USER GUIDE

AVR2050: BitCloud Developer Guide

Atmel MCU Wireless

Description

BitCloud® SDK provides Atmel® implementation of ZigBee® PRO stack, ZigBee Cluster Library (ZCL) and a set of reference applications for ZigBee Home Automation, ZigBee LightLink and ZigBee OEM devices.

This document describes key concepts for BitCloud stack configuration, network management, data exchange and other critical topics and is intended for use by application developers.

Features

- BitCloud architecture overview
- Application development concepts
- Stack configuration
- Network management
- Data exchange description
- Power management
- Hardware interfaces

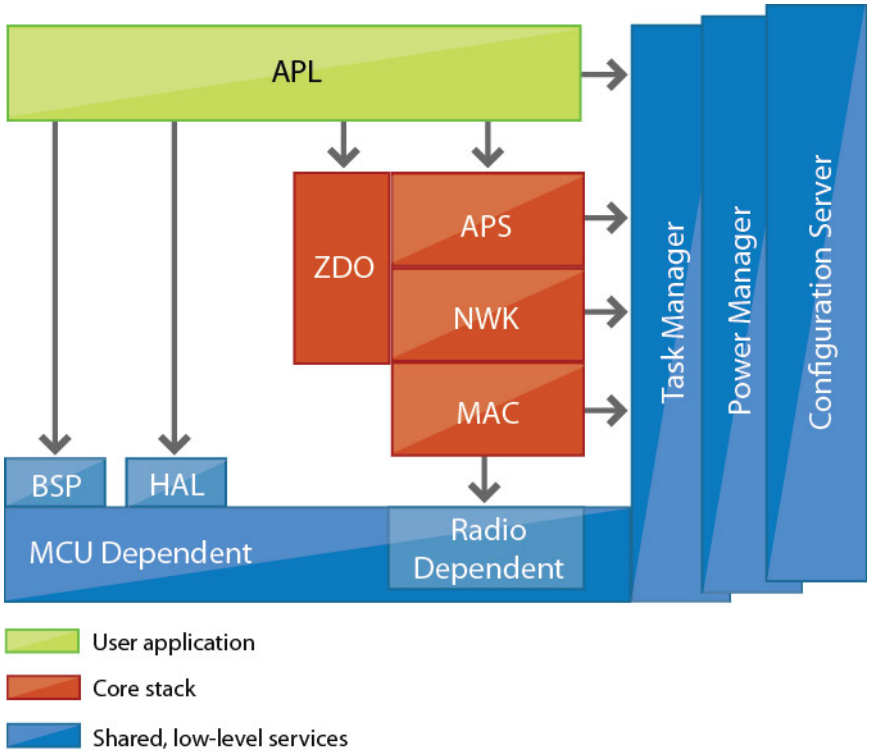
Table of Contents

1. BitCloud Architecture

1.1 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 [3].

Figure 1-1. BitCloud Software 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 2.2 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

1.2 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 prefix in `ZDO_GetLqiRssi()` function indicates that the function is implemented in ZDO layer of the stack.
2. Each API 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 requests (see Section 2.1.1) from the user application to the stack (for example, `ZCL_CommandReq()`)
    - Ind corresponds to the asynchronous indication of events propagated to the user application from the stack (for example, `ZDO_MgmtNwkUpdateNotf()`)
    - 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. BitCloud 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 should be designed to conform to the architecture pattern used in BitCloud components and to communicate with other components in the same manner as done 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 2.1, 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 2.1.1.

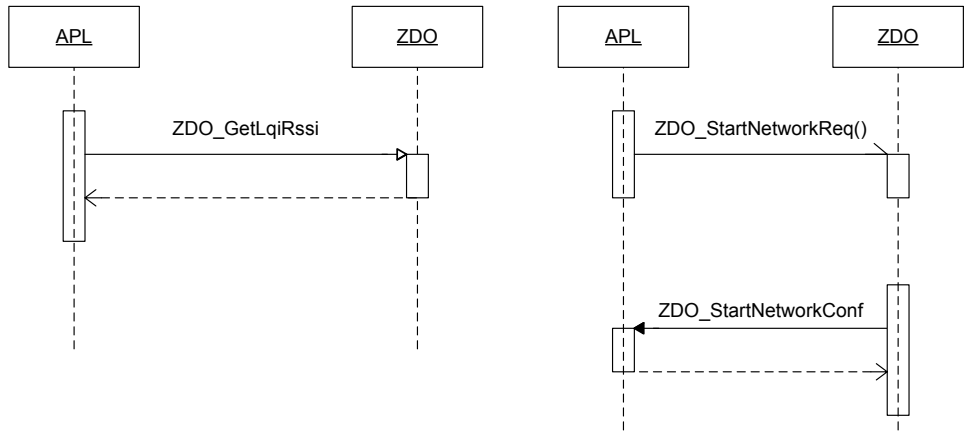
The life cycle of the entire BitCloud environment, including BitCloud components and the application, is maintained by a task scheduler. This is the core mechanism for controlling application flow. The task scheduler determines the sequence of task execution on different components. Actual task processing 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 2.2 details the task manager implementation.

In order to simplify application programming on the basis of described concepts, Section 2.3 illustrates the structure of a typical application developed on top of the BitCloud stack with code snippets. 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.

2.1 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 2-1. Synchronous vs. Asynchronous API Calls



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.

2.1.1 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 2.2.1.

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` file 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 but all others uses follow the same approach.

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.

Caution: Requests from higher layers may return status codes from lower layers. For example, a ZDO request may return status codes from NWK or MAC layers. The status code of the lower layer is always issued by the lower layer itself, although it may be redefined on the higher layer's header files. If the status code returned by a request is not found in header files of the layer that was used to send the request, search in header files of the lower layers (particularly, in the `nwkCommon.h` and `macCommon.h` files).

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 response 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 2.2.2 and 2.3 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 2.3.

System rule 1: BitCloud application should be organized as a set of callbacks that are executed on completion of API requests to the underlying layer.

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

2.2 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 uses (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 be executed ***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.

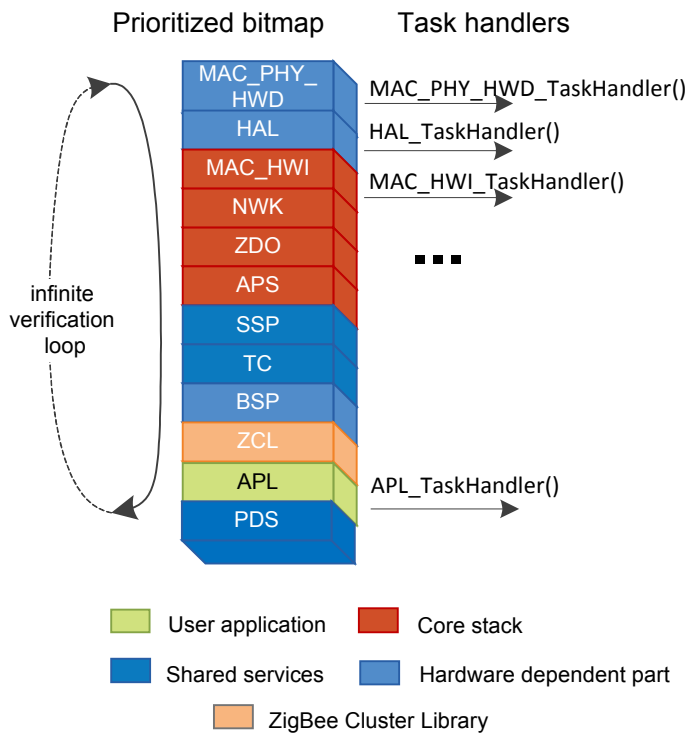
2.2.1 Task Manager Implementation

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 `<Component>_TaskHandler()` where `Component` represents the name of a layer (for example, APL, APS, NWK, HAL, etc.). Note that the source code for some component task handlers is hidden inside the Atmel BitCloud library. Changing any task handlers that are exposed in the source code, 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 respective 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 2-1 illustrates the task management process.

The operation of the task manager is maintained by an infinite loop residing in the `main()` function's body which is available in application code (for more details see Section 2.6.1). Its core structure is following: `main()` starts with a call 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 2-2. Control Flow Inside the Stack



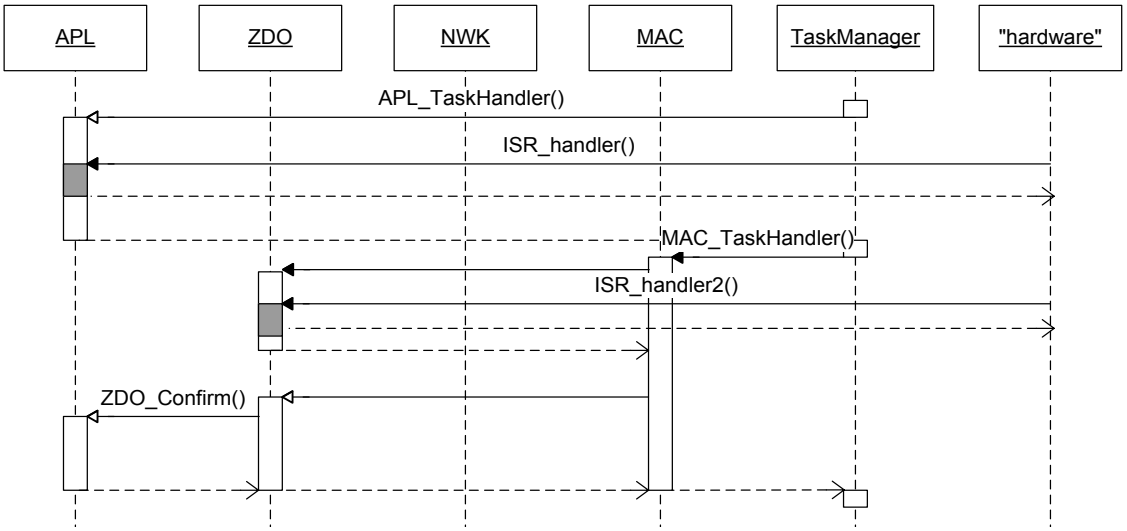
For additional information on task handling in ZigBee Light Link applications see Sections B.3.2 and B.4.

2.2.2 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 can be only in 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 2-3. 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 2-2 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 Atmel microcontrollers, flags are used, and so interrupts arriving during the masking will be saved.

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

2.3 Events Management

The stack uses the events mechanism to notify stack components and the application of various significant events happening with the device such as network start or loss and address conflict. Events are used in the stack's internal communications and are also available to the application.

The list of events used in the stack is defined in the `\Components\SystemEnvironment\include\sysEvents.h` file as `BcEvents_t` enumeration. An event is raised when the `SYS_PostEvent()` function is called with the event's ID and event's data (or `NULL` if no data is provided) specified in the argument. Raising an event causes the stack to call all handler functions subscribed to that event one-by-one. A handler is added to an event via the `SYS_SubscribeToEvent()` function.

The user can create its own events by adding new event IDs into the `BcEvents_t` enumeration. The user's event can then be raised as any other events via the `SYS_PostEvent()` function. The user can provide a pointer to any piece of data as the event's data, converted to the `SYS_EventData_t` type. A subscriber function will receive this pointer in its `data` argument.

The application can subscribe to any of the events defined in the `BcEvents_t` enumeration. To subscribe to an event the application needs a callback function to process the event, defined in the following way:

```
void eventCallback(SYS_EventId_t eventId, SYS_EventData_t data)
{
  ...
}
```

And an event receiver variable tied to the callback function:

```
//Defined globally
SYS_EventReceiver_t eventReceiver = { .func = eventCallback };
```

Note that the same event receiver (and the same callback function) may be used to subscribe to several events. To differentiate between events the callback function should use the `eventId` argument.

The application can then subscribe to an event, for example, the child joined event, calling the `SYS_SubscribeToEvent()` function somewhere in the application (the more common way is to process child's join notification in the `ZDO_MgmtNwkUpdateNotf()` function):

```
SYS_SubscribeToEvent(BC_EVENT_CHILD_JOINED, &eventReceiver);
```

The event will be raised when the stack discovers that a new child has joined. The callback function may process the event in the following way:

```
void eventCallback(SYS_EventId_t eventId, SYS_EventData_t data)
{
  //Cast event's data to the appropriate type
  ChildInfo_t *info = (ChildInfo_t *)data;

  //Access child node's info via data->shortAddr or data->extAddr
  ...
}
```

2.4 Memory Allocation

2.4.1 RAM Usage

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.

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 highly not recommended, even though they may be accessible from the C library linked with the stack and applications.

System rule 11: Dynamic memory allocation is not supported. The user must refrain from using standard `malloc()`, `calloc()`, and `realloc()` C library function calls.

Overall, the RAM demands of the BitCloud stack must be reconciled with that of the user application. The amount of RAM used by the stack data is to some extent a user-configurable value that is impacted by the value of ConfigServer parameters (see Chapter 3).

2.4.2 Flash Storage

Another critical system resource is flash memory.

The embedded microcontroller uses the flash memory mainly to store program code. The user has little control on the much flash space 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.

Additionally Persistent Data Server uses internal flash memory to store network and application parameters as described in Chapter 8.

2.4.2.1 Storing Variables in Flash Instead of RAM

To reduce RAM consumption, an application can place global or static variables holding constant data into Flash memory instead of RAM. When the application needs such variable it can easily read its value from Flash to a temporary RAM variable. For example, this may be applied to char strings or tables of function pointers.

2.4.2.2 Defining Variables Stored in Flash

To place a variable in Flash memory instead of RAM use the `PROGMEM_DECLARE()` macro function when defining the variable. Put the variable's definition in brackets. Depending on the platform, the macro adds necessary compiler directives or simply `const` to the definition. The following example shows how to store a char string in Flash:

```
PROGMEM_DECLARE(char NAME_WPANID[]) = "+WPANID";
```

It is possible to apply the macro to the whole definition of a structure, using `typedef`, like in the following example from the stack:

```
typedef PROGMEM_DECLARE(struct
{
  ClusterId_t          id;
  ZCL_ClusterOptions_t options;
  uint8_t              attributesAmount;
  ... //other fields
})ZCL_ClusterPartFlash_t;
```

A type defined this way can be used as a usual one to define a variable. A pointer of this type can be used as a field in common RAM structures.

```
//The structure to be stored in RAM
typedef struct
{
  ZCL_ClusterPartFlash_t     *partFlashAddr;  //A pointer to Flash memory
  ZCL_ClusterPartRam_t       *partRamAddr;
  bool                        needSynchronize;
} zclClusterImage_t;
```

To put a function pointer to Flash, first, define a type for the function pointer. Then put this type's name in the `PROGMEM_DECLARE()` macro and apply typedef to it to define the type for storing in Flash. Finally, define and initialize a variable. The following example illustrates the described approach:

```
//Define the function pointer type
typedef bool (* CommandIndHandler_t)(const CommandInd_t *const commandInd);
//Define the type for storing in Flash
typedef PROGMEM_DECLARE(CommandIndHandler_t) FlashCommandIndHandler_t;

//Define the function which pointer will be put to Flash
bool sampleInd(const CommandInd_t *const commandInd)
{
  ...
}

//Define a table of function pointers
static const FlashCommandIndHandler_t commandIndHandlers[] =
{
  [0] = sampleInd,
  [1] = anotherInd,
  ...
}
```

2.4.2.3 Reading Variables from Flash

A variable stored in Flash cannot be used as a usual variable stored in RAM. To access the value of such variable, the application should read it via the `memcpy_P()` function (in most cases it is a define for the standard `memcpy()` function). In the argument, provide address in RAM where to put the variable's value, address in Flash from which to read the value, and the size. See an example below.

```
//Define a char array placed in Flash
PROGMEM_DECLARE(char messageInFlash[]) = "Example string";
...
//The following code is inside some function
//Define a buffer in RAM
char messageBuffer[15];
//Copy a value from Flash to the buffer in RAM
memcpy_P(messageBuffer, messageInFlash, 15);
```

2.5 Other MCU Resource Usage

MCU 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 usage does not overlap with that of the stack. The hardware resources reserved for internal use by the stack are listed per supported platforms in [2].

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

2.6 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.

2.6.1 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 additional code into the body of the function, but the main function should always follow the structure provided: first, the `SYS_SysInit()` function is invoked to initialize the stack, 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 2-1.

2.6.2 Application Flow Control

There are three ways how application code can obtain execution control:

1. Through the task handler, following a `SYS_PostTask()` invocation as described in Section 2.6.2.1.
2. Through confirm callbacks invoked by underlying stack layers on request completion as indicated in Section 2.6.2.3.
3. Through asynchronous event notifications invoked by the stack as described in Section 2.6.2.4.

2.6.2.1 Application State Machine

This sub-Section describes a programming template shared by all reference applications provided with the BitCloud SDK. However developers are free to modify it to adjust for the needs of a customized application.

Following the task management implementation in BitCloud (see Section 2.2.1) application maintains global state data. The state is represented by variables for device role-dependent sub-states, various network parameters, sensor state, etc.

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 data which is shared between the callbacks and the application task handler.

For example, consider that the global state is represented by a single variable, `appState`, of type `AppState_t`, which is defined as custom enumerated type in the application code, and has a set of predefined values for different application states such as `APP_IN_NETWORK_STATE, APP_INITING_STATE`, etc. The definition of the state variable might be given in the global scope as follows:

```
AppState_t appState = APP_INITING_STATE;
```

Processing a state change requires setting of the `appState` variable to a target state and posting a task to the task manager via `SYS_PostTask(APL_TASK_ID)` function. This results eventually in calling the application task handler function (`APL_TaskHandler()`)which utilizes value of `appState` variable to switch to the code that has to be executed for the particular state as described in Section 2.6.2.2.

Following the state machine architecture described above helps conform to System rules that require all callbacks and task handlers to be executed in certain time duration (as given in Section 2.2). A good practice to ensure that such system rules are followed is to split long callback or tasks executions by changing the application state, posting application task and continue processing from the application task handler. This will guarantee that task manager will be able to handle pending stack tasks as well within require time limits. An example of such use is given in Section 2.6.2.3.

Approach described above can be a basis for extending the state machine. For example by creating additional sub-state machines that can be implemented within each global application state.

For additional information on task handling in ZigBee Light Link applications see Sections B.4 and B.3.2.

2.6.2.2 Application Task Handler

Every application implements a single application task handler function - `APL_TaskHandler()` to execute pending application tasks.

Upon device power-up/reset `APL_TaskHandler()` is called automatically after stack is fully initialized and is ready for use. After that it is application's responsibility to maintain application state machine and post application tasks using `SYS_PostTask(APL_TASK_ID)` function, to ensure that application tasks get processed in the `APL_TaskHandler()` as described in Section 2.6.2.1.

The code for APL_TaskHandler() in the application 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;
  }
}
```

For additional information on task handling in ZigBee Light Link applications see Sections B.4 and B.3.2.

2.6.2.3 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 `SYS_PostTask()` scheduler function. As described in Section 2.6.2.1 it is used in the application to defer processing of a network join to the application task handler, which will process it. 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.

To ensure execution of a user-defined callback at a specific time HAL timer can be used (see Section 9.7).

2.6.2.4 Stack events and 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)
```

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);
  }
}
```

Additionally, as described in Section 2.3, application can subscribe to other stack events that it is interested in.

3. Stack Configuration Parameters

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) and commonly start with `CS_` prefix.

This section gives a brief introduction to the ConfigServer interface itself, while a complete list and description of CS parameters can be found in the BitCloud Stack API Reference [3] that provides explanation of the parameter, its valid range, C-type and when a parameter value can be set.

3.1 Setting ConfigServer Parameters

CS parameters can be differentiated on when they can be set as follows:

- Only at compilation time (such as the parameters that impact memory allocation for different buffers and table)
- Only at time when the device is out of network (parameters that determine certain network related aspects of node behavior)
- At any time

Definitions of all CS parameters and their default values are contained in the `csDefaults.h` header file inside the stack.

Caution: Changing `csDefaults.h` file directly is not recommended. Instead, a developer is able to overwrite the default parameter values in the application header file, `configuration.h`. 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
```

This is the simplest method for assigning new compile-time values to CS parameters.

Management of CS parameters at run time is performed with the CS read/write functions, described in Section 3.1.1.

3.1.1 CS Parameter 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);
```

3.2 Parameter Persistence

Many ConfigServer parameters are defined as persistent items in `csPersistentMem.c` file and application can use those items or add own and keep them persistent over power cycles and resets. For more details see Chapter 8.

4. Network Management

As described in Section 1.1, the Atmel BitCloud architecture follows the structure of the ZigBee PRO specification [6], which allows applications to interact directly only with higher layers 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.

Appendix B provides additional information on the network management procedures specific for ZLL applications (for example touchlink).

4.1 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 that is 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 4.2.

Note that all ConfigServer parameters mentioned in this Section normally 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 BitCloud API Reference [3].

4.1.1 Device Type

ZigBee PRO specification [6] differentiates among three device types that can be present in a network: coordinator, router, and end device.

- The main responsibility of a **ZigBee coordinator (ZC)** 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 **ZigBee router (ZR)** 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
- A **ZigBee end device (ZED)** 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

4.1.1.1 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 5.9. If this parameter is set to true on an end device, parent polling will not be applied.

4.1.1.2 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);
```

4.1.2 Device Addressing

4.1.2.1 IEEE/extended Address

In accordance with the ZigBee standard [6], each device shall be uniquely identified by an extended non-zero (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 3.1. Before the network start procedure, the application shall ensure that the extended address is assigned uniquely on every device.

Note that if at compile time CS_UID is set to 0, Atmel reference applications attempt to read the extended address value from the user signature row of the device or external EEPROM (depending on the board) using the BSP_ReadUid() function. If the board doesn't support such mechanism then CS_UID value won't change and it is responsibility of the user either to update BSP_ReadUid() function or implement another way of ensuring that unique CS_UID is assigned before device initiates network join procedure.

4.1.2.2 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.

4.1.2.3 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 by the application to the CS_NWK_ADDR parameter. Note that static address assignment is Atmel-specific implementation, for ZigBee-compliant products only stochastic addressing mechanism shall be used.

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 4.2.1.3 or by reading `CS_NWK_ADDR` parameter.

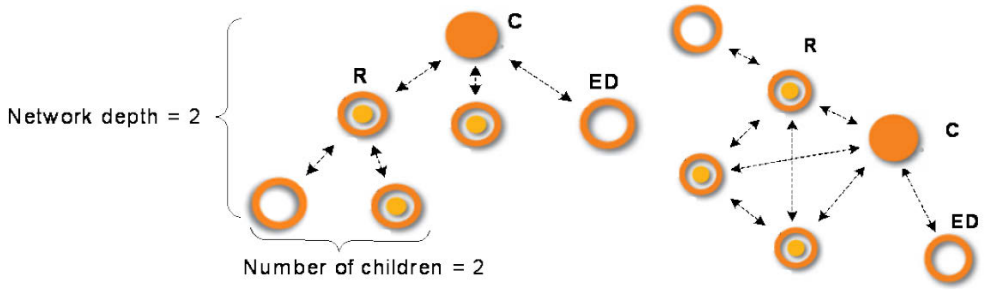
4.1.2.4 Address Conflict Resolution

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).

4.1.3 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 4-1. 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 4-1. Figure 4-1b shows the direct transmission links available in the network organized according Figure 4-1a, assuming all nodes are located within signal range of each other.

It is essential that in addition to the node parameters described in sections 4.1.1 and 4.1.2 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.

4.1.3.1 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
```

4.1.3.2 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.

4.1.3.3 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.

4.1.3.4 Differences 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 4.2.2.

4.1.4 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 Chapter 6).

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 Chapter 3.

4.1.4.1 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 ($b_{31}, \ldots, b_{27}$) of channel mask should be set to 0. The remaining 27 bits ($b_{26}, b_{25}, \ldots b_0$) indicate availability status for each of the 27 valid channels (1=supported, 0=unsupported).

Table 4-1 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 4-1. 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 = 2 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

4.1.4.2 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.

4.2 Basic Operations and Notifications

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.

In addition to general commands used to enter or leave the network (see Section 4.2.1), 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 this section, while detailed information with usage examples can be found in the BitCloud API Reference [3].

An important piece of functionality is provided by the ZDO component as ZigBee Device Profile requests (ZDP requests). See Section 4.3 for details.

4.2.1 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 from the application, 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.

4.2.1.1 Join Control Configuration

The stack provides intense control over network start to the application through the `CS_JOIN_CONTROL` parameter. The application can select the automatic mode, in which the stack chooses joining mode according to the configured network parameters, or choose the desired mode and specific options.

The parameter is a structure of NWK_JoinControl_t type. The type of the network start is set in the method field taking values from the NWK_JoinMethod_t enumeration. The default value for the method field is NWK_JOIN_BY_DEFAULT, which causes the stack automatically select values for other fields. That is, the stack selects the network start method to apply among other NWK_JoinMethod_t values and sets other fields.

All possible values for the method field are listed below:

- NWK_JOIN_BY_DEFAULT: automatic selection of the network starts method and other options. For example, if the coordinator type is selected as the ZigBee device type the stack will use the NWK_JOIN_VIA_FORMING method (network formation).
- NWK_JOIN_VIA_COMMISSIONING: the device just starts the network without scanning channels and without sending any frames to the air with known network parameters and node's parameters. This mode is supposed to be used when the node has already been in the network and has saved its parameters in the non-volatile memory (see Section 8.4). After hardware reset the node can restore them and start the network immediately, using this mode.
- NWK_JOIN_VIA_ASSOCIATION: the device tries to join a network through MAC association (as joining unknown network for the first time).
- NWK_JOIN_VIA_REJOIN: the device uses the NWK rejoining procedure. This means joining a network with known network parameters such as network PANID.
- NWK_JOIN_VIA_ORPHAN_SCAN: this method may be used to restore connection to the network. The device sends a special request asking which device could be its parent node. Only nodes that discover the joining device in their neighbor table will reply. This functionality is optional to the stack and is available on request.
- NWK_JOIN_VIA_FORMING: the device forms a new network and becomes the network's coordinator. To use this method correctly, the device type must be set to the coordinator.

Other fields set in the NWK_JoinControl_t structure are:

- secured: if the field set to true, frames sent during the network start procedure will be encrypted with a network key by the stack and incoming frames will be decrypted in the same manner. If the network key is unknown, but the field is set to true, frames will be sent unencrypted.
- discoverNetworks: if the field is set to true the stack will use active scanning to discover networks (information about discovered devices is added to the neighbor table); otherwise the stack uses the current neighbor table as is. In the latter case (the false value), one can fill the neighbor table before network start – the stack will try to join only to the devices from the table. In case of default joining method the field is set to true.
- annce: if the field is set to true, the device sends a device announcement frame once it joins a network, to notify other devices; otherwise device announcement is not sent. In case of default joining method, the field is set to true.

The application can set the CS_JOIN_CONTROL parameter at runtime using the CS_WriteParameter() function.

4.2.1.2 Parameters Required for Network Start

Table 4-2 lists ConfigServer parameters that shall be set in order to perform network start. The second column contains parameters' IDs as they should be provided to CS_WriteParameter() / CS_ReadParameter() functions to write or read their values. A number of parameters shall be set on the device no matter which type of network start is used. These parameters are also given in the table.

Association and rejoin procedures differ only in that, in case of rejoin, the extended PANID of the network shall be provided. The extended PANID is also needed for network formation, which can be initiated only by the coordinator: CS_DEVICE_TYPE shall be set to DEVICE_TYPE_COORDINATOR.

To perform commissioning, a device should know some parameters of the target network, which in other cases are received from the network: the working channel of the network, its short PANID, parent address (for end devices) and node's own short address. This method of joining is assumed to be used when a node has already been in the network it is going to join, and network parameters have been saved in the non-volatile memory. After hardware reset these parameters may be restored from the non-volatile memory, using the `PDS_Restore()` function (see Section 8.3). Then the node may use the `NWK_JOIN_VIA_COMMISSIONING` mode to restore its network state. Note that the neighbor table and the binding table are also saved to the non-volatile storage and can be restored with other parameters on reset. If this happen the node does not need to repeat binding procedures (see Section 5.8)

Table 4-2. Network Start Types and Parameters that shall be set to use them

Network start types	Parameters

All types	`CS_CHANNEL_MASK_ID`
	`CS_DEVICE_TYPE_ID`
	`CS_UID_ID`
	`CS_RX_ON_WHEN_IDLE_ID`
	`CS_NWK_ADDR_ID` if static addressing is used
	Security parameters:
	`CS_APS_TRUST_CENTER_ADDRESS_ID`
	`CS_ZDO_SECURITY_STATUS_ID`
	Security keys (depends on security type; see Chapter 6)

`NWK_JOIN_VIA_ASSOCIATION`	No specific parameters are set
`NWK_JOIN_VIA_REJOIN`	`CS_EXT_PANID_ID`

`NWK_JOIN_VIA_COMMISSIONING`	`CS_NWK_PARENT_ADDR_ID` (for end devices)
	`CS_NWK_LOGICAL_CHANNEL_ID`
	`CS_NWK_PANID_ID`
	`CS_NWK_ADDR_ID`
	
	Necessary *security information*: network keys and link keys (if required by the applied security type)
	
	Neighbor table and binding table may also be set

`NWK_JOIN_VIA_FORMING`	`CS_EXT_PANID_ID` `CS_DEVICE_TYPE_ID` is set to `DEVICE_TYPE_COORDINATOR`
`NWK_JOIN_VIA_ORPHAN_SCAN`	`CS_EXT_PANID_ID` A device that is supposed to accept the joining shall have an entry for this node (containing the node's extended address) in its neighbor table.

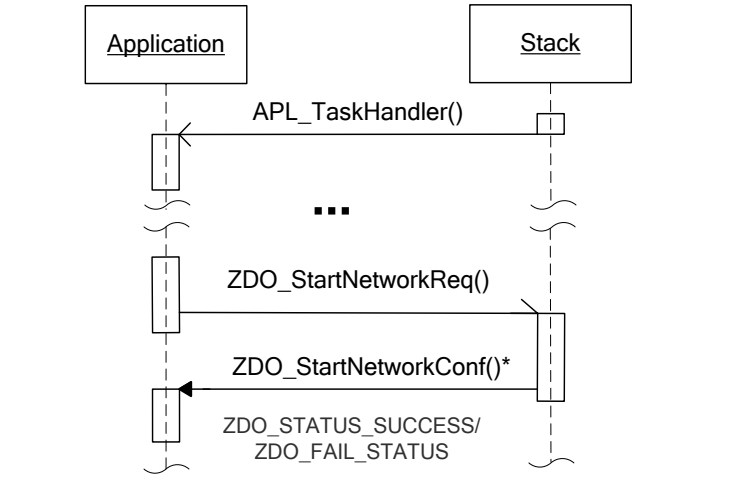
4.2.1.3 Network Start Request and Confirm

Figure 4-2 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, and appropriate join control configuration is selected (see Section 4.2.1.1), 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 [3].

If the network start failed the application can try to change the `CS_JOIN_ CONTROL` parameter via the `CS_WriteParameter()` function, setting a different joining method; see Section 4.2.1.1 for detail.

Figure 4-2. 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.

4.2.1.4 Network Formation Algorithm

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 4.1.4.2).

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 Wi-Fi 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 4.1.4.2).

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

4.2.1.5 Network join Algorithm

In default configuration (when in the CS_JOIN_CONTROL parameter, NWK_JOIN_BY_DEFAULT is chosen as a joining method) a node starts by sending a beacon request to collect information about neighbor devices. In other configurations, the application can specify whether a beacon request should be sent or not, which method, rejoin or association or another one to use, etc. (see Section 4.2.1.1).

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 *association*). 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.

4.2.1.6 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.

4.2.2 Parent loss and Network Leave

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

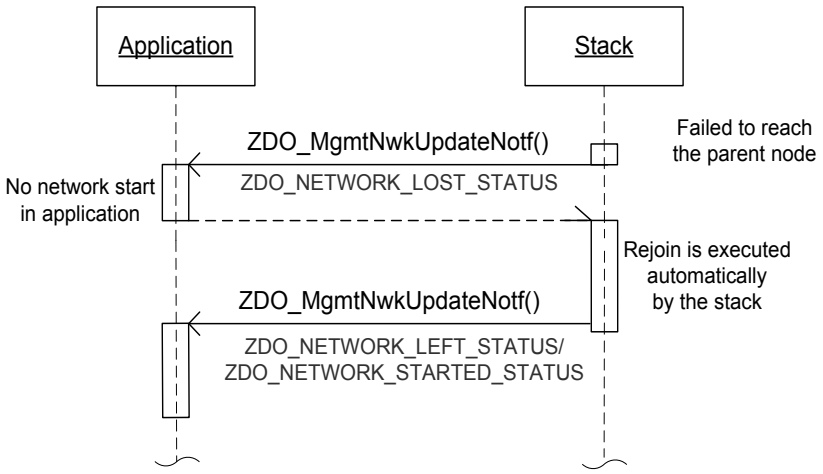
1. Parent loss. Because routers do not have dedicated parent nodes in a mesh topology, such a scenario is valid only for end devices.
2. The node is requested to leave the network. Such a request can be issued in one of two ways:
    a. By the application running on the node.
    b. By a remote node.

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

4.2.2.1 Parent Loss by an End Device

Figure 4-3 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 networks 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 4.2.1.

Figure 4-3. Parent Loss Sequence Diagram



4.2.2.2 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 5.9 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.

4.2.2.3 Network Leave on a ZDP Request

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 stack allows the application to initiate such a procedure on a node of any type using the `ZDO_ZdpReq()` function, which is used to issue ZDP requests serving many network-related tasks. To make a node leave the network the application should use a ZDP request of type `MGMT_LEAVE_CLID`. Refer to Section 4.3.6 for details on this type of ZDP requests and for code examples.

For example, the node can leave the network on own request. In this case, the request is not sent to the air, but goes down from ZDO to NWK and back to ZDO.

When a node leaves the network, the stack issues notification to the application via the `ZDO_MgmtNwkUpdateNotf()` function with the `ZDO_NETWORK_LEFT_STATUS` code.

4.2.3 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 4.2.1.4). 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 4-3 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 4-3. 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 4-3 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 may 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,
};
```

4.2.4 Network Update Notifications

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 4-4 provides descriptions and usage notes for the most common events. To find a complete list of statuses, refer to the Atmel BitCloud API Reference [3].

Table 4-4. 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 updated during network operation (for example due to frequency agility). 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 4.2.5. 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.
ZDO_SUCCESS_STATUS	Indicates that energy detection scan has been completed. See Section 4.3.5.1.

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, as well as influence the network state of single nodes. For example, the ZDP command of type `MGMT_LEAVE_CLID`, which has been discussed previously (see Section 4.2.2.3), makes it possible to force the current node, a single remote node, or a node with its children to leave the network. Another ZDP request, of type `MGMT_NWK_UPDATE_CLID`, can be used to change a working channel and perform energy detection scanning. This ZDP request is described in Section 4.3.5.

4.2.5 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 precede several events, switching logic according to the event ID received.

4.3 Network Management by ZigBee Device Profile Requests

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 commands are standardized, but the support of some commands is mandatory, and support of others is optional. ZDP requests are managed through the ZigBee Device Object, which is implemented by the stack and resides at application endpoint 0 (see Section 5.1.1 for information on application endpoints). 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 5.8)
- **Network control**, including commands for network leave, network update, obtaining the neighbor table from a remote node, etc.

Table 4-5 provides a list of ZDP requests supported in BitCloud. For a complete list of ZDP commands, refer to BitCloud API Reference [3] and the ZigBee PRO specification [6]. The specification also describes the response commands, though they can be sent only by the ZDO component and are not visible to the application.

Table 4-5. ZDP Requests Supported by the BitCloud Stack

Command ID	Description	Type of payload parameters
NWK_ADDR_CLID (0x0000)	Requests the short (network) address of a remote node with a given IEEE (extended) address. See Section 4.3.3.2.	ZDO_NwkAddrReq_t
IEEE_ADDR_CLID (0x0001)	Requests the IEEE (extended) address of a remote node with a given short (network) address. See Section 4.3.3.1.	ZDO_IeeeAddrReq_t
NODE_DESCRIPTOR_CLID (0x0002)	Request the node descriptor of a device	ZDO_NodeDescReq_t
POWER_DESCRIPTOR_CLID (0x0003)	Requests the power descriptor of a device	ZDO_PowerDescReq_t
SIMPLE_DESCRIPTOR_CLID (0x0004)	Requests the simple descriptor of the specified endpoint on a remote node.	ZDO_SimpleDescReq_t
ACTIVE_ENDPOINTS_CLID (0x0005)	Requests the list of all active endpoints on a remote device.	ZDO_ActiveEPReq_t
MATCH_DESCRIPTOR_CLID (0x0006)	Launches a search for devices that have endpoints supporting at least one of the specified clusters. See the example in Section 4.3.2.	ZDO_MatchDescReq_t
DEVICE_ANNCE_CLID (0x0013)	Requests the local node to send a device announcement frame notifying other nodes of device's presence and its capabilities.	ZDO_DeviceAnnceReq_t
BIND_CLID (0x0021)	Requests a remote node to insert an entry to its binding table. See Section 5.8.1.	ZDO_BindReq_t
UNBIND_CLID (0x0022)	Requests a remote node to remove an entry from its binding table with the specified cluster ID, source and destination endpoint and extended address.	ZDO_UnbindReq_t
MGMT_NWK_DISC_CLID (0x0030)	Requests a remote node to scan a network in its vicinity and report on the scan results.	Payload is not filled
MGMT_LQI_CLID (0x0031)	Obtains the list of a remote device's neighbors, along with corresponding LQI values. See example in Section 4.3.2	ZDO_MgmtLqiReq_t
MGMT_BIND_CLID (0x0033)	Retrieves the content of the binding table from a remote device.	ZDO_MgmtBindReq_t
MGMT_LEAVE_CLID (0x0034)	Requests network leave either for a current device or a remote node. See sections 4.2.2.3 and 4.3.6.	ZDO_MgmtLeaveReq_t

MGMT_PERMIT_JOINING_CLID (0x0036)	Configures the target node to allow nodes to enter the network via MAC association (joining the network for the first time) through this node. See Section 4.3.7.	ZDO_MgmtPermitJoiningReq_t
MGMT_NWK_UPDATE_CLID (0x0038)	Depending on payload configuration requests energy detection scanning, working channel change, or changing of other network parameters. See Section 4.3.5 for details.	ZDO_MgmtNwkUpdateReq_t

4.3.1 Sending a ZDP Request

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.

Warning: Make sure that the request object is not reused until the response is received and processed in the callback function.

The specific command type is specified in the reqCluster field, which takes a value from a range of predefined command IDs. However, in this context, cluster simply means a command.

The user shall also specify the address mode (dstAddrMode) used to address a node to which the request is sent. Depending on the selected address mode, either the short address (dstNwkAddr) or the extended address (dstExtAddr) shall be specified. The user also needs to set specific request parameters in the req.reqPayload field of type ZDO_ZdpReqFrame_t (for all commands except MGMT_NWK_DISC_CLID request, see Table 4-5. 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.

Caution: ZDP requests can be sent using either broadcast or unicast addressing – depending on the request's type. However, some requests, such as the match descriptor request, can be sent via both types of addressing.

The pointer to a callback function is specified in the ZDO_ZdpResp field. A function shall have the following signature:

```
void zdpResp(ZDO_ZdpResp_t *resp);
```

The callback's argument includes addressing information of the response's sender and the respPayload field containing the status field and the payload specific to request's type.

The callback function is called in different circumstances depending on the request's type. See details in the following sections.

4.3.1.1 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.

For all supported ZDP requests, except the match descriptor request, only one response command is expected. ZDO, receiving this response, calls the callback function with the status field set to the status code received in the response command. Then ZDO removes the request from the internal queues and ignores all other response commands to this request. If, however, timeout occurs when response has not yet been received then the response callback is called with the status field set to ZDO_RESPONSE_WAIT_TIMEOUT_STATUS.

For a request of MATCH_DESCRIPTOR_ID type ZDO expects several responses. In such case it always waits until timeout expires, calling the callback function for each received response, and indicates that reception of responses has completed with the ZDO_CMD_COMPLETED_STATUS code in the response callback. See how the match descriptor responses are processed in Section 4.3.2.

The CS_ZDP_RESPONSE_TIMEOUT parameter is set, by default, to an automatically calculated value, which takes into account only active (non-sleeping) devices. Devices that are sleeping when the request is sent are not expected to send the response. See Section 5.9 for detail and data exchange with end devices.

4.3.1.2 Processing a ZDP Request

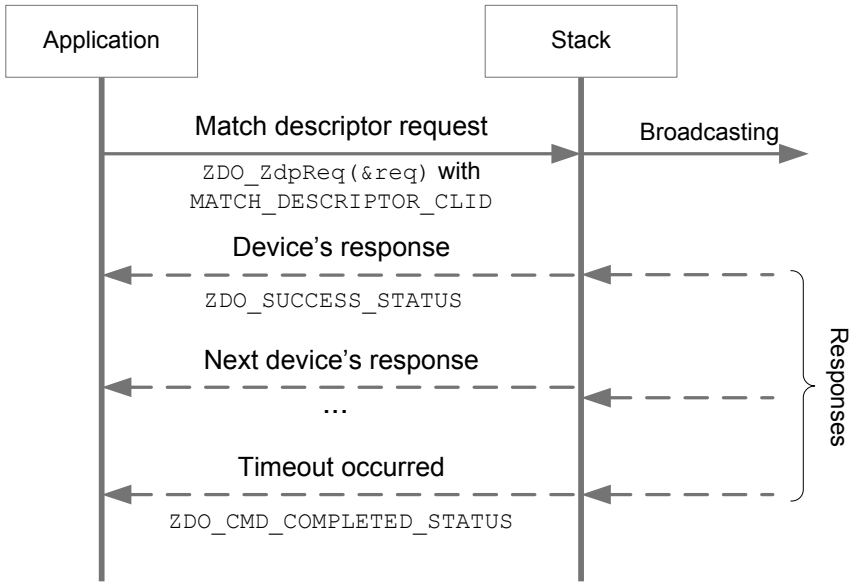
Incoming ZDP requests are processed by the ZDO component. The application is not notified directly about ZDP request the node receives, but the application can know about some of the requests through general notifications (for example, network leave notification – see Section 4.3.6) or by subscribing to some events (see Section 2.3).

4.3.2 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. The request is used for cluster binding between the nodes – see Section 5.8.

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. See Figure 4-4.

Figure 4-4. Service Discovery. Match Descriptor Request Sequence Diagram



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

```
static ZDO_ZdpReq_t zdpReq; //define a global variable...

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]`.

4.3.3 Device Discovery

4.3.3.1 IEEE Address Request

The ZDP request of `IEEE_ADDR_CLID` type is used to find out the unknown extended address of a device with the known short address. 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. The following code illustrates how to issue the request:

```
static ZDO_ZdpReq_t zdpReq; //Define a global variable
...
ZDO_IeeeAddrReq_t *ieeeAddrReq = &zdpReq.req.reqPayload.ieeeAddrReq;
zdpReq.ZDO_ZdpResp = zdpIeeeAddrResp; //Confirmation callback
zdpReq.reqCluster = IEEE_ADDR_CLID; //Type of request
zdpRequest.dstAddrMode = SHORT_ADDR_MODE; //Addressing mode
zdpRequest.dstNwkAddr = nwkAddr; //Address of the destination - the same as
                                 //in the payload
ieeeAddrReq->nwkAddrOfInterest = nwkAddr; //The short address of the device
                                          //which extended address is requested

ieeeAddrReq->reqType = SINGLE_RESPONSE_REQUESTTYPE;
ieeeAddrReq->startIndex = 0;
ZDO_ZdpReq(&zdpReq); //send the request
```

The short address of the device of interest is specified in the payload's nwkAddrOfInterest field. Two other fields, reqType and startIndex, should be set to SINGLE_RESPONSE_REQUESTTYPE and 0, respectively, if only the address of the target device is requested. But it is also possible to request extended addresses of all its child nodes (only end devices) – by having the reqType set to EXTENDED_RESPONSE_REQUESTTYPE. In this case, a single frame may not have enough room to enclose information about all child nodes. Receiving the response, the application should resend the request but with the startIndex field set to the position of the next child device (which equals the number of received entries), and repeat this until all children addresses are received.

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.

4.3.3.2 Network Address Request

The ZDP request of NWK_ADDR_CLID type does the opposite to the IEEE address request: it is used to find out the unknown short address of a device with the known extended address. See the following example:

```
static ZDO_ZdpReq_t zdpReq; //Define a global variable
...
ZDO_NwkAddrReq_t *nwkAddrReq = &zdpReq.req.reqPayload.nwkAddrReq;

zdpReq.ZDO_ZdpResp = zdpNwkAddrResp; //Confirmation callback
zdpReq.reqCluster = NWK_ADDR_CLID; //Type of request
zdpRequest.dstAddrMode = SHORT_ADDR_MODE; //Short addressing mode
zdpRequest.dstNwkAddr = BROADCAST_ADDR_ALL; //Send as a broadcast command

nwkAddrReq->ieeeAddrOfInterest = ieeeAddr; //The extended address of interest
nwkAddrReq->reqType = SINGLE_RESPONSE_REQUESTTYPE;
nwkAddrReq->startIndex = 0;
ZDO_ZdpReq(&zdpReq); //Send the request
```

The reqType and startIndex fields are used in the same way as for the IEEE address request (see Section 4.3.3.1).

Since the short address of the device is what should be found, a unicast frame cannot be used to reach the device. That is why the request is sent as a broadcast frame (see Section 5.7). The target device, receiving the request, sends back the ZDP response containing its short address (and short addresses of its end device children, if requested), while other nodes ignore the request.

4.3.4 LQI Request

The LQI ZDP request, of MGMT_LQI_CLID type, is used to retrieve the neighbor table of a remote node. Table entries will contain addressing information, LQI values, and other info.

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; // Globally defined variable
```

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; //Start index in the table
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;
        int i;
        for (i = 0; i < lqiResp->neighborTableListCount; i++)
        {
          char tmpStr[100];
          sprintf(tmpStr, "NWK addr: 0x%04x\r\nLQI: %u\r\n-----\r\n",
                  lqiResp->neighborTableList[i].networkAddr,
                  lqiResp->neighborTableList[i].lqi);
          strcat((char*)uartMsg, tmpStr);
        }
        //More code goes here
    }
}
```

Since the neighbor table is big enough not to fit into a single frame, only a limited number of items may be retrieved with a single request. But the whole neighbor table may be received by parts in several requests: in each request, the application should assign the payload's `startIndex` field to the position from which to start reading table entries.

In the code above and in the case of successful request execution, network addresses and LQI values from neighbor table entries received with the are formatted and written to a buffer, which can be, for example, sent to UART. The response payload, `zdpResp->respPayload.mgmtLqiResp` field, also includes the `startIndex` field (the same as in the request) and the `neighborTableEntries` field – the total number of entries in the target device's neighbor table.

4.3.5 Network Update Request

The ZDP command with type `MGMT_NWK_UPDATE_CLID`, or network update request, is designed to cope with multiple tasks, such as energy detection scan and changing a working channel. The actual task performed by the request depends on the value of the `scanDuration` field of the network update request's payload:

- **Energy detection scanning:** `scanDuration` field set to values from `ZDO_MGMT_ED_SCAN_DUR_0` (`0x00`) to `ZDO_MGMT_ED_SCAN_DUR_5` (`0x05`)
- **Changing the current channel:** `scanDuration` field set to `ZDO_MGMT_CHANNEL_CHANGE` (`0xFE`)
- **Changing the channel mask and the network manager's address:** `scanDuration` field set to `ZDO_MGMT_NWK_PARAMS_CHANGE` (`0xFF`)

4.3.5.1 Energy Detection Scan

Energy detection scan measures energy level in dBm on each of the provided channels. To perform energy detection scan send a ZDP request, setting `MGMT_NWK_UPDATE_CLID` as reqCluster and configuring the `req.reqPayload.mgmtNwkUpdateReq` field in the following way: set `scanDuration` to a value in the range from `ZDO_MGMT_ED_SCAN_DUR_0` (0x00) to `ZDO_MGMT_ED_SCAN_DUR_5` (`0x05`), provide channels to scan in the `scanChannels` field, and the number of scan attempts in the `scanCount` field. Actual duration of scanning depends exponentially on the `scanDuration` value.

The scanning is performed by the node specified in the request as the destination, which is identified by `dstAddrMode` and `dstNwkAddr` fields. To made the node that sends the request perform the scan its own address shall be specified. The payload's `scanCount` specifies how many times the specified channels shall be scanned. Separate response will be received for each scanning attempt.

See the example code below:

```
zdpReq.reqCluster = MGMT_NWK_UPDATE_CLID;
zdpReq.dstAddrMode = SHORT_ADDR_MODE;
zdpReq.dstNwkAddr = DST_SHORT_ADDRESS; //A value defined by the application

//Get current channel mask from Configuration Server
uint32_t chMask;
CS_ReadParameter(CS_CHANNEL_MASK_ID, &chMask);
zdpReq.req.reqPayload.mgmtNwkUpdateReq.scanChannels = chMask;

zdpReq.req.reqPayload.mgmtNwkUpdateReq.scanDuration = ZDO_MGMT_ED_SCAN_DUR_5;
zdpReq.req.reqPayload.mgmtNwkUpdateReq.scanCount    = 0x01; //One attempt
zdpReq.ZDO_ZdpResp = ZDO_ZdpResp; //The callback function is called when
                                  //the request is sent
ZDO_ZdpReq(&zdpReq);
```

Note that the callback function specified in the `ZDO_ZdpResp` field is called when the request is sent, and does not indicate that energy detection has been completed. In case the callback function reports success, the result of the scanning is received as a network update notification; the stack indicates it to the application, calling the `ZDO_MgmtNwkUpdateNotf()` function with status `ZDO_SUCCESS_STATUS`. The argument's `scanResult` field in this case contains scan data. The application may process this in the following way:

```
void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *nwkParams)
{
  switch (nwkParams->status)
  {
    case ZDO_SUCCESS_STATUS:
    {
      EDScan_t* edScan = &nwkParams->scanResult;
      // Process edScan's fields
    }
  }
}
```

`EDScan_t` structure obtained from the argument contains scan results:

- The `scanChannels` field is the bitmask of scanned channels (channels mask)
- The `totalTransmissions` field reports the total number of transmissions done during energy detection scan, and `transmissionsFailures` reports the number of failed transmissions

- Energy values measured for scanned channels are given in the `energyValues` array having `scannedChannelsListCount` items. Values are within the range from 0 to 84. Actually measured input power in dBm can be obtained by subtracting 91 from a value

4.3.5.2 Changing 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.

To change an operating channel send a ZDP request, setting `MGMT_NWK_UPDATE_CLID` as reqCluster and configuring the `req.reqPayload.mgmtNwkUpdateReq` field in the following way: set `scanDuration` to `ZDO_MGMT_CHANNEL_CHANGE` to indicate that this is a changing channel request and specify the new channel in the `scanChannels` field. To set a particular channel in the `scanChannels` field shift 1 via the `<<` operator by the channel's number (see the code example below).

The request may be sent to a particular node or broadcasted across the whole network (typically, such broadcast command is sent by the stack on the network manager node).

See the example code below:

```
//Define a variable in the file scope
ZDO_ZdpReq_t zdpReq;
...
zdpReq.ZDO_ZdpResp = appZdpNwkUpdateResp; //Set to the pointer of a callback
                                          // function to check request's status

zdpReq.reqCluster = MGMT_NWK_UPDATE_CLID;
zdpReq.dstAddrMode = SHORT_ADDR_MODE;
zdpReq.dstNwkAddr = DST_SHORT_ADDRESS; //The short address of the destination node

zdpReq.req.reqPayload.mgmtNwkUpdateReq.scanDuration = ZDO_MGMT_CHANNEL_CHANGE;
zdpReq.req.reqPayload.mgmtNwkUpdateReq.scanChannels = 1ul << NEW_CHANNEL;

ZDO_ZdpReq(&zdpReq);
```

The callback function's field in the example is set to `NULL`, although a function's pointer may be provided to check that request has been sent.

4.3.5.3 Changing Network Parameters

A ZDP request with the `reqCluster` field of the argument set to `MGMT_NWK_UPDATE_CLID` and the `scanDuration` field of the payload set to `ZDO_MGMT_NWK_PARAMS_CHANGE` (`0xFF`) is used to change the channel mask and the network manager's address parameters (the network update ID may be also changed via this request). The request is formed in the same way as the change channel request (see Section 4.3.5.2), but with `scanDuration` set to a different value and with the network manager short address provided in the `nwkManagerAddr` field (network update ID may be set in the `nwkUpdateId` filed).

4.3.6 Network Leave Request

A ZDP request with the `reqCluster` field of the argument set to `MGMT_LEAVE_CLID` makes the specified node and, if requested, all its children leave the network. This request has been introduced in Section 4.2.2.3.

To send a leave request configure an instance of the ZDO_ZdpReq_t type, setting the reqCluster field to MGMT_LEAVE_CLID. Request parameters include two address fields: the destination address (common to all ZDP requests) and the target address in the payload. The first field specifies the node to which the request is sent. This node, receiving the request, sends the NWK leave request to the node specified in the payload causing it to leave the network. If the destination node is the target node at the same time, the address in the payload (deviceAddr field) may be set to 0. In this case the destination node itself will leave the network.

The payload, req.reqPayload field of the argument, has two more fields, in addition to the deviceAddr field described above:

- The removeChildren bit field, set to 1, causes the target node to remove recursively all its children from the network by sending the NWK leave requests to all its children, before actually leaving the network. If removeChildren is set to 0 only the target node will leave the network. Note that if the device has sleeping children it will wait until they wake up and retrieve leave requests from it. Since sleep interval can be long the original request can expire by that time. So if it is required that only a certain node leaves the network and without delay, it is recommended to have this bit set to 0.
- The rejoin bit field indicates if the node shall rejoin the network after leaving it. Rejoin will happen it the field is set to 1 and will not if it is set to 0

See the example code below:

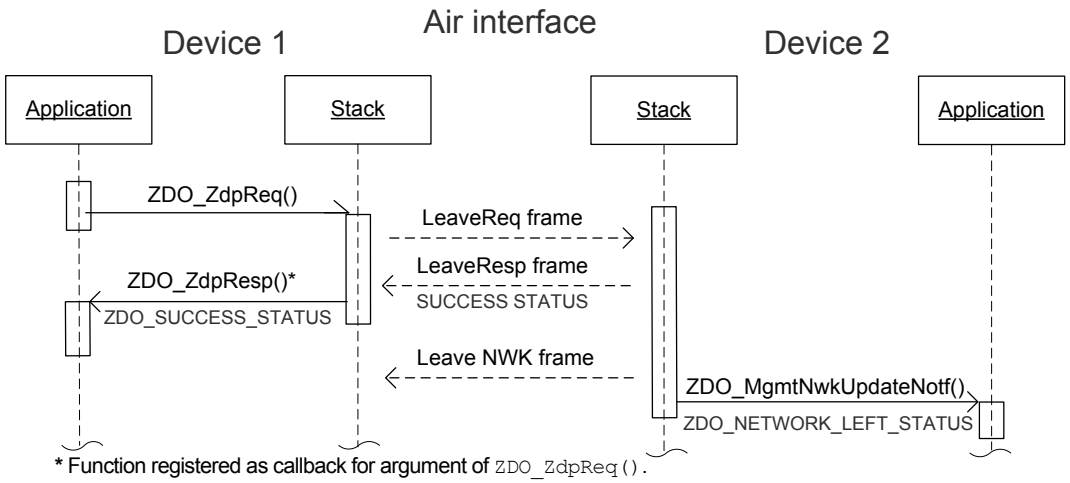
```
//Define a variable in the file scope
ZDO_ZdpReq_t zdpReq;
...
zdpReq.ZDO_ZdpResp = appZdpLeaveResp; //Set to the pointer of a callback function
                                      //to check request's status
zdpReq.reqCluster = MGMT_LEAVE_CLID;
zdpReq.dstAddrMode = EXT_ADDR_MODE;
zdpReq.dstExtAddr = DST_EXT_ADDRESS; //The extended address of the node that will
                                     //execute the request (via NWK commands)
zdpReq.req.reqPayload.mgmtLeaveReq.deviceAddr = TARGET_EXT_ADDRESS; //The address
                                          //of a device that will leave the network
zdpReq.req.reqPayload.mgmtLeaveReq.removeChildren = 0; //Children will stay
zdpReq.req.reqPayload.mgmtLeaveReq.rejoin = 0; //The node will not rejoin immediately

ZDO_ZdpReq(&zdpReq);
```

As the result of this code's execution the node with extended address TARGET_EXT_ADDRESS will leave the network, its children nodes will not leave the network, and it will not rejoin the network.

4.3.6.1 Leave Request to a Remote Note

The sequence diagram on Figure 4-5 shows how a node leaves the network upon a remote request when the destination node is the one that should leave the network.

Figure 4-5. Network Leave Sequence Diagram. Device 1 requests Device 2 to leave the Network

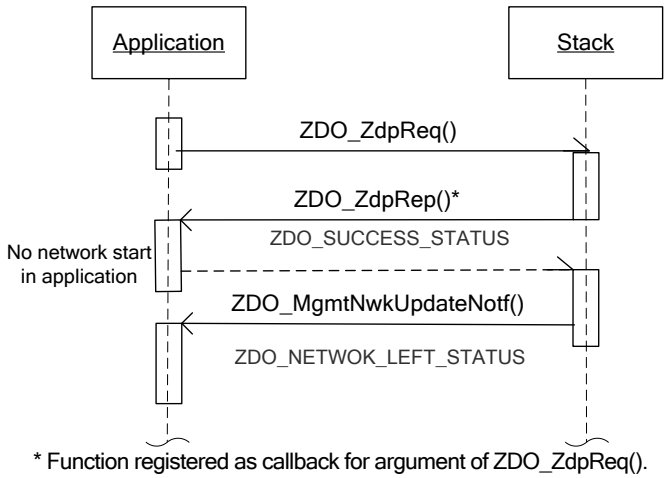


If the destination address does not equal zero (that is, a remote node is to leave the network), the stack replies with a callback as soon as it sends the leave request frame and receives the leave response frame with a success status, which means that the remote node has started the network leave procedure. If this procedure executes successfully, the application on the remote node, rather than the node that initiated the process, is notified by the `ZDO_MgmtNwkUpdateNotf()` function with status `ZDO_NETWORK_LEFT_STATUS`.

4.3.6.2 Network leave on own Request

The node can send a ZDP leave request to itself to leave the network. To send such a request, in the request parameters simply set 0 as the destination address and in the `deviceAddr` field in the payload. Setting real node's address in either of the fields or both will also work. Figure 4-6 illustrates network leave of a node on its own request with a sequence diagram.

Figure 4-6. Network leave Sequence Diagram (local call)



The application issues a ZDP request of type `MGMT_LEAVE_CLID`. The stack processes 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`.

4.3.7 Permit joining Request

The permit joining ZDP request, of `MGMT_PERMIT_JOINING_CLID` type, is used to forbid and permit joining to the target node by MAC association. The request affects the target node, which is the node specified as request's destination. The request may be sent as a unicast command to just one node or as a broadcast command.

The `CS_PERMIT_DURATION` parameter on a node sets the initial configuration. The default value in the stack is `0xFF`, which allows joining permanently, but some reference applications change it to `0x00` per ZigBee Profile requirements. The parameter shall be also used to change the permit join configuration at any time before network start. After the node enters a network, it can only change its permit joining configuration by addressing itself with the permit joining ZDP request of `MGMT_PERMIT_JOINING_CLID` type or by receiving same command from a remote node.

Joining via MAC association may be permitted for a given interval in seconds or forbidden permanently, depending on the payload's `permitDuration` field. This field specifies the duration of time, starting from the moment of request's reception, when joining by association is permitted. The limit values are:

- `0x00` – forbids joining permanently
- `0xFF` – permits joining permanently

Consider the following example:

```
static ZDO_ZdpReq_t zdpReq; //Define a global variable
...
ZDO_MgmtPermitJoiningReq_t *permit = &zdpReq.req.reqPayload.mgmtPermitJoiningReq;
zdpReq.ZDO_ZdpResp = zdoPermitJoiningResp; //The callback function
zdpReq.reqCluster = MGMT_PERMIT_JOINING_CLID;
zdpReq.dstAddrMode = SHORT_ADDR_MODE;
zdpReq.dstNwkAddr = 0x0001; //The node with 0x0001 short address will be addressed
permit->permitDuration = 5; //Permit joining for 5 seconds
ZDO_ZdpReq(&zdpReq);
```

5. Data Exchange

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.

The overview section (5.1) introduces important ZigBee concepts such as profiles, endpoints and clusters, and explains generally how data exchange is handled in the Atmel BitCloud stack.

Further sections cover more advanced topics. Section 5.2 provides detail instructions on how to implement application endpoint with desired clusters supported. Section 5.4 explains how to maintain cluster attributes locally on the node. While Sections 5.5 and 5.6 describe how to issue general and cluster-specific ZCL commands respectively. Multicasting and broadcasting subjects that allow data transmission to a group or to every node in the network are covered in Section 5.7. Establishing of virtual connections between endpoints via cluster binding is discussed in Section 5.8. Finally section 5.9 explains parent polling mechanism that is used for delivering data to sleeping end devices.

5.1 Overview

5.1.1 Application Endpoints and Data Transfer

In ZigBee PRO standard [6] data exchange on the application layer requires use of so called application endpoints.

Endpoints correspond to application objects, which are logically embraced into an application framework and serve as source and destination points for application data transfer in ZigBee networks. 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).

A node cannot participate in application data exchange 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 source node must specify a destination endpoint, too. Figure 5-1 illustrates this concept.

Figure 5-1. Data Exchange in ZigBee Applications

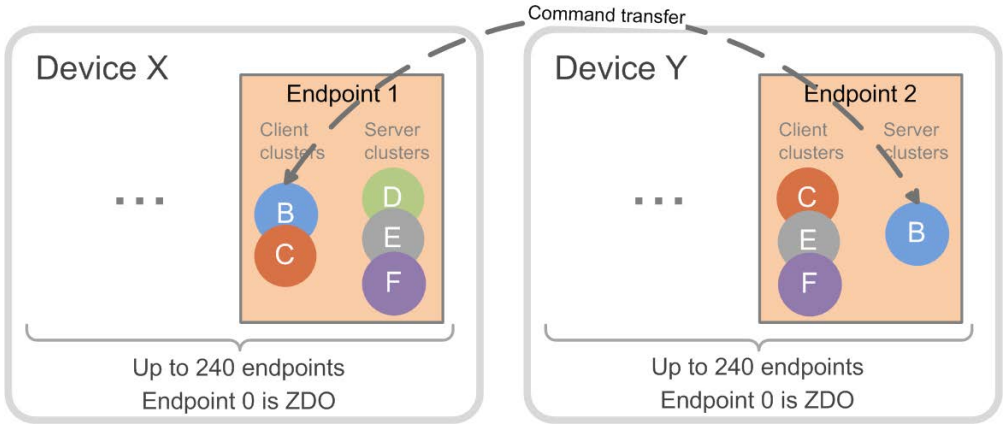


Figure 5-1 also shows that actual data exchange on application endpoints is performed within so called clusters that specify standard communication interface (commands, attributes and behavior) for a particular functionality (for example Time, On/Off control, Temperature). The set of clusters supported by an endpoint may be regarded as the description of endpoint functionality. More information on cluster notion is given in Section 5.1.2.

Each application endpoint is fully characterized by following parameters: endpoint ID, supported application profile (see Section 5.1.2.1), device ID (see Section 5.1.2.1), device version and lists of supported client and server clusters. This information composes a simple descriptor for the endpoint. The stack uses the simple descriptor to filter incoming frames. The indication of data reception is sent to the destination endpoint callback functions only when the message fits the profile and cluster information contained in the simple descriptor of the target endpoint. A simple descriptor is also employed in service discovery procedures, which allow a node to search for devices with particular profile and cluster configurations as described in Section 4.3.

A physical device is not restricted to use a single endpoint for all data interactions and may register several endpoints. This might be particularly useful if a physical device operates as two or more different logical devices. In this case clusters specific for logical devices may be registered on separate endpoints, while common clusters may be supported by a shared endpoint.

Detailed description and an example of endpoint registration are given in Section 5.2.

5.1.2 Clusters

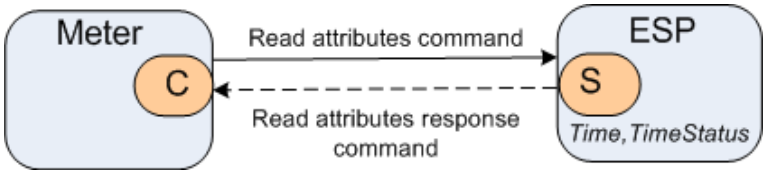
A *cluster* is a collection of *commands* and *attributes* that together form a communication interface to specific functionality; for example, time, level control, metering etc. Commands represent certain actions performed by the cluster, and attributes are cluster parameters manipulated by the application. ZigBee Cluster Library (ZCL) specification [6] provides cluster definitions that can be used by different application profiles.

Communication within a cluster involves two sides: a *client* and a *server*. Typically, the server stores and maintains attributes, and the client sends requests to manipulate attributes on the server. Some clusters allow clients to modify attributes, while for others this is possible only for the device holding the attribute. Such attributes are called read-only.

There is also a class of attributes that are stored on clients. Such attributes, which may be regarded to as local settings, are called client attributes. A device holding an attribute has full access to it and is able to write, read, and report the attribute value when needed. Reporting an attribute means sending the attribute's value to remote devices.

Figure 5-2 shows example of a cluster: a meter device serves as a client side of the Time cluster, while ESP supports the server-side Time cluster. The meter sends read attributes command (a general cluster command) to the ESP to obtain the Time attribute (holding the current time value) and receives a read attributes response command, containing the value of the Time attribute.

Figure 5-2. Interaction between ESP and Meter over the Time Cluster



How to use ZCL clusters, attributes, and commands in BitCloud application is described in Sections 5.2 to 5.6.

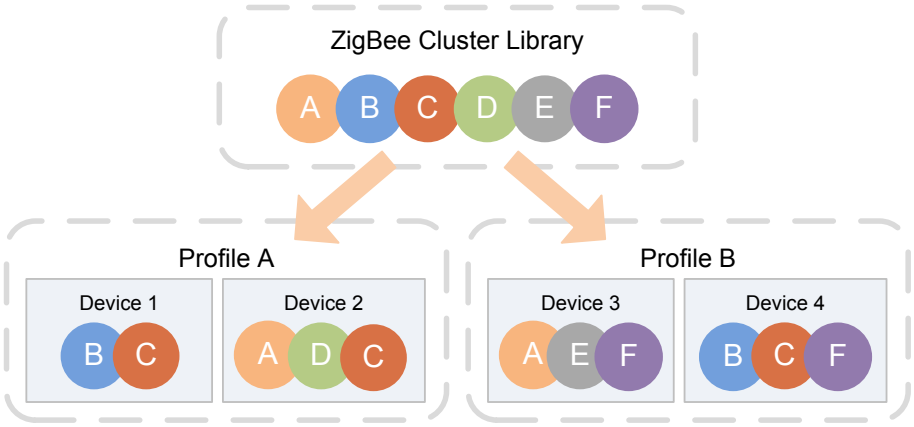
5.1.2.1 ZigBee Application Profiles

ZigBee application profiles describe specific application domains by defining functional blocks for all types of logical devices occurring in the domains. Public profiles released by ZigBee Alliance include Home Automation, LightLink, Smart Energy and others (see [9]).

Specification of a public ZigBee profile defines types of devices that can operate in the network. For each device type it provides the list of required clusters and the list of optional clusters, thus defining the device type. On the application level a profile is identified with its profile ID number.

Data exchange requires that communicating endpoints are assigned to the same profile. An application profile identifier is a mandatory identifier for every registered endpoint.

Figure 5-3. ZigBee Cluster Library and Public Profiles



Vendors should follow public profiles' specifications in order to certify devices they manufacture and keep them interoperable with device from other vendors. However, any vendor may extend the profile with its own, manufacturer-specific clusters, attributes or commands as described in Appendix A.

5.1.3 Packet 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 (the maximum number of hops in one direction) equals `2*CS_MAX_NETWORK_DEPTH`.

5.1.4 Delivery Mode by Destination Address

After an endpoint is registered, the application can use it to send data to endpoints on remote nodes using the corresponding ZCL API functions (see Sections 5.5 and 5.6). Depending on the destination address configuration data frames can be sent in unicast, broadcast, or multicast ways:

- A unicast message is delivered to a single node identified by its short address or extended address (see Section 4.1.2 for device addressing description), or as a remote node to which the current node is connected through binding (see Binding described in Section 5.8). In last two cases address resolution might need to be performed as described in Section 5.5.1.2 because over the air only short address is used for frame transmission.
- 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. Refer to Section 5.7 for more details.
- Multicasting means addressing to a group of nodes. For details concerning multicasting see Section 5.7.2

The application shall properly configure a data frame packet, clarify a destination (see Section 5.5.1.1), and issue a request to the stack. After that, the stack is fully responsible for delivering the data frame to the specified destination.

5.2 Cluster Implementation

A cluster shall be defined in specific way, before it can be used by an application endpoint (5.3).

Each cluster shall have its own header file accessible for the application. Default location for cluster definition files is he ZCL's `include` directory (`BitCloud/Components/ZCL/include/`).

The header file for a cluster defines all necessary types and constants for cluster's commands and attributes:

- Constants for attributes IDs
- Constants for commands IDs
- A C-structure embracing all attributes
- A C-structure embracing all commands
- The command payload definition for each supported command
- The macro for attributes instance initialization in the application
- The macro for commands instance initialization in the application
- The macro for cluster initialization in the application

Application will use the macros, contestants and types implemented in the cluster definition file when defining cluster instances and registering them on application endpoints as described in Section 5.3.

Clusters currently supported in the BitCloud SDK are already implemented in such way and Appendix B explains in details how to modify cluster definition files or create new ones.

5.3 Application Endpoint Implementation

The application should define and register one or several application endpoints for data exchange. Endpoints for profile-based applications are registered using the `ZCL_RegisterEndpoint()` function. Frames delivered to such endpoints are raised through the stack up to the ZCL component, which processes them, invoking application callback functions when required.

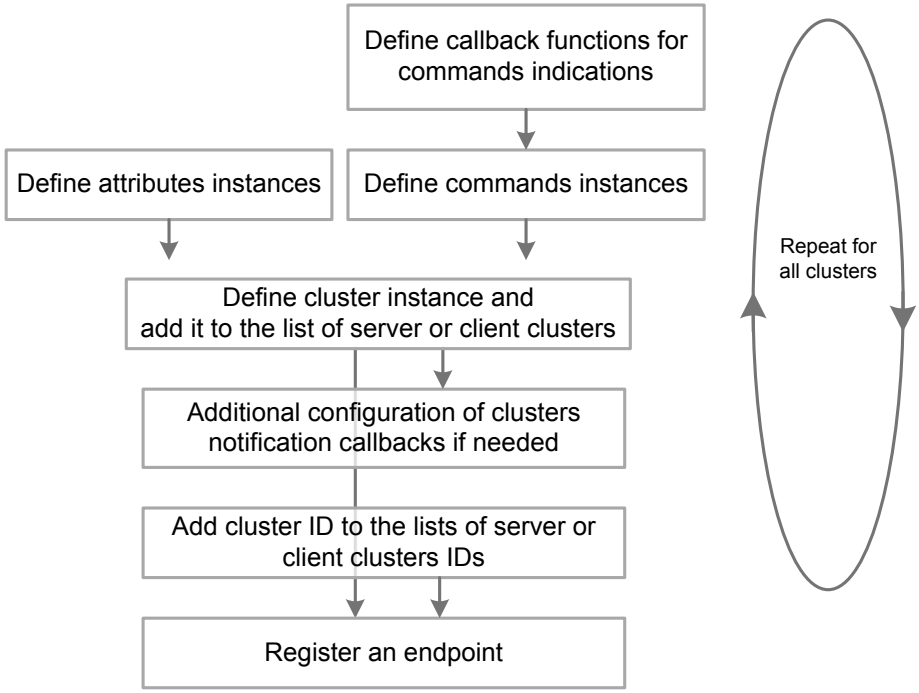
5.3.1 General Procedure

A ZCL endpoint is provided with lists of client and server clusters and list of their IDs. The former lists comprise elements of `ZCL_Cluster_t` type, which contain information about cluster attributes and commands. The mechanism of cluster definition and endpoint registration is illustrated on Figure 5-4.

In order to support certain clusters on the application endpoint the following steps shall be done:

1. Make sure that cluster definition header file (as described in Section 5.2) is available in the ZCL component (`\ZCL\include\` directory) or application code. This file shall define available cluster attributes and commands.
2. Decide whether the cluster should be a client or a server cluster.
3. Define callback functions for cluster-specific commands on the selected cluster side and define an instance of commands linking those callbacks (Section 5.3.3).
4. Define an instance of cluster attributes for the selected cluster side (Section 5.3.2).
5. Define the cluster instance, linking it with defined instances of commands and attributes and add the cluster instance to the list of client or server clusters for the target application endpoint (Section 5.3.4).
6. Define additional cluster-specific notification callbacks where needed (Section 5.3.5).
7. Add the cluster ID to the list of client or server cluster IDs for the target endpoint (Section 5.3.6).
8. Repeat steps 1 to 7 for all clusters to be supported by the endpoint.
9. Define an endpoint instance and link it with the clusters and cluster ID lists (Section 5.3.7).
10. Register endpoint (Section 5.3.7).

Figure 5-4. Configuration Clusters and Endpoint Registration in an Application



Caution: Instance of the OTAU cluster is defined inside the ZCL component and doesn't need to be configured by the application as described above. Pointer to the instance of OTA cluster may be obtained by calling `ZCL_GetOtauServerCluster()` / `ZCL_GetOtauClientCluster()` functions (for detail refer to [5]), and then added to the lists of clusters supported by the endpoint.

Note: An application can also register an endpoint via the `APS_RegisterEndpoint()` function, although frames received on such endpoint are not propagated to the ZCL component. Therefore if such endpoints are used for transferring ZCL commands, the application is responsible for processing APS payload. The application can also send raw APS data frames to any endpoint, using the `APS_DataReq()` function. For detail on APS APIs usage see Appendix C.

5.3.2 Defining Instance of Cluster Attributes

The header file for every cluster defines macros and types for server and client attributes, if they are supported by the cluster. To support a server cluster, the application should define an instance of server attributes type defined in the cluster header file. Otherwise it should define an instance of the client attributes type.

Note: By defining an attributes instance an application only allocates memory for storing attributes and do not assigns values to the attributes. Default values are also not assigned automatically to the attributes. The application should explicitly set desired values to all attributes, either directly through the attribute's instance or using API described in Section 5.4.

The body of an attributes instance declaration should contain an invocation of the corresponding macro. The macro name usually looks like this:

```
ZCL_DEFINE_<ClusterName>_CLUSTER_<SERVER or CLIENT>_ATTRIBUTES()
```

For example, an instance of the time cluster server attributes may be defined in the application this way:

```
static ZCL_TimeClusterServerAttributes_t timeAttributes =
{
    ZCL_DEFINE_TIME_CLUSTER_SERVER_ATTRIBUTES()
};
```

Where the ZCL_TimeClusterServerAttributes_t type and the ZCL_DEFINE_TIME_CLUSTER_SERVER_ATTRIBUTES() macro are defined in the zclTimeCluster.h file.

If the cluster does not have attributes this step is skipped. In this case the attributes parameter of the macro that initializes the cluster shall be set to NULL.

5.3.3 Defining Instance of Cluster Commands

The macro that defines indication callbacks for cluster-specific commands, and the commands payload types are defined in the cluster header file. The macro name looks like this:

```
DEFINE_<ClusterName>_CLUSTER_COMMANDS(<indications>)
```

The macro should be provided with indication callback functions for the cluster-specific commands to be received by the corresponding cluster side: client commands for the server cluster and server commands for the client cluster. Note that general cluster commands are supported by ZCL and do not require definition by application. The user should put callback functions in the right order, that is, as the commands are defined in the cluster header file. For commands that will be sent by the cluster set NULL to the macro's corresponding arguments.

The application may not support reception of a command, for example, because the command is intended for reception by a server, while the application is a client for a given cluster, or because this command is optional to support. In this case, put NULL for the corresponding command indication callback in the macro's argument.

For example, an application may define a commands instance for the server side of the messaging cluster in the following way:

```
static MessagingClusterCommands_t espMessagingCommands =
{
  DEFINE_MESSAGING_CLUSTER_COMMANDS(NULL, NULL,
                                    NULL, messageConfirmationInd)
};
```

As seen from the example, only the message confirmation command is supported for reception, so the first three arguments are NULL. Note that the instance of cluster-specific commands defines only commands that can be received by this cluster. Cluster-specific commands that can be sent are determined by the application code that forms and sends such commands as described in Section 5.6.

The indication function for the message confirmation command should have the following signature:

```
ZCL_Status_t messageConfirmationInd(ZCL_Addressing_t *addressing,
                                    uint8_t payloadLength,
                                    MessageConfirmation_t *payload)
```

The MessaginClusteCommands_t and MessageConfirmation_t types as well as the DEFINE_MESSAGING_CLUSTER macro are defined in the zclMessagingCluster.h file.

If the cluster does not receive cluster-specific commands the step described in this section is skipped. In this case the commands parameter of the macro that initializes the cluster shall be set to NULL.

5.3.4 Filling a List of Supported Clusters

Once instances of attributes and commands are defined, the application should define lists of server and client clusters. List elements must have the ZCL_Cluster_t type and should be defined with the help of special macros. A macro name may look like this:

```
DEFINE_<ClusterName>_CLUSTER()
```

The macro takes three parameters:

- Indication of whether the cluster is server or client
- A pointer to attributes instance (`NULL` if the cluster does not have attributes)
- A pointer to commands instance (`NULL` if the cluster does not have commands)

For example, consider a device that serves as a server for the messaging cluster, the identify cluster, and the basic cluster. In this case, the list of server clusters may be defined in the following way:

```
static ZCL_Cluster_t deviceServerClusters[] =
{
  DEFINE_MESSAGING_CLUSTER(ZCL_SERVER_CLUSTER_TYPE,
                    NULL, // no attributes
                       &espMessagingCommands),
  DEFINE_IDENTIFY_CLUSTER(ZCL_SERVER_CLUSTER_TYPE,
                    &espIdClusterServerAttributes,
                    &espIdClusterServerCommands),
  ZCL_DEFINE_BASIC_CLUSTER_SERVER(&basicClusterServerAttributes)
};
```

The last macro defines the server side of the basic cluster and does not take arguments except for the attributes instance.

5.3.5 Setting Indication Functions for Clusters

Once clusters' structures are defined, the application should specify indication functions for each cluster if they are required. The following callback functions may be specified:

- Indication function for reception of unsolicited (non-requested) default response commands is configured by the `ZCL_DefaultRespInd` field of the cluster's instance (see Section 5.3.6)
- The callback function set to the `ZCL_AttributeEventInd` field of the cluster's instance indicates events related to reading, writing and configuring of cluster attributes
- Attributes reporting indication function is set to the `ZCL_ReportInd` field of the cluster instance (see Section 5.5.3 for detail). The function is called when an attribute's report for the cluster is received

5.3.6 Filling a List of Supported Cluster IDs

The application must fill lists of server and client clusters' IDs for each endpoint and specify them in the endpoint registration parameters. Cluster identifiers may be observed in the `clusters.h` file. Elements in the list should have the `ClusterId_t` type. For the stack the order of elements in the clusters IDs list does not matter.

The example below shows definition of client cluster IDs list for Energy Service Portal (ESP).

```
static ClusterId_t espClientClusterIds[] =
{
  SIMPLE_METERING_CLUSTER_ID,
  BASIC_CLUSTER_ID
};
```

5.3.7 Application Endpoint Registration

To register an application endpoint, configure a global instance of the `ZCL_DeviceEndpoint_t` type. Fill the simple descriptor fields of the instance and provide lists of server and client clusters. In the simple descriptor (the `simpleDescritor` field), specify the endpoint number, the profile ID, the profile device ID, amount and lists of server and client clusters IDs.

Finally pass the pointer to the endpoint instance to the `ZCL_RegisterEndpoint()` function.

For example, the endpoint for the Energy Service Portal device (ESP) may be registered with the following code:

```
#define IN_CLUSTERS_COUNT  (ARRAY_SIZE(espServerClusterIds))
#define OUT_CLUSTERS_COUNT (ARRAY_SIZE(espClientClusterIds))
ZCL_DeviceEndpoint_t appEndpoint; //This variable must be global or static

appEndpoint.simpleDescriptor.endpoint = APP_ENDPOINT_ESP;
appEndpoint.simpleDescriptor.AppProfileId = PROFILE_ID_SMART_ENERGY;
appEndpoint.simpleDescriptor.AppDeviceId = ZSE_ENERGY_SERVICE_PORTAL_DEVICE_ID;
appEndpoint.simpleDescriptor.AppInClustersCount = IN_CLUSTERS_COUNT;
appEndpoint.simpleDescriptor.AppInClustersList = espServerClusterIds;
appEndpoint.simpleDescriptor.AppOutClustersCount = OUT_CLUSTERS_COUNT;
appEndpoint.simpleDescriptor.AppOutClustersList = espClientClusterIds;
appEndpoint.serverCluster = espServerClusters;
appEndpoint.clientCluster = espClientClusters;

ZCL_RegisterEndpoint(&appEndpoint);
```

In the example, `espServerClusterIds` and `espClientClusterIds` variables define lists of server and client clusters IDs (Section 5.3.6); `espServerClusters` and `espClientClusters` variables define lists of server and client clusters instances (Section 5.3.4). The `APP_ENDPOINT_ESP` constant specifies endpoint ID; it is a value between 1 and 240 defined by the application.

Note: When an endpoint is registered via the ZCL function frame reception will be indicated only by callback functions registered for the cluster commands.

5.4 Local Attribute Maintenance

To read a local attribute (stored on the device) the `ZCL_ReadAttributeValue()` function should be called, specifying endpoint ID, cluster ID, cluster side (server or client), and two pointers to which the function will write the attribute data type and the attribute value.

For example, to read the Server Time Attribute of the Time cluster registered as a server cluster on `APP_ENDPOINT`, the following code may be used:

```
uint16_t timeout;
uint8_t attrDataType;
ZCL_ReadAttributeValue(APP_ENDPOINT,
          TIME_CLUSTER_ID,
          ZCL_CLUSTER_SIDE_SERVER,
          ZCL_TIME_CLUSTER_SERVER_TIME_ATTRIBUTE_ID,
          &attrDataType,
          (uint8_t *)&timeout);
```

Note that the attribute value pointer (`&timeout`) should be cast to the `uint8_t*` type. The attribute ID can be found in the cluster's header file.

To write a local attribute the `ZCL_WriteAttributeValue()` function should be used. This function takes the set of parameters similar to those of the `ZCL_ReadAttributeValue()` function described above. The attribute value must be converted to the little-endian format using one of the `CPU_TO_LE*` macros. The same attribute as in the previous example can be modified via the following code:

```
ZCL_WriteAttributeValue(APP_ENDPOINT_ESP,
          TIME_CLUSTER_ID,
          ZCL_CLUSTER_SIDE_SERVER,
          ZCL_TIME_CLUSTER_SERVER_TIME_ATTRIBUTE_ID,
          ZCL_UTC_TIME_DATA_TYPE_ID,
```

```
            (uint8_t *)&timeout);
```

Note: To find out attribute's data type open the header file containing definitions for the cluster and skip to the section where the attributes are defined via the `DEFINE_ATTRIBUTE()` macro. For example, see the following definition:

```
DEFINE_ATTRIBUTE(time,
                 ZCL_READWRITE_ATTRIBUTE,
                 ZCL_TIME_CLUSTER_SERVER_TIME_ATTRIBUTE_ID,
                 ZCL_UTC_TIME_DATA_TYPE_ID),
```

Attribute's data type is `ZCL_UTC_TIME_DATA_TYPE_ID`, set in the last argument.

5.5 General Cluster Commands

General cluster commands are supported by all clusters and are used for discovering, reading, writing, and reporting cluster attributes. Cluster-specific commands are discussed in Section 5.6.

The following sections describe how to send general cluster commands (Section 5.5.1) and give examples on how to read and write remote attributes using the corresponding general cluster commands (Section 5.5.2), configure and use reportable attributes (Section 5.5.3) as well as report attributes from the application, and how to discover attributes supported on remote devices (Section 5.5.4).

Table 5-1 lists all general commands implemented in BitCloud SDK. Enumerator type `ZCL_GeneralCommandId_t` defines the values and names for the general command IDs.

Table 5-1. Commands Supported by all Clusters by Default

Command ID value	Command
0x00	Read attributes
0x01	Read attributes response
0x02	Write attributes
0x03	Write attributes undivided
0x04	Write attributes response
0x05	Write attributes no response
0x06	Configure reporting
0x07	Configure reporting response
0x08	Read reporting configuration
0x09	Read reporting configuration response
0x0A	Report attributes
0x0B	Default attributes
0x0C	Discover attributes
0x0D	Discover attributes response
0x0E	Read attributes structured
0x0F	Write attributes structured
0x10	Write attributes structured response

5.5.1 Sending a General Cluster Command

General cluster commands are sent via the `ZCL_AttributeReq()` function that has the pointer to an instance of the `ZCL_Request_t` type as an argument. The structure behind the pointer includes complete destination specification (device address, destination endpoint, cluster ID etc.), command ID, source endpoint ID, payload length, and payload itself.

The argument also includes the `defaultResponse` field for configuring default response bit. If it is set to `ZCL_FRAME_CONTROL_ENABLE_DEFAULT_RESPONSE` then ZCL default response command will be sent from the destination but not for the commands that already require command-specific ZCL response (e.g. read attribute command). Reception of unsolicited default responses is indicated by a special callback function (see Section 5.3.5).

Consider the example:

```
ZCL_Request_t zclReq; //In the global scope
...
zclReq.id = …; // specify command ID, e.g. ZCL_READ_ATTRIBUTES_COMMAND_ID
zclReq.endpointId = APP_ENDPOINT_ESP; //Source application endpoint
zclReq.ZCL_Notify = ZCL_AttributeResp; //The callback function

//Setting destination address
zclReq.dstAddressing.addrMode    = APS_NO_ADDRESS; //Use binding
zclReq.dstAddressing.profileId   = PROFILE_ID_SMART_ENERGY;
zclReq.dstAddressing.clusterId   = SIMPLE_METERING_CLUSTER_ID;
zclReq.dstAddressing.clusterSide = ZCL_CLUSTER_SIDE_SERVER;

//Disable default response (won't be sent for commands with specific ZCL response)
//but can be enabled for requests w/o response (e.g. attribute reports)
zclReq.defaultResponse = ZCL_FRAME_CONTROL_DISABLE_DEFAULT_RESPONSE;

zclReq.requestLength   = …; // specify payload length
zclReq.requestPayload  = …; // point to payload memory

//execute the request
ZCL_AttributeReq(&zclReq);
```

Note: How to fill the `req.requestLength` and `req.requestPayload` fields is not shown in this example. BitCloud ZCL provides `ZCL_PutNextElement()` function that can be used to simplify filling and parsing the payload and calculating its length correctly. Section 5.5.2.1 and Section 5.5.2.2 provides separate instructions on how to set `requestLength` and `requestPayload` fields for read attribute and write attribute commands respectively.

Caution: The `ZCL_Request_t` instance used to send the request as well as the payload must not be re-used until a response for the command is received.

Sections 5.5.2, 5.5.3, 5.5.3.3, and 5.5.4 provide examples on how to use `ZCL_AttributeReq()` for read and write commands, attribute reporting and attribute discovery commands.

5.5.1.1 Specifying Destination Address

`ZCL_Request_t` type used as argument variable for ZCL command requests has `dstAddressing` structure of `ZCL_Addressing_t` type that specifies destination information.

First of all, for successful delivery of the ZCL command `profileID`, `clusterID` and `clusterSide` fields of `dstAddressing` shall be set correctly for the outgoing command.

`dstAddressing.addrMode`, `dstAddressing.addr` and `dstAddressing.endpointId` fields specify actual address information for the destination node.

`dstAddressing.addrMode` defines the addressing mode to be used for the request. It shall be set as follows:

- `APS_SHORT_ADDRESS` if unicast or broadcast short address set in `dstAddressing.addr.shortAddress` field shall be used as destination. Section 5.7.1 provides more information on broadcast transmissions
- `APS_GROUP_ADDRESS` if group address set in `dstAddressing.addr.groupAddress` field shall be used as destination. For more details on group transmissions see Sections 5.7.2
- `APS_EXTENDED_ADDRESS` if extended address set in `dstAddressing.addr.extAddress` field shall be used as destination. Note that for over the air transmission stack always relies on the short address so if `APS_EXTENDED_ADDRESS` mode is used then address discovery might need to take place as described in Section 5.5.1.2
- `APS_NO_ADDRESS` if binding table entries shall be used as destination. If this addressing mode is used, there is no need to specify addr and endpointId fields as they will be taken from the binding table. However it is not recommended to use this mode due to lack of status updates on delivery to multiple destinations. Instead application should retrieve target entries from the binding table and perform transmission to them using other addressing mode. More details on transmission to bound targets are described in Section 5.8.2

As described in Section 5.1.1 to perform data exchange in addition to destination address also destination endpoint shall be specified for ZCL commands. `dstAddressing.endpointId` shall be set to the target endpoint.

For example, the following code shows how to configure addressing via the short address (assuming `dstShortAddr` and `dstEndpointId` are set by an application):

```
ZCL_Request_t zclReq; // global scope
ShortAddr_t dstShortAddr;
Endpoint_t  dstEndpointId;

...
zclReq.dstAddressing.sequenceNumber    = ZCL_GetNextSeqNumber();
zclReq.dstAddressing.addrMode          = APS_SHORT_ADDRESS;
zclReq.dstAddressing.addr.shortAddress = dstShortAddr; //destination short address
zclReq.dstAddressing.endpointId        = dstEndpointId; // destination endpoint
zclReq.dstAddressing.profileId         = PROFILE_ID_SMART_ENERGY;
zclReq.dstAddressing.clusterId         = PRICE_CLUSTER_ID;
zclReq.dstAddressing.clusterSide       = ZCL_CLUSTER_SIDE_SERVER;
...
```

Note: ZCL automatically maintains sequence numbers for general cluster commands. So there is no need to set `sequenceNumber` field in the `dstAddressing` structure. For a cluster-specific command the application shall use the `ZCL_GetNextSeqNumber()` function to obtain the value for this field.

5.5.1.2 Automatic Short Address Resolution

The stack automatically resolves situations when extended (MAC or IEEE) address of the destination is given. For example, if the extended addressing mode is used to identify the destination, APS needs the short address of the destination device, too, to be able to send the frame over the air. If the short address corresponding to the provided extended address is not known, a ZDP request will be used to discover the short address.

Other cases when address resolving is applied include:

- Binding (see Section 5.8) is used to identify the destination, but the short address of one of the bound devices is not known
- A frame should be encrypted with a link key (see Section 6.2), but the short addressing mode is used, and the corresponding extended address (which is necessary to find a link key entry in the APS key-pair set) is not known
- The stack receives an encrypted frame from the sender which extended address is not known

The application can also use address resolving if necessary. In such case, it should call the `ZDO_ResolveAddrReq()` function, specifying the address that is known in the argument's `dstAddress` field and its type in the `dstAddrMode` field. The callback function shall be also specified. The application may also send ZDP requests itself: an IEEE address ZDP request or a network address ZDP request, to discover required addresses.

5.5.1.3 Processing Response Callback

The callback function specified in the `ZCL_Notify` field of the command's parameters is called to notify application on the request execution status.

The callback function should have the following signature (the name of the function is arbitrary):

```
void ZCL_AttributeResp(ZCL_Notify_t *ntfy);
```

Depending on the type of ZCL command this notification callback will be called at various events.

The `ntfy->status` field indicates the status of command's execution and is closely linked to the `id` field which specifies the type of the notification and has enumerator type `ZCL_NotifyId_t` with following values:

- `ZCL_APS_CONFIRM_ID`: to update on the status of request transmission and reception of APS Acknowledgement if one is required. For ZCL requests that do not require any ZCL response only this type of notifications will be returned in `ZCL_Notify` callback
- `ZCL_ZCL_RESPONSE_ID`: to indicate successful or non-successful reception of a specific ZCL response command if such is expected for executed ZCL request
- `ZCL_ZCL_DEFAULT_RESPONSE_ID`: to indicate successful or non-successful reception of ZCL default response command if such is required for executed ZCL request

Response commands for general cluster commands are sent automatically by the stack. The application's impact is limited to configuration of clusters and commands.

5.5.2 Read/write a Remote Attribute

To read or write an attribute on a remote device, the `ZCL_AttributeReq()` function is used (see Section 5.5.1). A single request can access values of several attributes from the same cluster.

The argument is of `ZCL_Request_t*` type. The id field of the argument shall be set to `ZCL_READ_ATTRIBUTES_COMMAND_ID` to read attributes' values and to `ZCL_WRITE_ATTRIBUTES_COMMAND_ID` to write attributes' values. The `requestPayload` and `requestLength` fields of the `ZCL_Request_t` instance shall be filled as described in following sections.

5.5.2.1 Payload for Read Attributes Command

To form the payload for the read attributes command use the `ZCL_PutNextElement()` function. With the help of this function attribute IDs are written one by one to the buffer in a correct way and the overall payload length is calculated. The buffer is then used as the payload for the read attribute command. See instructions below:

1. Define variables of `ZCL_NextElement_t` and `ZCL_ReadAttributeReq_t` types. For example:

```
ZCL_NextElement_t element;
ZCL_ReadAttributeReq_t readAttrReqElement;
```

2. Define a sufficiently large buffer (but note that it should include only attribute IDs). The buffer must be defined globally and may be a variable of the `uint8_t*` type.
3. Configure `element` in the following way:

```
element.payloadLength = 0;
element.payload = buffer;
element.id = ZCL_READ_ATTRIBUTES_COMMAND_ID;
element.content = &readAttrReqElement;
```

The buffer variable is assumed to be defined somewhere in the application to be used as the buffer.

4. For each requested attribute repeat the following steps:
   c. Set readAttrReqElement.id to the target attributes identifier.
   d. Call ZCL_PutNextElement(&element).
5. Set the requestLength field of the ZCL_Request_t instance to element.payloadLength.
6. Set the requestPayload field of the ZCL_Request_t instance in the request to buffer.

5.5.2.2 Payload for Write Attributes Command

Forming the payload for a write attributes command involves almost the same steps as for the read attributes command:

1. Define a variable of ZCL_NextElement_t type and a request element. The latter may have the ZCL_WriteAttributeReq_t type. For example:

```
ZCL_NextElement_t element;
ZCL_WriteAttributeReq_t writeAttrReqElement;
```

In this case the type field of the ZCL_WriteAttributeReq_t instance should be set to the constant specifying the attribute's data type; for example, ZCL_U64BIT_DATA_TYPE_ID. The data type for a given attribute is defined by the ZCL specification and should coincide with the data type assigned to the attribute on the destination devices.

The attribute's value should be written to the value field. However, this field in fact has the unit8_t[1] type. A one-byte value may be assigned to value[0]. But values occupying more than one byte in memory cannot be written directly. A possible workaround is to allocate sufficiently long buffer and cast it to the ZCL_WriteAttributeReq_t* type. Another solution is to define a custom structure containing subsequent ID, type, and value fields, with the value field of the actual attribute's size. For example, for 64-bit unsigned type the structure may be defined in the following way:

```
typedef struct PACK
{
  ZCL_AttributeId_t id;
  uint8_t type;
  uint64_t value;
} APP_Write64BitAttributeReq_t;
```

And the request element will be defined in this way:

```
APP_Write64BitAttributeReq_t writeAttrReqElement;
```

2. Define a sufficiently large buffer to hold all request elements for all attributes to be written. The buffer must be defined globally and may be a variable of the uint8_t* type.
3. Configure element in the following way:

```
element.payloadLength = 0;
element.payload = buffer;
element.id = ZCL_WRITE_ATTRIBUTES_COMMAND_ID;
element.content = &writeAttrReqElement;
```

The buffer variable is assumed to be defined somewhere in the application to be used as the buffer.

4. For each requested attribute repeat the following steps:
   a. Set readAttrReqElement.id to the target attribute's ID.
   b. Write the attribute's value to the writeAttrReqElement.value field. For a ZCL_WriteAttributeReq_t instance, cast this field to a pointer to the attribute's data type and dereference it, for example:

```
*(uint64_t*)writeAttrReqElement->value = VALUE;
```

   c. Call ZCL_PutNextElement(&element). Check the returned value: ZCL_END_PAYLOAD_REACHED_STATUS will be retuned when the last element fitting into the payload is written.

5. Set the `requestLength` field of the `ZCL_Request_t` instance to `element.payloadLength`.
6. Set the `requestPayload` field of the `ZCL_Request_t` instance in the request to `buffer`.

5.5.2.3 Parsing the response payload

In the callback function for a general cluster command's response, the application should first check the status of the command's execution (same as described for notification callback in Section 5.5.1.3). Parsing response payload is needed in case of:

- Successful execution of the read attributes command
- Error in execution of a write attributes command for several attributes at once (the payload will contain IDs of attributes for which writing operation has failed)

For the write attributes response the success status means that *all* attributes have been written successfully, while the error status means that *some* attributes have not been written. In the latter case the response payload contains a separate status report for each attribute (see the ZCL specification [7] [6]).

To process the response payload, the application should configure an instance of `ZCL_NextElement_t` type and use the `ZCL_GetNextElement()` function to extract subsequent attribute records:

7. Define a variable of `ZCL_NextElement_t` type and a variable of the `ZCL_ReadAttributeResp_t` type (for read attributes response) or of the `ZCL_WriteAttributeResp_t` type (for write attributes response). For example:

```
ZCL_NextElement_t element;
```

and

```
ZCL_ReadAttributeReq_t *temp = NULL;
```

or

```
ZCL_WriteAttributeReq_t *temp = NULL;
```

8. Configure element in the following way:

```
element.payloadLength = resp->responseLength;
element.payload = resp->responsePayload;
element.content = NULL;
```

For read attributes response:

```
element.id = ZCL_READ_ATTRIBUTES_RESPONSE_COMMAND_ID;
```

For write attributes response:

```
element.id = ZCL_WRITE_ATTRIBUTES_RESPONSE_COMMAND_ID;
```

9. For each requested attribute repeat the following steps:
   d. Call `ZCL_GetNextElement(&element)`.
   e. Cast `temp` to `ZCL_ReadAttributeResp_t*` for read attributes response and to `ZCL_WriteAttributeResp_t*` for write attribute response.
   f. Access the value as `*(<AttrType>*)value` or check the `id` and `status` fields (the `status` field describes the result of read attributes command).

For example:

```
ZCL_GetNextElement(&element);
temp = (ZCL_ReadAttributeResp_t *) element.content;
if (ZCL_SUCCESS_STATUS == temp->status)
{
  uint64_t value = *(uint64_t*)temp->value;
  //Process the received value
```

```
}
```

10. Check the value returned by `ZCL_GetNextElement()`. When the last element from the payload is read, the function returns `ZCL_END_PAYLOAD_REACHED_STATUS`.

5.5.3 Attribute Reporting

Reporting a cluster's attribute means delivering the value of a particular cluster attribute to the remote endpoint supporting this cluster. There is a special ZCL Attribute Report command used for that.

A cluster attribute may be reported automatically by the BitCloud based on configured intervals or level change as described in Sections 5.5.3.1 and 5.5.3.2. The application may also report the value of an attribute at any time via the `ZCL_AttributeReq()` function (see Section 5.5.3.3).

5.5.3.1 Automatic Attribute Reporting

Whether an attribute is automatically reportable or not is determined by the attribute declaration in the cluster header file. A reportable attribute is defined using the `DEFINE_REPORTABLE_ATTRIBUTE` macro which, in comparison to the macro for non-reportable attributes, has two additional arguments for minimum and maximum reporting intervals in seconds (see Section A.2.1).

To enable attribute reporting on the reporter side, configure the minimal and maximum reporting intervals by setting arguments of the macro that defines cluster's attributes and initiate automatic attribute reporting when expected by application via `ZCL_StartReporting(void)` function. Once this function is called then stack will start reporting of all reportable attributes per their configuration. If certain attribute changes its reportable status or reporting configuration at any time after `ZCL_StartReporting()` is called, new configuration will be automatically taken into account without any additional actions required from the application.

The minimum reporting interval puts the lower boundary on the interval in seconds between two reports of the attribute: if attribute reporting is requested earlier (e.g. due to the level change as described in Section 5.5.3.2) the stack will wait until the interval elapses and then will send the report. The maximum reporting interval specifies specifies the interval in seconds between two automatically sent attribute reports. If the maximum value is set to `0xFFFF` then no reports for the current attribute will be ever sent.

For example, the following code sets the minimum reporting interval to 4 seconds and maximum reporting interval to 5 seconds for the simple metering cluster attributes:

```
SimpleMeteringServerClusterAttributes_t meterAttributes =
{
  DEFINE_SIMPLE_METERING_SERVER_ATTRIBUTES(4, 5)
};
…
// single start for all reportable attributes
ZCL_StartReporting(void);
```

ZCL will send attribute report frames containing reporting values periodically to all devices bound to the cluster that holds this attribute. APS acknowledgement is not requested, and the application does not receive notification on the delivery status.

End devices with sleep-when-idle started (see Section 7.1) will automatically wake up to perform scheduled reporting. If sleep-when-idle is not used then stack will verify the need for periodic reporting only on the scheduled wake ups. Hence sleep-when-idle is strongly recommended for use on sleeping end device that require automatic attribute reporting.

5.5.3.2 Reportable Change

The application may configure additional attribute reporting by setting the reportable change value. If it is set for an attribute and the stack encounters that the attribute's value has changed, from the moment of the last report, by the value greater than the reportable change, an additional report is generated.

The last reported value and the reportable change are stored as fields of the attribute's structure of the same data type as the attribute's value.

To set a reportable change value the application should call the `ZCL_SetReportableChange()` function. The arguments include endpoint ID, cluster ID, cluster side and attribute ID to locate the attribute, and attribute data type and the reportable change to be set. The argument's type for the reportable change is `uint8_t*`, so this argument may point a value of any data type.

Each time when `ZCL_WriteAttibute(..)` function is used to change the attribute value, stack will automatically check the change level and send the report if needed. If the attribute value is changed directly via memory access then application can use `ZCL_ReportOnChangeIfNeeded(void *attr)` function to request stack to verify the change level and to send the report if needed. If attribute value is changed by a ZCL Write Attribute Request command received from a remote node, stack will automatically analyze the change level and perform attribute reporting on value change if needed.

5.5.3.3 Manual Attribute Reporting

The `ZCL_AttributeReq()` function should be used to report the attribute's value manually by sending the report attributes command. The `id` field of the argument shall be set to `ZCL_REPORT_ATTRIBUTES_COMMAND_ID`.

A node may report several attributes in a single report attributes command. The payload is filled using the `ZCL_PutNextElement()` function in a similar way as for the write attributes command (the command ID is different). IDs and values of the attributes are put to the payload one by one. See Section 5.5.2.2.

5.5.3.4 Receiving Attribute Reports

To handle attribute reporting on the recipient side, the reporting indication callback shall be set. This is usually done after the client clusters have been configured and before an endpoint is registered. To set a report indication callback, assign the callback function pointer to the `ZCL_ReportInd` field of the reporter cluster (which is supported as a client cluster on the recipient) before registering an endpoint.

For example, for ESP, considering the simple metering cluster is the first in the client clusters list, reporting indication is configured as follows:

```
espClientClusters[0].ZCL_ReportInd = ZCL_ReportInd;
```

The callback function is declared in the following way:

```
void ZCL_ReportInd(ZCL_Addressing_t *addressing,
                   uint8_t reportLength,
                   uint8_t *reportPayload)
{
  ... //Parse reportPayload to retrieve attribute value
}
```

The `addressing` field contains all information about the reporter including the device's address and the reporter cluster id. The `reportPayload` parameter points to the buffer containing received attributes' information (id, data type and value). The attributes reports are retrieved using the same mechanism as for reading attributes (see Section 5.5.2). That is, define an instance of the `ZCL_NextElement_t` type and configure it for parsing reporting payload in the callback function's body:

```
...
ZCL_NextElement_t element;
ZCL_Report_t *temp;

element.id = ZCL_REPORT_ATTRIBUTES_COMMAND_ID;
element.payloadLength = reportLength;
element.payload = reportPayload;
```

```
element.content = NULL;
```

After that get the information about the next reported attribute:

```
ZCL_GetNextElement(&element);
temp = (ZCL_Report_t *) element.content;
```

The `temp` variable is containing the attribute information including id, data type, and value.

5.5.4 Attribute Discovery

With the help of the attribute discovery command an application can find out what attributes are supported by a certain cluster on a remote node.

To send an attribute discovery command use the `ZCL_AttributeReq()` function (see Section 5.5.1) with appropriately configured argument:

- Specify `ZCL_DISCOVER_ATTRIBUTES_COMMAND_ID` as the command's ID
- Set `defaultResponse` to `ZCL_FRAME_CONTROL_ENABLE_DEFAULT_RESPONSE` if default response is needed (otherwise no ZCL frames will be sent in response)
- Configure the payload for the attribute request, using the `ZCL_PutNextElement()` function

5.5.4.1 Command Options

The attribute discovery command has two parameters. One parameter is the start attribute ID that defines the ID value to start the search from. Attributes with IDs less than the start attribute ID will not be included in the response. Another parameter specifies the maximum amount of attributes to be returned in the command's response.

The start attribute ID parameter is required, because only limited number of attributes may fit into the response frame. So not all attributes supported by a cluster may be discovered with a single request. In such case the device that initiates attribute discovery can send the first discovery command with start attribute ID equal to 0. Upon processing the response, it can check whether all attributes have been received or not and set the start attribute ID for the next command to the biggest attribute ID among the discovered attributes. The device may send subsequent attribute discovery commands until all attributes are received.

5.5.4.2 Forming a Command

The code examples below show how to configure an attribute discovery payload and command's parameters. In the example, the reference ESP application finds out what attributes are supported by the meter with a particular extended address.

First, define some global variables are defined:

```
static ZCL_Request_t discoverAttrReq;
static uint8_t simpleMeteringBuf[20];
static void ZCL_DiscoverAttributesResp (ZCL_Response_t *resp);
```

`simpleMeteringBuf` may be a shared buffer used for other generic commands as well.

Define some auxiliary local variables and prepare the payload for the attribute request:

```
ZCL_NextElement_t element;
ZCL_DiscoverAttributesReq_t discoverAttrReqElement;

element.payloadLength = 0;
element.payload = simpleMeteringBuf;
element.id = ZCL_DISCOVER_ATTRIBUTES_COMMAND_ID;
element.content = &simpleMeteringReqElement;
ZCL_PutNextElement(&element);
```

Note: The use of the instance of ZCL_NextElement_t type: provided it with a buffer, specify the type of a particular command that is going to be sent, and point it to an instance of ZCL_DiscoverAttributesReq_t type. The ZCL_PutNextElement() function copies the command's parameters and ID to the buffer in a correct way and sets the length of the obtained payload.

Configure command parameters:

```
discoverAttrReqElement.startAttributeId = 0;
discoverAttrReqElement.maxAttributeIds = 6;

discoverAttrReq.id = ZCL_DISCOVER_ATTRIBUTES_COMMAND_ID;
discoverAttrReq.ZCL_Notify = ZCL_DiscoverAttributesResp;
discoverAttrReq.endpointId = APP_ENDPOINT_ESP;
discoverAttrReq.dstAddressing.addrMode = APS_EXT_ADDRESS;
discoverAttrReq.dstAddressing.addr.extAddress = METER_EXT_ADDRESS;
discoverAttrReq.dstAddressing.endpointId = APP_ENDPOINT_METER;
discoverAttrReq.dstAddressing.profileId = PROFILE_ID_SMART_ENERGY;

//ID of the cluster for which attributes shall be discovered
discoverAttrReq.dstAddressing.clusterId = SIMPLE_METERING_CLUSTER_ID;
//The side of the cluster on the remote device
discoverAttrReq.dstAddressing.clusterSide = ZCL_CLUSTER_SIDE_SERVER;

discoverAttrReq.defaultResponse = ZCL_FRAME_CONTROL_ENABLE_DEFAULT_RESPONSE;
discoverAttrReq.requestLength = element.payloadLength;
discoverAttrReq.requestPayload = simpleMeteringBuf;
```

Alternatively, a short address can be used for addressing. Using binding is not recommended. The network may contain several devices that support the provided cluster, with different attributes supported by the cluster on each device. In this case, the first response received by the device will be propagated to the application, and so information contained in the frame may not be valid for all devices. The recommended approach to attribute discovery is sending unicast frames to specific addresses.

Since the configuration of the request has been completed, the command is sent by the following line:

```
ZCL_AttributeReq(&discoverAttrReq);
```

5.5.4.3 Processing the Response

The response callback function's pointer is assigned to the ZCL_Notify field of the request's parameters. The callback function is called with a success status when the response frame is received. If an error occurred during transmission of the frame from the initiator case the callback function is called with an error status. The callback function will also report an error if the receiving side fails to execute the request and the response with an error code is received.

Note: The application on the receiving side does not have to do anything on reception of general-cluster commands, which are fully processed by ZCL.

The application should use the ZCL_GetNextElement() function to parse the response payload. This function helps extract an instance of ZCL_DiscoverAttributesResp_t type, which is used to access directly the information about attributes through its attributeInfo field. Another field, discoveryComplete, indicates whether all attributes has been received.

The following code demonstrates how the callback function may be organized:

```
void ZCL_DiscoverAttributesResp(ZCL_Notify_t *ntfy)
{
```

```
  ZCL_NextElement_t element;
  ZCL_DiscoverAttributesResp_t *temp = NULL;
  uint8_t attrAmount = 0;

  if (ZCL_SUCCESS_STATUS == ntfy ->status)
  {
    element.id = ZCL_DISCOVER_ATTRIBUTES_RESPONSE_COMMAND_ID;
    element.payloadLength = ntfy->responseLength;
    element.payload = ntfy->responsePayload;
    element.content = NULL;

    attrAmount = (element.payloadLength - 1)/
                          sizeof(ZCL_DiscoverAttributesRespRec_t);

    ZCL_GetNextElement(&element);
    temp = (ZCL_DiscoverAttributesResp_t *) element.content;

    for (uint8_t i = 0; i < attrAmount; i++)
    {
//Here you can access the attribute's ID and type via
//temp-> attributeInfo[i], which is of ZCL_DiscoverAttributesRespRec_t //type
    }

    if (false == temp->discoveryComplete)
    {
      //...
    }
  }
  else
  {
    //...
  }
```

If `temp->discoveryComplete` is `false` the remote device has more attributes than included into this response. To get information about these attribute you should issue new attribute discovery command, setting the `startAttributeId` field of `discoverAttrReqElement` to the ID of the last attribute from the `attributeInfo` list plus 1.

5.6 Cluster-specific Commands

Applications that implement public profiles typically communicate by sending commands related to specific clusters. In this case a command is transferred between endpoints that support different sides of the same cluster (that is, from a server to a client or from a client to a server).

General cluster commands (for example, read/write/report/discover attributes) are sent via the `ZCL_AttributeReq()` function, as described in Section 5.5.1. Cluster-specific commands are sent via the `ZCL_CommandReq()`, which usage is described in the sections below.

A profile application can still send data using the `APS_DataReq()` function, as described in Section C.4. However, sending a ZCL command via this function is more complicated then when the ZCL API is used, since the frame is not processed by the ZCL component. Note that a frame with application data that is not a standard ZCL command should not be sent to a ZCL endpoint as it will ignore the frame. For such data, a destination endpoint must be registered via the `APS_RegisterEndpoint()` function.

5.6.1 Sending a Cluster-specific Command

To send a cluster specific command the `ZCL_CommandReq()` function is used with the pointer to an instance of the `ZCL_Request_t` type as an argument that is used very similar as in `ZCL_AttributeReq()` function for general cluster commands described in Section 5.5.1.

Argument's fields include command ID, source endpoint, destination addressing information, command payload and command payload's length (see Section 5.6.2), pointers to a confirmation callback function (see Section 5.6.3), and a field configuring default response.

Note that for a cluster-specific command the application shall use the `ZCL_GetNextSeqNumber()` function to obtain ZCL sequence numbers set it to the `sequenceNumber` field in the `dstAddressing` structure of the argument of `ZCL_CommandReq()`.

If a cluster-specific command requires specific ZCL response command per its specification in cluster definition file (see Section A.3), then `ZCL_Notify` callback for the ZCL request will be triggered only after such response command is received, or until timeout for it reception expires. Immediately after such `ZCL_Notify` the callback function for the ZCL response command as specified for supported cluster commands (see Section 5.3.3) will be called as well.

5.6.2 Format the Command's Payload

For most of the cluster-specific commands the command's payload is set to the pointer to an instance of a special type defined for each command in the cluster header file. The payload length can be calculated with the help of the `sizeof` operator applied to this type.

For example, the payload for a get current price command may be configured as follows:

```
// Allocating global variables for the request and the payload
ZCL_GetCurrentPrice_t getCurrentPricePayload;
ZCL_Request_t getCurrentPriceReq;
...

// Filling payload's fields
getCurrentPricePayload.requestorRxOnWhenIdle = 1;

// Filling request's fields
getCurrentPriceReq.id = GET_CURRENT_PRICE_COMMAND_ID;
getCurrentPriceReq.ZCL_Notify = getCurrentPriceResponse;
... // and others

// Filling request's payload fields
getCurrentPriceReq.requestPayload =
                              (uint8_t*)& getCurrentPricePayload;
getCurrentPriceReq.requestLength = sizeof(ZCL_GetCurrentPrice_t);
...
//Send the configured request
ZCL_CommandReq(&getCurrentPriceReq);
```

Note: The pointer to the payload variable is cast to the `unit8_t*` type. The C-type for payload is `getCurrentPricePayload`. It is defined in the header file for the price cluster, `zclPriceCluster.h`.

After sending a ZCL command via the `ZCL_CommandReq()` function, the application may receive several types of notifications via registered callback functions:

- The callback function assigned to the `ZCL_Notify` field of request parameters indicates whether the command has been sent successfully and

- If the transmitted command implies a specific response, the indication callback registered in the cluster for this response command is called when the response is received

Caution: The `ZCL_Request_t` instance used to send the request as well as the payload must not be re-used until a response for the command is received.

Note: APS acknowledgements for a command are enabled or disabled in the macro that defines the command in the cluster header file in the `COMMAND_OPTION` macro's invocation. See Step 5b in Section A.3.

5.6.3 Manufacturer-specific Commands

Manufacturer-specific commands, which are commands not defined in the ZCL specification, are defined in the cluster's header file in the same way as a standard cluster-specific command (see Section A.3).

The application can a manufacturer-specific command, using the `ZCL_CommandReq()` function. For this purpose, the destination addressing structure includes the `manufacturerSpecCode` field, which shall be filled by the application. ZCL does not do anything special with such commands, but only adds the `manufacturerSpecCode's` value to the command's frame if the value is not zero. A command's destination node can access this value in the addressing structure passed to the argument of the indication callback function. ZCL will not reject the frame, and it is up to the application to decide further actions with the received command.

5.7 Broadcast and Multicast Transmissions

In addition to unicast (node-to-node) transmissions, 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 broadcast messages is not guaranteed, it shall not be applied for 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 explicitly aware of other members of the same group. On the network level, multicasting is based on broadcasting.

Note: An end device sends its own data including broadcast and multicast to the nodes in the network only via its parent.

5.7.1 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. Whether broadcast ZCL request will be responded to with a ZCL response depends on a particular cluster.

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.

5.7.1.1 Broadcast Implementation

On the network level, the broadcast procedure is performed as follows: after the API function to send a broadcast request is called, the node sends the frame up to three times over the air. Each router node in the network, 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 up to 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.

5.7.1.2 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.

5.7.2 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.

5.7.2.1 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.

ZCL group cluster can be used to add and remove a remote device to a group as well as to check the group membership and obtain other group-related information.

5.7.2.2 Sending a Data Frame to a Group

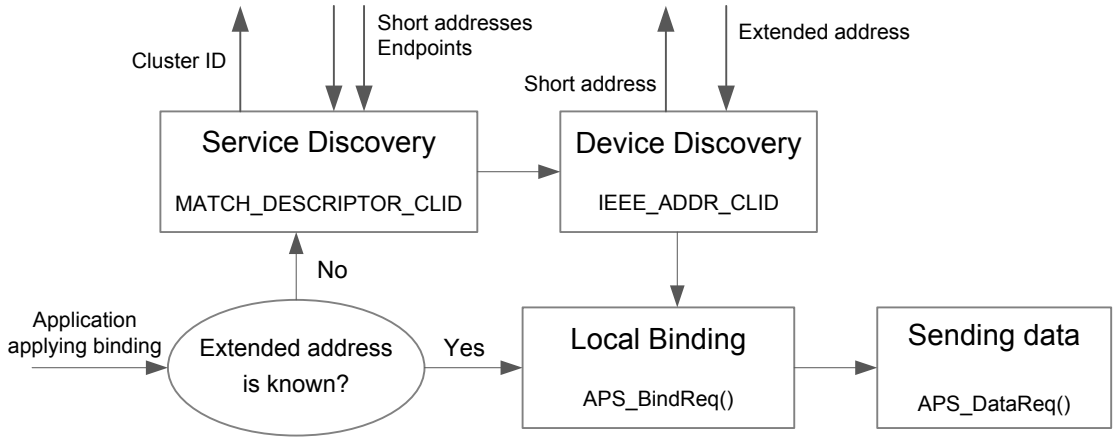
As described in Section 5.5.1.1 issuing a message to a group is done by executing data transmission requests with the addressing mode set to `APS_GROUP_ADDRESS` and the destination address field assigned the group address (specified by the members of the group in `APS_AddGroupReq()` parameters). The destination endpoint shall be set to `0xFF`.

5.8 Cluster Binding

Cluster 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 BitCloud stack keeps entries for each binding in the local 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 5-5. Applying Cluster Binding



To bind to a cluster, the application must know the extended address of the target device and its 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 (see Section 4.3.2). But since binding requires extended addresses, the application should perform also address discovery operation to set up a correspondence between extended and short addresses (see Section 4.3.3.1). Both service and address discovery are carried out with the help of ZDP requests.

Once the information about target has been collected, the application can bind to the devices by inserting corresponding entries into the binding table (see Section 5.8.1). It is also possible to request a remote device to add an entry to its binding table via a ZDP request. 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`.

After all preparation is completed, the data can be transmitted to all nodes that support a specific cluster. Figure 5-5 illustrates the described process and Table 5-2 summarizes the ZDP requests concerned with both service discovery, device discovery and binding operations performed during binding.

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.

Table 5-2. 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. See Section 4.3.3.2.
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 4.3.3.1.
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 4.3.2.
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. Optional to use.
UNBIND_CLID	Requests a remote node to remove an entry from its binding table with the specified source and endpoint addresses. Optional to use.

For instructions and examples of how to issue a particular ZDP request, refer to Section 4.3. More examples can be observed in the sample applications provided with the SDK. For a complete list of ZDP requests, refer to the BitCloud API Reference [3].

5.8.1 Implementing Local Binding

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 5.8.1.1). 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 to execute local binding
```

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.

5.8.1.1 Local 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
```

Note that group members that do not support a cluster specified within the request will decline the message.

5.8.2 Sending Data to a Bound Target

After cluster binding has been established, the application can start sending cluster commands to the bound destination(s). Although it is possible to use the address mode `APS_NO_ADDRESS` for that as described in Section 5.5.1.1, it is recommended that application retrieves the destination information from the binding table and performs command transmission using `APS_SHORT_ADDRESS`, `APS_EXT_ADDRESS` or `APS_GROUP_ADDRESS` modes. This will allow proper status tracking of frame transmission to each of the bound destinations.

Obtaining the values in the binding table are possible with `APS_NextBindingEntry(ApsBindingEntry_t *entry)` function that returns the next valid entry in the binding table. As a starting point `entry` argument shall be set to `NULL`. While browsing through the binding table application needs to compare the clusterId and source endpoint to find the destination to be used. Here is an example on how this can be done:

```
ApsBindingEntry_t *bindingEntry = NULL;
Endpoint_t srcEndpointId;

// loop through the binding table
while (NULL != (bindingEntry = APS_NextBindingEntry(bindingEntry)))
{
  // skip entries with a different source endpoint
  if (bindingEntry->srcEndpoint != srcEndpointId)
```

```
        continue;
    //skip entries with a different cluster bound
    if (bindingEntry->clusterId != clusterId)
        continue;

    //access destination information
    // bindingEntry->dstAddrMode - for destination addressing mode
    // group address for group addressing mode:
    if (APS_GROUP_ADDRESS == bindingEntry->dstAddrMode)
       // bindingEntry->dst.group – group address if
    else
    {
      // unicast extended address and endpoint ID
      // bindingEntry->dst.unicast.extAddr
      // bindingEntry->dst.unicast.endpoint
    }
}
```

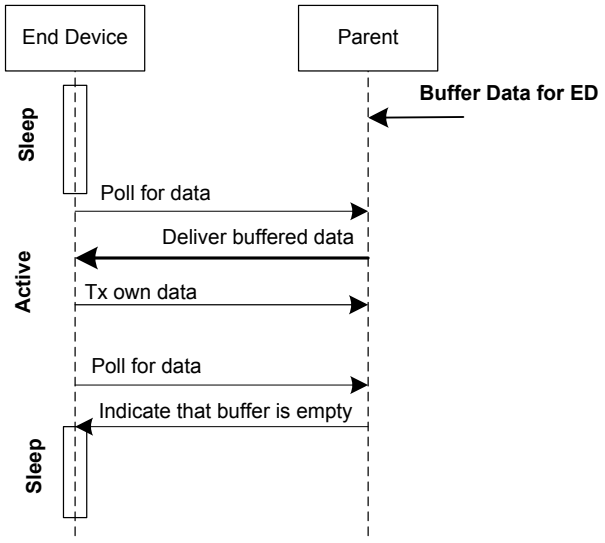
Note that binding table doesn't contain destination short address and hence address discovery might be triggered as described in Section 5.5.1.2.

5.9 Parent Polling Mechanism

In ZigBee networks, an indirect data delivery mechanism is used to send data to sleeping end devices over the last hop (that is, from a parent node to its end device child node). The parent does not transfer data destined to the end device child directly, but buffers the data frames instead. An end device sends a data poll request to its parent and if the buffer contains messages for the child, the parent then delivers the buffered data right after such data poll requests.

Figure 5-6 illustrates parent polling mechanism and retrieval of the buffered frame from the parent.

Figure 5-6. End Device Polling Mechanism



5.9.1 Parent Polling Configuration on End Devices

Polling is handled by the lower stack layers but application can control it with the help of certain ConfigServer parameters and requests to the stack.

To use parent polling, an end device shall have CS_RX_ON_WHEN_IDLE set to false (see the example in Section 4.1.1.2), indicating that the radio is kept in listening mode only for short periods of time when the device is actually sending and receiving data frames and hence data polling mechanism will be used for frame reception. After an end device enters a network, it cannot change CS_RX_ON_WHEN_IDLE as a usual parameter via the ConfigServer interface.

If sleep-when-idle feature (see Section 7.1) is started then an end device under normal conditions after network join will poll its parent with interval configured in CS_END_DEVICE_SLEEP_PERIOD parameter. Such polling mode can be called *"slow polling"* as normally these data polls are meant as notification of end device presence to the parent and hence CS_END_DEVICE_SLEEP_PERIOD can have larger values from several seconds to several hours.

However during ongoing data transactions when stack expects a response for pending request (for example an APS Ack or a ZDO/ZCL response) stack will automatically reduce the polling interval and send them every CS_INDIRECT_POLL_RATE until expected response is received. This mode can be considered as "*fast polling*" as CS_INDIRECT_POLL_RATE is normally kept rather small (100-1000ms) to ensure fast retrieval of expected responses from the parent.

In addition to automatic control from the stack, application can request the *"fast polling"* mode as well by subscribing to BC_EVENT_POLL_REQUEST event and returning 1 in the data argument if it wants the next poll to occur in CS_INDIRECT_POLL_RATE ms. It doesn't need to change the data argument if there is no need for a fast poll from application perspective. Here is a code example how this can be done.

```
//define event callback function, event receiver and a status flag
static void isPollRequired(SYS_EventId_t eventId, SYS_EventData_t data);
static SYS_EventReceiver_t appPollRequestEventReceiver = { .func = isPollRequired};
static bool fastPollRequiredFlag; // 1 means is poll is requried and 0 means is not
…
// subscribe to the event
SYS_SubscribeToEvent(BC_EVENT_POLL_REQUEST, &appPollRequestEventReceiver);
…
//event callback handler
static void isPollRequired(SYS_EventId_t eventId, SYS_EventData_t data)
{
  bool *check = (bool *)data;
  if (BC_EVENT_POLL_REQUEST == eventId)
    *check |= fastPollRequiredFlag;
}
```

Caution: Same as for any other BitCloud events same memory is used for the data argument by all BC_EVENT_POLL_REQUEST event receivers. Hence bitwise or operation (|) is be used in the event callback handler to ensure that if a single component requires a fast poll in CS_INDIRECT_POLL_RATE interval then the data value stays as true independent on values set in other event callback functions.

With sleep-when-idle enabled there's generally no need to change CS_INDIRECT_POLL_RATE and CS_END_DEVICE_SLEEP_PERIOD parameters over time unless special polling behavior is needed for certain periods.

When sleep-when-idle feature is not used (see Section 7.2) then stack will send data polls with CS_INDIRECT_POLL_RATE interval when device is awake.

An end device application also can 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. Additionally an application can issue data poll requests manually with the NWK_SyncReq() function.

5.9.2 Polling Configuration on fully functional Devices

Special care should be taken when configuring fully functional devices (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 hence 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 configured on the parent, then messages for such a child may be dropped.

6. Network and Data Security

Security considerations are of high importance for wireless networks in many applications. In ZigBee networks, security serves three main purposes:

1. Authentication: to disallow network join for devices that cannot provide appropriate credentials.
2. Data privacy: to protect message from reading by non- authorized nodes.
3. Integrity control: to protect data frames against modifications.

BitCloud SDK help secure network operation by providing flexible security configuration.

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.

6.1 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 6-1.

Table 6-1. 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 may use a link key specific for them to encrypt application payload of a frame destined to the peer device.

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.

6.1.1 Selecting Security Mode

To enable required security mode, the application must be compiled with the correct BitCloud library. The choice of a particular library can be specified using by selecting corresponding configuration from IDE (Atmel Studio or IAR Embedded Workbench®) or by specifying 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_Atmega256rfr2_Atmega256rfr2_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.

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

For ZCL frames encryption level configurable per cluster side and is done in corresponding cluster header file. Consider following example that sets security level for On/Off client cluster in `zclOnOffCluster.h` file to network layer encryption only (Section 6.2.2):

```
#define ONOFF_CLUSTER_ZCL_CLIENT_CLUSTER_TYPE(clattributes, clcommands) \
{ \
   .id               = ONOFF_CLUSTER_ID, \
   .options          = { \
           .type     = ZCL_CLIENT_CLUSTER_TYPE, \
```

```
            .security = ZCL_NETWORK_KEY_CLUSTER_SECURITY, \
                        }, \
    .attributesAmount = ZCL_ONOFF_CLUSTER_CLIENT_ATTRIBUTES_AMOUNT, \
    .attributes       = (uint8_t *)clattributes, \
    .commandsAmount   = ZCL_ONOFF_CLUSTER_COMMANDS_AMOUNT, \
    .commands         = (uint8_t *)clcommands \
}
```

If encryption on the APS layer (See Section 6.3.3) is required as well then `.options.security` field shall be set to `ZCL_APPLICATION_LINK_KEY_CLUSTER_SECURITY`.

6.1.1.1 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.

6.1.2 Trust Center

Enabling security introduces an additional network role called the trust center. The trust center is responsible for device authentication, the distribution and management of encryption keys, and other security parameters.

ZigBee network can be operated in a distributed and centralized trust center mode. Centralized trust center network has a single device functioning as the network trust center. While in a network operating in distributed trust center mode all router devices play the trust center role for authenticating joining devices.

In BitCloud `CS_APS_TRUST_CENTER_ADDRESS` parameter is used to set the extended address of the trust center.

In a centralized trust center operation actual trust center device shall set this parameter to the same value as own `CS_UID`. For cases when a joining device has to be bound to join to a particular trust center device it shall set `CS_APS_TRUST_CENTER_ADDRESS` parameter to the extended address of the target trust center. In cases when target trust center is unknown the `CS_APS_TRUST_CENTER_ADDRESS` shall be assigned to the special universal extended address, `APS_SM_UNIVERSAL_TRUST_CENTER_EXT_ADDRESS`. Upon join and authentication to a network with centralized trust center stack will automatically overwrite `CS_APS_TRUST_CENTER_ADDRESS` parameter with a value that corresponds to the actual extended address of network trust center.

To operate in a distributed trust center mode all nodes in a network shall set `CS_APS_TRUST_CENTER_ADDRESS` to `APS_SM_UNIVERSAL_TRUST_CENTER_EXT_ADDRESS`.

6.1.3 Security Status

Another parameter to be taken into consideration for security configuration is `CS_ZDO_SECURITY_STATUS`. As shown in Table 6-2, 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 6-2. `CS_ZDO_SECURITY_STATUS` Parameter Values

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. Network key is not predefined on joining devices and is delivered during authentication process. Encryption on APS layer is performed with link key set for a particular pair of devices.
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).

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.

6.2 Standard Security Mode

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.

Trust center can update the network key as described in Section 6.4.

6.2.1 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 APS 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 `APS_SM_UNIVERSAL_TRUST_CENTER_EXT_ADDRESS (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.

6.2.2 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 6-1. ZigBee Frame with Standard Security

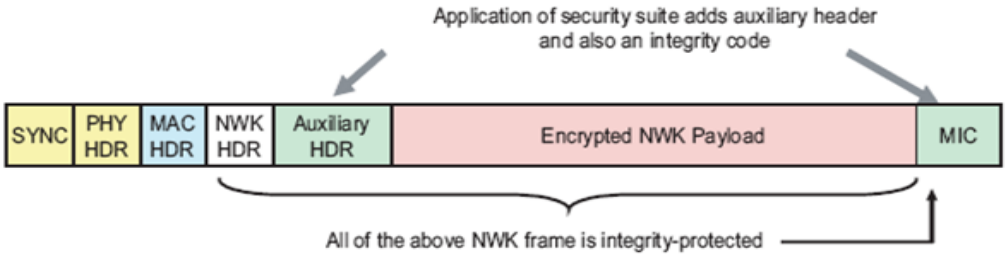


Figure 6-1 illustrates the structure of a frame transmitted to the network. In addition to the encrypted NWK payload, an auxiliary header and an integrity code are inserted into the frame. The integrity code is a hash value for headers and the payload and protects them from being changed.

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.

6.3 Standard Security with Link Keys

Standard security mode with link keys support extends the standard link security mode with a possibility to use link keys for authentication and additional encryption on the APS level. However it also relies on the network layer encryption using network key as described in Section 6.2.2. Initialization and update of network keys is described in Section 6.4.

When standard security with link keys is used, the `CS_ZDO_SECURITY_STATUS` parameter shall be set to `1`. As described in Table 6-2, using a value of `1` for the parameter requires a link key to be predefined on devices for communication with the network 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 table.

6.3.1 Network Join with a Predefined Trust Center Link Key

Authentication of a device attempting to join the network with security status set to 1 is described below.

In networks with centralized trust center a parent device that is used as network entry point, automatically notifies the trust center device about joining device with a special APS update request command. The trust center then should make a decision on whether to allow the device to enter the network or not.

In a distributed trust center network there is no APS update request command sent to a single trust center node. Instead authentication decision takes place on the parent only.

To be able to authenticate joining devices, the trust center has to know extended addresses and corresponding link keys for the devices. They should be inserted into the link key-pair table. This can be done either directly via `APS_SetLinkKey()`. If extended addresses of the joining device cannot be known in advance then global link keys can be defined on a trust center as described in Section 6.3.2.

The joining 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.

If a joining device is allowed to enter the network then either the network trust center device (in networks with centralized trust center) or the parent itself (networks with distributed trust center) sends the APS Transport key command that contains current NWK key and has payload encrypted with the trust center link key. Thus unlike with standard security mode, 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 APS transport key message it verifies whether it shall continue with the join to this network. First of all it tries to decrypt the message with own predefined trust center link key. If it succeeds then it compares 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. If joining device has `CS_APS_TRUST_CENTER_ADDRESS` set to `APS_SM_UNIVERSAL_TRUST_CENTER_EXT_ADDRESS (0xFFFFFFFFFFFFFFFF)` the device will accept any trust center device and then will set the `CS_APS_TRUST_CENTER_ADDRESS` parameter to the received trust center extended address.

The application on the joining device shall also 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. This can be done with `APS_SetLinkKey()` function, passing to it the trust center extended address and the link key value. If the extended address of the trust center is not known in advance or distributed trust center is used then `CS_APS_TRUST_CENTER_ADDRESS` shall be set to universal TC address as described in Section 6.1.2.

How to set a trust center link key on a non-trust center device is highlighted by the following example:

```
//Set a link key to a constant defined by the application
uint8_t linkKey[SECURITY_KEY_SIZE] = LINK_KEY;
ExtAddr_t tcAddr;
CS_ReadParameter(CS_APS_TRUST_CENTER_ADDRESS_ID, & tcAddr);
APS_SetLinkKey(&tcAddr, linkKey);
```

6.3.2 Global Link Keys

Global link keys are the link keys added to the APS key-pair set table and 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 searches an entry with this extended address in the APS key-pair set table. 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).

6.3.3 APS Data Encryption with Link Key

In addition to encrypting the NWK payload with a network key, standard security with link keys provides a possibility for encryption of the application payload with a link key, which can be uniquely defined for each pair of communicating nodes.

Figure 6-2. ZigBee Frame with Extended Standard Security

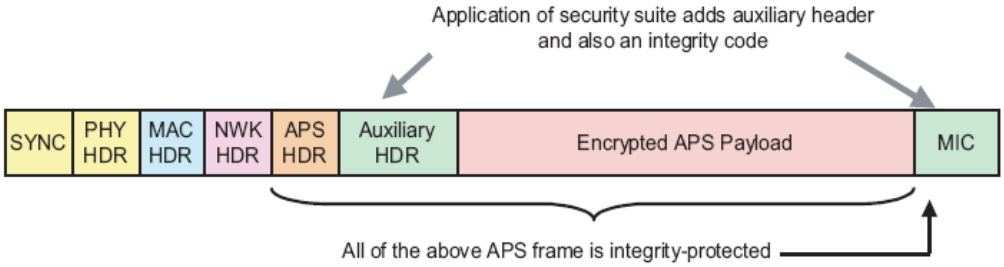


Figure 6-2 illustrates how a frame is formatted when the standard security with link keys mode is applied. First, the APS payload, which is set by the application in the data request, is encrypted with the use of the link key. An auxiliary header is inserted after the APS header. After that, the entire NWK payload is encrypted with the use of the network key, as shown on Figure 6-1. 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.

6.3.3.1 Requesting the Link Key for Communication with a Remote Device

When standard security with link keys is enabled, communication between any two nodes in the network can be performed with encryption of APS payload using a link key established for such pair. 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 Table 5-2). 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
  }
}
```

6.4 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.

6.4.1 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. A key can be activated as the main one for use via `NWK_ActivateKey()`.

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.

6.4.2 Keys Distribution

The trust center node 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.

6.4.2.1 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.

7. Power Management

In ZigBee networks, power consumption is often a major concern as many device types such as remote controllers, wall sensors, switches and others are expected to be battery-powered.

Independently of its networking status (joined to a network or not), a device can be either in active mode or sleep mode. Based on ZigBee PRO spec, only ZigBee end devices can be put into sleep mode because indirect data delivery via buffering on a parent is defined for end devices only (see Section 5.9).

After being powered up, a node always starts in active mode, with its MCU fully turned on.

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

This section describes mechanisms implemented in BitCloud for power management.

7.1 Sleep-when-idle

The simplest and the most efficient way to control switching between sleep and active modes is using sleep-when-idle feature implemented in BitCloud.

With this feature BitCloud task scheduler will automatically attempt to put an end device into sleep every time when it has no pending tasks to do (see Section 7.1.1). Stack will also automatically wake up the device to process callbacks for any running timers as described in Section 9.7.

By default sleep-when-idle mode is not started and MCU stays in active state after power up. By calling `SYS_EnableSleepWhenIdle(void)` function application will immediately turn on sleep-when-idle operation mode. To stop it completely `SYS_DisableSleepWhenIdle(void)` shall be called. Both functions can be used at any time.

7.1.1 Going into Sleep

If sleep-when-idle mode is running then BitCloud stack will automatically put a device into sleep when there are no pending tasks within the stack or application. Such check is performed in following steps:

- The first step checks for any posted tasks in the task manager.
  This is verified during execution of `SYS_RunTask()` in the infinite loop in `main()` function (see Section 2.2.1 for more information on task manager operation). If there any posted tasks the stack will keep the device in active mode. As soon as all posted tasks are handled the stack will proceed to step 2.
- During the second step the stack checks whether BitCloud components and the application are actually ready to go into sleep mode.
  The stack does this by posting a special `BC_EVENT_BUSY_REQUEST` event and collecting responses. Depending on their states and ongoing transactions the stack components and application may not allow going into sleep mode (for example if busy with RF communication or doing some encryption/decryption work, etc.) even when there is no actual task posted for immediate handling.
  If an application wants to have additional control on when the device is put into sleep it shall subscribe to the `BC_EVENT_BUSY_REQUEST` event and set the value of the data argument accordingly. Here is a code example on how this can be done:

```
//define event callback function, event receiver and a status flag
static void isAppReadyToSleep(SYS_EventId_t eventId, SYS_EventData_t data);
static SYS_EventReceiver_t appBusyEventReceiver = { .func = isAppReadyToSleep};
static bool readyToSleepFlag; // 1 means is ready to sleep and 0 means is not
…
// subscribe to the event
SYS_SubscribeToEvent(BC_EVENT_BUSY_REQUEST, &appBusyEventReceiver);
…
//event callback handler
```

```
static void isAppReadyToSleep(SYS_EventId_t eventId, SYS_EventData_t data)
{
  bool *check = (bool *)data;
  if (BC_EVENT_BUSY_REQUEST == eventId)
    *check &= readyToSleepFlag;
}
```

Caution: Same as for any other BitCloud events same memory is used for the `data` argument by all `BC_EVENT_BUSY_REQUEST` event receivers. Hence bitwise and operation (`&`) shall be used in event callback handler to ensure that if a single component is not ready for sleep then the `data` value stays as `false` independent on values set in other event callback functions.

- Finally during the third step the stack components, MCU and RF transceiver are put into sleep mode.
  Right before going into sleep mode stack calculates sleep duration. The sleep time is set as the time interval left until expiration of the next HAL timer. If there're no running timers then sleep interval will be set to the value of `CS_END_DEVICE_SLEEP_PERIOD` parameter. If this parameter is set to `0` then MCU is put into power down mode and will wake up only on external interrupt.

7.1.2 Wake up Procedure

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

7.1.2.1 Wake up on a Scheduled Time

In the scheduled approach, a node wakes up automatically after specific time interval. As described in Section 7.1.1 BitCloud stack automatically calculates sleep interval based on the next expiring timer started with `HAL_StartAppTimer()` function inside the stack or application. On end devices the BitCloud stack uses timers in a very limited cases, mainly for tracking time of over-the-air transactions that require some response, for periodic parent polling (Section 5.9) and for automatic ZCL attribute reporting (Section 5.5.3.1).

For the application there is no visible difference between cases when a HAL timer expires while device is in sleep state or when it expires during awake state. BitCloud stack will execute the function registered as callback for corresponding timer and no additional steps are expected from the application. The stack is fully operation and ready to perform data exchange if needed.

The application is notified about the scheduled wake up via the `ZDO_WakeUpInd()` function that is executed prior to the timer callback function that has triggered device wake up. There is no need to perform any additional actions in `ZDO_WakeUpInd()` to wake up the stack. It is called for notification purpose as the callback function for the timer might be present inside the stack.

After handling timer callback and related activity the stack will automatically attempt to put device back into sleep following the common procedure described in Section 7.1.1.

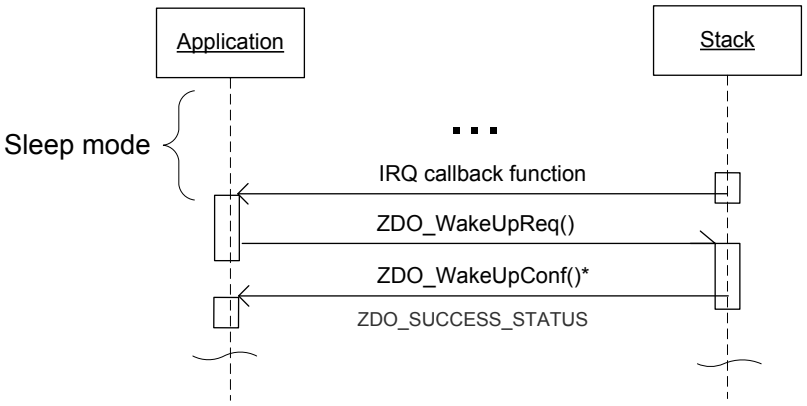
While the device is sleeping, the MCU is found in the power save mode, consuming much less power than in the idle state.

7.1.2.2 Wake up on an External Hardware Interrupt

In the IRQ triggered approach, the MCU is switched to active mode upon a registered IRQ event (see Section 9.6 for more details on IRQ handling in BitCloud).

Wake up on an external hardware interrupt is handled in a different way comparing to a wake up on scheduled time described in Section 7.1.2.1, where BitCloud stack is fully operational when a timer callback is called. The registered external interrupt callback function is executed as part of the interrupt service routine (ISR) and the networking components of the stack are not automatically awaked by that. In order to bring the whole stack back to active operation, the application must call the `ZDO_WakeUpReq()` function. After the `ZDO_WakeUpConf` callback registered for this request returns `ZDO_SUCCESS_STATUS`, the stack, the RF chip, and the MCU are fully awake and ready for data exchange. Figure 7-1 shows the control flow for an IRQ triggered wake up.

Figure 7-1. IRQ Triggered Wake-up



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

Here is a code example how this can be done.

```
//define event callback function and event receiver
static void wakeUpEventHandler (SYS_EventId_t eventId, SYS_EventData_t data);
static SYS_EventReceiver_t wakeUpEventListener = {.func = wakeUpEventHandler};

//define stack wake up request and callback function
static ZDO_WakeUpReq_t zdoWakeUpReq;
static void ZDO_WakeUpConf(ZDO_WakeUpConf_t *conf);
…
// subscribe to the event
SYS_SubscribeToEvent(HAL_EVENT_WAKING_UP, & wakeUpEventListener);
…
//event callback handler
static void wakeUpEventHandler (SYS_EventId_t eventId, SYS_EventData_t data)
{
  // wake up the stack if device is awaked on the external interrupt
  if (HAL_WAKEUP_SOURCE_EXT_IRQ == *(HAL_WakeUpSource_t *)data)
  {
    zdoWakeUpReq.ZDO_WakeUpConf = ZDO_WakeUpConf;
    ZDO_WakeUpReq(&zdoWakeUpReq);
  }
}
…
static void ZDO_WakeUpConf(ZDO_WakeUpConf_t *conf)
{
  if(ZDO_SUCCESS_STATUS == conf->status)
…
}
```

An IRQ callback must not perform any long operations, such as issuing asynchronous requests. It is also not recommended to call other API functions, since the stack is not fully awakened at that moment.

7.2 Sleep Control without Sleep-when-idle

It is possible for the application to fully control the sleep behavior on the device. In such case sleep-when-idle modes shall be kept disabled (in default or using `SYS_DisableSleepWhenIdle(void)` function).

In order to put a device into sleep mode when sleep-when-idle is disabled, an application shall call 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 device will be put into sleep for `CS_END_DEVICE_SLEEP_PERIOD` interval. After this interval expires the device will wake up and indicate this to the application via `ZDO_WakeUpInd()` function. All application timers that have expired during the sleep will be triggered upon wake up. If `CS_END_DEVICE_SLEEP_PERIOD` is set to `0` then the MCU will be put into power down mode and can wake up on external interrupt only.

Wake up on external interrupt is handled in the same way as when sleep-when-idle mode is enabled (see Section 7.1.2.2).

7.3 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 5.9.2. 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.

8. Persistent Data

The Persistent Data Server (PDS) component of the BitCloud stack implements interfaces and functionality for storing and restoring data in a non-volatile (NV) memory storage. Section 8.1 describes how to configure the PDS scheme.

In PDS particular pieces of persistent data are called files and groups of parameters are called directories. Section 8.2 describes how to define such persistent items, while Section 8.3 gives an overview of PDS API functions that can be used to store and restore them.

Finally Section 8.4 explains how application can maintain internal stack data in PDS. Section 8.5 provides an example how to create own persistent item for application data.

8.1 PDS Configuration

Wear-leveling PDS implemented in `BitCloud/Components/PersistentDataServer/wl/` supports only internal flash as non-volatile memory. The main feature behind the wear leveling PDS is the mechanism designed to extend the lifetime of the NV storage as well as to protect data from being lost when a reset occurs during writing to the NV. This mechanism is based on writing data evenly through the dedicated area, so that the storage's lifetime in not limited by the number of reading and writing operations performed with more frequently used parameters. For this purpose, the non-volatile storage organized as a cyclic log with new versions of data being written at the end of the log, not in place where the previous versions of the same data are stored.

To use PDS with wear leveling, the `PDS_ENABLE_WEAR_LEVELING` label shall be set to `1` in application's `configuration.h` file, and configure other PDS parameters as follows:

- Define `PDS_NO_BOOTLOADER_SUPPORT` if application will run as standalone without AVR2054 bootloader [4]. If Serial/OTA bootloader is used with the application then this define shall be not present as Flash access functions from the bootloader area will be used
- `PERSISTENT_NV_ITEMS_PLATFORM` is normally set to `NWK_SECURITY_COUNTERS_MEM_ID` to ensure that network security counters are not erased on network rejoins and factory new resets
- `PERSISTENT_NV_ITEMS_APPLICATION` is normally set to `0xFFFu` to indicate that application will handle its data in PDS by itself (see Section 8.5)
- If amount of application data files and directories exceeds default values then redefine `APPLICATION_MAX_FILES_AMOUNT` and `APPLICATION_MAX_DIRECTORIES_AMOUNT` parameters with the new ones. See Section 8.5
- Make sure the PDS area reserved for wear-leveling is defined in the application linker script via `D_NV_MEMORY_START` and `D_NV_MEMORY_SIZE` parameters

8.2 Defining Files and Directories

In PDS particular pieces of persistent data are called files (or items), and groups of files are called directories. Note that directories are just the way to refer to particular files, and a file can belong to several directories at once. Files and directories contain the meta-information about the data that allows its maintenance within the NV – file descriptors and directory descriptors.

The PDS component defines a number of file units for individual stack parameters and directories to group files, for more subtle control (see Section 8.4). The application may define its own items (see Section 8.5).

8.2.1 File Descriptors

A file descriptor consists of the following parts:

- `memoryId`: memory identifier associated with a file
- `size`: the size of file's data

- ramAddr: the pointer to item's entity in RAM (that is, to a variable holding file's data), if one exists, or NULL otherwise
- fileMarks: file marks, specifying specific characteristics of the file.

  File marks may be set either to following values:
  - SIZE_MODIFICATION_ALLOWED: indicates that size of the file can be different in new firmware image after over-the-air upgrade. Usually is set for files storing table data, such as binding table, group table and others.
  - ITEM_UNDER_SECURITY_CONTROL: no impact, works same as NO_FILE_MARKS
  - NO_FILE_MARKS: no special characteristics for the file

A file descriptor tied to some data in RAM is defined by using the PDS_DECLARE_FILE macro in the code that may be used by both the stack and the application:

```
PDS_DECLARE_FILE(memoryId, size, ramAddr, fileMarks)
```

Section 8.5 provides an example how application can define a persistent data file for own data.

8.2.2 Directory Descriptors

Directory descriptors are special entities describing a group of file. A directory descriptor is defined in the code (the stack's or the application's one) and is placed to the separate flash memory segment.

The directory descriptor consists of the following parts:

- list: pointer to the list of files IDs associated with the directory. This list should be placed in the flash memory (by the use of the PROGMEM_DECLARE macro – see an example below).
- filesCount: the amount of files associated with the directory
- memoryId: memory identifier associated with the directory

A directory is declared via the PDS_DECLARE_DIR macro in the following way:

```
PDS_DECLARE_DIR(const PDS_DirDescr_t csGeneralParamsDirDescr) =
{
  .list       = CsGeneralMemoryIdsTable,
  .filesCount = ARRAY_SIZE(CsGeneralMemoryIdsTable),
  .memoryId     = BC_GENERAL_PARAMS_MEM_ID
};

//Where CsGeneralMemoryIdsTable is the list of objects defined in the following way:
PROGMEM_DECLARE(const PDS_MemId_t CsGeneralMemoryIdsTable[]) =
{
  CS_UID_MEM_ID,
  CS_RF_TX_POWER_MEM_ID,
  //other parameters in this list
}
```

Note that each file in this list shall be defined using the PDS_DECLARE_FILE macro as described in Section 8.2.1. Linker will form the complete list of directory descriptors at compile time.

Section 8.5 provides an example how application can define a directory for persistent data files.

8.3 PDS API Functions

The API functions are used to perform actions with data prepared for storing in the NV memory and identified via memory IDs: with files and directories. The functions are listed and explained in Table 8-1. Most of the functions have the `memoryId` argument, which should be set to the ID of a file or a directory. For details on using the API see BitCloud API Reference [3], which is available with the BitCloud SDK.

Caution: Although the HAL component also provides read/write functions for Flash, it is strongly recommended to use the PDS component for such purposes to eliminate the risk of overwriting stack-specific variables.

Table 8-1. PDS API Functions.

Function	Description
`PDS_Store(memoryId)`	Writes the object (a file or a directory) with the specified memory ID to the NV storage
`PDS_StoreByEvents(memoryId)`	The specified stack object will be written to the NV storage each time one of the events, to which the object is subscribed, occurs. Such mapping is done inside PDS via `EVENT_TO_MEM_ID_MAPPING` macro.
`PDS_Restore(memoryId)`	Reads the latest version of the object from the NV storage
`PDS_Delete(memoryId)`	Marks a file or all files in a directory as deleted. The space occupied by deleted files is considered unoccupied.
`PDS_IsAbleToRestore(memoryId)`	Determines if the specified object can be restored from the NV storage

To save data attributed to a file or a directory into the NV storage the application should call the `PDS_Store()` function, providing memory identifier of a file or a directory as an argument. For example, to force saving of all BitCloud stack parameters use the following code:

```
PDS_Store(BC_ALL_MEMORY_MEM_ID);
```

To be able to store several specific files at once, define a directory containing these files (this may be done at compile time only) and provide the memory ID of this directory to the `PDS_Store()` function.

To restore data from the NV storage to the data variables the application should call the `PDS_Restore()` function, providing an identifier of a file or a directory to be restored. Before trying to restore data, the application may check that restoring of a specific item is possible, using the `PDS_IsAbleToRestore()` function. If the item 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 directory with the `BC_ALL_MEMORY_MEM_ID` identifier:

```
if (PDS_IsAbleToRestore(BC_ALL_MEMORY_MEM_ID))
  PDS_Restore(BC_ALL_MEMORY_MEM_ID);
```

8.4 Maintaining Stack Data in the NV Storage

PDS implements a simple interface to maintain in persistent memory important network-related ConfigServer parameters including various internal tables (neighbor table, binding table, etc.). PDS memory identifiers for such files and directories are declared in and `Components/PersistentDataServer/include/wl/include/wlPdsMemIds.h` file. The files and directories themselves for stack data are defined in the CS component in the `/ConfigServer/src/csPersistentMem.c` file (to understand how files and directories are defined see Section 8.2).

The application can maintain stack parameters as follows:

- Rely on the BitCloud stack for automatic storing of stack parameters on pre-defined events that are likely to change their values. For this the application should call the `PDS_StoreByEvents()` function once to specify a file or a directory that should be taken under maintenance by events (for example, `BC_ALL_MEMORY_MEM_ID` – to store all stack files).
- Store stack data at arbitrary moments by itself, using the `PDS_Store()` function

Stack data should also be restored in the application via the `PDS_Restore()` function, for example during initialization, as described in Section 8.3.

8.5 Maintaining Application Data in the NV Storage

An application can maintain arbitrary application data in the NV storage, using simple interface of the PDS component. For the application data, the user shall define file descriptors, tying them to particular variables, and, if necessary, define directories to organize files in groups. The PDS API is then used to store and restore the defined items.

The application can use `APP_DIR1_MEM_ID` and `APP_DIR2_MEM_ID` identifiers for application directories and identifiers formatted as `APP_PARAM<number>_MEM_ID` for files.
Among the memory IDs there are some IDs intended to be used by the application (`APP_*`), and the application can define more IDs if needed. There are also two important constants set in this file: `APPLICATION_MAX_FILES_AMOUNT` limiting the number of files that can be defined in the system and `APPLICATION_MAX_DIRECTORIES_AMOUNT` limiting the number of directories.
The application should first define file descriptors, using the `PDS_DECLARE_FILE` macro for each file, providing it with the ID, the size of data, the pointer to the RAM memory storing the data itself, and file marks.

For example, consider an application that needs to store instances called scenes and server attributes of the Scenes cluster.

```
//The variables that store the application data
extern Scene_t scenes[MAX_NUMBER_OF_SCENES];
extern ZCL_SceneClusterServerAttributes_t scenesClusterServerAttributes;

PDS_DECLARE_FILE(APP_PARAM1_MEM_ID,
                 sizeof(Scene_t)*MAX_NUMBER_OF_SCENES,
                 &scenes,
                 NO_FILE_MARKS);

PDS_DECLARE_FILE(APP_PARAM2_MEM_ID,
                 sizeof(ZCL_SceneClusterServerAttributes_t),
                 &scenesClusterServerAttributes),
                 NO_FILE_MARKS);
```

The next step is to define a separate list of memory IDs assigned to the files, using the `PRGMEM_DECLARE` macro:

```
PROGMEM_DECLARE(const PDS_MemId_t appMemoryIdsTable[]) =
{
  APP_PARAM1_MEM_ID,
  APP_PARAM2_MEM_ID,
}
```

Finally, the application should define a directory descriptor, using the `PDS_DECLARE_DIR` macro, in the following way:

```
PDS_DECLARE_DIR(const PDS_DirDescr_t appMemoryDirDescr) =
{
  .list       = appMemoryIdsTable,
  .filesCount = ARRAY_SIZE(appMemoryIdsTable),
  .memoryId   = APP_DIR1_MEM_ID    //Directory ID
};
```

The provided code samples completely define auxiliary structures to store the application data in the NV storage. PDS API function (see Section 8.3) such as `PDS_Store()` and `PDS_Restore()` can be used with `APP_PARAM1_MEM_ID`, `APP_PARAM2_MEM_ID` and `APP_DIR1_MEM_ID` as an argument to maintain the contents of these variables in the NV storage.
To simplify the file and directory management in the application code, it is also possible to map own memory ID to the target one for example:

```
#define APP_LIGHT_DATA_MEM_ID APP_DIR1_MEM_ID
#define APP_LIGHT_SCENE_CLUSTER_SERVER_ATTR_MEM_ID APP_DIR1_MEM_ID
```

9. Hardware Control

In addition to the ZigBee networking functionality 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 component also implements a number of drivers that provide interfaces used in Over-the-Air upgrade (ISD/OFD) and implement additional serial interfaces (VCP and USB FIFO).

Table 9-1. HW Interfaces Supported in BitCloud with API

Interface	ATmega256(4)RFR2	Comments
UART /USART	Yes	See Section 9.1
SPI	Yes	See Section 9.2
TWI	Yes	See Section 9.3
ADC	Yes	See Section 9.4
GPIO	Yes	See Section 9.5
IRQ	Yes	See Section 9.6
WDT	Yes	
PWM	Yes	
USB	N/A	
VCP(2)	N/A	
USB FIFO(2)	Yes	
Internal Flash	Yes	
EEPROM	Yes	
1-wire	Yes	
Battery monitor	Yes	
ISD (for OTA Server)	Yes	See Section 9.8.1
OFD (for OTA client)	Yes	See Section 9.8.1

Sections below provide additional information on how some of the HW interfaces can be accessed via BitCloud API. Detailed API description can be found in [ref].

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 [3].

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.

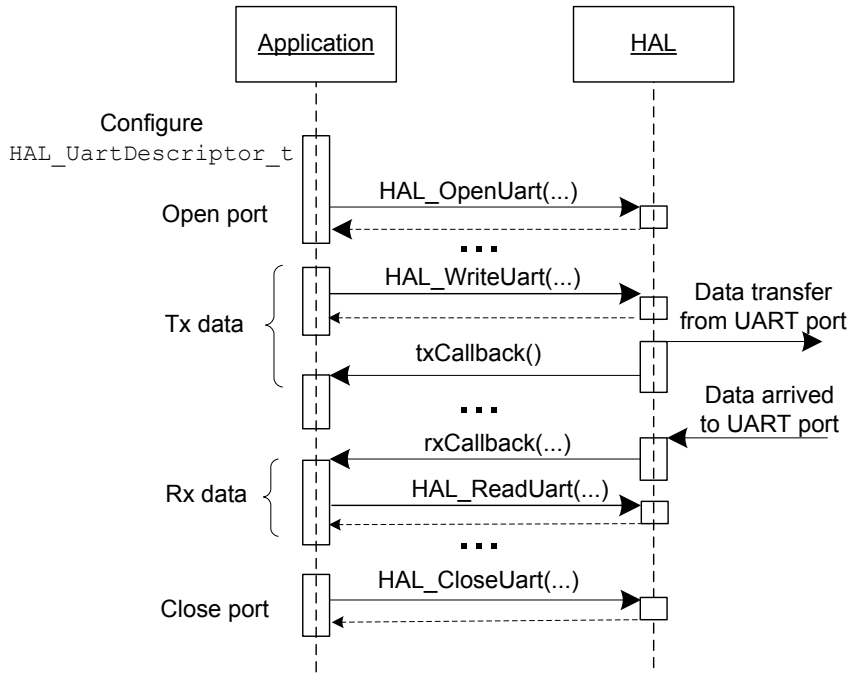
9.1.1 USART Callback Mode

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

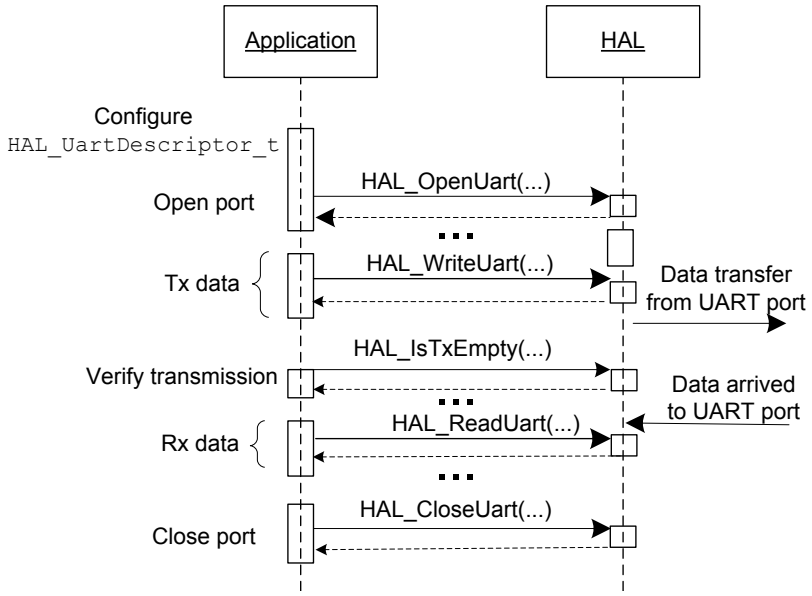
In the polling mode, USART Tx/Rx operations utilize corresponding cyclic buffers of the USART descriptor. Thus, the buffer pointer as well as the buffer length should be set to non-zero for the direction to be deployed in polling mode, while the corresponding callback function should be NULL.

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

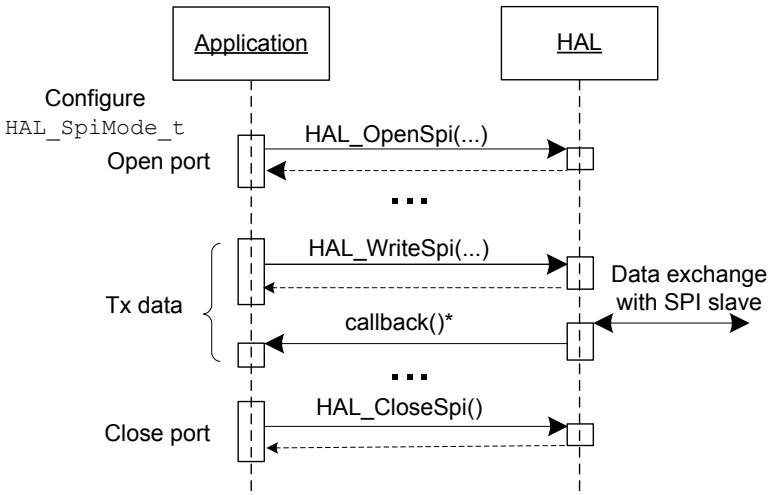
In addition to data read/write operations, the Atmel BitCloud stack provides an API for managing the CTS / RTS / DTR lines of the USART port that supports hardware flow control (depending on the platform). For a detailed description of the corresponding functions, see [3].

9.2 SPI Bus

Depending on the MCU platform, the HAL component of the BitCloud stack implements the SPI protocol either on the USART bus or on a pure SPI bus (when available) or both. 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 9-3 illustrates a sequence diagram of SPI-related API calls.

Figure 9-3. 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.

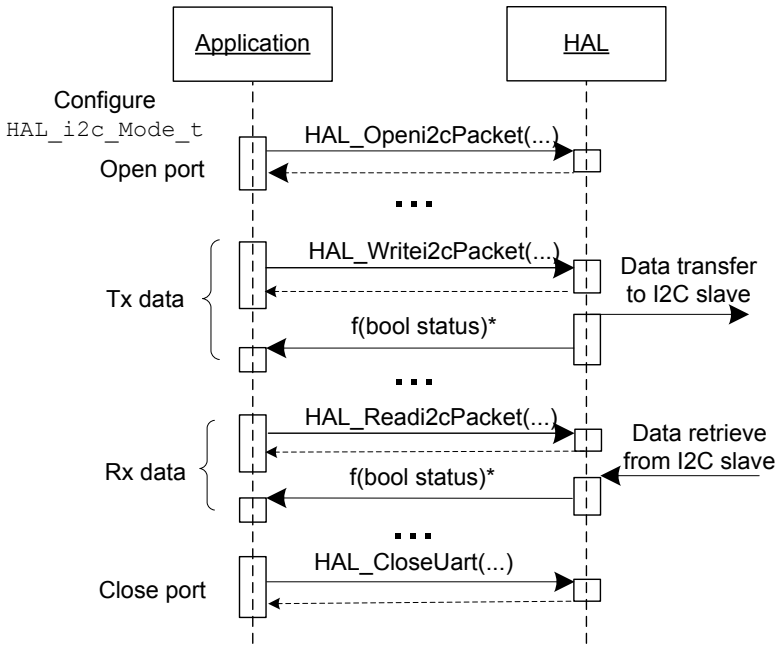
9.3 Two-wire Serial Interface Bus

For technical details about TWI modules as I^2C slaves, 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 9-4 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 9-4. Data Exchange over the TWI Bus



As shown in the figure above, TWI write/read operations are executed in asynchronous manner (see Section 2.1). 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 ADC

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

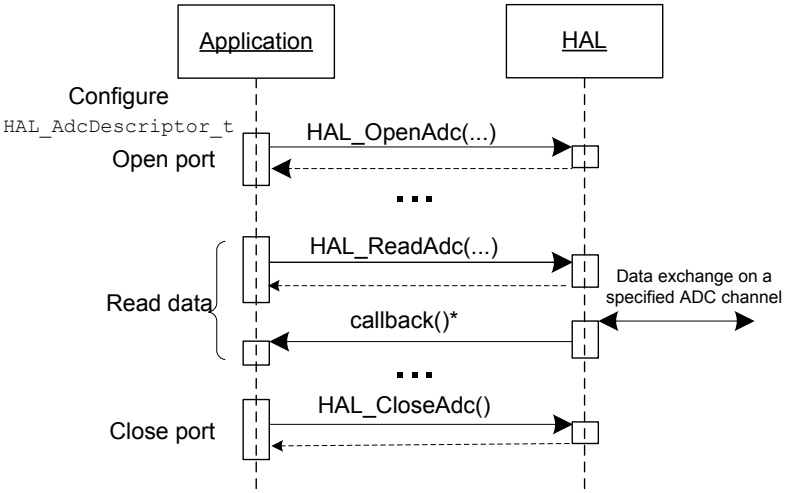
As with other interfaces, the ADC interface must be first configured and enabled using the `HAL_OpenAdc()` function with an argument pointing to a global variable of `HAL_AdcDescriptor_t` that contains configuration parameters. The `HAL_OpenAdc()` function returns status indicating whether the opening 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



9.5 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 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.

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

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

9.7 Timers

The stack provides a high-level application timer interface, which relies on a low-level implementation on top of hardware timer.

The `HAL_AppTimer_t` structure defines the timer interval (in milliseconds), its operation mode (whether it is a one-shot timer or a repeating timer), 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. Here is an example how this can be done:

```
// global definition of the timer instance and its callback
static HAL_AppTimer_t activityTimer;
static void activityTimerFired(void);
…
// configure timers
activityTimer.mode     = TIMER_ONE_SHOT_MODE;
activityTimer.interval = ACTIVITY_TIMER_PERIOD;
activityTimer.callback = activityTimerFired;
…

// start the timer
HAL_StartAppTimer(&activityTimer);
…

// callback function called when timer expires
static void activityTimerFired(void)
{
  …
}
```

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 (or ISD - for server side of OTAU cluster)
- Serial Flash driver (or OFD - for client side of OTAU cluster)
- USB FIFO driver
- and others

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 [5].

9.9 RF Control

The stack provides the functionality of controlling RF registers. Two functions `RF_RegWriteReq()` and `RF_RegReadReq()`, nominally a part of the MAC-PHY components, serve to write and read, respectively, radio chip registers.

To use the functions configure an instance of `RF_RegAccessReq_t` type: specify the 16-bit address in the `addr` field, the 8-bit value in the `value` field (for the writing request), and the callback function of the following signature:

```
void RF_RegAccessConf(RF_RegAccessConf_t *conf);
```

In the callback function, process the status of the operation (either success or failure), and access the value (for the reading request) of the specified register in the `value` field.

To see the list of registers supported by an RF chip refer to the RF chip's documentation.

Appendix A. Extending the Cluster Library

The user might need to add a new cluster not implemented in BitCloud. It is possible because it affects only header files and does not require modifications of the BitCloud source code, which is not available for users.

To add a new cluster:

- Create `.h` file in the `BitCloud\Components\ZCL\include\` directory (how to fill this file is described in Section A.1). Name the file this way:

  `zcl<Name>Cluster.h`
- Where `<Name>` is the cluster's name. For example, `Alarms`
- Add a constant for the cluster ID to the enumeration in the `clusters.h` file; the cluster ID should be a 16-bit value in the little endian format. Add a line in the following format to the enumeration:

```
<NAME>_CLUSTER_ID = CCPU_TO_LE16(<ID>)
```

- For example, for the Alarms cluster this line will be:

```
ALARMS_CLUSTER_ID = CCPU_TO_LE16(0x0009)
```

Makefiles need not to be changed, because all the files from the ZCL's `include` directory are included during compilation.

A.1 Filling the Cluster's Header File

The header file for a cluster should contain all definitions related to the cluster enumerated in Section 5.2. It might be useful to take an existing cluster header file as a template. The following instruction shows how to define correctly all necessary structures:

1. Include general ZCL headers:

```
#include <zcl.h>
#include <clusters.h>
```

2. Define constants for amounts of client and server attributes and commands. At first you may assign zero values to the constants increasing them when adding attributes and commands:

```
#define ZCL_<Name>_CLUSTER_SERVER_ATTRIBUTES_AMOUNT 0
#define ZCL_<Name>_CLUSTER_CLIENT_ATTRIBUTES_AMOUNT 0
#define ZCL_<Name>_CLUSTER_COMMANDS_AMOUNT 0
```

3. Define macros that will be used to initialize the cluster in the application. These macros fill an instance of the `ZCL_Cluster_t` type, which will represent the cluster in the application. Two macros are required for a client and a server and one macro that will switch between first two.
   a. The macro defining the client cluster type:

```
#define <Name>_CLUSTER_ZCL_CLIENT_CLUSTER_TYPE(clattributes, clcommands) \
  { \
    .id               = <Name>_CLUSTER_ID, \
    .options          = {.type       = ZCL_CLIENT_CLUSTER_TYPE, \
                         .security   = <Security>, \
           .ackRequest = 1}, \
    .attributesAmount = ZCL_<Name>_CLUSTER_CLIENT_ATTRIBUTES_AMOUNT, \
    .attributes       = (uint8_t *)clattributes, \
    .commandsAmount   = ZCL_<Name>_CLUSTER_COMMANDS_AMOUNT, \
    .commands         = (uint8_t *)clcommands \
  }
```

   b. The macro defining the server cluster type:

```
#define <Name>_CLUSTER_ZCL_SERVER_CLUSTER_TYPE(clattributes, clcommands) \
  { \
    .id                = <Name>_CLUSTER_ID, \
    .options           = {.type = ZCL_SERVER_CLUSTER_TYPE, \
                          .security = <Security>, \
                          .ackRequest = 1}, \
    .attributesAmount  = ZCL_<Name>_CLUSTER_SERVER_ATTRIBUTES_AMOUNT, \
    .attributes        = (uint8_t *)clattributes, \
    .commandsAmount    = ZCL_<Name>_CLUSTER_COMMANDS_AMOUNT, \
    .commands          = (uint8_t *)clcommands \
  }
```

`<Security>` should be substituted with the appropriate security level either `ZCL_NETWORK_KEY_CLUSTER_SECURITY` or `ZCL_APPLICATION_LINK_KEY_CLUSTER_SECURITY` depending on desired

c. The macro that will be used to initialize the cluster in the application (see Section 5.3.4):

```
#define DEFINE_<Name>_CLUSTER(cltype, clattributes, clcommands)
<Name>_CLUSTER_##cltype(clattributes, clcommands)
```

`cltype` in the argument may be set to either `ZCL_SERVER_CLUSTER_TYPE` or `ZCL_CLIENT_CLUSTER_TYPE` to point to the corresponding macro. The macro will switch between first two.

The application should use this macro to initialize the cluster instance.

4. If the cluster has attributes.
   a. Define a type that will carry information about all cluster attributes:
      `ZCL_<Name>ClusterAttributes_t`
   b. Define a macro that will initialize attributes in the application:
      `#define ZCL_DEFINE_<Name>_CLUSTER_SERVER_ATTRIBUTES() \`
      For client attributes replace the `SERVER` par in the macro name with `CLIENT`. Add a definition for each attribute using the `DEFINE_ATTRIBUTE` macro (explained in the next step).
   c. Add attributes one by one as described in Section A.2.

5. If the cluster has cluster-specific commands
   a. Define a type that will carry information about all commands:
      `ZCL_<Name>ClusterCommands_t`
   b. Define a macro that will initialize commands in the application:
      `#define <Name>_CLUSTER_COMMANDS() \`
      Add a definition for each command using the `DEFINE_COMMAND` macro (explained in the next step).
   c. Add commands one by one as described in Section A.3.

Note: Only cluster-specific commands should be defined for the cluster. General commands are supported in BitCloud for all clusters by default.

To use the cluster in the application, add an instance of the `ZCL_Cluster_t` type representing this cluster to the list of client of server clusters prepared for an endpoint. An instance of the `ZCL_Cluster_t` type can be initialized easily with the help of the macros defined in the cluster header file. For detail see Section 5.3.4.

A.2 Add a New Attribute to a Cluster

The cluster header file contains all information about its commands and attributes. To add a new attribute to the cluster, the user shall modify this file as described below.

Note: The instruction below is given for a non-reportable attribute. A reportable attribute is defined in a slightly different way as described in Section A.2.1.

6. Increase attributes amount (either server or client) by one. Attributes amount is given by two defines. For server attributes it is:

   `ZCL_<ClusterName>_CLUSTER_SERVER_ATTRIBUTES_AMOUNT`

7. Define a constant for the attribute identifier. Attribute IDs are typically named in the following way (for a client attribute replace `SERVER` with `CLIENT`):

   `ZCL_<ClusterName>_CLUSTER_SERVER_<AttributeName>_ATTRIBUTE_ID`

   Note that you must convert the attribute id value to little endian. For example:
   `CCPU_TO_LE16(0x0006)`.

8. Extend the attributes type definition or create one, if the cluster did not have attributes before. The attributes type is usually named as follows (for a client attribute replace Server with Client):

   `ZCL_<ClusterName>ClusterServerAttributes_t`

   The attributes type contains definitions of C structures for each attribute. Each structure consists of `id`, `type`, `properties`, and `value` fields for a non-reportable attribute and contains some additional fields for a reportable attribute (see Section A.2.1). For example, the time cluster attributes type is defined like this:

```
typedef struct PACK
{
  struct PACK
  {
    ZCL_AttributeId_t id;
    uint8_t type;
    uint8_t properties;
    ZCL_UTCTime_t value;
  } time;
  ... //Other attributes
} ZCL_TimeClusterServerAttributes_t;
```

9. Extend the macro that is used to define attributes instance or create one, if the cluster did not have attributes before.

   a. If the macro has not been defined earlier, first, add the line (for a client attribute replace `SERVER` with `CLIENT`)

```
#define ZCL_DEFINE_<ClusterName>_CLUSTER_SERVER_ATTRIBUTES() /
```

   b. If the macro is already defined, add a line (one line for each attribute) in the following format:

```
DEFINE_ATTRIBUTE(<AttrName>, <AttrId>,  <AttrType>)
```

   `<AttrName>` is the name of the corresponding attribute type field, `<AttrId>` is the constant name for the attribute identifier, and `<AttrType>` specifies the attribute type. Available attribute types are wrapped in the `ZCL_AttributeType_t` enumeration defined in the `zcl.h` file.

After all described steps are done the attribute can be used as all attributes defined earlier. Note that the code that configures a cluster in the application (see Section 5.3.2) should not be changed, if a new attribute has been added. It defines a variable for all attributes at once due to the macro that hides clusters definitions.

A.2.1 Adding Reportable Attributes

For a reportable attribute, the attribute's value is automatically sent to other devices with a given period. To define a reportable attribute follow the instruction in Section A.2 with the following changes:

10. The C-type defined for the attribute must include additional fields: `reportCounter`, `minReportInterval`, `maxReportInterval`, and `timeoutPeriod` of the `ZCL_ReportTime_t` type and `reportableChange` of the same type as the `value` field. For example, the attributes type for the simple metering cluster is defined like this:

```
typedef struct PACK
{
  struct PACK
  {
    ZCL_AttributeId_t id;
    uint8_t           type;
    uint8_t           properties;
    uint8_t           value[6];
    ZCL_ReportTime_t  reportCounter;
    ZCL_ReportTime_t  minReportInterval;
    ZCL_ReportTime_t  maxReportInterval;
    uint8_t           reportableChange[6];
    ZCL_ReportTime_t  timeoutPeriod;
  } currentSummationDelivered;
  ...//Other attributes
} SimpleMeteringServerClusterAttributes_t;
END_PACK
```

11. Initialize the attribute via the `DEFINE_REPORTABLE_ATTRIBUTE()` macro function, which has two additional arguments for the minimum and maximum reporting intervals. This implies that the attributes initialization macro should have the same additional arguments. For example, see the following definitions from the `zclSimpleMeteringCluster.h` file:

```
#define DEFINE_SIMPLE_METERING_SERVER_ATTRIBUTES(min, max) \
  DEFINE_REPORTABLE_ATTRIBUTE(currentSummationDelivered, ZCL_READONLY_ATTRIBUTE,
CCPU_TO_LE16(0x0000), ZCL_U48BIT_DATA_TYPE_ID, min, max), \
```

A.3 Add a New Cluster Command

To add a new cluster-specific command to a cluster, modify the header file that contains cluster definition. The user shall define the command ID, the payload format, and extend the definition of the C type that wraps all commands and the macro which fills the fields of an instance of this type. Follow the instruction provided below.

12. Increase the amount of cluster commands by one. The amount of cluster commands is defined in the following way:

```
#define <ClusterName>_CLUSTER_COMMANDS_AMOUNT
```

13. Define a constant for the command ID. Command IDs are typically defined in the following way:

```
#define <CommandName>_COMMAND_ID <value>
```

Note that a value should not be converted to the little-endian format.

14. Define the command payload format:

```
typedef struct PACK
{
  ...
} ZCL_<CommandName>_t;
```

Command payload is fully command-specific. The PACK macro ensures that the fields will be located in memory one-by-one without gaps.

15. Extend the commands type definition or create one, if the cluster did not have commands before. The commands type usually is named as follows:

    ZCL_<ClusterName>ClusterCommands_t

    The commands type contains definitions of C structures for each command. Each structure consists of `id`, `options`, and indication handler fields. For example, the price cluster commands type is defined like this:

```
typedef struct PACK
{
  struct PACK
  {
    ZCL_CommandId_t id;
    ZclCommandOptions_t options;
    ZCL_Status_t (*getCurrentPrice)(
                               ZCL_Addressing_t *addressing,
                               uint8_t payloadLength,
                               ZCL_GetCurrentPrice_t *payload);
  } getCurrentPriceCommand;
  ... //Other commands
} ZCL_PriceClusterCommands_t;
```

16. Extend the macro that is used to define commands instance or create one, if the cluster did not have commands before.

    g. Add or modify the first line of the macro, which look like this:

```
#define <ClusterName>_CLUSTER_COMMANDS(<indications>)
```

    `<indications>` represents the macro argument, which consists of indication handlers for each command.

    h. Add the line with command definition after the last macro's line. Use the `DEFINE_COMMAND` macro:

```
DEFINE_COMMAND(<Name>, <ID>, <Options>, <Indication>)
```

    `<Name>` must equal the name of the corresponding field in the commands structure. `<ID>` is set to the command id, define in Step 2. <`Options`> configure command options with the help of the `COMMAND_OPTIONS` macro, for example:

```
COMMAND_OPTIONS(CLIENT_TO_SERVER, ZCL_THERE_IS_RELEVANT_RESPONSE, ZCL_COMMAND_ACK)
```

    The `ZCL_COMMAND_ACK` flag indicates that the receiver of the command will send an acknowledgement frame on the APS level upon receiving the command. If the flag is left out, acknowledgement will not be sent.

    `<Indication>` must equal the name of the indication handler given in the macro argument.

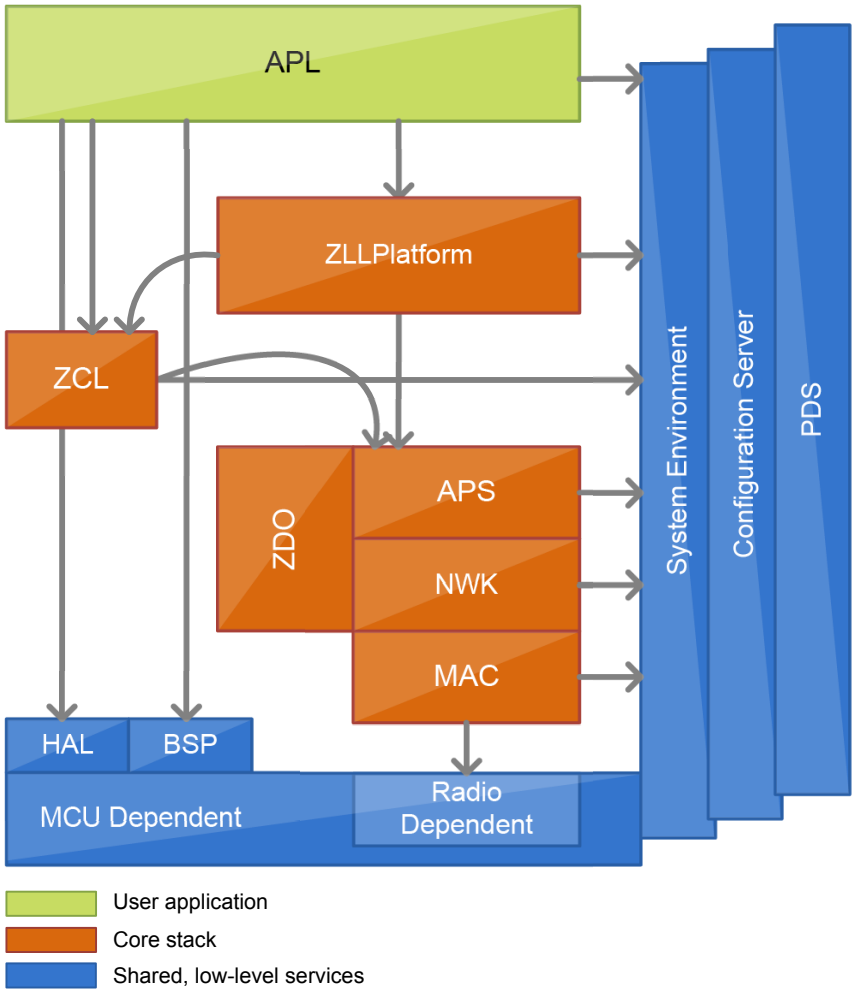
This completely defines a new cluster command. In the code that configures the cluster in the application, modify the invocation of the macro extended in Step 5 by specifying one more argument. Or if the cluster did not have commands before, define an instance of the commands type and pass its pointer to the cluster definition in the clusters list (see Section 5.3.3 for details).

Appendix B. ZLL Overview

B.1 Architecture

The ZigBee Light Link (ZLL) profile uses clusters from the standard ZigBee Cluster Library (ZCL). The architecture of BitCloud stack with ZLL is shown in Figure B-1. Unlike other applications based on BitCloud, ZLL applications do not interact with APS and ZDO directly, but involve an intermediate layer. This intermediate layer also implements ZLL-specific tasks hiding much of details of such operations as touchlink commissioning from the application. Also, the intermediate layer provides a wrapper for task management, which makes implementation of application state machine easier. The application still uses ZCL component of BitCloud to send cluster commands as well as HAL for establishing serial connection with a PC, PDS for storing data in non-volatile memory and ConfigServer for maintaining stack parameters.

Figure B-1. BitCloud ZigBee Light Link Architecture



B.2 Device Types

The ZigBee Light Link profile defines two general types of devices. The first type includes various lighting devices, and the second type consists of controllers – devices that perform remote control over lighting devices. In applications lighting devices act as ZigBee routers in the network, and controller devices either as ZigBee end devices or as ZigBee routers. Lists of ZLL devices pertaining to lighting devices and to controllers are given in Table B-1.

Table B-1. ZigBee Light Link Devices and Corresponding ZigBee Device Types

Category	ZLL device	ZigBee type
Lighting devices	On/off light On/off plug-in unit Dimmable light Dimmable plug-in unit Color light	Router
Controller devices	Color controller Color scene controller Non-color controller Non-color scene controller Control bridge On/off sensor	End device / Router

ZLL networks have no ZigBee coordinators, and so there is no dedicated device that is responsible for starting a ZLL network. Instead, a lighting device receives network parameters from a controller via inter-PAN data exchange and joins the network as if it had been already formed by the coordinator. Then a controller joins the network via association to the lighting device. The whole procedure of transferring parameters to a lighting device and making it a part of a network is called touchlink commissioning.

After the network start devices communicate with each other by sending usual ZCL commands: cluster-specific commands (defined in a cluster) and general commands (related to all clusters, such as reading and writing of attributes). Typically, controllers send commands to lighting devices to change light status.

Note that binding is not used in ZLL. Controllers store information about linked lights and can send a unicast command to each node identified by its network address. Alternatively, the group cluster can be used to create a group, add nodes to it and use group addressing whenever it is required to affect a group of lights.

B.3 ZLLPlatform General Topics

This chapter focus on some general tasks related to application's organization and describes how they can be approached with the help of the API of the ZLLPlatform component.

B.3.1 Initialization

Before using any of the ZLLPlatform modules, the application must call the module's initialization function. Such functions may be called all at once on application's start. Note that part of the modules must be initialized *after* MAC layer's initialization conducted via the `N_Cmi_InitMacLayer()` function. For reference, see `init()` and `initMacLayerDone()` functions in the reference application provided with BitCloud ZLL SDK. The latter function is passed as an argument to the `N_Cmi_InitMacLayer()` function and is executed once the MAC layer is initialized.

B.3.2 Finite State Machine

The ZLLPlatform package includes the `N_FSM` component used to implement a *finite state machine* in the application. With the help of this component, the state machine is represented in the application as a transition table which defines all the transitions between application's states. A transition consists of the event that initiates the transition, the condition that should be satisfied in order to trigger the action, the action, and the new state entered by the application on execution of the action.

Technically, a transition table is an array of elements of `N_FSM_Transition_t` type. This array must consist of groups of elements each starting with an element that defines a state and followed by one or several elements defining transitions from this state. An element defining a state is declared via the `N_FSM_STATE()` macro taking the state's identifier (a value from an enumeration). Transitions are declared via the `N_FSM()` macro, which should be provided with four arguments: the event, the condition, the action, and the new state.

There are a number of predefined macros that can be used to substitute arguments of the N_FSM() macro:

- Set N_FSM_NONE as the condition (action) if no condition should be checked (no action performed) for a certain transition
- Set N_FSM_ELSE as the condition to signify that the transition should be processed if the above transition (defined in the previous line) is being processed and its condition is false
- Set N_FSM_SAME_STATE as the new state if the application should be left in the same state after the transition

All parameters of a transition are set by some enumerated identifiers. What is actually executed to calculate conditions or perform actions is defined by two functions: one for conditions, and the other for actions, which are provided when the transition table is registered.

A transition table is registered via the N_FSM_DECLARE() macro. In the arguments specify the variable's name, which is later used to refer to the state machine, the transition table and its size, states' entry-exit actions table and its size, and two functions (for checking conditions and for performing actions).

An example of finite state machine definition is given in following Section B.3.3.

B.3.3 Example Transition Table

The example code below demonstrates implementation of a state machine based on usage of the N_FSM component. First, the application should define enumerations for all components of a transition:

```
enum colorSceneRemote_states //States
{
  sIdle,
  sWaiting,
  sScanning,
  //Other states
}

enum colorSceneRemote_events  //Events
{
  eTimer,
  eButtonPressed,
  eButtonReleased,
  //Other events
}

enum colorSceneRemote_conditions //Conditions
{
  cIsFactoryNewOrInDistributedTrustCenterMode,
  //Other conditions
}

enum colorSceneRemote_actions //Actions
{
  aShowDefault
  aStartWaitTimer,
  aStopTimer,
  //Other actions
}
```

Then the table of entry-exit actions for each state should be defined. Each state may be assigned with two functions: one will be executed each time the state is entered, the other each time the state is exited. Note that when the new state of a transition is set via the N_FSM_SAME_STATE macro and this transition occurs then the current state is not considered entered one more time and the entering function (as well as exiting) is not called.

```
//Functions to be called on entry into states
static void IdleEntry(void) { ... }
static void ScanningEntry(void) { ... }

//Defining a table of entry-exit functions for states
//Exit functions are not specified
static const N_FSM_StateEntryExit_t s_entryExit[] =
{
  N_FSM_ENTRYEXIT(sIdle, IdleEntry, NULL),
  N_FSM_ENTRYEXIT(sScanning, ScanningEntry, NULL),
  //Other states
}
```

Define each entry using the `N_FSM_ENTRYEXIT()` macro. Specify the state in the first argument and function pointers in the other two arguments. If one of the functions is not assigned to a state set `NULL` in its place. If no functions should be called on entering or exiting of a state, do not add an entry for this state.

Next, the application defines the transition table:

```
static const N_FSM_Transition_t s_transitionTableZllRemote[] =
{
  N_FSM_STATE(sIdle),
  N_FSM(eButtonPressed,   N_FSM_NONE,  aStartWaitTimer,  sWaiting),
  N_FSM(eTimer,           N_FSM_NONE,  aShowDefault,     N_FSM_SAME_STATE),

  N_FSM_STATE(sWaiting),
  N_FSM(eButtonReleased,  N_FSM_NONE,  aStopTimer,       sIdle),
  //More transitions under sWaiting

  //Other states and transitions from them
}
```

Once the transition table is defined it should be registered via the `N_FSM_DECLARE()` macro, which creates the finite state machine's instance:

```
N_FSM_DECLARE(fsm,   //Finite state machine (new variable)
              s_transitionTableZllRemote, //Transition table
              N_FSM_TABLE_SIZE(s_transitionTableZllRemote), //Its size
              s_entryExit,      //A table of entry/exit actions for states
              N_FSM_TABLE_SIZE(s_entryExit), //Its size
              PerformAction,
              CheckCondition);
```

The macro is provided with the finite state machine variable, a transition table, a table of entry-exit actions for states, and functions for performing actions and checking conditions. The `fsm` variable is defined by the above expression. Later it is used to refer to the state machine, for example, to post an event – to make the state machine process it:

```
N_FSM_PROCESS_EVENT(&fsm, &s_currentState, eTimer);
```

`s_currentState` is assumed to be a local `unit8_t` variable holding current state value. Additionally, one or two 32-bit arguments may be passed to the event. For this purpose, use `N_FSM_PROCESS_EVENT_1ARG()` to pass one argument and `N_FSM_PROCESS_EVENT_2ARGS()` to pass two arguments. The arguments are received in the perform-actions and check-conditions functions as the second and the third arguments.

Functions for performing actions and checking conditions may be defined in the following way:

```
//arg1 and arg2 hold data passed to the event
static void PerformAction(N_FSM_Action_t action, int32_t arg1, int32_t arg2)
{
  switch (action)          //Switch among actions
  {
    case aShowDefault:     //As cases, process specific actions
      ShowDefault();
      break;

    case aStartWaitTimer:
      StartWaitTimer();
      break;

    //Processing of other actions
  }
}

//The checking-conditions function must return true of false
static bool CheckCondition(N_FSM_Condition_t condition,
                           int32_t arg1, int32_t arg2)
{
  switch (condition)     //Switch among conditions
  {
    case cIsFactoryNewOrInDistributedTrustCenterMode:
      return IsFactoryNewOrInDistributedTrustCenterMode();

    //Check other conditions
    default:
      return TRUE;
  }
}
```

B.4 Tasks Management

The ZLLPlatform layer extends the task management mechanism used in BitCloud (implemented in the System Environment component). The application must manually specify task handler functions of ZLLPlatform's modules used by the application, while initializing the `N_Task` component. That is, the application defines an array of function pointers, in the following way:

```
static const N_Task_HandleEvent_t s_taskArray[] =
{
  ColorSceneRemote_EventHandler,       //Implemented in the application
  N_ConnectionEndDevice_EventHandler,  //Other handlers are implemented in
  N_LinkInitiator_EventHandler,        //ZLLPlatform
  N_DeviceInfo_EventHandler,
  N_LinkTarget_EventHandler,
  N_ReconnectHandler_EventHandler,
}
```

These functions will be called consequently while processing tasks posted to the ZLL component of the stack (via execution of `SYS_PostTask(ZLL_TASK_ID)`; see Section 2.2.1).The application can provided its own function for processing these tasks, placing it the first in the list, like the `ColorSceneRemote_EventHandler` function in the example.

The taskArray variable is then passed to the N_Task_Init() function:

```
N_Task_Init((uint8_t)N_UTIL_ARRAY_SIZE(s_taskArray), s_taskArray);
```

The own task handler function must take an argument of N_Task_Event_t type and return a bool value – FALSE returned will mean that the event is not handled (and, thus, is a subject for handling by other functions in the list), and TRUE will mean that the event has been handled. Consider the following example:

```
bool ColorSceneRemote_EventHandler(N_Task_Event_t evt)
{
  switch (evt)
  {
    case EVENT_TIMER:  //An example timer event is sent to the state machine
                       //named fsm; see Section B.3.3
      N_FSM_PROCESS_EVENT(&fsm, &currentState, eTimer);
      break;

    //Processing other events

    default:
      //Event not handled
      return FALSE;
  }

  //Event handled
  return TRUE;
}
```

The usage of the N_Task component does not replace the standard usage of APL_TaskHandler() function, which processes application-specific tasks posted by execution of SYS_PostTask(APL_TASK_ID). For more details on task management in BitCloud applications refer to Section 2.2.

B.5 Memory Allocation

To safely allocate memory at runtime, the application can use the N_Memory component. Memory is allocated from the heap via the N_Memory_AllocChecked() function and released via the N_Memory_Free() function.

For example, the application can allocate a buffer dynamically on reception of scan responses, which is demonstrated by the code samples below. First, define a variable at a file scope:

```
static N_LinkInitiator_Device_t *foundDevices = NULL;
```

Then memory is allocated and assigned to this variable in the following way:

```
foundDevices = (N_LinkInitiator_Device_t *)
N_Memory_AllocChecked((size_t)(sizeof(*foundDevices) * numDevices));
```

The numDevices variable is assumed to hold the number of received responses. When the memory is not longer needed, it must release by calling the N_Memory_Free() function:

```
N_Memory_Free(foundDevices);
foundDevices = NULL;
```

Once the memory is released, the variable pointing to it must be set to NULL for safety.

B.6 Internal Flash Access

Flash access in AVR® may be performed only from No Read-While-Write code memory section (refer to the documentation on the MCU). During flash access only the code from NRWW section may be executed, if attempt to execute code from usual area is performed while the flash access command is not finished – the result will be unpredictable. Thus all code which should be executed during the flash access shall be placed in NRWW section.

Also it is necessary to say that every flash access operation is very continuous in time and it should be kept in mind when some time critical procedure is implemented. For example, since writing to flash may take up to 10 ms UART communication can be corrupted (some bytes may be lost). Flash access driver has a feature that may help in such situations. It is possible to register poll handler which will be called periodically during flash access procedure. First the polling handler and the polling stopped handler shall be declared:

```
/* The Uart_PollingHandler will be called periodically during each flash access
procedure. */
NRWW_SECTION_DECLARE static void Uart_PollingHandler(void)
{
  /* Manually handle incoming data. */
}

/* The Uart_ PollingStopped will be called once at the end of each flash access
procedure. */
NRWW_SECTION_DECLARE static void Uart_PollingStopped(void)
{
  /* Finalize received data. */
}
```

This poll handler should be registered in the flash driver:

```
static HAL_PollDuringFlashWritingHandler_t handler = {
  .poll = Uart_PollingHandler,
  .finished = Uart_PollingStopped
};
HAL_RegisterPollDuringFlashWritingHandler(&handler);
```

Also it is possible to unsubscribe from polling during flash access:

```
HAL_UnregisterPollDuringFlashWritingHandler(&handler);
```

Flash memory interface needs some additional actions to be able to work. If PDS with Flash storage is used some of the following steps may be skipped otherwise all steps should be performed to enable Flash memory access.

1. Linker script shall contain definition of the following sections:
    a. For GCC compiler:
        i. .boot_section - section in No Read-While-Write code memory (regarding NRWW refer to the documentation on the MCU). The same boot section size shall be set by fuses;
        ii. .access_point - section with size of pointer to function, shall be placed in the very last bytes of code space and shall;
        iii. .nrww_section - No Read-While-Write (NRWW) code memory section as it is described in the MCU documentation; This step is not necessary if polling during flash access is not used.
        iv. .wwf_poll_access_section - section with size of pointer to function, shall be placed in the NRWW code space related to the application. It is not necessary to place this section at specified address.

b. IAR:

i. BOOT_SECTION - section in No Read-While-Write code memory (regarding NRWW refer to the documentation on the MCU). The same boot section size shall be set by fuses;

ii. NRWW_SECTION - No Read-While-Write (NRWW) code memory section as it is described in the MCU documentation;

2. Boot flash section which is programmed by fuses shall have enough size to fit the boot section declared in linker script. If the boot section occupies the whole No Read-While-Write (NRWW) code memory section than it is not possible to use polling mechanism since there is no memory to place polling handlers.

3. If application uses flash memory access interface and is not intended to be used with bootloader then somewhere in code the following macro shall be declared (see HAL\flash.h):

   `DECLARE_FLASH_ACCESS_ENTRY_POINT();`

4. In both cases when it is supposed to use bootloader or not following macro shall be declared:
   `DECLARE_POLL_DURING_FLASH_WRITING_ACCESS_POINT()`

   It defines a pointer to the poll handler which will be called during the flash writing process.

5. If it is not supposed to use the program memory access interface just don't declare `DECLARE_FLASH_ACCESS_ENTRY_POINT()` and `DECLARE_POLL_DURING_FLASH_WRITING_ACCESS_POINT()`. If access points are not declared the linker will strip out the whole flash access driver saving the program memory space.

B.7 Sleeping Device Management

ZLL relies on sleep-when-idle feature (see Section 7.1) for sleep management.

Within the ZLL Platform the module N_LowPower is responsible for power management and should be initialized using `N_LowPower_Init()` function. After the initialization it is possible to subscribe to `N_LowPower_SleepStatus_EnterSleep` and `N_LowPower_SleepStatus_LeaveSleep` events using the `N_LowPower_Subscribe()` function like it is shown below:

Firstly the callback function should be defined:

```
void N_LowPower_Callback(N_LowPower_SleepStatus_t status,
                         N_LowPower_SleepLevel_t   level)
{
      ...
}
```

After that the defined callback function should be registered in the `N_LowPower` module:

```
N_LowPower_Subscribe(N_LowPower_Callback);
```

B.8 System Mutex Implementation

BitCloud provides the User with the ability to synchronize a set of independent procedures within the application. For this purpose System mutex may be used. Its definition can be found in sysMutex.h file. The mutex is similar to binary semaphore with one significant difference: the mutex has an owner. It means that a module (a component) locks a mutex only it can unlock it. If a module tries to unlock a mutex it hasn't locked (thus doesn't own) then an error condition is encountered and, most importantly, the mutex is not unlocked.

To use the System mutex it should be created as it shown below:

```
CREATE_MUTEX(arbitraryMutexName); /* Memory allocation for
arbitraryMutexName mutex */
```

In the modules which are supposed to use the mutex it should be declared using the following macro:

```
        DECLARE_MUTEX(arbitraryMutexName);
```

After that the structure of type `SYS_MutexOwner_t` should be created within the module. This structure will be used in every operation with the mutex. It should be initialized as it shown below:

```
        static SYS_MutexOwner_t arbitraryOnwer =
        {
.context = &userSpecificData, /* Purpose of this parameter is to save owner's
execution context. It's up to the owner how to use this parameter. Mutex
implementation must not modify this parameter. */
.SYS_MutexLockConf = arbitraryMutexNameLockConf /* This function is called in the
case when the SYS_MutexLock() operation was postponed and now the mutex is locked by
the owner. */
        };
```

When the mutex owner structure is declared it is possible to use the mutex for synchronization:

```
void someFunction(void)
{
 if (!SYS_MutexLock(&arbitraryMutexName, &arbitraryOnwer))
        return; // Wait the callback function.
 // Work with the resource which is locked by the mutex.
 SYS_MutexUnlock(&arbitraryMutexName, &arbitraryOnwer);
}
void arbitraryMutexNameLockConf(void)
{
 // Work with the resource which is locked by the mutex.
 SYS_MutexUnlock(&arbitraryMutexName, &arbitraryOnwer);
}
```

B.9 ZLL Device Configuration

B.9.1 Basic Configuration

Along with initialization of all ZLLPlatform's modules used by the application (described in Section B.3.1), the `N_DeviceInfo` module must be also initialized via the `N_DeviceInfo_Init()` function. A controller device may be configured, using this function, like it is shown below:

```
N_DeviceInfo_Init(N_DeviceInfo_Profile_ZLL,
                  TOUCHLINK_RSSI_CORRECTION,
                  TOUCHLINK_RSSI_THRESHOLD,
                  BUTTONLINK_RSSI_THRESHOLD,
                  TRUE,      //Can assign network addresses to others
                  TRUE,      //Can initiate touchlink and buttonlink
                  0xF5u);
```

For full specification of the arguments see the `N_DeviceInfo_Init()` entry in [3].

B.9.2 Factory New State

After programming, ZLL devices are in the factory new state, which usually means that a device has not yet entered any networks. After successful network join the device is no longer in the factory new state, though it should be able to return to the default state on user's interaction.

The application can check if the device is in the factory new state via the `N_DeviceInfo_IsFactoryNew()` function, which returns `TRUE` if the device is factory new and `FALSE` otherwise.

The stack does not provide a special function for resetting a device to the factory new state. To make the device factory new, the application should simply remove all data saved in persistent memory and perform hardware reset, which is achieved by executing the following lines:

```
PDS_Delete(PDS_ALL_EXISTENT_MEMORY);
HAL_WarmReset();
```

B.9.3 Network Parameters

The channel mask, which determines which radio channels can be used by the device, consists for ZLL applications of the primary channel mask and the secondary channel mask. The primary and secondary channel masks are fixed by the ZLL specification. However, these parameters may be changed during testing and debugging of the application. The reference application provided with the BitCloud ZLL SDK declares them as `APP_PRIMARY_CHANNELS_MASK` and `APP_SECONDARY_CHANNELS_MASK` in the `configuration.h` file.

The primary and secondary channel masks must be explicitly set in the application by using the `N_DeviceInfo` module's API functions, as shown below:

```
N_DeviceInfo_SetPrimaryChannelMask(APP_PRIMARY_CHANNELS_MASK);
N_DeviceInfo_SetSecondaryChannelMask(APP_SECONDARY_CHANNELS_MASK);
```

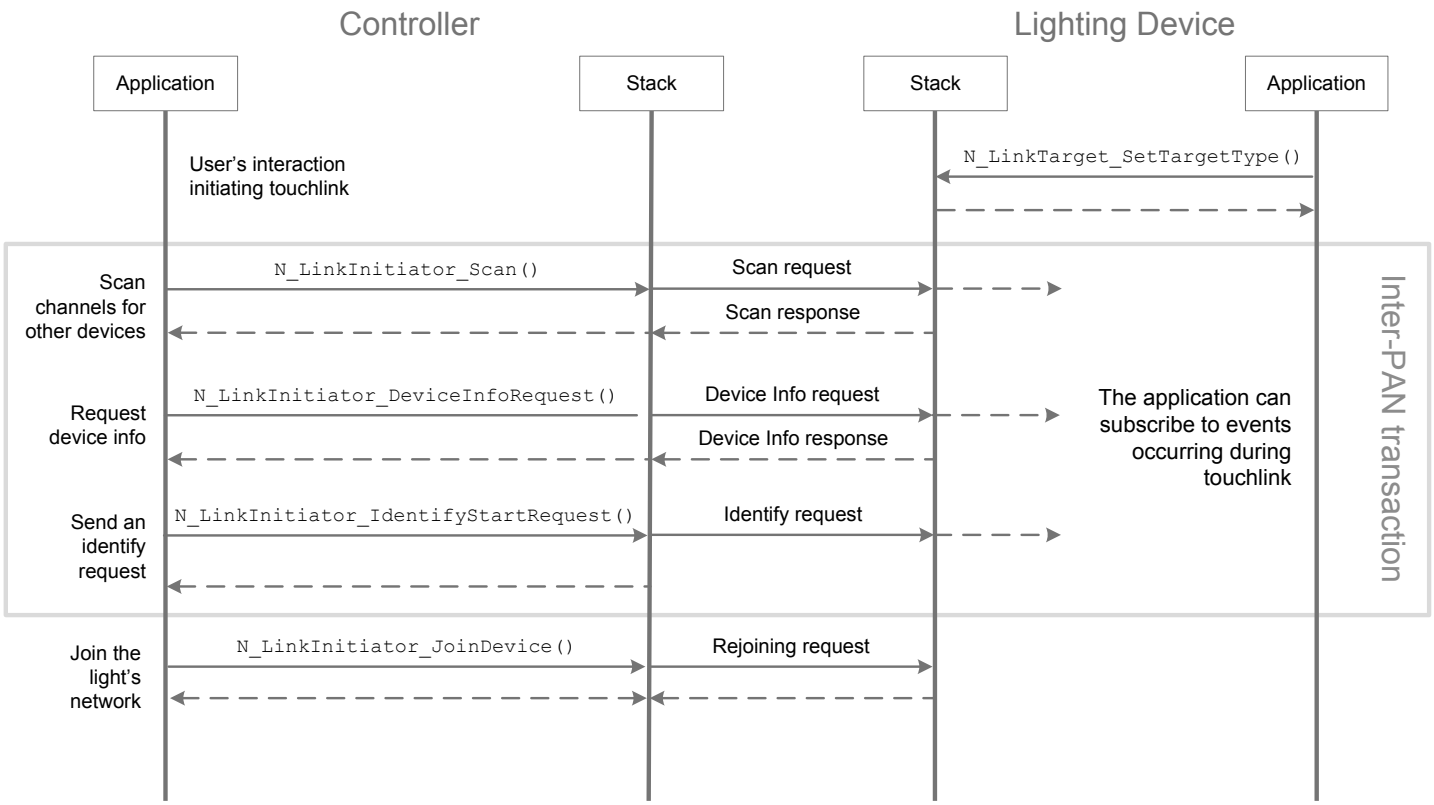
B.10 Touchlink Commissioning

For ZigBee Light Link applications network formation and join happen in the entirely different way. In common ZigBee networks a dedicated device called the coordinator forms the network with required parameters. In Light Link networks there is no network formation in the sense of common ZigBee networks.

The primary source of examples for the procedures described below is the ZLL demo application provided with the SDK. See `\Applications\ZLLDemo\ColorSceneRemote\src\colorSceneRemoteFsm.c` and `\Applications\ZLLDemo\Light\src\light.c` files.

Figure B-2 illustrates a typical sequence of application's actions for touchlink commissioning of a lighting device (steps not required by the ZLL specification [8] are marked as optional). The procedure is discussed in detail in section B.10.1 and B.11.

Figure B-2. Touchlink Commissioning of a Factory new Lighting Device



B.10.1 Controller's Side

A Light Link network is started after the commissioning of the first lighting device performed by a factory new controller. The commissioning procedure is usually initiated by pressing a special button on the controller brought close to the target light device.

B.10.2 Inter-PAN Mode

The thouchlink procedure includes sending inter-PAN commands. To make inter-PAN communication possible, the stack must be put into the corresponding inter-PAN mode. The *initiator* inter-PAN mode is switched on automatically when the application performs initiator-specific actions, such as scanning or sending the identify command. In touchlink with a light device, a controller is always the initiator, and so the controller's application does not need to set the initiator inter-PAN mode explicitly.

However, to enable touchlink between two controllers the *target* inter-PAN mode must be enabled on the controller. This is achieved by calling the `N_LinkTarget_SetTargetType()` function, making the controller simultaneous initiator and target of the touchlink. The target inter-PAN mode must be later disabled (via the same function) if it is decided on the application level that this controller is the initiator and not the target. The target inter-PAN mode may be enabled and, further, disabled during touchlink with the light as well, to support both cases. For details on touchlink between two controllers see Section B.12.

B.10.3 Scanning

The controller must discover the touchlink target by scanning ZLL channels. Scanning can be initiated as soon as the target type is set, as described in Section B.13. To initiate scanning, the application should call the `N_LinkInitiator_Scan()` function with the first argument set to `N_LinkInitiator_ScanType_Touchlink` for scanning only on primary channels. To scan the secondary channels as well, set the first argument to `N_LinkInitiator_ScanType_Touchlink | N_LinkInitiator_ScanType_IncludeSecondaryChannels`. In the second argument, provide an array of `N_LinkInitiator_Device_t` type to store results of the scanning, and specify its size in the third one and the confirmation callback function in the fourth. An example is given below.

First, define the array for storing information about discovered devices:

```
N_LinkInitiator_Device_t foundDevices[TOUCHLINK_SCAN_MAX_DEVICES];
```

The memory for the buffer may also be allocated dynamically from the heap. Refer to Section B.5 for details.

To perform scanning only on primary channels, execute:

```
N_LinkInitiator_Scan(N_LinkInitiator_ScanType_Touchlink,
                     foundDevices,                        //The buffer
                     (uint8_t) TOUCHLINK_SCAN_MAX_DEVICES, //Buffer's size
                     ScanDone);                           //The callback function
```

To scan for devices on both the primary and the secondary channels, execute:

```
N_LinkInitiator_Scan(N_LinkInitiator_ScanType_Touchlink |
                     N_LinkInitiator_ScanType_IncludeSecondaryChannels,
                     foundDevices,
                     (uint8_t) TOUCHLINK_SCAN_MAX_DEVICES,
                     ScanDone);
```

The stack fills the `foundDevices` array with scanning results sorted by link quality level. The element with the best RSSI level is the first in the array. This is the primary target of the touchlink procedure.

The callback function called on completion of scanning is defined in the following way:

```
void ScanDone(N_LinkInitiator_Status_t status, uint8_t numDevicesFound)
{
  switch (status)
  {
    case N_LinkInitiator_Status_Ok:
      ... //Scanning has been completed successfully
      break;

    case N_LinkInitiator_Status_Stopped:
      ... //Scanning has been stopped
      break;

    default:
      break;
  }
```

The `status` parameter indicates whether the scanning has been successful. `numDevicesFound` contains the number of discovered devices.

B.10.4 Requesting Device Info

Optionally, the application can request additional endpoint information from the touchlink target discovered during the scanning procedure, by calling the `N_LinkInitiator_DeviceInfoRequest()` function. In the argument, the application must provide an `N_LinkInitiator_Device_t` instance, an array of `N_InterPan_DeviceInfo_t` for storing information about target device's endpoints, its size, and the callback function that will be called on completion of the asynchronous request. The array for endpoints info must be at least as big as the value of the `scanResponse.numberSubDevices` field of the scan response instance (which contains the number of endpoints registered on the target device).

For example, consider `foundDevices` contains sorted scanning results from Section B.10.3. Define an endpoints info array. Since its size will be determined at runtime, the memory for it may be allocated dynamically via the `N_Memory_AllocChecked()` function (see more details on memory allocation in Section B.5):

```
//Define a variable of the file scope
static N_InterPan_DeviceInfo_t *epInfo = NULL;
...
epInfo = (N_InterPan_DeviceInfo_t *)N_Memory_AllocChecked((size_t)(sizeof(*epInfo)
                                  * foundDevices[0].scanResponse.numberSubDevices));
```

The device info request to the first element of the `foundDevices` array, which should be considered the most suitable touchlink target, is then issued in the following way:

```
N_LinkInitiator_DeviceInfoRequest(&(foundDevices[0]),
                                  epInfo,
                                  foundDevices[0].scanResponse.numberSubDevices,
                                  DeviceInfoRequestDone);
```

The callback function is defined this way:

```
void DeviceInfoRequestDone(N_LinkInitiator_Status_t status)
{
  //More code
}
```

B.10.5 Identifying the Lighting Device

Optionally, after sending the device info request or immediately after scanning if the device info request is not sent, the application can send identify request to the touchlink target discovered by the scanning procedure. The purpose of sending the identify command is to provide a capability for the user to cancel the touchlink procedure, for example, if the target has been selected incorrectly.

To start identifying, the application should call the `N_LinkInitiator_IdentifyStartRequest()` function, providing an `N_LinkInitiator_Device_t` instance, the identifying duration, and the callback function that will be called once the request is sent. In the following example the identify request is sent to the first element of the `foundDevices` array from Section B.10.3 containing scan responses (the one with the best link quality):

```
N_LinkInitiator_IdentifyStartRequest(&(foundDevices[0]),          //Target
                                     N_LINK_IDENTIFY_TIME_DEFAULT, //Duration
                                     IdentifyRequestDone);         //Callback
```

The callback function is defined in the following way:

```
void IdentifyRequestDone(N_LinkInitiator_Status_t status)
{
  //More code
}
```

The callback function is called as soon as the identify request is sent to the air. At this point the application may start a timer to allow the user to find out if the right touchlink target has been selected and cancel the touchlink otherwise. To support the latter, the application may watch some user-interaction event like releasing of the button pressed to initiate the touchlink. In this case, to stop indentifying the target the `N_LinkInitiator_IdentifyStopRequest()` should be called:

```
N_LinkInitiator_IdentifyStopRequest(&(foundDevices[0]), IdentifyRequestDone);
```

If the button is not released, the application should wait until the timer elapses and proceed to joining the network.

B.10.6 Joining the Network

To complete touchlink, the controller's application must call the `N_LinkInitiator_JoinDevice()` function. If the controller is not factory new, this will make the target light join the network of the controller. Otherwise, the light will form a new network and the controller will join it. As the first argument, provide an `N_LinkInitiator_Device_t` instance (it may be an element of the `foundDevices` array from Section B.10.3 containing scan results). The second argument determines whether the communication check is skipped (`TRUE`) or performed (`FALSE`). In the last argument, specify the callback function.

The function may be called in the following way:

```
N_LinkInitiator_JoinDevice(&(foundDevices[0]), FALSE, JoinDeviceDone);
```

Where the callback function is defined as follows:

```
void JoinDeviceDone(N_LinkInitiator_Status_t status,
                    uint16_t targetNetworkAddress)
{
  //More code
}
```

The `status` field provides result code of the joining operation. The `targetNetworkAddress` field contains the address of the network formed or joined by the target light.

B.10.7 Sending Add Group Request and Saving Light's Info

Once the touchlink procedure is completed and both the controller and the light join the network, the controller may add the lighting device to a group. This is achieved by sending the add group command of the groups cluster. The command is sent using the `ZCL_CommandReq()` function, as a cluster-specific command, with `APS_SHORT_ADDRESS` as the addressing mode to the short address of the lighting device (which may be obtained from the `scanResponse.networkAddress` field of the `N_LinkInitiator_Device_t` instance received by the application with scan results). For details on sending ZCL commands refer to Section 5.6.

B.11 Lighting Device's Side

On startup, a factory-new light typically tries to discover and join a ZigBee network. Regardless of how its attempts finish, the light should be ready to become the target of a touchlink procedure; that is, *target* inter-PAN mode should be enabled. To enable the inter-PAN mode call the `N_LinkTarget_SetTargetType()` function from the application. The first parameter may be set to `N_LinkTarget_TargetType_Hybrid` to support both touchlink and buttonlink or to `N_LinkTarget_TargetType_Touchlink` to support only touchlink. An example is given below.

```
N_LinkTarget_SetTargetType(N_LinkTarget_TargetType_Hybrid, setTargetTypeDone);
```

The callback function is called on completion of the operation and takes no parameters:

```
void setTargetTypeDone(void)
```

```
{
  ...
}
```

No other actions are required on the light for performing touchlink. All interactions with the link initiator are implemented in ZLLPlatform transparently for the light's application.

B.11.1 Subscribing to Events

The link target's application can subscribe to a number of events, such as scanning, reception of identify and device info requests, etc. To subscribe to the events, the application should define a table of callback functions that will be called on occurrence of the corresponding events and pass this table to the `N_LinkTarget_Subscribe()` function, as shown in the following example:

```
//Define a callback function for network joins.
//Note that different types of indications
//may require distinguishing sets of arguments
static void joinNetworkIndication(uint16_t groupIdFirst, uint16_t groupIdLast)
{
  //Code goes here
}

//Define a table of callback functions
static const struct N_LinkTarget_Callback_t targetCallbacks =
{
  NULL,  //  ScanIndication,
  NULL,  //  IdentifyStartIndication,
  NULL,  //  IdentifyStopIndication,
  NULL,  //  ResetIndication,
  joinNetworkIndication,
  NULL,  //  UpdateNetworkIndication,
  NULL,  //  ReverseJoinIndication,
  -1
};
...
//While initializing the application, pass the
//table to the API function
N_LinkTarget_Subscribe(&targetCallbacks);
```

For details on available indication types, refer to documentation of thee `N_LinkTarget_Callback_t` type in [3].

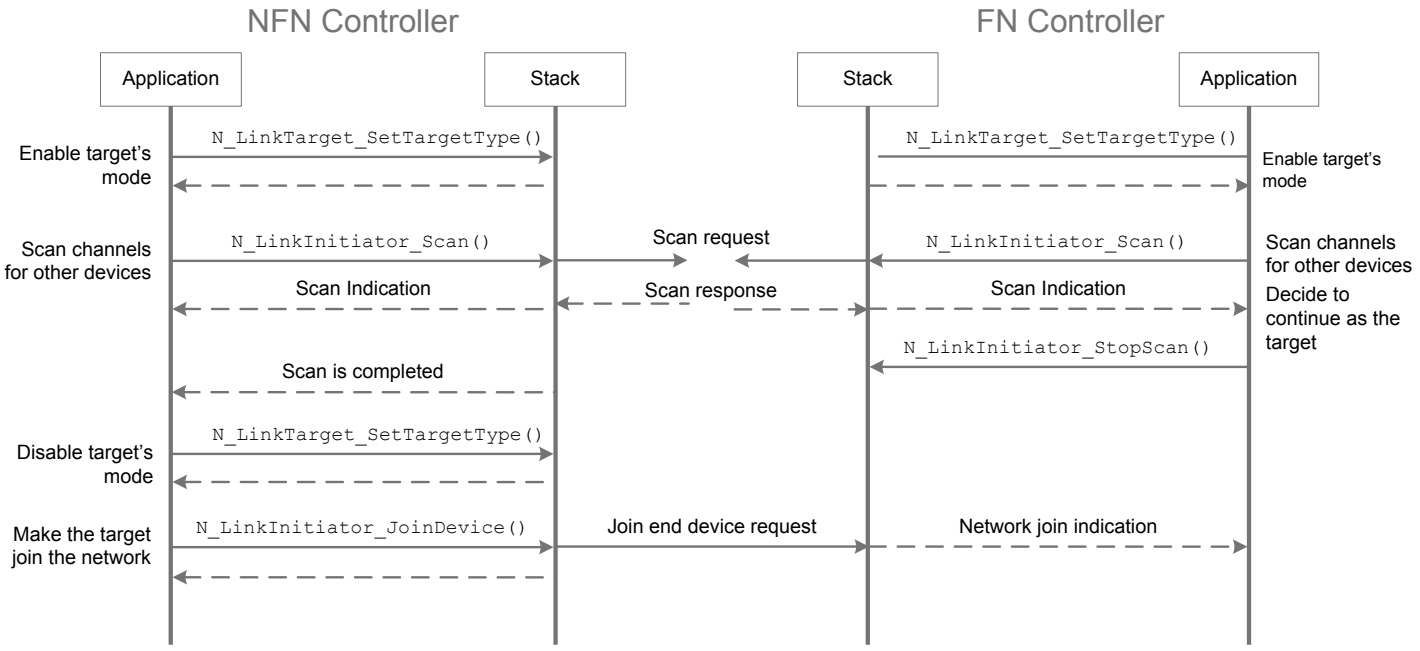
B.12 Touchlink between Two Controllers

Touchlink may happen between a non-factory new controller device and a factory new one. As a result, the factory new controller joins the ZLL network of the non-factory new controller and may do touchlink with the lighting devices from this network.

Note: If a Light Link network consists of one controller and several lighting devices, then making a touchlink between another controller and one of the lighting devices will create a new network. Therefore to control the lighting devices through a controller that has not belonged to the network, the controller should join the network – this is achieved using touchlink between two controllers.

Touchlink procedure between two controllers starts with both devices performing scanning (see Section B.10.3). The procedure may be initiated by buttons pressed simultaneously on both controllers. The whole touchlink procedure between two controllers is illustrated by Figure B-3.

Figure B-3. Required Steps for Touchlink between Two Controllers



B.13 Setting Target Type

Each of the controllers participating in the touchlink procedure starts it as both the initiator and the target. The initiator inter-PAN mode is activated automatically when initiator-specific requests (such as scanning) are issued. To enable the target mode, making the controller capable of responding to inter-PAN requests as a link target, the application should call the `N_LinkTarget_SetTargetType()` function with the first parameter set to `N_LinkTarget_TargetType_Touchlink`. For example:

```
N_LinkTarget_SetTargetType(N_LinkTarget_TargetType_Touchlink, SetTargetTypeDone);
```

The callback function, called on completion of the operation, is defined in the following way:

```
static void SetTargetTypeDone(void)
{
  ...
}
```

B.13.1 Processing Scan Indication

Both participating controllers should start scanning channels. As a target, each controller will receive scan indication from the other one. The scan indication may be raised up to the application and processed in a callback function. At this point, the application on each controller can decide which of the two devices should be the initiator and which one the target. In a typical scenario, the factory-new controller becomes the target, and the non-factory-new controller stays the initiator. The target then should stop scanning and cancel further initiator's activities, while the initiator should disable the target mode and continue in the same way as in touchlink with a light.

To process scan indication, the application should call the `N_LinkTarget_Subscribe()` function to subscribe to the scan indication event, like in the following example:

```
//Define a table of callback functions. The callback for the scan
//indication goes the first. Other indications may be also processed
//by the application
```

```
static const N_LinkTarget_Callback_t linkTargetCb =
{
  ScanIndication,             //The callback function for the scan indication
  IdentifyStartIndication,
  IdentifyStopIndication,
  NULL,
  JoinNetworkIndication,
  UpdateNetworkIndication,
  ReverseJoinIndication,
  -1
};
//Subscribe to the events
N_LinkTarget_Subscribe(&linkTargetCb);
```

The callback function itself should be defined as follows:

```
static void ScanIndication(N_InterPan_ScanRequest_t *pScanRequest)
{
  ...
}
```

The argument holds the parameters of the received scan request. To check if the scan request's sender is factory new, use the `pScanRequest->clsnInfo` field. If `pScanRequest.clsnInfo & N_INTERPAN_CLSN_INFO_FACTORY_NEW` equals 0 then the sender is not factory new. To check if the current device is factory new, check if the `N_DeviceInfo_IsFactoryNew()` function returns `TRUE`.

If, processing the scan indication, the controller's application finds out that it is factory new and the scan request's sender is not factory new, the application should stop scanning by calling the `N_LinkInitiator_StopScan()` function (without parameters) and wait to be added to the initiator's network.

B.13.2 Disabling Target Type

The controller that completes scanning and continues as the touchlink initiator should disable target's mode. The application should call the `N_LinkTarget_SetTargetType()` function again, this time passing `N_LinkTarget_TargetType_None` value as the first argument:

```
N_LinkTarget_SetTargetType(N_LinkTarget_TargetType_None, SetTargetTypeDone);
```

Once the target mode is disabled, the controller should proceed as the touchlink initiator, sending the join network request to the target via the `N_LinkInitiator_JoinDevice()` function (see Section B.10.6). ZLLPlatform will know that the target is a controller and will send the join end device request (if the controller is an end device) instead of the join router request sent to the light.

B.14 Joining a Home Automation Network

A ZLL device may join a non-ZLL network, namely, a Home Automation profile network. For this purpose, the application can use either core BitCloud functions or ZllPlatform functions. In the latter case the application first performs standard ZigBee scanning, discovering other devices, and then joins directly to the most suitable device. For scanning, the `N_Connection_AssociateDiscovery()` function is used. The function may be executed in several modes: discover networks with any PANID, with a particular PANID, or with all PANIDs except the specified one. Consider a code example:

```
//Define a variable for holding beacon data
static N_Beacon_t beacon;

//Define a callback function
```

```
static void assocDiscoveryDone(N_Connection_Result_t result)
{
  if (N_Connection_Result_Success == result) //A network has been found
    ... //More code
}

...
//Initiate scanning
N_Connection_AssociateDiscovery(&beacon,
                                N_Connection_AssociateDiscoveryMode_AnyPan,
                                NULL,    //Specific PANID - not needed when
                                         //searching for any networks
                                assocDiscoveryDone);
```

To join to the network, call the `N_Connection_AssociateJoin()` function, specifying the same beacon variable in the first argument, as in the following example:

```
//Callback function's definition
static void assocJoinDone(N_Connection_Result_t result)
{
  ...
}
...
//Join by association to the selected device
N_Connection_AssociateJoin(&beacon, assocJoinDone);
```

Note: In common BitCloud applications both discovery and join may be performed by a single API function of the ZDO component `ZDO_StartNetworkReq()`.

Also, make sure that the following settings are done:

- Channel mask should be first set to the primary channel mask, which is set by the `APP_PRIMARY_CHANNELS_MASK` parameter and equals `0x2108800` (marks channels 11, 15, 20, and 25)
- Both `CS_NWK_UNIQUE_ADDR` and `CS_NWK_PREDEFINED_PANID` are set to false to indicate that network PANID and short address are not predefined, but will be determined during network start. For details on setting ConfgServer parameters see Chapter 3

The transport key command received by the device during joining is decrypted with the help of HA trust center link key set inside the `N_Security` module of the ZLLPlatform layer.

A controller and a lighting device that belong to the same HA network and performed touchlink may use the HA network as infrastructure, passing messages through any router from this network. This way a controller may deliver a command to a lighting device even if the latter is not directly accessible for the controller.

Using standard ZigBee commissioning, ZLL device may also join a ZLL network, if one of the routers in this ZLL network permits joining temporarily (by default the permit joining flag should set to false on all devices in a ZLL network). Associated security keys are described in Section B.16. Since a device may encounter either an HA or a ZLL network while performs joining by association, the stack distinguishes between two types of networks and selects the appropriate link key to decrypt the transport key command.

B.15 Clusters and Addressing

After touchlink, devices in a Light Link network use cluster-specific commands for communication. Working with clusters including sending and processing commands is the same as for common profiles (for details refer to Sections 5.5 and 5.6).

Most of the clusters employed by the Light Link devices duplicate clusters already used by other profiles, but may require slightly different definitions. That is why the BitCloud ZigBee Cluster Library implementation includes separate versions for such clusters. The definitions files for these clusters are located in the `\Components\ZCL\include\` directory and have `Zll` in their name. For the lists of clusters supported by demo application refer to [2].

Service discovery and binding are not used. Commands are sent from a controller device, using group addressing, unless an individual device is selected.

A controller should add a lighting device to a group shortly after touchlink by sending Add Group command of the Groups cluster. By default, commands may be sent to the default group of devices (with addressing mode set to `APS_GROUP_ADDRESS`). The controller's application should provide a possibility to select a single lighting device, for example, upon button's press or other form of user's interaction. Once a user presses such button, the application should send a unicast identify command (sent with an `APS_SHORT_ADDRESS` addressing mode) to the next lighting device from the list of linked lighting devices. In response to the identify command, the lighting device shall indicate itself to the user (for example, by blinking). Then the application may send all commands requested by a user as unicast messages addressed to the selected device's short address until the controller is inactive for a given period of time or the selection button is pressed one more time and the next lighting device from the list is selected.

Group addressing is described in details in Section 5.7.2.2.

B.16 Security

ZLL applications use standard security with link keys described in Section 6.3. The link keys are used only for authentication during touchlink or network association. For ZCL data exchange in a ZLL network the APS payload is not encrypted and only the NWK payload is secured with the network key.

The security parameters used in the SDK are set in `N_Security_Calc.h` and `N_Security_Calc.c` files located in the `\ZLLPlatform\ZLL\N_Security\include\` directory inside ZLLPlatform. The parameters are described in Table B-2.

Table B-2. ZLL Security Parameters

Parameter	Description
`ZLL_DEVELOPMENT_KEY_INDEX`	Index of a development key (0)
`ZLL_MASTER_KEY_INDEX`	Index of a master key, which must be used by commercial ZLL devices
`ZLL_CERTIFICATION_KEY_INDEX`	Index of a certification key, which should be used during certification and may be used during development
`ZLL_SUPPORTED_SECURITY_KEYS_BITMASK`	Bitmask of supported security keys
`ZLL_SECURITY_KEY`	Specifies the key used to encrypt or decrypt the network key during touchlink commissioning
`ZLL_NWK_KEY`	The value of the test network key that is used instead of generating a network key value during development if the application sets the `APP_ZLL_FIXED_PAN` flag to 1 (see the `configuration.h` file of the demo application)
`ZLL_PREINSTALLED_LINK_KEY`	Specifies the key used to encrypt or decrypt the transport key command payload during classical ZigBee commissioning
`ZLL_DEFAULT_HA_TC_LINK_KEY`	Specifies the key used to decrypt the transport key command payload when joining a Home Automation network

B.16.1 Security in Touchlink Commissioning

A network key is generated randomly on a controller during touchlink commissioning. The touchlink target (either a light or a controller) receives the network key during touchlink commissioning. To secure network key transfer during touchlink commissioning the network key is encrypted with the help of the ZLL security key, either a certification or a master key. All operations related to the network key are maintained internally by the ZLLPlatform layer.

The keys are set in the `N_Security_Calc.c` file located at `ZLLPlatform\ZLL\Z_Security\src\` directory.

B.16.2 Security in Standard ZigBee Commissioning

A ZLL device may also use classical ZigBee commissioning to join a ZLL network (joining an HA network is described in Section B.14). By default, none of devices in a ZLL network permits joining by association. To make joining to a ZLL network possible, joining must be permitted on one of the routers, typically, by sending a permit joining ZDP request from a controller to a certain lighting device.

A ZDP request should be sent to the unicast address by calling the `ZDO_ZdpReq()` function. In the argument the `reqCluster` field is set to `MGMT_PERMIT_JOINING_CLID`, `dstAddrMode` to `SHORT_ADDR_MODE`, and `dstNwkAddr` to the target short address. In the request's payload, the `req.reqPayload.mgmtPermitJoiningReq` field, the `permitDuration` field should be set to the duration of permitting in seconds. The `0xff` value sets the duration to the upper bound, 11520 seconds. The `0x00` value prohibits joining.

During classical ZigBee commissioning the transport key command sent to a joining device is encrypted using the ZLL pre-installed link key set by the `ZLL_PREINSTALLED_LINK_KEY` parameter. During development and certification the certification value given by the ZLL specification is used. Commercial ZLL devices shall use the secret key distributed among certified manufacturers.

Appendix C. APS Data Exchange

Although application normally uses ZCL primitives for data exchange as described in Chapter 5. However ZCL implementation relies on the APS layer and application can also perform data transmission and reception directly on the APS layer as well. This section describes the APS interfaces implemented in BitCloud ZigBee PRO stack.

C.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.

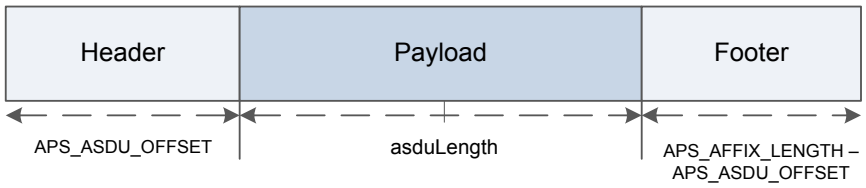
```
//Specify endpoint descriptor
static SimpleDescriptor_t simpleDescriptor = {1, 1, 1, 1, 0, 0, NULL, 0, NULL};
//variable for registering endpoint
static APS_RegisterEndpointReq_t endpointParams;…
//Set application endpoint properties
…
endpointParams.simpleDescriptor = &simpleDescriptor;
endpointParams.APS_DataInd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

C.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 C-1. See the ZigBee PRO specification [6] for more details on ASDU.

Figure C-1. ASDU Format



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

Table C-1. Maximum APS Payload Size for Different Stack Configurations

Option	Maximum ASDU size
No security	95
Standard security (encryption on NWK layer only)	77
Standard security with link keys (encryption on NWK and APS layers)	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]; // Heade
  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. Cluster binding is described in Section 5.8.

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)
```

C.3 Transmission Options

A data request is additionally configured through the `txOptions` field of the data request parameters that has `APS_TxOptions_t` bit field type. 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 C-2. Besides, following sections add comments on the use of txOptions where these transmission options are applicable.

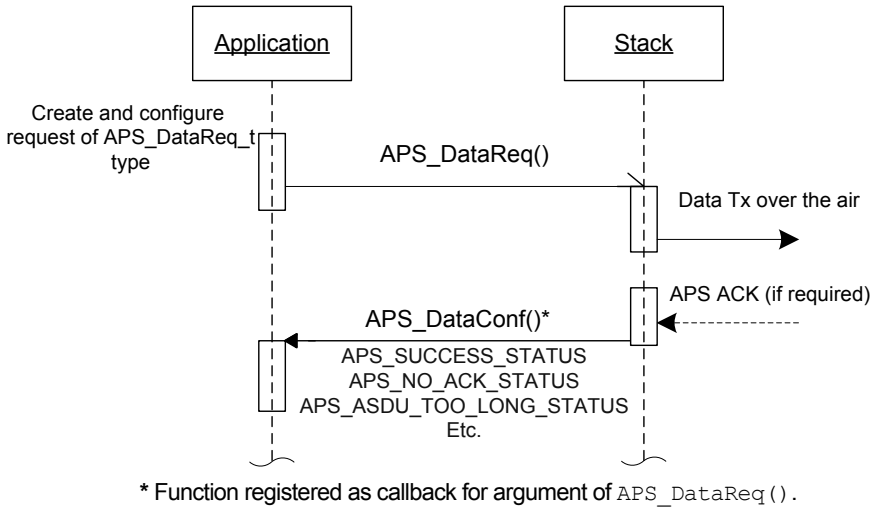
Table C-2. Fields of txOptions for Data Frame Configuration

Option	Description
`securityEnabledTransmission`	Indicates if the application payload shall be encrypted; see Section 6.3.3
`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 C.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 5.7.1.1
`noRouteDiscovery`	If the route to the destination is unknown do not start route discovery and report error

C.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 C-2 illustrates sequence diagram on the source node during data transmission.

Figure C-2. Data Transmission Sequence



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.

C.4.1 Retries in Case of Failure

A data frame sent from the application passes through APS to NWK to MAC. Each layer waits until the next underlying layer processes the data request and reports the status. If a layer does not receive frame delivery acknowledgement (for a unicast frame) in time it does not report the failure immediately to the next higher layer, but retries the transmission several times. Durations between successive retries increase.

The MAC layer is responsible for delivering a frame over the first hop, to the right neighbor node. MAC timeouts are measured in milliseconds. While transmitting a frame, MAC makes up to 3 retries (4 attempts in total), each time waiting for a reply from the neighbor node it addresses. All 4 attempts, following the IEEE 802.15.4 specification, occur during 30ms, and so the MAC layer reports to the NWK layer within 30ms to any data request.

The NWK does not perform retries for the frames initiated on the current node and, discovering an error status in the MAC response, directs it immediately to APS.

APS repeats transmission of a data frame if acknowledgement has been requested by the application but not received in time, or a failure status is received from lower layers for some reason. The maximum number of retries is set by the `CS_APS_MAX_FRAME_RETRIES` parameter. Timeouts for the retries are calculated by an optimized formula taking into account some network parameters. First time APS waits for an acknowledgement frame for about 1.5s; then the duration increases exponentially.

APS does not retry to send a data frame if one of the following statuses is received:

- `NWK_BAD_CCM_OUTPUT_STATUS (0xCE)`
- `NWK_MAC_TRANSACTION_EXPIRED_STATUS (0xF0)`
- `NWK_ROUTE_DISCOVERY_FAILED_STATUS (0xD0)`

C.4.2 Removing Routing Table Entries if too Many Failed Attempts

The NWK component keeps information about routes to specific nodes in the routing table. Each routing table entry includes the rate field, which shows how often data frames are sent successfully to this route (on the MAC layer). If transmission of a frame fails on the MAC layer several times (each one's including 3 retries) NWK removes this entry from the routing table.

Caution: If a destination node is also present in the neighbor table, routing may not be applied when sending a data frame to it. An entry in the neighbor table is preferred to the one from the routing table if the link cost in the neighbor table entry is lower. The neighbor table entry can also be removed – if the destination node loses connection to the network and the node does not receive link status frames from it for a certain time.

A routing table entry is also removed when the node receives a special command from a remote node that discovers that it does not know the route for a frame it should transfer (no corresponding entry in the routing table). If an entry exists, but a failure happens on the MAC layer on a remote node, this node decreases entry's rate. So if the initiator node will persist is sending frames through such route, the rate of the corresponding routing table entry on a remote node will eventually decrease to zero, the entry will be removed, and the remote node will send a no-such-route command to the initiator.

Once NWK removes a routing table entry, it reports error to APS. If the limit for retries on APS has not been reached, APS tries to send a data request one more time. This time NWK will find out that no corresponding entry exists in the routing table and will start route discovery.

C.5 Data Frame Reception

As described in Section C.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.

C.6 APS 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 rely on the 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.

C.6.1 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 Chapter 6.

Table C-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.

C.6.2 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 C.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.

C.6.3 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.

Appendix D. References

[1] AVR2050: BitCloud Developer's Guide (this document)
[2] AVR2052: Atmel BitCloud Quick Start Guide
[3] BitCloud API Reference (available in BitCloud SDK)
[4] AVR2054: Serial Bootloader User Guide
[5] AVR2058: BitCloud OTAU User Guide
[6] ZigBee PRO specification (05-3474r20)
[7] ZigBee Cluster Library specification (07-5123r04)
[8] ZigBee Light Link Profile specification (11-0037-10)
[9] ZigBee standards overview

Appendix E. Revision History

Doc. Rev.	Date	Comments
8199I	15.03.2014	Updated for BitCloud SDK v3.0.0. Integrated content of AVR2056 Profile Suite Developer Guide and AVR2061 BitCloud ZLL Developer Guide.
8199H	15.04.2012	Release for BitCloud SDK v1.14.0

Atmel | Enabling Unlimited Possibilities®

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

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

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

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