_CLID type destined to the remote node. In this case, unbinding can be performed with a ZDP request of type UNBIND_CLID.

6.6.3 Binding to a group

Binding can be established not only with distinct devices, but also with a whole group. In this case, a node that wishes to bind to a group has to know the group address. The binding itself is set via the APS_BindReq() function, as in the case of unicast binding. Invocation of this function inserts an entry to the binding table, and is done similarly to the unicast case. Consider the following example:

```
APS_BindReq_t apsReq; //binding request parameters need not to be given by a
                      //global variable since function call is synchronous
CS_ReadParameter(CS_UID_ID, &apsReq.srcAddr); //read own extended address
apsReq.srcEndpoint = APP_ENDPOINT; //assign to application-defined constant
apsReq.clusterId = APP_CLUSTER; //assign to application-defined constant
apsReq.dstAddrMode = APS_GROUP_ADDRESS;
apsReq.dst.group = groupAddr; //set to the address of the group (16bit value)
APS_BindReq(&apsReq); //synchronous call to APS
```

Data transmission is done in the same way as in the unicast case. Note that group members that do not support a cluster specified within the request will decline the message.

6.6.4 ZDP requests maintaining service and device discovery

Service discovery represents a set of procedures for obtaining information about network devices that support specific endpoint configurations as well as for obtaining information about services provided by a specified device.

Device discovery represents two reverse procedures: one for obtaining the extended address of a node with a specified short address, and another for discovering a node in the network with a specific extended address. The first procedure is accomplished with a single unicast request, since the location in the network of the device is known. The second procedure consumes a greater amount of time, since the presence of the device is unknown and requires the node to propagate a series of broadcast request to discover it.

Table 6-3 summarizes the ZDP requests concerned with both service and device discovery.

Table 6-3. ZDP requests maintaining service and device discovery.

Status	Description
`NWK_ADDR_CLID`	Requests the short address of a remote device with a known extended address. Broadcasting is used because location in the network is unknown.
`IEEE_ADDR_CLID`	Requests the extended address of a remote device with a known short address. Accomplished with a single unicast request since the network location is given. See the example in Section 6.5.1.2.
`MATCH_DESCRIPTOR_CLID`	Launches a search for devices that registered endpoints supporting at least one of the specified clusters. Callback is initiated for each discovered device with a list of suitable endpoints registered on a device. See the example in Section 6.5.1.1.
`SIMPLE_DESCRIPTOR_CLID`	Requests a simple descriptor of the specified endpoint on a remote node.
`BIND_CLID`	Requests a remote node to insert an entry to its binding table with the specified source and endpoint addresses.
`UNBIND_CLID`	Requests a remote node to remove an entry from its binding table with the specified source and endpoint addresses.

For instructions and examples of how to issue a particular ZDP request, refer to Sections 5.3.2 and 6.5.2. More examples can be observed in the sample applications provided with the SDK. For a complete list of ZDP requests, refer to the Atmel BitCloud API Reference.

6.7 Fragmentation

When an application needs to send an amount of data larger than what actually fits into a frame, it can manage on its own by breaking the data into smaller pieces and issuing multiple data requests, or it can apply the Atmel BitCloud fragmentation feature, which compliant with the ZigBee PRO standard. In the latter case, the stack handles all implementation details so that the application needs only to configure a data request to apply fragmentation.

6.7.1 The maximum data frame payload size

The maximum size of an application payload for a data frame sent to the air is indicated by the APS_MAX_ASDU_SIZE constant. The actual value depends on stack configuration options, such as the security level. Note that different stack configurations require different library versions to be used. If security is enabled, a data frame contains additional information, such as auxiliary headers, hash values for encrypting the message, etc. Hence the maximum payload size of a single frame decreases. For details concerning security, refer to Section 8. Table 6-1 shows payload size limits applied in the BitCloud stack. Note that fragmentation requires an additional header to be inserted into the frame and thus consumes some bytes in addition to the values in the table, decreasing the maximum payload size.

6.7.2 Enabling fragmentation in a data request

To enable fragmentation, the fragmentationPermitted field in txOptions should be set to 1 in the data request parameters. Additionally, fragmentation requires acknowledgement transmission to ensure delivery of all pieces of data. The following sample code illustrates the usage:

```
APS_DataReq_t messageParams; //global variable
...
messageParams.txOptions.acknowledgedTransmission = 1;
messageParams.txOptions.fragmentationPermitted = 1;
...
```

The ASDU buffer should be defined as for a common request (see Section 6.2.2 for an example), except that the data section length can be larger than the maximum payload size. In most cases, it can be as large as the application needs to hold the whole message payload. However, the amount of data transmitted with a single request to APS is limited, because the stack has to allocate enough memory to store all parts of a fragmented message, if it receives one. The maximum is equal to CS_APS_MAX_BLOCKS_AMOUNT × CS_APS_BLOCK_SIZE provided the latter value is non-zero, and CS_APS_MAX_BLOCKS_AMOUNT × APS_MAX_ASDU_SIZE if CS_APS_BLOCK_SIZE is set to 0.

Note that the asduLength field should be set to the overall data length.

The confirmation callback for the request is executed when all data fragments are delivered and the stack receives acknowledgement for the last frame. On the destination endpoint, the indication is raised when all fragments have been received. An argument passed to the indication callback is a pointer to the structure that contains the whole message sent by the originator.

6.7.3 Node parameters affecting fragmentation

Several ConfigServer parameters affect fragmentation processing. The stack splits data into a number of blocks, and this number is limited by the CS_APS_MAX_BLOCKS_AMOUNT parameter mentioned above. The default value of 0 implies no restriction. Another parameter, CS_APS_BLOCK_SIZE, specifies the block size, that is, the size of the parts to which a message is split. To set the block size to the maximum possible value, assign the parameter a value of 0, which is the default value.

While transmitting data frames for the current data request, the stack sends a certain number of frames, then stops sending frames to wait for an acknowledgement, and then continues when receives it. The number of frames to be sent before waiting for an acknowledgement is controlled by the CS_APS_MAX_TRANSMISSION_WINDOW_SIZE parameter, which is set to 3 by default. It is strictly required that this parameter be the same for both the originator and the destination. Otherwise, it will not be possible to synchronize the communication because the communicating nodes will not recognize a correct moment for responding.

Section 7
Power management

In ZigBee networks, power consumption is often a major concern because in many applications, not all devices can be powered from a wall socket. The Atmel BitCloud stack provides a simple API that allows switching between active and sleep modes as well as turning off the radio chip to reduce power consumption.

The BitCloud stack supports power management mechanisms not only on end device nodes, but also on fully functional devices, that is, on routers and coordinators, thus providing full, flexible control over power consumption in the network. The API and logic for sleeping device management is the same for end devices and fully functional devices. However, employing sleeping routers and a sleeping coordinator in the network requires additional considerations for the synchronization of devices.

7.1 Sleeping device management

7.1.1 Active and sleep modes

Independently of its networking status (joined to a network or not), a device can be either in active mode or sleep mode.

After being powered up, a node always starts in active mode, with its MCU fully turned on and RF chip ready to perform Rx/Tx operations. An application can call any BitCloud commands and will receive responses through registered callbacks and notifications.

In sleep mode, the RF chip and the MCU are put in special low power states. In the BitCloud stack, only the functionality required for MCU and radio wake ups remains active. Thus, the application cannot perform any radio Tx/Rx operations, communicate with external periphery, etc.

7.1.1.1 Putting a device into sleep mode

In order to put a device into sleep mode, an application calls the ZDO_SleepReq() function with an argument of the ZDO_SleepReq_t type. After that, the confirmation callback provided with the argument will indicate the execution status, and if ZDO_SUCCESS_STATUS is returned, the node will enter sleep mode after executing the callback. The function can be called at any time, but if it is called before the device enters a network, the device type must be configured so that the stack can choose which parameter to use as a sleep period.

A failure status indicating that the node will not enter sleep mode may be returned for various reasons. For example, the stack will not proceed to sleep mode if the confirmation for any asynchronous request issued either by the application or by the underlying stack layers is being awaited. In this case, the application calls the ZDO_SleepReq() function one more time to put the device into sleep mode.

There are two ways to wake up a node (that is, to switch from sleep mode to active mode): scheduled and IRQ triggered.

7.1.1.2 Constraints on sleep mode for a fully functional device

For routers and the coordinator, there are many more stack states for which a device cannot enter sleep mode immediately, and thus a sleep request will report a failure status. On fully functional devices, this can occur when:

- There is a packet waiting for delivery to an end device child after the child wakes up and sends a polling request
- The device is participating in a network transaction, such as transferring a routed data frame to a remote node

In active mode, the stack on a fully functional device uses only one state of the radio chip. The application can put it into another state either by calling an Atmel BitCloud API function or by manipulating the radio chip directly (which is not recommended). If the radio chip on a device is not in the state used by the stack, then putting the device into sleep mode may fail. Additionally, after wake up, the state will not be restored and the radio chip will be put into the common state employed by the stack. And so it is not recommended to change the state of the radio chip shortly before entering sleep mode.

7.1.2 Wake up after the specified sleep period (the scheduled approach)

In the scheduled approach, a node wakes up automatically after the time interval in milliseconds specified by the non-zero value of a special ConfigServer parameter, as described in Table 7-1.

Table 7-1. Parameters specifying the sleep period.

Parameter	**Use**
CS_END_DEVICE_SLEEP_PERIOD	Used on end devices as the sleep duration. Used on routers and the coordinator as the sleep duration of end device children.
CS_FFD_SLEEP_PERIOD	Used on routers and the coordinator as the sleep duration. Not used on end devices.

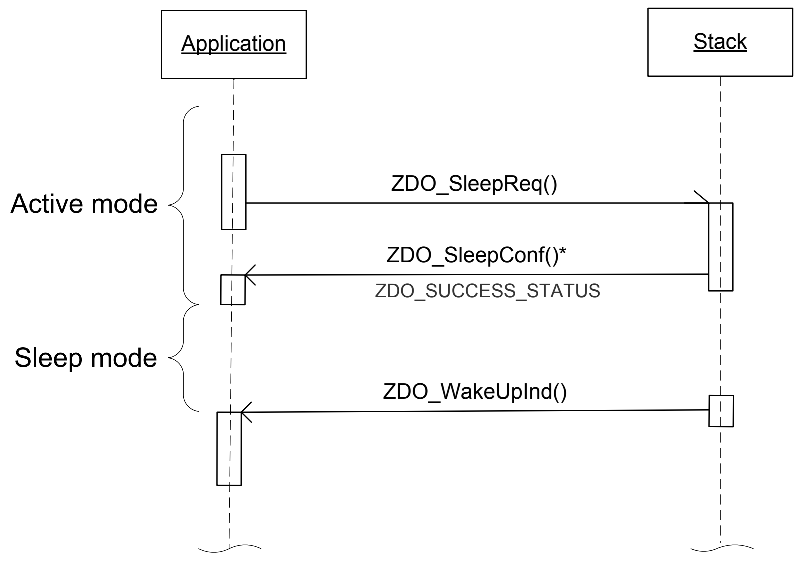
The value of a sleep period parameter is set in milliseconds, but not all values are distinguished from each other, and thus the resolution for a sleep period parameter does not equal 1ms. Resolutions for different platforms are given in Table 7-2. Note that in case of an Atmel XMEGA® platform, if an ultra low power oscillator is used, then the resolution is approximate and an accurate value is not reached.

Table 7-2. Sleep period resolutions for different hardware platforms.

Platform	**Sleep period resolution**
Atmel ZigBit, megaRF	31.25ms
Atmel XMEGA	15.625ms

The application is notified about the switch to active mode via the ZDO_WakeUpInd() function. Figure 7-1 shows the sequence diagram of stack calls for sleep and scheduled wake-up procedures.

Figure 7-1. Scheduled wake up.



* Function registered as a callback in the `ZDO_SleepReq()` argument.

While the device is sleeping, the MCU is found in the power save mode, consuming much less power than in the idle state. Since the MCU still has to wake up in time, a special clock should be involved to measure the time. Depending on the MCU, such timing can be done either using an internal oscillator (for example, on Atmel XMEGA) or an external one (for example, on Atmel ATmega128RFA1). Please refer to the development board documentation for the target MCU for more detail.

7.1.2.1 Synchronizing sleeping end devices and their parent devices

The user application is fully responsible for synchronizing sleeping end devices with sleeping routers and coordinator. Note that the polling mechanism is only implemented for end devices. Data sent to a router or the coordinator while it is sleeping is not cached on its parent automatically like it is for end devices.

Values for sleep periods on end devices and fully functional devices should be chosen carefully. A parent node must be awakened when its sleeping child wakes up, otherwise network connections may become broken.

Special attention should be given to configuring the CS_END_DEVICE_SLEEP_PERIOD parameter on routers and the coordinator, which use it to calculate time estimates for child device wake ups, as described in Section 6.4. If the end device sleep period is modified at run time on a sleeping end device, then its parent node should be notified about this by the application.

7.1.3 Wake up on a hardware interrupt (the IRQ triggered approach)

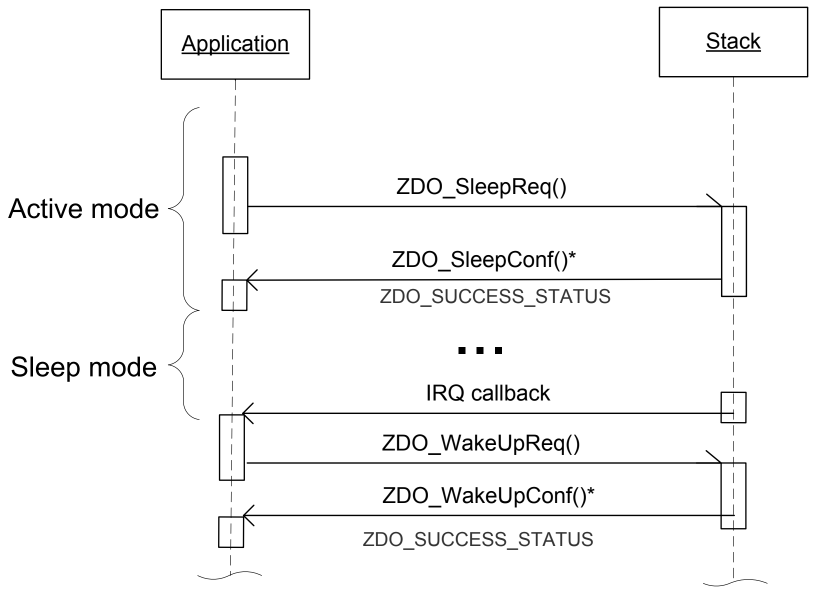
In the IRQ triggered approach, the MCU is switched to active mode upon a registered IRQ event. However, because a notification about such an event is issued by the HAL component directly, the network stack is not aware of it. In order to bring the whole stack back to active operation, the application must call the ZDO_WakeUpReq() function. After the callback registered for this request returns ZDO_SUCCESS_STATUS, the stack, the RF chip, and the MCU are fully awake.

An IRQ callback must not perform any long operations, such as issuing asynchronous requests. It is also not recommended to call API functions, since the stack is not fully awakened. A typical scenario for wake up on an IRQ event is calling the ZDO_WakeUpReq() function. It is the confirmation callback for this request that can perform any appropriate tasks, such as changing the application state, which can then be processed by the application task handler.

Note that not all IRQ numbers can be used to register a callback for a device's wake up. To find out which IRQ numbers can be used to register a callback for a device's wake up, refer to the MCU specifications.

Note also that if an IRQ callback is called from sleep mode and a sleep timer is scheduled according to a sleep period parameter, the timer will be stopped and will be restarted again upon the next ZDO_SleepReq() call.

Figure 7-2. IRQ triggered wake up.



* Function registered as a callback in the `ZDO_WakeUpReq()` argument.

Scheduled and IRQ triggered methods can be combined in a single application to manage power consumption. However, if a sleep period parameter (see Table 7-1) is set to zero, then only IRQ triggered wake up can switch the node from sleep to active mode. In this case, the MCU employs the power down mode, rather than to the power save mode, thus consuming even less power. Details are platform-specific, and can be observed in the MCU's documentation.

7.1.4 Suspending RF chip operation only

In several scenarios it might be required to keep the MCU operating (for example, for data processing purposes) while the RF chip is put into a special, low power consumption state called the TRX_OFF state. In this state, the RF chip cannot receive any frames from the outside, but is able to send a frame to the air immediately after receiving such a command from the MCU. This state is also called the clock state, because the crystal oscillator continues running and the RF chip can be used by the MCU as the main clock source.

This is supported in the Atmel BitCloud stack via its ability to turn off the polling mechanism on an end device, and then setting the RF chip to the desired state. The radio is set to the TRX_OFF state by having the application call the ZDO_StopSyncReq() function. The RF chip is restored to its normal state (and polling enabled) via the ZDO_StartSyncReq() function. When using this mechanism, it is important to take into account that a parent node expects data polls from its child node, and if not received for a certain period of time it will remove the end device from its neighbor list (for more detail on the polling mechanism, refer to Section 6.4). Note that the described feature is available for end devices only.

By putting the radio into the TRX_OFF state and disabling polling support, the node stops sending poll requests to the parent node automatically. However, the application is able to send a poll request to its parent manually through the NWK_SyncReq() function addressing the NWK layer directly.

7.2 Sleeping mesh

In BitCloud applications all devices in the network, including routers and the coordinator, may be put into the sleep mode (see Section 7.1.1.1). Making all devices in the network able to sleep allows decreasing power consumption on all devices; routers may also work on batteries. But applying such scenario is not always possible and requires additional considerations.

When a device programmed with BitCloud-based application goes into the sleep mode it does not become switched off, being in a special hardware state of low power consumption. Transition to the active state happens with a minimum delay and, more important, all RAM data on a device, which includes network parameters and routing, neighbor, and binding tables, are restored. This allows all devices in the network, when they wake up, to return to the normal operation in the network immediately.

For the best efficiency, all devices should go to the sleep mode and return to the active state simultaneously. However, very accurate synchronization (millisecond timescale) is not required.

Making all routers sleeping devices may not be relevant for application with large data throughput, since, as means for minimizing power consumption, all devices should be in the active state for as short period as possible.

So the main points in organizing a sleeping mesh network are the following ones:

- All nodes may be routers and, thus, be able to transfer data to any accessible device.
- All nodes operate on a short duty cycle. The fraction of time in the active mode is minimized, and the sleeping period is stretched.
- A node being in the sleep mode is not switched off. Network parameters and information about routes and neighbor devices is not lost.
- All nodes in the network wake up simultaneously, exchange application data and go back to sleep. Note that this approach is different from fine-grained synchronization on millisecond timescale, since it does not require any special hardware beyond what is already required for a ZigBee PRO node.

Section 8 Security

Security considerations are of high importance for wireless networks in many applications. In ZigBee networks, security serves two main purposes: disallowing network joins for devices that cannot provide appropriate credentials (authentication) and encrypting transmitted data (data privacy and integrity control). BitCloud SDKs help secure network operation by providing a choice between three modes: no security, standard security and standard security with link keys.

Enabling security introduces an additional network role called the trust center and a parameter called the network key. Each secured network must contain one device functioning as the trust center. The trust center is responsible for device authentication, the distribution and management of encryption keys, and other security parameters. The network key must be the same on all nodes in the network. It is used to authenticate devices joining the network and to encrypt transmitted data.

Higher security modes offer two levels of frame's encryption. In addition to network keys, they employ link keys specific for each pair of devices. Note that in a secured network unencrypted frames received by a device are ignored and do not reach the application.

Secured network operation is maintained by a set of ConfigServer parameters and functions at the APS level. Applying security sets certain constraints on application performance and memory usage, which can be managed by these parameters and functions. The details are provided by the following sections.

8.1 Security modes

To enable the ZigBee security mechanisms, four distinct Atmel BitCloud libraries are supplied in SDKs, for each of the supported security modes, which are shown in Table 8-1.

Table 8-1. Security modes for ZigBee networks.

Security mode	Description
No security	No authentication for network join required. Frames are not encrypted.
Standard security	Common network key is used to encrypt NWK payload of a frame.
Standard security with link keys (Stdlink security)	In addition to the network key, each communicating pair of nodes should know the link key in order to encrypt application payload of a frame with this key.
High security (not available in a standard BitCloud package, may be provided on demand)	Link keys can be created from master keys using the SKKE procedure. Additional security restrictions are also applied.

Security modes that provide greater security level include all features of less secured modes and add their own. Stdlink security offers two levels of encryption of a data frame: on the network level and on the APS level, although the application is able to send an unencrypted frame as well.

To enable the required security mode, the application must be compiled with the correct library. The choice of a particular library can be specified using the CONFIG_NAME parameter in the application makefile used by the GCC or IAR™ compiler if command line compilation is applied. For example, having

```
CONFIG_NAME = Makefile_All_StdlinkSec_Stk600_Atmega128rfa1_8Mhz_Gcc
```

selected in the makefile means that the library with standard link security support will be used when compiling the application for the specified platform. If, however, *_Sec_* is present in the selected CONFIG_NAME, then the library with standard security will be used.

If IAR Embedded Workbench® is used, then selection of a suitable workspace configuration implies compilation with a proper BitCloud library.

8.1.1 Security status and trust center address

After the appropriate library is chosen, the BitCloud stack should be configured properly to fit the application requirements for security.

The first parameter to be taken into consideration is CS_ZDO_SECURITY_STATUS. As shown in Table 8-2, it can take values 0 and 3 for standard security, and determines whether the network key is predefined on a device or not. Other values are not supported. The security status must be the same for all nodes in the network. A device is not able to join a network that employs a security status different from the one specified on the device.

Table 8-2. Authentication type with respect to the CCS_ZDO_SECURITY_STATUS parameter value.

CS_ZDO_SECURITY_STATUS	Security mode	Description
0	Standard security	The network key is predefined.
1	Standard security with link keys	Authentication is performed with a predefined trust center link key.
3	Standard security	*For standard security:* the network key is not predefined, but rather transported from the trust center.

The CS_ZDO_SECURITY_STATUS value 2 is reserved for high security mode (which is not supported in public BitCloud packages).

Another mandatory parameter used by the stack for various operations supporting security is CS_APS_TRUST_CENTER_ADDRESS, which is set to the extended address of the trust center. However, in some cases, for example, for testing purposes, devices can join the network and participate in data exchange without knowledge of the trust center extended address. For that, the CS_APS_TRUST_CENTER_ADDRESS is assigned the special universal extended address, APS_SM_UNIVERSAL_TRUST_CENTER_EXT_ADDRESS.

Once the parameters are configured, the application can operate with a chosen security mode. When the device joins the network, it has to authenticate itself. If authentication fails, the device is not allowed to enter the network. Otherwise, it can start communicating with other nodes in the network after successful authentication.

8.1.2 Switching security off

On a node with security support, the application can switch off the security (encryption/decryption and authentication procedures) at runtime by setting the CS_SECURITY_ON parameter to false. This should only be done before the device enters the network. After the device joins the network, CS_SECURITY_ON should not be changed. The parameter is applicable to both standard security and standard security with link keys.

8.2 Standard security

The main principle of standard security is to use a single key, called the network key, by all nodes in a network for authentication and the encryption of data frames.

Standard security may be used with one of two values specified by the CS_ZDO_SECURITY_STATUS parameter, 0 and 3. If the parameter equals 0, the network key must be predefined on each device in the network and stored in the CS_NETWORK_KEY parameter. If the parameter equals 3, the network key is not present on a device until it joins the network. CS_ZDO_SECURITY_STATUS is required to be the same on all network devices.

8.2.1 Device authentication during network join

Authentication is performed during the network start after a device establishes a connection to a parent node, but before the network start confirmation is raised on the device. The parent sends an update device request to the trust center. If the network employs a predefined network key, the trust center responds with an encrypted transport key command retransmitted to the joiner by its parent without decrypting it. Since the joiner knows the network key, it is able to decrypt the message and compare the received trust center extended address to the predefined value of the CS_APS_TRUST_CENTER_ADDRESS parameter. Thus, the device ensures that the response has been sent by a reliable trust center, and the so-called network authentication is a success.

However, if CS_APS_TRUST_CENTER_ADDRESS is set to 0xFFFFFFFFFFFFFFFF the device accepts any trust center and sets the CS_APS_TRUST_CENTER_ADDRESS parameter to the received trust center extended address. The same logic is applied in case of standard security with link keys.

If the network does not use a predefined network key, the response from the trust center contains the network key value. The parent of the joining device decrypts the message before sending it to the child, because the child will not be able to decrypt it. That is why network joining without a predefined network key must be handled very carefully. For example, a network join can be permanently forbidden for all nodes with the use of a permit join ZDP request. But when the joining of a new device is needed, it can be temporally permitted on a single node, which can reduce the signal power so that the unencrypted signal will reach only the joining device located in its immediate neighborhood.

8.2.2 NWK payload encryption with the network key

After a device enters a network, the network key is used to encrypt an NWK payload, that is, the payload of the message sent from the NWK layer to the underlying MAC while processing the data transmission. The NWK payload includes the APS payload filled by the application during data request issuing as well as APS headers.

Figure 8-1. ZigBee frame with standard security.

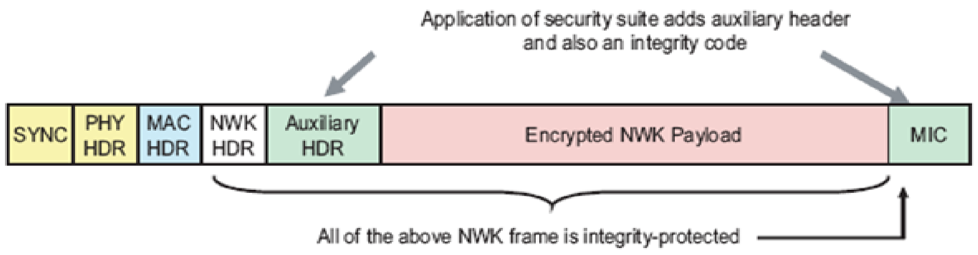

Application of security suite adds auxiliary header and also an integrity code
SYNC
PHY HDR
MAC HDR
NWK HDR
Auxiliary HDR
Encrypted NWK Payload
MIC
All of the above NWK frame is integrity-protected

Figure 8-1 illustrates the structure of a frame transmitted to the network. In addition to the encrypted NWK payload, an auxiliary header and an integrity code are inserted into the frame. The integrity code is a hash value for headers and the payload and protects them from being changed.

The stack automatically encrypts all the messages sent to the air, and so from the application's point of view, data transmission is performed as without security.

8.3 Standard security with link keys

In addition to encrypting the NWK payload with a network key, standard security with link keys provides encryption of the application payload with a link key, which is unique for each pair of communicating nodes.

Figure 8-2. ZigBee frame with extended standard security.

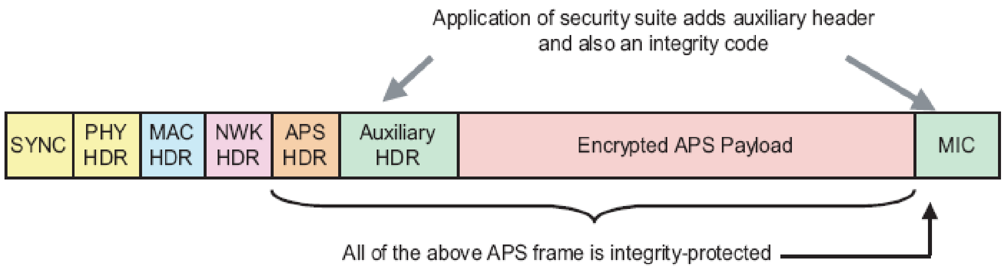

Application of security suite adds auxiliary header
and also an integrity code
SYNC
PHY HDR
MAC HDR
NWK HDR
APS HDR
Auxiliary HDR
Encrypted APS Payload
MIC
All of the above APS frame is integrity-protected

Figure 8-2 illustrates how a frame is formatted when the standard security with link keys mode is applied. First, the APS payload, which is set by the application in the data request, is encrypted with the use of the link key. An auxiliary header is inserted after the APS header. After that, the entire NWK payload is encrypted with the use of the network key, as shown on Figure 8-1. As with standard security, the integrity code is appended to the frame to ensure the safety of the NWK payload. If the NWK payload is changed, the destination device is able to detect it by calculating it own integrity code and comparing it to the one received.

When standard security with link keys is used, the CS_ZDO_SECURITY_STATUS parameter is set to 1. As described in Table 8-2, using a value of 1 for the parameter requires a link key to be predefined on devices for communication with the trust center. Pairs of link keys and corresponding extended addresses are stored on a device in a special internal structure referred to as the APS key-pair set.

Authentication of a device attempting to join the network is performed in two steps. First, upon receiving an update request from a parent of the device, the trust center should make a decision on whether to allow the device to enter the network or not. The following two sections describe this authentication in more detail.

8.3.1 Network join with a predefined trust center link key

The application shall insert an entry with the trust center link key and the trust center extended address into the APS key-pair set before the network start. To cope with this, the application shall use the APS_SetLinkKey() function, passing to it the trust center extended address and the link key value. How to process the task is highlighted by the following example:

```
//Set a link key to the constant defined by the application
uint8_t linkKey[SECURITY_KEY_SIZE] = LINK_KEY;
//Assume that TRUST_CENTER_EXT_ADDR is defined by the application
ExtAddr_t extAddr = CCPU_TO_LE64(TRUST_CENTER_EXT_ADDR);
APS_SetLinkKey(&extAddr, linkKey);
```

To be able to authenticate joining devices, the trust center has to know extended addresses and corresponding link keys for the devices, which should also be inserted into the key-pair table via

APS_SetLinkKey() in a similar way, as show above.

During the network join, the trust center receives an update device request from the parent of the joining device. The device is admitted if an entry with the device's extended address exists in the trust center APS key-pair set. If there is no such entry, the device authentication fails. In this case, the trust center application receives a notification via the ZDO_MgmtNwkUpdateNotf() function marked with status equal to ZDO_NO_KEY_PAIR_DESCRIPTOR_STATUS and the extended address of the joining device in the deviceExtAddr field. Thus, the application on the trust center can try to obtain a link key for the device from some external resource, and if this happens, the device can successfully join the network on its next attempt.

After the device is authenticated on the trust center, the trust center sends to the joining device a transport key command frame containing the network key and encrypted with the link key. If the device can be reached by the signal, the trust center sends the frame to it directly. Otherwise, it employs tunneling to transfer the frame through the device's parent. Note that unlike standard security, in standard security with link keys the network key is never transmitted over the air in an unencrypted frame. Once the joining device receives the message with the network key inside, it initiates a special command exchange with the parent using the obtained network key to establish a secure connection with it. After this procedure is completed, the device receives the network start confirmation via the callback function registered with the network start request.

8.3.2 Data exchange in secured networks

8.3.2.1 Requesting the link key for communication with a remote device

With standard security with link keys enabled, communication between any two nodes in the network requires a link key. If a node has to send data to a remote node and does not know the link key associated with this pair of nodes, it must request it from the trust center via the APS_RequestKeyReq() function. The function must be provided with extended addresses of the trust center and the remote node. Often, only the short address of the remote node is known. To obtain the corresponding extended address, the application can issue a ZDP request (see Section 6.5.1). To check whether a link key is already known, the APS_FindKeys() function can be applied. Consider the following example illustrating the usage of this API:

```
APS_RequestKeyReq_t apsRequestKeyReq; //a variable in a global scope
...
ExtAddr_t extAddrToBindWith;
...
//First, check whether a link key is already known
if (!APS_KEYS_FOUND(APS_FindKeys(&extAddrToBindWith)))
{
  ExtAddr_t tcAddr;
  //Read the trust center address to a prepared variable
  CS_ReadParameter(CS_APS_TRUST_CENTER_ADDRESS_ID, &tcAddr);

  apsRequestKeyReq.destAddress = tcAddr;
  apsRequestKeyReq.keyType = APS_APP_KEY_TYPE;
  apsRequestKeyReq.partnerAddress = extAddrToBindWith;
  apsRequestKeyReq.APS_RequestKeyConf = apsRequestKeyConf; //Confirmation callback

  APS_RequestKeyReq(&apsRequestKeyReq);
}
```

A confirmation callback reports a success if a link key has been received and stored in the APS key-pair set, accompanied by the extended address of the remote node. The trust center transfers the established link key to the remote node as well in order to make it possible for that node to decrypt messages which will come from the node that initiated the link key request. The confirmation callback may perform an actual data request deferred earlier because of the unknown link key:

```
static void apsRequestKeyConf(ZS_ApsRequestKeyConf_t *conf)
{
  if (APS_RKR_SUCCESS_STATUS == conf->status)
  {
    //Perform a data request or any other appropriate actions
  }
}
```

8.3.2.2 *Sending a data frame*

Data transmission is requested as usual via the APS_DataReq() function (refer to Section 6.2 for details). If a short address is used for the destination address, the stack searches for the corresponding extended address in the address map table. An entry containing short and extended addresses is inserted in this table automatically upon the completion of the device discovery ZDP request. The extended address extracted from the address map table is used to get a link key value from the APS key-pair set. Eventually, the obtained link key value is applied to encrypt the application payload.

Request parameters for APS_DataReq() are configured in the same way as without security, except for the txOptions field, which helps manage payload encryption. If txOptions.securityEnabledTransmission is set to 0, the APS payload is not encrypted with the use of the link key, and the frame is secured only by encryption on the network layer with the network key. Thus, to benefit from two-level encryption, the application must always ensure that securityEnabledTransmission in txOptions is set to 1.

Note that if a mapping between certain short and extended addresses has been added to the address map table, it is not guaranteed that it will always be there. If the table is full and a new entry needs to be inserted, the stack overwrites an existing entry. If the stack fails to obtain an extended or short address from the address map table while processing an APS data request, it initiates a callback with an error status. To avoid such situations, the extended address of the sender may be added to the auxiliary header of the data frame by setting includeExtendedNonce field of txOptions to 1 (reducing the available payload size). However, the recipient still has to know the link key for the extended address of the sender to encrypt the frame.

All tx options are explained in Table 6-2.

8.3.3 Network key authentication

After the trust center authenticates the device and exchanges necessary information with it, the device has to establish an authenticated connection with the parent. So network key authentication happens at the end of device's authentication procedure during network join, when the joining device receives the network key. The purpose of the parent authentication is to ensure that both the device and its parent keep the same network key.

The procedure consists of four frame transmissions between the joining device and its potential parent. If the procedure has success, the parent node finally accepts the joining device, thus finishing the authentication process and the network join for the device.

8.3.4 Global link keys

Global link keys are the link keys added to the APS key-pair set paired with addresses from the range

0xFFFFFFFFFFFFFFFA – 0xFFFFFFFFFFFFFFFF (global link key addresses). When a device receives a frame the stack checks if it knows the extended address corresponding to the short address of the sender. Then if the extended address is discovered the stack search an entry with this extended address in the APS key-pair set. If the stack does not find the sender's extended address or the APS key-pair set does not contain the entry with a link key for it, the stack tries global link keys one-by-one to decrypt the frame (if entries with global link key addresses exist in the APS key-pair set).

8.4 Multiple network keys

BitCloud stack allows the network to have several network keys on every device, although only one network key should be active and used to encrypt data frames in a given moment in the whole network. This feature is available for all security modes.

8.4.1 Setting network keys on a node

To enable multiple networks keys the user should set the CS_NWK_SECURITY_KEYS_AMOUNT parameter to a value greater than one, indicating the maximum number of network keys that may be stored on a node simultaneously.

The stack keeps an internal table for network keys (which size is specified by the mentioned CS parameter). A network key is identified by its sequence number, which is simply an integer number specified by the user. The first network key to be used as active is the network key that is specified in the CS_NETWORK_KEY parameter or received from the trust center in case the network key is not predefined. The application can insert additional network keys in the table for the future use by calling the NWK_SetKey() function. It is recommended to do this before network join. The network key that will be used first is stored in a special variable inside the stack called the default network key.

Note that the application can use the same network key value under different sequence numbers. Such keys will be regarded by the stack as different network keys.

8.4.2 Keys distribution

Further on the trust center can distribute new network keys as well as other security keys with the help of the APS_TransportKeyReq() function. Note that security keys are distributed differently depending on the security mode. For standard security and standard security with link keys the trust center can issue a single broadcast command to reach all devices at once. In this key the data frame with the command is encrypted only by the network key.

The application on the receiving node is not required to perform any specific actions, because the stack saves the received security information automatically.

8.4.3 Switching to a new network key

New network keys should be distributed beforehand. Note that while a new key is being distributed, the network continues to use the old network key. When all devices receive the new network key, the trust center should initiate a switch to the new network key, making it active on all devices, by calling the APS_SwitchKeyReq() function.

Note that in case of broadcast key distribution the trust center cannot verify that all devices have received the new key value, and so it might be useful to distribute new keys in advance, when the switch to the new network key is not going to happen.

Upon receiving a switch key command, the stack decrypts the command frame and makes the network key that has the sequence number received with the command active.

8.5 Permission table

The permission table, a special object for keeping addresses of devices permitted to the network, makes controlling of network joins even more flexible. By default the permission table feature is not used. To switch it on specially compiled BitCloud libraries should be used. The permission table works in all security modes.

If the permission table is enabled, only devices with addresses presented in the table will be allowed to join the network. For instance, an entry for a device with its link key or master key may exist in the APS key pair set, but if device's address is not included in the permission table, the device will not join the network.

The permission table feature empolys additional API functions with the *TC_* prefix. To add devices to the permission table use the TC_AddDeviceToPermissionTable(ExtAddr_t *extAddr) function, which takes a pointer to the device's extended address as an argument. A device can be removed from the permission table with a call to TC_RemoveDeviceFromPermissionTable(ExtAddr_t *extAddr).

The CS_MAX_TC_ALLOWED_DEVICES_AMOUNT parameter specifies the size of the permission table.

Section 9
Hardware control

In addition to the ZigBee networking functionality described in Section 5, the Atmel BitCloud API also provides extensive support for common hardware interfaces, such as USART, TWI, SPI, ADC, GPIO, IRQ, etc. The hardware abstraction layer (HAL) component of the BitCloud stack is responsible for all interactions between Atmel modules and external periphery.

This section gives a brief overview of the main hardware interfaces generally supported by the BitCloud stack. What hardware interfaces are supported by HAL depends on the MCU type as shown in Table 9-1. HAL also enables a number of drivers that provide interfaces used in Over-the-Air upgrade and implement additional serial interfaces (VCP and USB FIFO).

Table 9-1. Interfaces Supported in BitCloud on Different MCUs.

Interface	ATxmega256A3/D3	ATmega1281	ATmega128RFA1	SAM3S	SAM7X	UC3
UART	x	x	x	x	x	x
USART	x	x	x	x	x	x
DTR		x				
SPI	x	x(1)	x		x	
TWI	x	x	x		x	
ADC	x	x	x		x	
WDT	x	x	x		x	
IRQ	x	x	x	x	x	x
SLEEP	x	x	x			
GPIO	x	x	x	x	x	x
PWM		x	x			
USB	n/a	n/a	n/a	x	x	
VCP(2)				x	x	
USB FIFO(2)		x	x			
EEPROM	x	x	x	x(3)	n/a	n/a
1-wire		x(4)			x	

Notes: 1. UART operating in the SPI mode
2. Implemented as a driver
3. Emulated in Flash
4. Implemented in software

9.1 USART bus

BitCloud stack provides API for support of a universal synchronous/asynchronous receiver/transmitter (USART) interface.

In order to enable communication over a USART interface, an application must first configure the corresponding USART port using a static global variable of HAL_UsartDescriptor_t type. It requires the setting of all common USART parameters, such as synchronous/asynchronous mode, baud rate (see the hardware platform datasheet for the maximum supported value), flow control, parity mode, etc. Note that although the USART name is generally used in this section, the referenced API is the same for asynchronous and synchronous interfaces (UART /UART). For asynchronous interface operation, the mode in the USART descriptor must be set to USART_MODE_ASYNC.

Data reception and transmission over a USART can be separately configured for operation either in callback or in polling mode, as described in sections below. The detailed structure of HAL_UsartDescriptor_t is given in the BitCloud Stack API Reference [5].

USART settings should be applied using the HAL_OpenUsart()function with an argument pointing to a global variable of HAL_UsartDescriptor_t type with the desired port configuration. The returned value indicates whether the port is opened successfully and can be used for data exchange.

When there is no more need to keep a USART port active, the application should close it using the HAL_CloseUsart() function.

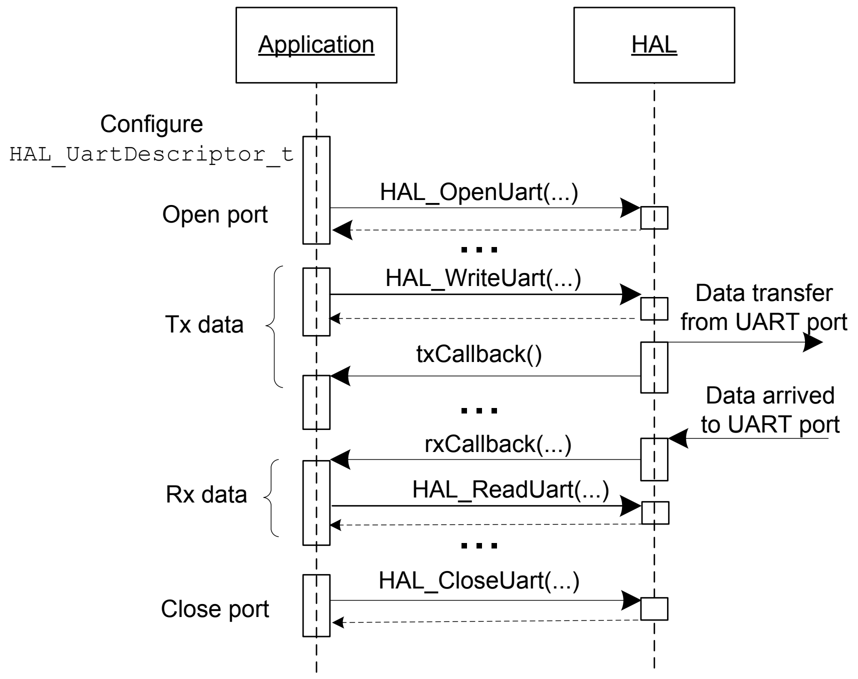
9.1.1 USART callback mode

The code snippet below shows how to configure a USART port so that both Tx and Rx operations are executed in the callback mode.

```
HAL_UsartDescriptor_t appUsartDescriptor;
static uint8_t usartRxBuffer[100]; // any size maybe present
…
appUsartDescriptor.rxBuffer        = usartRxBuffer; // enable Rx
appUsartDescriptor.rxBufferLength  = sizeof(usartRxBuffer);
appUsartDescriptor.txBuffer        = NULL; // use callback mode
appUsartDescriptor.txBufferLength  = 0;
appUsartDescriptor.rxCallback      = rxCallback;
appUsartDescriptor.txCallback      = txCallback;
…
HAL_OpenUsart(&appUsartDescriptor);
…
```

Figure 9-1 illustrates the corresponding sequence diagram for a USART deployed in callback mode.

Figure 9-1. USART data exchange in callback mode.



For data transmission, the HAL_WriteUsart() function shall be called with a pointer to the data buffer to be transmitted and the data length as arguments. If the returned value is greater than 0, the function registered as txCallback in the USART descriptor will be executed afterwards to notify the application that data transmission is finished.

The USART is able to receive data if the rxBuffer and rxBufferLength fields of the corresponding USART descriptor are not NULL and 0, respectively. For callback mode, the rxCallback field shall point to a function that will be executed every time data arrives in the USART's rxBuffer. This function has the number of received bytes as an argument. Knowing this number, the application shall move the received data from USART rxBuffer to the application buffer using the HAL_ReadUsart() function.

9.1.2 USART polling mode

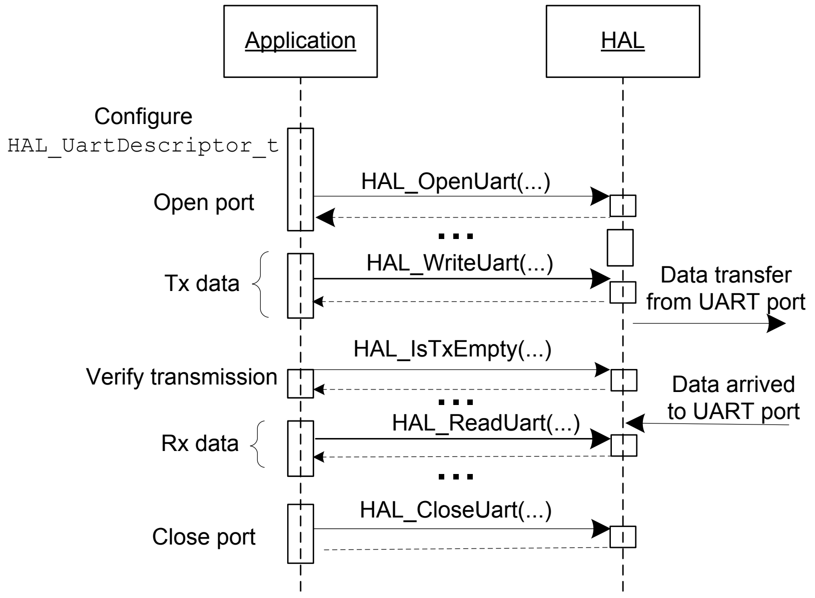
In the polling mode, USART Tx/Rx operations utilize corresponding cyclic buffers of the USART descriptor. Thus, the buffer pointer as well as the buffer length should be set to non-zero for the direction to be deployed in polling mode, while the corresponding callback function should be NULL.

Figure 9-2 illustrates the sequence diagram for Tx/Rx operations in polling mode. The main difference in Tx operation between callback and polling modes is that in the latter case, after calling HAL_WriteUsart(), all data submitted as an argument for transmission is cyclically moved to the txBuffer of the USART descriptor. Hence, the application can immediately reuse memory occupied by the data. The HAL_IsTxEmpty() function can be used to verify whether there is enough space in the txBuffer, as well as how many bytes were actually transmitted.

In contrast to using callback mode, in polling mode the application is not notified about the data reception event. However, as with callback mode, received data in polling mode is automatically stored in the cyclic rxBuffer and the application can retrieve it from there using the HAL_ReadUsart() function.

In the case of a txBuffer/rxBuffer overflow, the rest of the incoming data will be lost. To avoid data loss, the application should control the number of bytes reported as written by HAL_WriteUsart(), and if possible use hardware flow control.

Figure 9-2. USART data exchange in polling mode.



9.1.3 CTS / RTS / DTR management

In addition to data read/write operations, the Atmel BitCloud stack provides an API for managing the CTS / RTS / DTR lines of the USART port that supports hardware flow control (depending on the platform). For a detailed description of the corresponding functions, see [5].

9.2 External interrupts

BitCloud API provides functions for registering external interrupts on the MCU.

To register an interrupt call the HAL_RegisterIrq() function, specifying an interrupt number, an interrupt mode, and a callback function. The mode determines what signal level will generate an interrupt. Generating an interrupt causes execution of the callback function. What interrupt numbers are available depends on the MCU as shown in Table 9-2.

Table 9-2. Interrupt numbers available on different MCUs.

MCU	IRQ numbers
AVR32	IRQ_0, IRQ_1, IRQ_2, IRQ_3, IRQ_4, IRQ_5IRQ_6, IRQ_7
SAM7X	IRQ_0, IRQ_1
AVR	IRQ_5, IRQ_6, IRQ_7
XMEGA	IRQ_*PC*, where *P*- port name, *C* - interrupt number. For example: IRQ_D1
SAM3S	IRQ_PORT_*X*, where *X* stands for the port name

Once the interrupt is registered, it must be enabled to start using it. To enable an interrupt call the HAL_EnableIrq() function. From this moment the pin assigned to the interrupt is listened to by HAL, and if the interrupt occurs, a callback function will be called.

An interrupt can be disabled and unregistered at anytime by the corresponding HAL functions.

9.3 Two-wire serial interface bus

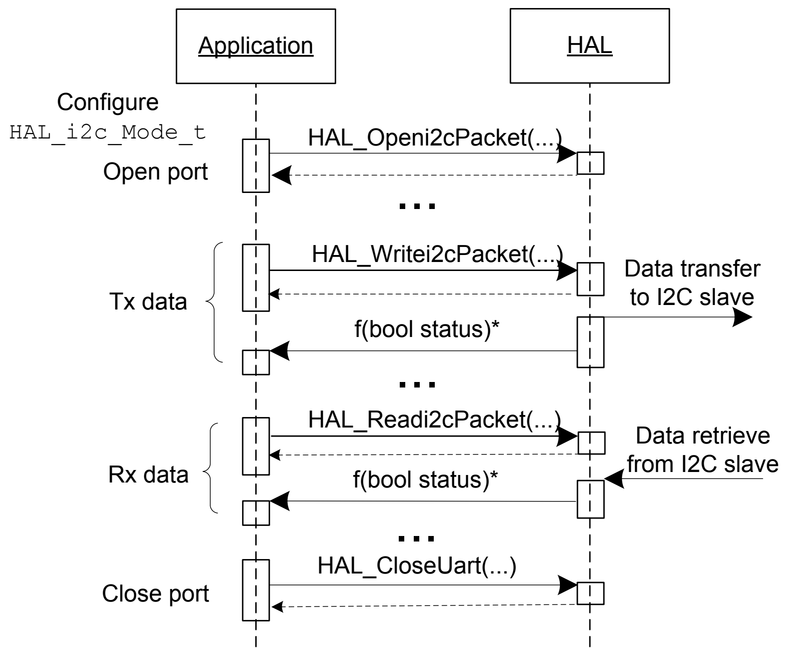
For technical details about TWI modules as I^{2}C slaves, please refer to:

http://www.atmel.com/dyn/resources/prod_documents/doc2565.pdf.

The Atmel BitCloud application can perform only master device functionality of the TWI protocol. Similar to any other hardware interface, the two-wire interface must first be configured and enabled for communication. After that, actual data read/write procedures can be performed with a remote TWI slave device.

Figure 9-3 provides a reference for TWI data exchange. To start using TWI bus, call the HAL_Openi2cPacket() function pointing it to an instance of the HAL_i2cMode_t type (the only parameter is a clock rate). After that use the HAL_Writei2cPacket() and HAL_Readi2cPacket() functions to send and receive data over the opened TWI port, respectively.

Figure 9-3. Data exchange over the TWI bus.



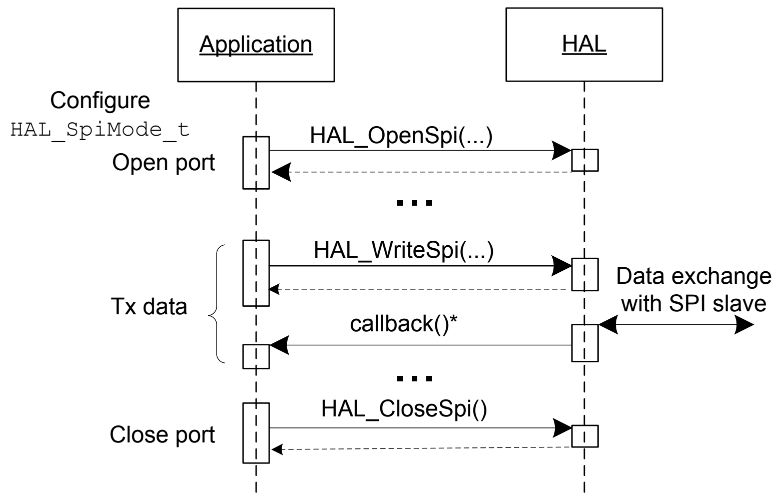
As shown in the figure above, TWI write/read operations are executed in asynchronous manner (see Section 3.1). That is, after calling HAL_WriteI2cPacket() or HAL_ReadI2cPacket(), the application should not perform any actions with the TWI bus nor with the memory allocated to the argument of the HAL_i2cParams_t type until the registered callback function is returned.

9.4 SPI bus

Depending on the MCU platform, the HAL component of the BitCloud stack implements the SPI protocol either on the USART bus (AVR platform) or on a pure SPI bus, when available. However, corresponding API calls are defined in the spi.h file and are independent of the underlying platform. The only difference appears in SPI configurations, namely in the fields of static variables of HAL_SpiDescriptor_t type.

The BitCloud application supports only SPI master mode. Figure 9-4 illustrates a sequence diagram of SPI-related API calls.

Figure 9-4. Data exchange over a SPI bus.



As with other interfaces, the SPI bus must be first configured and enabled by executing the HAL_OpenSpi() function with arguments pointing to a global variable of the HAL_SpiDescriptor_t type that contains configuration parameters and to a callback function.

Data transmission can then be performed using an asynchronous HAL_WriteSpi() request. For synchronous data exchange between master and slave devices, the HAL_ReadSpi() function should be used. If the callback function is not set to NULL, the application leaves the function where HAL_WriteSpi() or HAL_ReadSpi() is called, and the callback function informs the application that the SPI transaction has finished. If the callback argument in HAL_OpenSpi() was NULL, a SPI transaction will be performed during the actual call of the read/write function.

Finally, if no further data exchange is expected, the SPI bus should be closed in order to free occupied resources.

9.5 GPIO interface

The BitCloud API provides an extensive set of commands to manage the GPIO interface on both standard GPIO pins as well as on pins reserved for other interfaces that can also be used in GPIO mode (see the corresponding platform datasheet for information about such pins).

GPIO-related functions and macro names are defined in the gpio.h file of the HAL_HWD component. Function calls have the following form: GPIO_#pin_name#_#function_name#(). To execute the desired function for a particular pin, the corresponding macros for that pin must be used in the function body. Macro mapping to pin names is given in the gpio.h file. The following manipulations can be performed with GPIO-enabled pins:

- Configure a pin as either input or output. Examples for pins GPIO0 and USART1_TXD:

```
GPIO_0_make_in(); // configure GPIO0 pin for input
GPIO_USART1_TXD_make_out(); // configure pin for output
```

- Determine whether a pin is configured for input or output. Example for pin ADC_INPUT_1:

```
uint8_t pinState = GPIO_ADC_INPUT_1_state();
```

- Enable an internal pull-up resister. Example for GPIO0 pin:

```
GPIO_0_make_pullup();
```

- Set/toggle the logical level on an output pin. Example for pin TWI_CLK:

```
GPIO_I2C_CLK_set(); // set to logical level "1"
GPIO_I2C_CLK_clr(); // set to logical level "0"
GPIO_I2C_CLK_toggle(); // toggle logical level
```

- Read the current logical level on an input pin. Example for pin TWI_CLK:

```
uint8_t pinLevel = GPIO_I2C_CLK_read();
```

9.6 ADC

The Atmel BitCloud stack provides a platform-independent API for an analog-to-digital converter (ADC) interface.

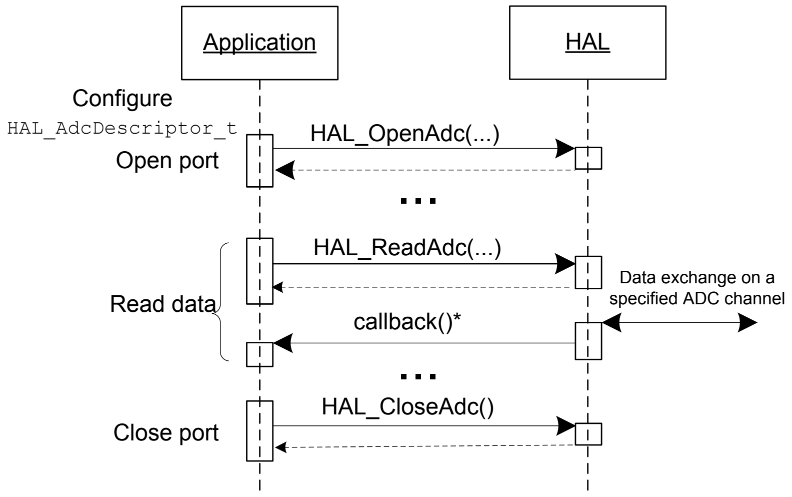
As with other interfaces, the ADC interface must be first configured and enabled using the HAL_OpenAdc() function with an argument pointing to a global variable of HAL_AdcDescriptor_t that contains configuration parameters. The HAL_OpenAdc() function returns status indicating whether the opening operation was executed successfully. A failed status may be returned for the following reasons:

- The ADC is already opened
- The function argument points to NULL
- The resolution is higher than RESOLUTION_10_BIT
- An incorrect value for the voltage reference is passed (param->voltageReference)
- RESOLUTION_10_BIT is used with a sample rate higher than 9600

After a successful ADC initialization, reading from the desired ADC channel can be done using the HAL_ReadAdc() function, with the HAL_AdcDescriptor_t variable (registered previously by HAL_OpenAdc()) provided as an argument. The result is returned via the callback function specified in the ADC descriptor. The HAL_CloseAdc() function is used to close the ADC interface.

Figure 9-5 illustrates an example sequence diagram of ADC reading.

Figure 9-5. Reading the ADC channel.



9.7 Other HAL functionality

The HAL component of the Atmel BitCloud stack also provides support for the following interfaces:

- 1-Wire®

- IRQ
- Application timer, system timer
- Watchdog, warm reset

The HAL_ReadUid() function reads a UID value from the UID chip if it is present on the development board. This function can also be implemented to read UID from any other source. For more detail see [6].

The BitCloud Stack API Reference [5] provides information about the API functions that are used to manage the interfaces listed above.

Although available in the HAL component, some functions are intended for internal execution inside the stack, and applications should avoid using them:

- HAL_ReadEeprom()/HAL_WriteEeprom()
- Sleep timer

9.8 Drivers

In addition to the described interfaces, the Hardware Abstraction layer includes a set of drivers that implement particular hardware interactions used in Over-the-Air upgrade and enable special serial interfaces. The drivers include:

- Image Storage driver (OTAU)
- Serial Flash driver (OTAU)
- VCP driver
- USB FIFO driver

To find out what platforms support VCP and USB FIFO driver see Table 9-1.

9.8.1 OTAU drivers

The Image Storage driver and the Serial Flash driver are used only with Over-the-Air upgrade by some internal components. These drivers are not intended to be used directly by the application, but the users might need to modify them to implement interaction with custom devices.

OTAU functionality is powered by the OTAU cluster, which uses the Image Storage driver on the OTAU server to communicate with the image storage. An OTAU server is a device, which, being a part of the network, distributes firmware images to other devices in the network by transferring data over the air. In reference implementation provided in the BitCloud package the image storage is a PC-based application, and the OTAU server exchanges data with it via UART. However, the image storage can be any suitable source of data.

The Serial Flash driver resides on each upgradable device in a ZigBee network. Its functions are to transfer a firmware image received over the air to the external flash memory and to swap firmware images upon completion of the upgrade.

For details on OTAU see [7]. Some helpful information on implementing a custom serial flash driver is found in [6].

9.8.2 Using VCP interface

The Virtual COM Port interface emulates direct UART output to PC's COM port. That is why the VCP interface API is the same as for UART/USART except that VCP functions share the VCP prefix, rather than HAL.

So to start sending data to a virtual COM port, call the VCP_OpenUsart() function providing it with an instance of the HAL_UsartDescriptor type to configure the port. The same way as UART/USART, the VCP can operate in the callback mode or in the polling mode.

To write and read data VCP_WriteUsart() and VCP_ReadUsart() functions are generally used, though there are some differences depending on whether the callback or the polling mode is applied. See Section 9.1 for more detail.

Section 10
Memory and resource allocation

10.1 RAM

RAM is a critical system resource used by the Atmel BitCloud stack to store run-time parameters like neighbor tables, routing tables, children tables, etc. As sometimes required by the stack, the values of certain RAM parameters are stored in EEPROM so that their values can be restored after hardware reset. The call stack also resides in RAM, and is shared by the BitCloud stack and the user application. To conserve RAM, the user must refrain from the use of recursive functions, functions taking many parameters, functions which declare large local variables and arrays, or functions that pass large parameters on the stack.

System rule 8: Whole structures should never be passed on the stack; that is, used as function arguments. Use structure pointers instead.

System rule 9: Global variables should be used in place of local variables, where possible.

User-defined callbacks already ensure that structures are passed by pointer. The user must verify that the same is true for local user-defined functions taking structure type arguments.

System rule 10: The user application must not use recursive functions. It is recommended that the maximum possible depth of nested function calls invoked from a user-defined callback be limited to 10, each function having no more than two pointer parameters each.

Overall, the RAM demands of the BitCloud stack must be reconciled with that of the user application. Fortunately, the amount of RAM used by the stack data is a user-configurable parameter. The configuration server, ConfigServer or CS, is distributed in source code (see Table 2-1) to allow the user to redefine the key runtime parameters of the stack. Among these parameters are some directly affecting the size of run-time tables stored in RAM. The table below lists such parameters and provides an easy formula to compute RAM consumption based on the value of a parameter (p):

Table 10-1. ConfigServer parameters and RAM consumption.

Parameter name	**Description**	**RAM consumption in bytes (no security/ standard security)**
CS_NEIB_TABLE_SIZE	Number of entries in the neighbor table	55 × p
CS_ROUTE_TABLE_SIZE	Number of entries in the routing table	7 × p
CS_DUPLICATE_REJECTION_TABLE_SIZE	Number of entries in the duplicate filtering table	5 × p
CS_ADDRESS_MAP_TABLE_SIZE	Number of entries in the address map table	11 × p
CS_ROUTE_DISCOVERY_TABLE_SIZE	Number of entries in the temporary table for route discovery	17× p
CS_NWK_BUFFERS_AMOUNT	Number of entries in the NWK data indications buffer	182 × p / 219 × p

Parameter name	Description	RAM consumption in bytes (no security/ standard security)
CS_APS_DATA_REQ_BUFFERS_AMOUNT	Number of entries in the APS data requests buffer	93 × p / 102 × p
CS_APS_ACK_FRAME_BUFFERS_AMOUNT	Number of entries in the APS acknowledgements buffer	35 × p / 53 × p
CS_NWK_BTT_SIZE	Number of entries in the broadcast transaction table	5 × p

The BitCloud stack's overall contribution to RAM consumption can be computed by summing up the last column of the table and substituting the value of each parameter for the respective p. With reasonable values of CS parameters, the user can expect about 5KB to 6KB of RAM to be consumed by the stack. The remaining RAM is accessible to the user application data and the call stack. In addition to the user-tunable CS parameters, there is a fixed RAM allocation dedicated to the BitCloud stack.

At compile time, RAM consumed by the data stored in RAM may be determined by executing the following command. This command is usually appended to the makefile of all provided sample applications:

avr-size -d app.elf

The result will be presented as follows:

```
text       data     bss      dec       hex      filename
118482     714      6068     125264    1e950    app.elf
```

The number of bytes consumed by data will be **data + bss.** Note that this value does not include the portion of RAM used by the call stacks, which varies at run time and depends on the depth of the stacks and the number of function parameters. The user may inspect the value of the call stack pointer (pointing to the top of the call stack) at run time by running the application in debug mode. An important property of the system is that all memory allocation is static, that is, the amount of RAM consumed by the BitCloud stack and the user application data is known at compile time.

Most C programmers are familiar with C library functions such as malloc(), calloc(), and realloc(), which are used to allocate a block of RAM at run time. The use of these functions is strictly prohibited, even though they may be accessible from the C library linked with the stack and applications.

System rule 11: Dynamic memory allocation is strictly prohibited. The user must refrain from using standard malloc(), calloc(), and realloc() C library function calls.

10.2 Flash storage

Another critical system resource is flash memory. The embedded microcontroller uses the flash memory to store program code. The footprint of the application in flash may be determined by running the following command:

avr-size -d app.elf

The result will be presented as follows:

```
text       data     bss      dec       hex      filename
```

```
118482        714          6068          125264        1e950         app.elf
```

The number of bytes consumed will be **text + data.** Unlike with RAM, the user has little control of how much flash space is consumed by the underlying BitCloud stack. Since the stack libraries are delivered as binary object files, at link time (part of the application building process) the linker ensures that mutual dependencies are satisfied; that is, that the API calls used by the application are present in the resulting image, and that the user callbacks invoked by the stack are present in the user application. The linking process does not significantly alter the amount of flash consumed by the libraries.

10.3 EEPROM and PDS

EEPROM constitutes nonvolatile storage available on most microcontrollers. Since EEPROM is another resource shared by both the BitCloud stack and the application to store their nonvolatile data, the use of EEPROM is arbitrated by a special API called the persistence data server (PDS) [5]. In general, EEPROM has linear addresses, and so in order to protect the EEPROM portion occupied by stack's parameters from being written over by the application code, the PDS uses a special offset to write and read all application data. Although the HAL component also provides read/write functions for EEPROM, it is strongly recommended to use the PDS component for such purposes, because the HAL_ReadEeprom() and HAL_WriteEeprom() functions access memory directly at a given address and hence do not eliminate the risk of overwriting internal stack-specific variables.

10.3.1 Saving stack data in EEPROM

BitCloud stack data including Configuration Server parameters and various internal tables (neighbor table, binding table, etc.) can be stored in EEPROM and restored after device restarts, using the PDS API. The PDS component defines a number of persistent data tables (PDTs) units to separate groups of stack data for more subtle control. The whole list of stack PDTs is given in Table 10-2. Identifiers for the stack PDTs are defined in PDS in the `/PersistentDataServer/include/pdsDataServer.h` file. The actual contents of PDTs is set in the CS component in the `/ConfigServer/src/csPersistentMem.c` file (to understand how PDT's layout is defined see Section 10.3.2).

By default no stack data is stored in EEPROM. The application may use one of the following strategies:

- Store stack data each time when one of the pre-defined events happens. The application should call the PDS_StoreByEvents() function once to specify what PDTs should be taken into account: either PDS_BITCLOUD_MEMORY (to store all stack PDTs) or a bitmask of some of the PDTs. The events and PDTs that should be stored upon happening of these events are defined in PDS in the `/PersistentDataServer/src/pdsEvents.c` file as the `pdsMemoryMap` variable.
- Store stack data periodically with a given interval. The application should call the PDS_StoreByTimer() function once to specify the interval and what PDTs should be stored.
- Store stack data at arbitrary moments, using the PDS_Store() function.

The argument of all PDS functions mentioned above as well as most of the other PDS functions should be set to a bitmask specifying to which PDTs the action will be applied. PDTs' identifiers defined by PDS may be chained via the | operator to form such a bitmask.

Table 10-2. Division of BitCloud stack data.

Group	PDT ID	Comment
General stack parameters	PDS_GENERAL_PARAMS	Include device information (extended address, short address, etc.), network information (PANID, channel mask, etc.) and other important parameters
Extended set of parameters	PDS_EXTENDED_PARAMS	More ConfigServer parameters
Neighbor table	PDS_NEIGHBOR_TABLE	
Group table	PDS_GROUP_TABLE	
Binding table	PDS_BINDING_TABLE	
Security tables of the NWK component	PDS_NWK_SECURITY_TABLES	Valid when security is used (either standard or standard link mode)
Security tables of the APS component	PDS_APS_SECURITY_TABLES	Includes APS key-pair set (pairs of extended addresses and link keys); is valid for standard link security (see Section 8.3)
Permission table	PDS_PERMISSION_TABLE	

Stack data should also be restored somewhere in the application via the PDS_Restore() function. For example, during initialization the application may check if stack data can be restored from EEPROM by calling the PDS_IsAbleToRestore() function and then restore these data via PDS_Restore() called with the same argument, if the former function returns true.

10.3.2 Defining a persistent data table

An application can save arbitrary application data in EEPROM, using the PDS component of the stack. The application can define its own persistent data tables (PDTs) to store application data. There are two PDT identifiers, PDS_APP_MEMORY_1 and PDS_APP_MEMORY_2, defined in PDS specifically for application use, though the application may define custom identifier. PDTs defined by the application should be filled them with links to some globally defined structures containing application data. The application saves and restores data by calling the PDS_Store() and PDS_Restore() functions, which save and restore, accordingly, the whole persistent data table.

Note: Some specific applications (for example, ZigBee Light Link profile) may define their own names for PDTs' identifiers PDS_APP_MEMORY_1 and PDS_APP_MEMORY_2. The stack uses its own PDT labeled with PDS_BITCLOUD_MEMORY.

To define a PDT an application should first define a utility type with fields having types of data structures that will be saved in PDT. This is done to describe the structure of saved data and to determine the sizes of stored instances. For example:

```
BEGIN_PACK
typedef struct PACK
{
  Scene_t                                   SCENE_TABLE[MAX_NUMBER_OF_SCENES];
  ZCL_SceneClusterServerAttributes_t        SCENE_ATTRIBUTES;
} AppPersistentDataOffsets_t;
END_PACK
```

A PDT, which structure is defined by the `AppPersistentDataOffsets_t` type, will be used to store scenes and server attributes of a Scene Cluster.

The next step for the application is to use the PROGMEM_DECLARE macro to provide actual variables to the PDT and to define a variable that will hold the PDT, in the following way:

```
extern Scene_t scenes[MAX_NUMBER_OF_SCENES];
extern ZCL_SceneClusterServerAttributes_t scenesClusterServerAttributes;
PROGMEM_DECLARE(const PDS_PersistentDataDescr_t appPersistentDataTable[]) =
{
  APP_PDDESCR(SCENE_TABLE, scenes),
  APP_PDDESCR(SCENE_ATTRIBUTES, &scenesClusterServerAttributes),
};
```

The PDT will be represented via the `appPersistentDataTable` variable.

The PDT is already defined, but the application should now assign it to one of two available memory blocks, using the PDB_REC macro. For example, the following code marks a PDT with the PDS_APP_MEMORY_1 label:

```
//Define auxiliary constants
#define APP_PERSISTENT_ITEMS_AMOUNT (ARRAY_SIZE(appPersistentDataTable))
#define APP_PERSISTENT_MEMORY_SIZE  (sizeof(AppPersistentDataOffsets_t))

//Define a descriptor for the PDT
PDB_REC(const PDS_PersistentDataTableDescr_t appPdbRec) =
{
  .pdt = appPersistentDataTable, //A variable defined by the PROGMEM_DECLARE macro
  .pdtSize = APP_PERSISTENT_ITEMS_AMOUNT,
  .dataSize = APP_PERSISTENT_MEMORY_SIZE,
  .memoryId = PDS_APP_MEMORY_1
};
```

The provided code samples completely define a PDT. The application can now access data through the `scenes` and `scenesClusterServerAttributes` variables. PDS_Store() and PDS_Restore() functions are used to save and restore the contents of these variables to EEPROM (see Section 10.3.3).

10.3.3 Saving and restoring data

PDS operates with blocks of memory called persistent data tables (PDTs). The application may use PDTs with PDS_APP_MEMORY_1 and PDS_APP_MEMORY_2 identifiers or define its own identifiers (see Section 10.3.2). The BitCloud stack splits its data for convenience, defining a number of PDTs (see Section 10.3.1).

To save data attributed to a PDT into EEPROM the application should call the PDS_Store() function, providing PDT's identifier as an argument. The function may be used to save data from several PDTs at once: the PDTs' identifiers provided in the argument should be chained in a bitmask using the | operator. For example, to force saving of an application PDT with PDS_APP_MEMORY_1 identifier and BitCloud PDTs containing stack parameters execute the following code:

```
PDS_Store(PDS_APP_MEMORY_1 | PDS_BITCLOUD_MEMORY);
```

Note that the PDS_Store() function launches an asynchronous storing process (lasting up to several seconds), which is executed in background, not blocking the application flow. For synchronous storing the PDS_BlockingStore() function should be used. The function returns, and the application execution continues, only when the storing procedure is completed. There is also an option to configure PDS to save data periodically, using the PDS_StoreByTimer() function, and upon happening of specific events, using the PDS_StoreByEvents() function.

To restore data from EEPROM to the data variables the application should call the PDS_Restore() function, providing a bitmask of PDTs' identifiers of PDTs to be restored. Before trying to restore data the application may check that restoring of the specified PDTs is possible, using the PDS_IsAbleToRestore() function. If any of the PDTs specified in the argument cannot be restored (because, for instance, it has not been filled with data and saved) the function will return false. For example, the application may use the following code to restore an application PDT with the PDS_APP_MEMORY_1 identifier:

```
if (PDS_IsAbleToRestore(PDS_APP_MEMORY_1))
  PDS_Restore(PDS_APP_MEMORY_1);
```

10.3.4 Resetting non-volatile storage and other functions

Non-volatile memory may be cleared by the application's calling the PDS_ResetStorage() function. The function should be provided with a bitmask of persistent data tables (PDTs) to be cleared, as an argument. A number of other PDS functions may be used to control which PDTs have been restored, etc. For the full specification of PDS functions see [5].

The two functions PDS_ReadUserData()/PDS_WriteUserData() operate with EEPROM data on a lower level. The functions read from and write to non-volatile memory at a given offset from the beginning of the remaining memory, not occupied by persistent data tables (PDTs). This means that these functions cannot overwrite stack data, as well as PDTs defined by the application, and are safe to be used. The 0 offset points to the beginning of the user data section. Negative offsets cannot be applied.

10.4 Other resources

Additional hardware resources include microcontroller peripherals, buses, timers, IRQ lines, I/O registers, etc. Since many of these interfaces have corresponding APIs in the hardware abstraction layer (HAL), the user is encouraged to use the high-level APIs instead of the low-level register interfaces to ensure that the resource use does not overlap with that of the stack. The hardware resources reserved for internal use by the stack are listed in [4].

System rule 12: Hardware resources reserved for use by the stack must not be accessed by the application code.