BitCloud

User Guide

8199B–MCU Wireless–02/09

Table of Contents

Section 1
Introduction

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

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

BitCloud's target audience is system designers, embedded programmers and hardware engineers evaluating, prototyping, and deploying wireless solutions and products. BitCloud is delivered as a software development kit, which includes (1) extensive documentation, (2) standard set of libraries comprising multiple components of the stack, (3) sample user applications in source code, as well as (4) a complete set of peripheral drivers for the supported platforms.

1.1 Intended Audience and Purpose

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

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

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

1.2 Key Features

- Full ZigBee PRO and ZigBee compliance
- Easy-to-use C API and serial AT commands available
- Ultimate in data reliability with true mesh routing
- Large network support (100s of devices)
- Optimized for ultra low power consumption (5-15 years battery life)
- Extensive security API
- Over-the-air software update capability
- Flexible and easy to use developer tools

1.3 Supported Platforms

Currently, different releases of BitCloud SDK support the following hardware platforms:

- SDK for ATAVRRZRAVEN:
  - AVRRAVEN and RZUSBSTICK provided as a part of ATAVRRZRAVEN kit; http://www.atmel.com/dyn/resources/prod_documents/doc8120.pdf
- SDK for ZigBit:
  - ATZB-DK-24 (ZDK)
- SDK for ZigBit Amp
  - ATZB-DK-A24 (ZDK Amp)
- SDK for ZigBit 900
  - ATZB-DK-900 (ZDK 900)

1.4 Related Documents

1. GNU 'make'. http://www.gnu.org/software/make/manual/make.html
2. make (software). http://en.wikipedia.org/wiki/Make_(software)
3. 053474r17ZB_TSC-ZigBee stack profile Pro Specification http://www.zigbee.org/en/spec_download/download_request.asp
4. AVR2052: BitCloud Quick Start Guide
5. AVR2051: BitCloud Stack Documentation

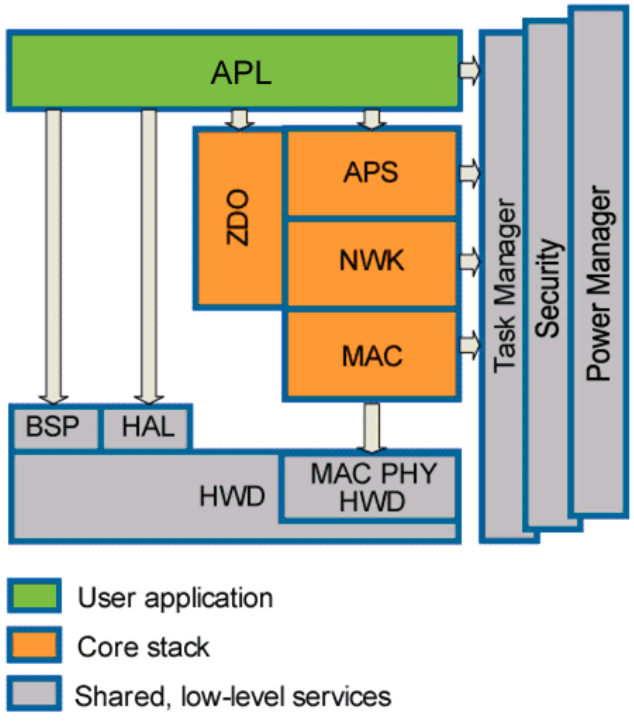
Section 2

BitCloud Architecture

2.1 Architecture Highlights

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, BitCloud contains additional layers implementing shared services (e.g. task manager, security, and power manager) and hardware abstractions (e.g. 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. BitCloud API reference manual provides detailed information on all public APIs and their use [5].

Figure 2-1. 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 API which enable main network management functionality (start, reset, formation, join). ZDO also defines ZigBee Device Profile types, device and service discovery commands implemented by the stack.

There are three service vertical components including: task manager, security, and power manager. These services are available to the user application, and may also be utilized by lower stack layers.

Task manager is the stack scheduler which mediates the use of the MCU among internal stack components and user application. The task manager implements a priority based co-operative scheduler specifically tuned for multi-layer stack environment and demands of time-critical network protocols. "Task scheduler, priorities, and preemption" on page 3-3 describes 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 design-in and smooth integration with a range of external peripherals (IRQ, TWI, SPI, UART, 1-wire).

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

2.2 Naming Conventions

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

1. Each API function name is prefixed by the name of the layer where the function resides. For example, ZDO_GetLqiRssi API is contributed by the ZDO layer of the stack.
2. Each function name prefix is followed by an underscore, _, separating the prefix from the descriptive function name.
3. The descriptive function name may have a Get or Set prefix, indicating requesting that some parameter is returned or setting that parameter in the underlying layer, respectively (e.g. HAL_GetSystemTime).
4. The descriptive function name may have a Req, Request, Ind, or Conf suffix, indicating the following:

    - Req and Request correspond to the asynchronous (See "Concurrency and interrupts" on page 3-4.) requests from the user application to the stack (e.g. APS_DataReq).
    - Ind corresponds to the asynchronous indication or events propagated to the user application from the stack (e.g. ZDO_NetworkLostInd).
    - By convention, function names ending in Conf are the 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 macros variable names are in capital letters.

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

2.3 File System Layout

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

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

Table 2-1. BitCloud SDK file system layout

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

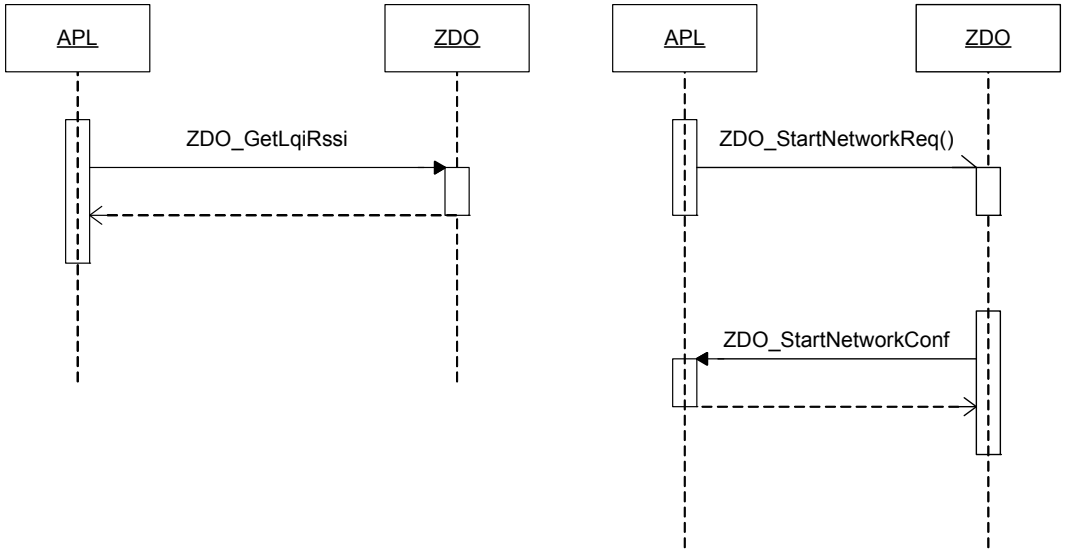
Section 3

User Applications Programming

3.1 Event-driven programming

Event-driven systems are a common programming paradigm for small-footprint embedded systems with significant memory constraints and little room for the overhead of a full operating system. Event-driven or event-based programming refers to programming style and architectural organization which pairs each invocation of an API function with an asynchronous notification (and result of the operation) of the function completion 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 when the request is serviced.

Figure 3-1. Synchronous vs. Asynchronous calls.



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

3.2 Request/confirm and indication mechanism

All applications based on the BitCloud SDK are written 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. There is a generic type of user-defined callback which is responsible for executing application-level code called the task handler. APL_TaskHandler is the reserved callback name known by the stack as the application task handler. Invocation of the APL_TaskHandler is discussed in "Task scheduler, priorities, and preemption" on page 3-3.

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

For example, consider ZDO_StartNetworkReq(&networkParams) request call, which requests ZDO layer to start the network. The networkParams argument is a structure defined in zdo.h as ZDO_StartNetworkReq_t:

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

The first field is a structure used to the stack's response (actual network parameters) to the request. The last field is the actual callback pointer. The ZDO_StartNetworkReq(&networkParams) request is paired up with a user-defined callback function with the following signature:

```
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confirmInfo)
{
}
```

Therefore actual request is preceded by an assignment to the callback pointer as follows:

```
networkParams.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
```

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

Note that the need to decouple the request from the answer is especially important when the request can take an unspecified amount of time. For instance, when requesting the stack to start the network, the underlying layers may perform an energy detecting scan which takes significantly longer than we are willing to block for. "Concurrency and interrupts" on page 3-4 and "Typical application structure" on page 3-6 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 which is not a reply to any specific request. For this, there is a number of user-defined callbacks with fixed names which are invoked by the stack asynchronously. These include events indi-

cating loss of network, readiness of the underlying stack to sleep, or notifying that the system is now awake.

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

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

3.3 Task scheduler, 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 coordinates, 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), the commodity operating system core like Windows are typically very complex in comparison to single-task systems like BitCloud. In BitCloud context, there is a single application running on top of the stack, thus most of 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 timeslice among different applications (multiplexing them transparently on one CPU) so that the 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 aware of the need to yield the resources that it's using (primarily, the processor) to the underlying stack. Another benefit compared to the preemptive Operating System is that only one stack is used. That saves a considerable amount of data memory.

Returning to the example with user callbacks, if 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, so ZDO_StartNetworkConf runs under ZDO's priority level. Users should exercise caution when executing long sequences of instructions, including instructions in nested function calls, in the scope of a callback invoked from another layer.

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

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

The strategy for callbacks executing longer than 10 ms is to defer execution. Deferred execution is a strategy to 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);
```

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 under the application's priority level. In other

words, the application task handler runs only when all higher priority tasks have completed. This permits longer execution time in a task handler's context versus a callback context.

System rule 5: Application task handler runs only when all tasks of higher priority have completed.

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

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

3.4 Application timers

Thus far, the three ways that control flow enters application code are: (1) through task handler following a SYS_PostTask invocation, (2) through confirm callbacks invoked by underlying stack on request completion, and (3) through asynchronous event notifications invoked by the stack. Note that neither one of the three has a time bound for when the invocation is to be expected. One way to ensure execution of a user-defined callback at a specific time in the future is by using a timer.

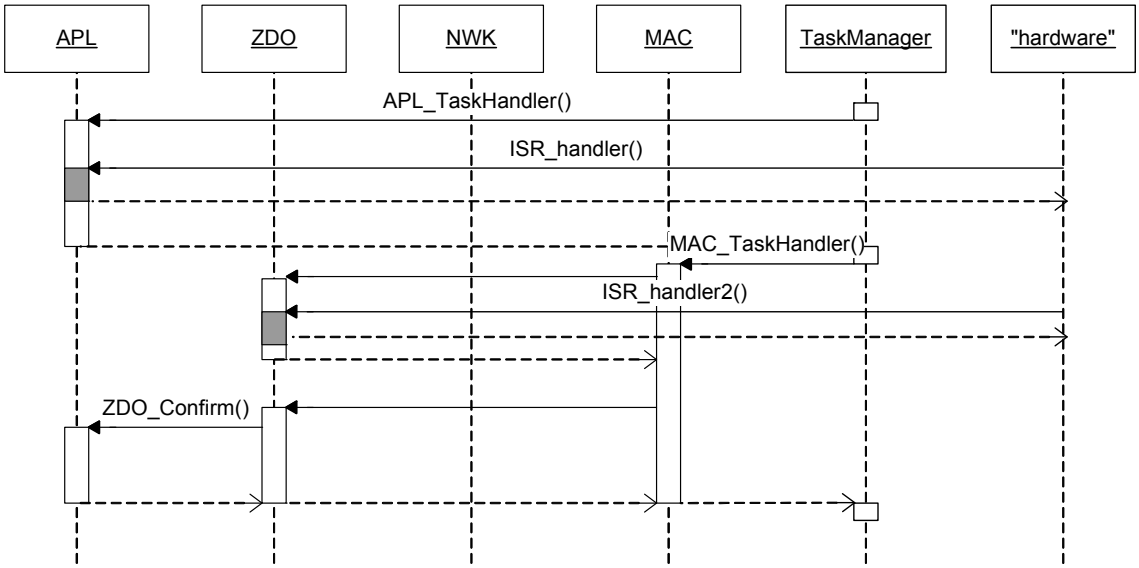
The stack provides a high-level application timer interface, which uses a low-level hardware timer. The timer interface is part of HAL and its use is illustrated in many sample applications (see BlinkApp.c in Blink folder for more detail). The general idea is that a HAL_AppTimer_t structure defines the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

3.5 Concurrency and interrupts

Concurrency refers to several independent threads of control executing at the same time. In preemptive systems with timeslicing 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 unpredictability of interruption and the fact that the two functions may share data, the application developer must ensure atomic access to all shared data.

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

Figure 3-2. Normal and interrupt control flow in the stack.



Even though timeslicing 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 (e.g. UART or SPI channel). This is analogous to handling hardware interrupts in any other system.

Figure 3-2 on page 3-5 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 interrupt service routine or interrupt handler. After the interrupt handler completes, the control is returned to the application task handler. Once the task handler finishes, the control is returned to the scheduler, which selects a MAC layer task to run next. While the MAC_TaskHandler is running, it invokes a confirm callback in 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 which will execute on 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 an user-defined interrupt handler is:

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

The 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 an execution of another application callback. If the interrupt handler accesses global state also accessed by any of the application callbacks than access to that share state must be made atomic. Failure to provide atomic access can lead to data races, i.e. non-deterministic code interleavings 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's 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 the AVR microcontroller, flags are used, so interrupts arriving during the masking will be saved.

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

3.6 Typical application structure

A BitCloud application differs significantly in its organization from a typical non embedded C program. As discussed previously,

1. Every application defines a single task handler, which contains in its scope the bulk of the application's code (including code accessible through nested function calls).
2. Every application defines a number of callback functions contributing code which executes when an asynchronous request to the underlying layer is serviced.
3. Every application defines a number of callbacks with known names which execute when an event is processed by the stack.
4. Every application maintains global state that is shared state information between the callbacks and the task handler.

The main function is embedded in the stack itself. Upon the stack initialization, the control passes from main to the task scheduler which begins invoking task handlers in order of priority (from highest to lowest) eventually invoking APL_TaskHandler, which is the initial entry point into the application. Following the initial call to the application task handler, the control flow passes between the stack and the callbacks as shown in Figure 3-2 on page 3-5.

Application code may be arbitrarily split up among several C files (see WSNDemo application and Makefile as an example). For simplicity's sake, we consider here a standard single file application omitting large portions of the code for illustration purposes.

```
/******************************
  #include directives
******************************/
...
#include <taskManager.h>
#include <zdo.h>
#include <configServer.h>
#include <aps.h>
/******************************
 FUNCTION PROTOTYPES
******************************/
...
/******************************
  GLOBAL VARIABLES
******************************/
AppState_t appState = APP_INITING_STATE;
...
/******************************
 IMPLEMENTATION
******************************/
/******************************
```

```
 Application task handler
*****************************/
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;
    }
}
/*****************************
 Confirm callbacks
*****************************/
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confirmInfo)
{
    ...
    if (ZDO_SUCCESS_STATUS == confirmInfo->status) {
        appState = APP_IN_NETWORK_STATE;
        ...
        SYS_PostTask(APL_TASK_ID);
    }
}
void ZDP_LeaveResp(ZDP_ResponseData_t *zdpRsp)
{
    ...
}
/*****************************
 Indication callbacks
*****************************/
void ZDO_NetworkLostInd(ZDO_NetworkLostInd_t *indParams)
{
    ...
    if (APP_IN_NETWORK_STATE == appState) {
        appState = APP_STARTING_NETWORK_STATE;
        ...
        SYS_PostTask(APL_TASK_ID);
    }

}

void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *nwkParams)
{
    ...
}
```

The above skeleton code is representative of the vast majority of user applications.

Invariably, there is a global state, represented in our case by appState variable that is accessed by callbacks and the task handler. In real-world applications, state is represented by a number of variables for device role-dependent sub-states, various network parameters, sensor state, etc. Note that a typical

task handler switches on a state variable to execute the code for each particular state, which is a programming template shared by all sample applications provided with the SDK. The application developer is well-served by considering their application first in terms of a state machine. The mapping from a state machine description to code is then natural given the event-driven programming style.

Note also the use of SYS_PostTask scheduler function. For example, the application purposefully defers processing of a network lost indication to the task handler, where it is dealt with fully. The callback simply changes the global state and returns to the ZDO layer from which it is invoked. This style of programming is consistent with cooperative multitasking system setup, and it permits the stack to handle higher priority tasks before the returning to the deferred action.

Section 4

ConfigServer Interface

The BitCloud stack provides an extensive set of configuration parameters which determine different aspects of network and node behavior. These parameters are accessible for application via Configuration Server interface (ConfigServer, CS for short). This chapter gives only a brief introduction into the CS interface, while complete list and description of CS parameters can be found in BitCloud API documentation [5].

All CS parameters can be divided into two categories: persistent and non persistent. Persistent parameters are stored in power independent EEPROM memory and their values are accessible for application and the stack after node HW reset. Non persistent parameters are stored in RAM memory and upon HW reset are reinitialized with their default values. configServer.h file includes comments next to parameter ID indicating whether certain CS parameter is persistent or not.

4.1 Reading/writing CS parameters

configServer.h file contains definitions of all CS parameters with their default values.

However, if necessary application is able to assign new values to CS parameters using any or both of the methods described in details below:

- Definition in application Makefile
- CS read/write functions

4.1.1 CS parameter definition in Makefile

The simplest method to assign a value to a CS parameter is to define it in the Makefile of the application. In such case default value assignment in configServer.h file is skipped and value assigned in the Makefile is applied to the CS parameter. As an example; the following line added in Makefile of any sample application sets RF output power to 3 dBm:

```
CFLAGS += -DCS_RF_TX_POWER=3
```

In BitCloud sample applications CFLAGS are automatically initiated during project build procedure so adding just a line as shown above is sufficient to assign desired value to the parameter.

Although described method is fairly simple it allows parameter configuration only at compile time and does not support run-time modifications.

4.1.2 CS Read/Write functions

In order to perform parameter read/write procedure at run-time ConfigServer provides corresponding API functions: CS_ReadParameter and CS_WriteParameter. Both functions require parameter ID and a pointer to parameter value as arguments. Parameter ID identifies which CS parameter the function is applied to and is constructed by adding "_ID" at the end of CS parameter name. It is important that sec-

ond argument points to a global variable, because stack doesn't allocate memory for the parameter and hence application shall take care about allocating and preserving such memory. The best and simplest approach is to use global variables. When reading ConfigServer parameters, local variables can be used.

Code below gives an example how application can read and write RF output power (CS_RF_TX_POWER parameter):

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

Section 5

Networking overview

As already described in “Architecture Highlights” on page 2-1, the BitCloud architecture follows the structure of the ZigBee PRO specification which allows applications to interact directly only with APS and ZDO components of the core stack. Such approach significantly simplifies application development as well guarantees that application has no impact on networking protocol and hence always behaves compliant to the ZigBee PRO specification.

This chapter gives brief introduction into main networking features of BitCloud stack as well as describes corresponding interactions between application and core stack API.

5.1 Network formation and join

In ZigBee PRO and ZigBee standards only coordinator is capable to create a new network. Router and end devices can only join an existing network already formed by the coordinator. Further "network start" term is used if from application point of view there is no difference between network formation and network join procedures. BitCloud application shall perform network start procedure in 3 steps which are described in details in this section:

1. Configuring network parameters
2. Network start request
3. Network start confirmation.

5.1.1 Network start parameters

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

1. Supported modulation scheme, so called channel page (CS_CHANNEL_PAGE)
2. Supported frequency channel, specified via 32-bit channel mask (CS_CHANNEL_MASK)
3. 64-bit Extended PAN ID (CS_EXT_PAN_ID)
4. Security parameters (See “Security” on page 5-12.)

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

CS_CHANNEL_PAGE defines the modulation type to be used by the device. This parameter is valid only for 868/916 MHz bands and is ignored for the 2.4 GHz band.

CS_CHANNEL_MASK is the 32-bit field which determines the frequency channels supported by the node. The 5 most significant bits (b31,..., b27) of channel mask shall be set to 0. The rest 27 bits (b26, b25,...b0) indicate availability status for each of the 27 valid channels (1 = supported, 0 = unsupported).

Table 5-1 on page 5-2 shows channel distribution among IEEE 802.15.4 frequency bands, as well as provides data rates on the physical level for different channel pages.

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

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

Example 1: Coordinator's transceiver operates in the 2.4 GHz band and shall form a network only on channel 17 (decimal).

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

b) Channel mask for such case shall be set as CS_CHANNEL_MASK =

$= 0_{31}0_{30}0_{29}0_{28}0_{27}0_{26}0_{25}0_{24}0_{23}0_{22}0_{21}0_{20}0_{19}0_{18}1_{17}0_{16}0_{15}0_{14}0_{13}0_{12}0_{11}0_{10}0_{9}0_{8}0_{7}0_{6}0_{5}0_{4}0_{3}0_{2}0_{1}0_{0} =$
$= 0x00020000$

Example 2: Coordinator operates in 915 MHz band and should support 250 kbps data rate. Channels 3, 4, and 9 shall be enabled for network start.

a) CS_CHANNEL_PAGE=2

b) CS_CHANNEL_MASK=

$= 0_{31}0_{30}0_{29}0_{28}0_{27}0_{26}0_{25}0_{24}0_{23}0_{22}0_{21}0_{20}0_{19}0_{18}0_{17}0_{16}0_{15}0_{14}0_{13}0_{12}0_{11}0_{10}1_{9}0_{8}0_{7}0_{6}0_{5}1_{4}1_{3}0_{2}0_{1}0_{0} =$
$= 0x000000218$

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

During network formation coordinator selects another network identifier: 16-bit (network) PAN ID which is used in frame headers during data exchange instead of heavy 64-bit extended PANID. By default network PANID is generated randomly and if coordinator during network formation detects another network with same extended PANID it automatically selects different network PANID in order to avoid conflicts during data transmissions.

However such mechanism leads to sometimes undesired behavior: if coordinator node is reset and initiates network start again with the same extended PANID and on the same channel, it may find routers from his previous network present there and hence will form a new network with different network PANID. But often it is required that coordinator joins the same network where it was before and hence can participate in data exchange. In order to force coordinator to do so, network PANID shall be predefined on application level prior to network start as shown in example below:

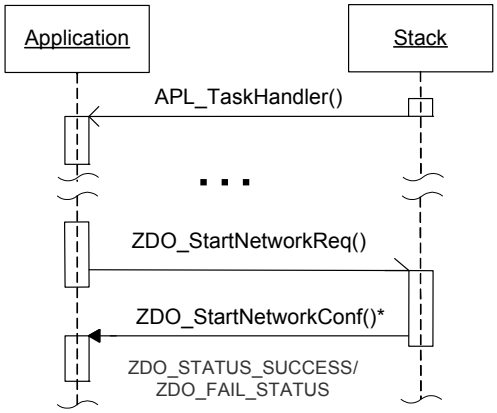
```
bool predefPANID=true; //global variable
uint16_t nwkPANID=0x1111; //global variable
...
CS_WriteParameter(CS_PREDEFINED_PAN_ID_ID, &predefPANID);
CS_WriteParameter(CS_NWK_PANID_ID, &nwkPANID);
```

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

5.1.2 Network start request and confirm

The BitCloud application is fully responsible for initiating the network start procedure. "Network start request and confirm" on page 5-3 shows the sequence diagram for network start procedure which is the same for all types of devices. Once network parameters described in "Network start parameters" on page 5-1 are configured properly, the application can initiate network start procedure by executing asynchronous call ZDO_StartNetworkReq(). After finishing the network start/join procedure the ZDO component informs the application about the result according to registered callback function with argument of ZDO_StartNetworkConf_t type that contains status of the performed procedure as well as information about started network, obtained network address for the node, etc. Status ZDO_STATUS_SUCCESS is received if the procedure is executed successfully while status equal ZDO_FAIL_STATUS means that network start has failed. Detailed info on ZDO_StartNetworkConf_t can be found in [5].

Figure 5-1. Network start sequence diagram.



* Function registered as callback for argument of `ZDO_StartNetworkReq()`

After an end device joins the network it parent node receives a notification via ZDO_MgmtNwkUpdateNotf() function with argument having status field set to ZDO_CHILD_JOINED_STATUS (0x92). Child node extended and network addresses are returned as argument fields as well. However such notification is not issued upon router network join events because there is no dedicated parent for router nodes.

5.2 Parent loss and network leave

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

1. Parent loss. Since in mesh topology routers do not have dedicated parent nodes, such scenario is valid only for end devices.
2. node is requested to leave the network. Such a request can be issued by two sources:

   a) by the application running on the node

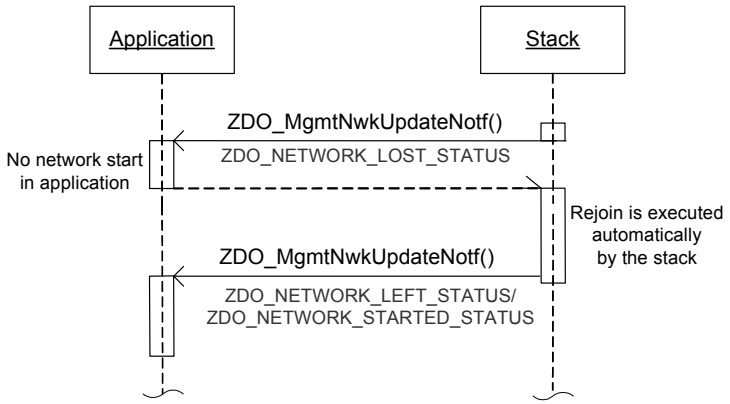
   b) by a remote node

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

5.2.1 Parent loss by end device

Figure 5-2 on page 5-4 shows the notification messages issued by the stack on an end device if the connection to the parent node is lost. First notification with status ZDO_NETWORK_LOST_STATUS is issued via ZDO_MgmtNetworkUpdateNotf() function when the end device cannot reach its current parent node. After receiving this message the application may perform some actions but shall not initiate a network rejoin procedure. It will be started automatically when stack thread receives control. Stack tries to find a new parent for the node to enter the network again. The result is reported to the application via ZDO_MgmtNwkUpdateNotf() function. Status ZDO_NETWORK_STARTED_STATUS means that node has successfully rejoined the network with network parameters indicated in argument of ZDO_MgmtNwkUpdateNotf(). If the network rejoin procedure has failed the application receives notification with status ZDO_NETWORK_LEFT_STATUS. After this, the application is responsible for changing network parameters if necessary and initiating a new network start procedure as described in “Network formation and join” on page 5-1.

Figure 5-2. Network start sequence diagram.



5.2.1.1 Child loss notification

It is often important for a parent node to be able to register when its child is lost, i.e. is out of the network. As mentioned above because only end devices can be associated as child nodes, such notification will not be triggered on a router node if another router is turned off or is out of signal reach.

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

So in order to be sure that child node is out of the network and not just in sleep mode, parent node shall know about end device's sleep period length (i.e. have the same CS_END_DEVICE_SLEEP_PERIOD as child nodes). Parent implies that its child periodically wakes up and issues poll request (see Section 5.3.4.1). So if during time interval 3*(CS_END_DEVICE_SLEEP_PERIOD + CS_INDIRECT_POLL_RATE) a child doesn't deliver any poll frames parent node assumes that child has left it and issues ZDO_MgmtNwkUpdateNotf() function with status ZDO_CHILD_REMOVED (0x93). In its argument extended address of the child node is indicated.

5.2.2 Network leave

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

```
static ZDO_ZdpReq_t zdpLeaveReq;
…
//set corresponding cluster ID
zdpLeaveReq.reqCluster = MGMT_LEAVE_CLID;
zdpLeaveReq.dstAddrMode = EXT_ADDR_MODE;
zdpLeaveReq.dstExtAddr = 0; // for own node address shall be 0
zdpLeaveReq.ZDO_ZdpResp = ZDO_ZdpLeaveResp; // callback

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

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

If in the code above destination address is set to address of a remote node then after calling ZDO_ZdpReq(&zdpLeaveReq), node will transmit a network command frame to the destination node requiring it to leave the network.

Sequence diagram for network leave procedure is shown on Figure 5-3 on page 5-6 (node leaves network on its own request) and Figure 5-4 on page 5-6 (node leaves network upon remote request).

Figure 5-3. Network leave sequence diagram (local call).

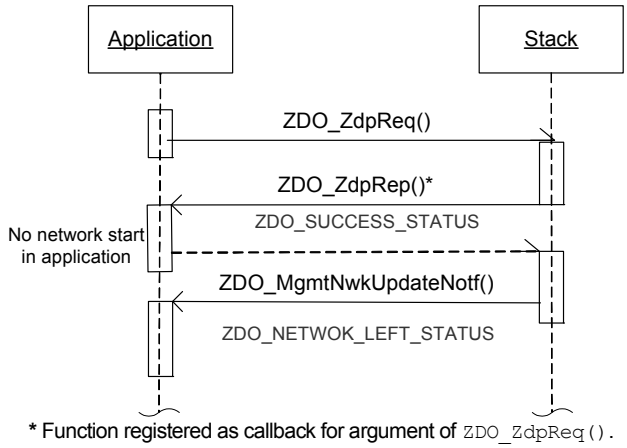
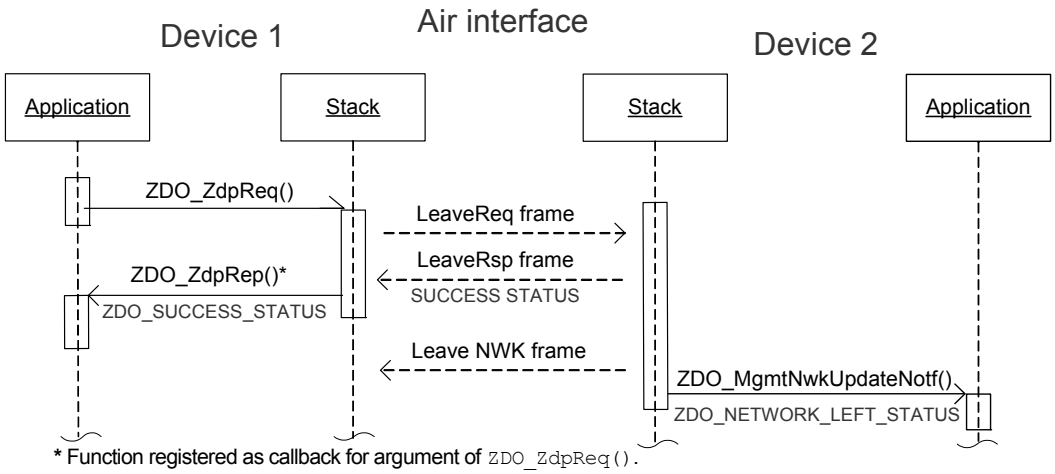


Figure 5-4. Network leave sequence diagram. Device1 requests Device2 to leave Network.



5.3 Data exchange

Obviously the purpose of establishing a ZigBee network as described in "Network formation and join" on page 5-1 is to perform data exchange between remote nodes as required by application functionality. This section gives an overview about node configuration, packet transmission parameters and BitCloud API functions responsible for communication on application level.

5.3.1 Application endpoint registration

Within ZigBee the application data is exchanged between so called application endpoints that represent certain applications on each device. In order to enable communication on application level each node shall register at least one endpoint using APS_ReqisterEndPoint() function with an argument of APS_RegisterEndpointReq_t type. The argument specifies endpoint descriptor (simpleDescriptor field) which includes such parameters as endpoint ID (a number from 1 to 240), application profile ID, number and list of supported input and output clusters. In addition APS_DataInd field specifies indication call-back function which will be called upon data reception destined for this endpoint.

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

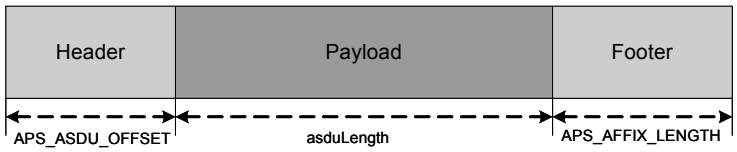
```
// Specify endpoint descriptor
static SimpleDescriptor_t simpleDescriptor = {1, 1, 1, 1, 0, 0, NULL, 0, NULL};
// variable for registering endpoint
static APS_RegisterEndpointReq_t endpointParams;…
// Set application endpoint properties
…
endpointParams.simpleDescriptor = &simpleDescriptor;
endpointParams.APS_DataInd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

5.3.2 ASDU configuration

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

Because the stack requires the ASDU as a contiguous block in the RAM memory with specific characteristics, the application shall construct such structure as shown in Figure 5-5 on page 5-7. See also [3] for more details on ASDU.

Figure 5-5. ASDU format.



Maximum allowed application payload size is limited to 84 bytes for unsecured transmission and to 53 bytes if the standard security mechanism is turned on as described in"Security" on page 5-12.

Code extract below provides an example, how to create a data request with correctly structured ASDU:

```
// Application message buffer descriptor
BEGIN_PACK
typedef struct
{
  uint8_t header[APS_ASDU_OFFSET]; // Header
  uint8_t data[APP_ASDU_SIZE]; // Application data
  uint8_t footer[APS_AFFIX_LENGTH - APS_ASDU_OFFSET]; //Footer
} PACK AppMessageBuffer_t;
END_PACK
static AppMessageBuffer_t appMessageBuffer; // Message buffer
APS_DataReq_t dataReq; // Data transmission request
…
dataReq.asdu = appMessageBuffer.data;
dataReq.asduLength = sizeof(appMessageBuffer.data);…
```

If direct addressing scheme is used then source node shall have full knowledge about destination of data packet, i.e. node network address, application profile ID, application endpoint and supported input clusters. In indirect addressing scheme these parameters are set during binding procedure.

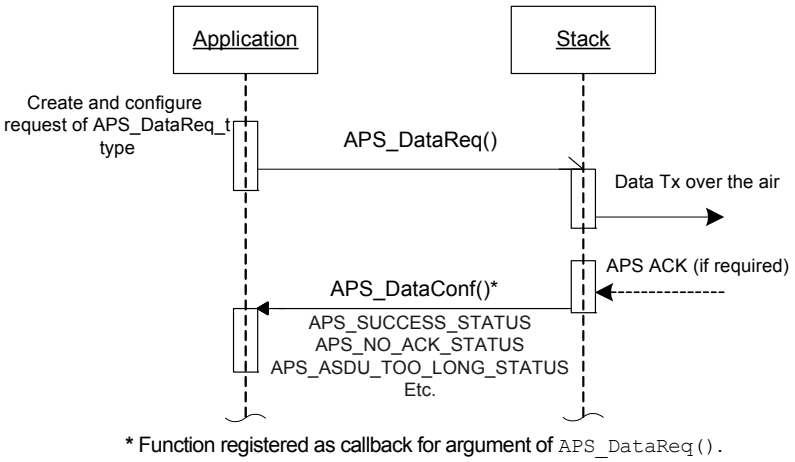
More details about structure of APS_DataReq_t can be found in [5].

5.3.3 ASDU transmission

After a data request is created and all parameters are set as required, the application can transfer it to APS layer to perform actual transmission over the air using APS_DataReq() function.

Figure 5-6 on page 5-8 illustrates sequence diagram on the source node during data transmission.

Figure 5-6. Data transmission.



If the application sends data to the destination for the first APS_DataReq() will automatically perform route request and find most reliable path to desired destination. Moreover, during the time this path is kept updated taking into account possible changes in network topology and link qualities. The application can set the maximum number of hops allowed for transmission of the frame (and hence limit the latency) by setting corresponding number in 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 in txOptions field of data request to 1. In such 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 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 such case if all parameters are set correctly confirmation callback with APS_SUCCESS_STATUS is called after transceiver has sent frame over the air.

instance of APS_DataReq_t can be reused only after the corresponding confirmation callback is executed.

5.3.3.1 Broadcast transmission

In addition to unicast (node-to-node) transmissions, applications can send data in broadcast manner where data frames are destined for all network nodes or nodes with specific properties. Following predefined enumerators can be used by the BitCloud application as destination address for a broadcast message (corresponding hexadecimal values can be set instead as well):

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

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

On network level, the broadcast procedure is performed as follows: after the APS_DataReq() function with broadcast data request is called, the transceiver sends the frame 3 times over the air. Each node after receiving a copy of this frame (only one copy is accepted, others are ignored) decreases transmission radius by one and if it is greater than zero broadcasts the message three times to its neighbors. Such procedure repeats on other nodes until transmission radius is exhausted.

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 dstEndpoint field in data request to APS_BROADCAST_ENDPOINT (or 0xFF). Such broadcast on node level can be performed for unicast transmissions as well.

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

5.3.4 ASDU reception

As described in “Application endpoint registration” on page 5-6 in order to enable data exchange on application level, the node shall register at least one application endpoint with an indication callback function for data reception procedure.

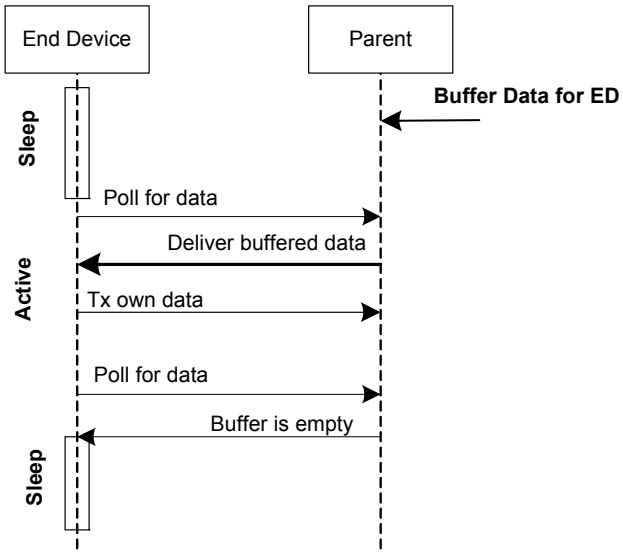
After a frame destined for the node is received by the transceiver, the stack verifies whether the destination endpoint indicated in the frame header has a corresponding match among endpoints registered on the node. In case such endpoint exists, corresponding indication function, as specified in APS_DataInd field of endpoint registration request (APS_RegisterEndpointReq_t) will be executed with argument of

APS_DataInd_t type. This argument contains application message (asdu field) as well as information about source and destination address, endpoint, profile, etc.

5.3.4.1 End device polling

In ZigBee network a polling mechanism is used to deliver data to end devices over the last hop (i.e. between parent node and child node). Figure 5-7 on page 5-10 shows general principle of polling mechanism.

Figure 5-7. End device polling mechanism.



Upon frame reception destined for its child node or broadcast frame with non-exhausted transmission radius and destination address equal 0xFFFF, parent node buffers the frame and waits for poll request from the child.

In awake mode end device polls its parent node periodically every CS_INDIRECT_POLL_RATE ms. Note that polling is done on the network layer and is transparent to application. After receiving data request from end device parent node shall transmit a single buffered frame and indicate whether there're more frames to be transmitted or not so that end device issues data request after reception of current frame is finished. When buffered data is retrieved from the parent node, end device can switch to sleep mode.

5.4 Power management

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

Following ZigBee PRO standard BitCloud supports power management mechanisms on end device nodes only. To avoid issues in network stability router and coordinator nodes shall be always in active mode.

5.4.1 Active and Sleep modes for end device nodes

Independent on its networking status (joined network or not) an end device node can be either in active or in sleep mode.

After being powered up a node always starts in active mode and has its MCU completely turned on and RF chip ready to perform Rx/Tx operations. Application can call any BitCloud commands and will receive registered callbacks and notifications.

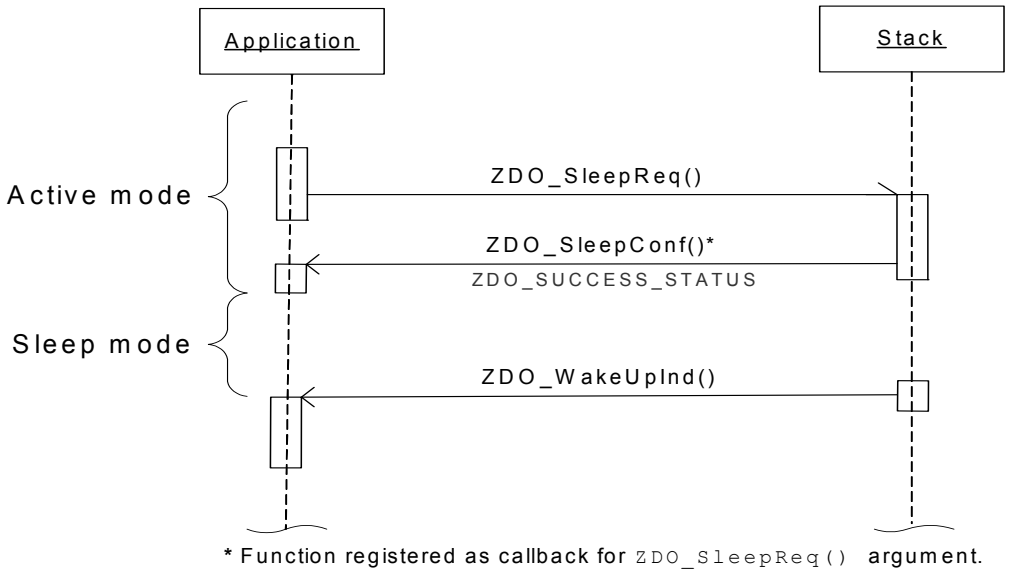
In sleep mode RF chip is turned off while MCU is put in a special low power state. In BitCloud stack only functionality responsible for MCU and radio wake ups is active. Thus application cannot perform any radio Tx/Rx operations, communicate with external periphery, etc.

In order to put end device node into the sleep mode application shall call ZDO_SleepReq() function with argument of ZDO_SleepReq_t type. After that registered confirmation callback will indicate the execution status and if ZDO_SUCCESS_STATUS is returned node will enter the sleep mode.

There are two ways to wake up node (i.e. to switch from sleep to active mode): scheduled and IRQ-triggered.

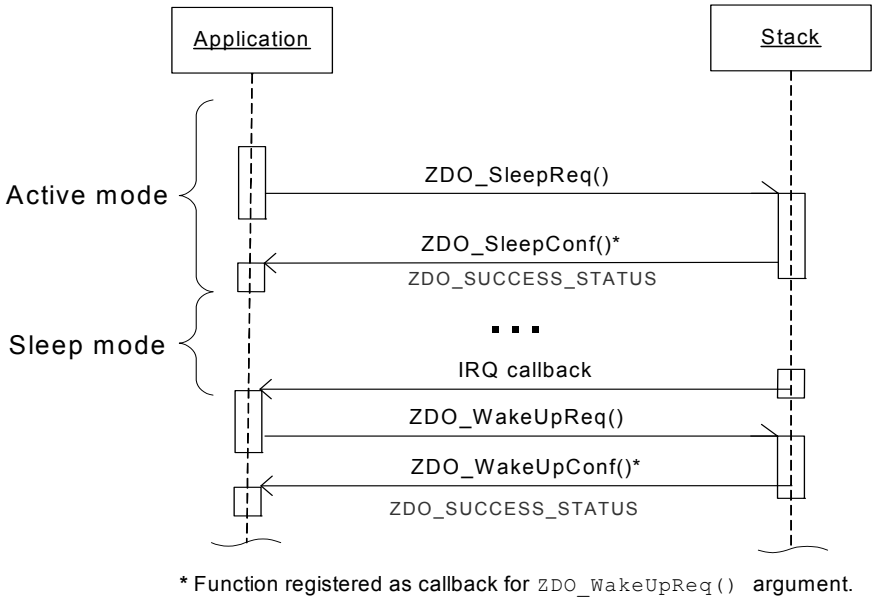
In scheduled approach node wakes up automatically after time interval in milliseconds specified in ConfigServer component as CS_END_DEVICE_SLEEP_PERIOD. Application is notified about the switch to active mode via ZDO_WakeUpInd function. Please note that CS_END_DEVICE_SLEEP_PERIOD shall not be modified at run-time. Figure 11 shows sequence diagram of stack calls for sleep and scheduled wake up procedures.

Figure 5-8. Scheduled wake up



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

Figure 5-9. IRQ-triggered wake up



* Function registered as callback for `ZDO_WakeUpReq()` argument.

Scheduled and IRQ-triggered methods can be combined in a single application to manage power consumption. If CS_END_DEVICE_SLEEP_PERIOD parameter is set to zero then only IRQ-triggered wake up can switch node from sleep to active mode.

5.4.2 Turning off RF chip only

In several scenarios it might be required to keep MCU operating (e.g. for data processing purpose) while RF chip shall be turned off to reduce power consumption. Such case is supported in BitCloud stack via possibility to turn off polling mechanism on end device (see Section Error! Reference source not found.) and hence RF chip as well. In order to turn radio off application shall set CS_AUTO_POLL parameter to false and in order to enable data polling it shall be set to true.

5.5 Security

Note: BitCloud for ATAVRRZRAVEN in this release supports only the **no security** option.

BitCloud provides two levels of security in a ZigBee network: no security and standard network level security. Hence there are two types of libraries located in the lib/ directory: one without security and ones that include security support. The choice of a particular library shall be done in the Makefile when compiling an application. In the BitCloud sample application, such choice is done based on the values of the SECURITY_MODE parameter.

If security support is required, in addition to correct libraries included, the CS_ZDO_SECURITY_STATUS and CS_NETWORK_KEY parameters shall be configured to enable proper network operation.

- CS_ZDO_SECURITY_STATUS=0. All devices in a network must have the same preconfigured network key (CS_NETWORK_KEY) defined in the Makefile.
- CS_ZDO_SECURITY_STATUS=3. One of the devices in a network (usually a coordinator) must be configured as Trust Center, i.e. to have network key (CS_NETWORK_KEY) defined in the Makefile. All other devices in the network must be provided with CS_APS_TRUST_CENTER_ADDRESS to be able to establish connection with the Trust Center, but must have no CS_NETWORK_KEY defined.

Note that switching on security option enlarges the program memory size, so that for some programs it may become too large to fit into the available memory. In such case, decreasing such parameters as CS_NEIB_TABLE_SIZE, CS_ROUTE_TABLE_SIZE, CS_MAX_CHILDREN_AMOUNT and other parameters that impact memory allocation as described in Section 6.

As mentioned in “ASDU transmission” on page 5-8, maximum application payload for encrypted frames are 53 bytes.

Section 6

Hardware Control

In addition to the ZigBee networking functionality described in Section 5, the BitCloud API also provides an extensive support of common HW interfaces such as UART, TWI, SPI, ADC, GPIO, IRQ, etc., Hardware Abstraction Layer (HAL) component of BitCloud is responsible for all interactions between Atmel modules and external periphery.

This chapter gives a brief overview over main HW interfaces generally supported in BitCloud.

Note: This section provides a future reference to the external interfaces that can be supported by different releases of BitCloud, the current release of BitCloud for ATAVRRZRAVEN doesn't support them.

6.1 UART bus

Note: UART bus is not supported in HAL for the current release of BitCloud for ATAVRRZRAVEN.

The BitCloud stack supports two channels of Universal Asynchronous Receiver/Transmitter (UART) interface, identified as UART_CHANNEL_0 and UART_CHANNEL_1.

In order to enable communication over the UART interface, the application shall first configure the corresponding UART port using static global variable of HAL_UartDescriptor_t type. It requires setting of all common UART parameters such as synchronous/asynchronous mode, baudrate (see HW platform datasheet for maximum supported value), flow control, parity mode, etc. In addition data reception and transmission over UART can be separately configured for operation either in callback or in polling mode as described in sections below. Detailed structure of HAL_UartDescriptor_t is given in BitCloud Stack Documentation [5].

UART settings shall be applied using HAL_OpenUart()function. Returned value indicates whether port is opened successfully and can be used for data exchange.

When there is no more need in keeping UART port active application shall close it using HAL_CloseUart() function.

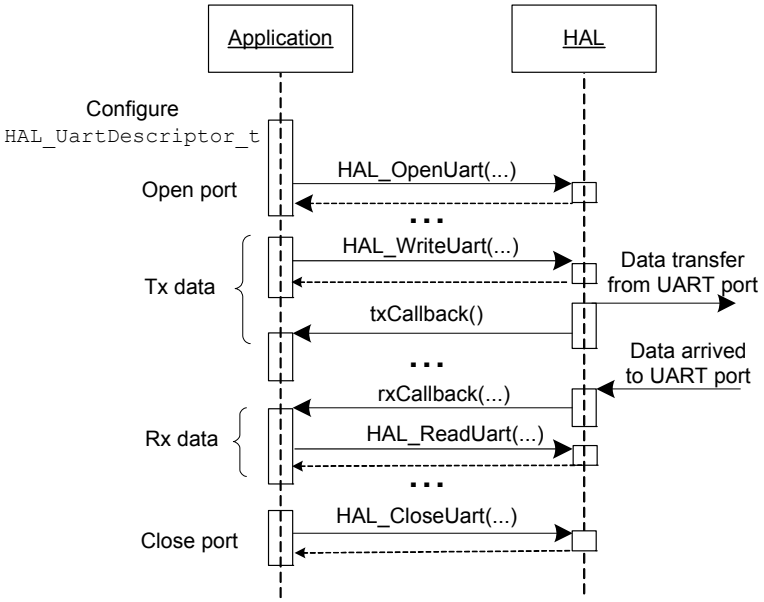
6.1.1 UART callback mode

Code snippet below shows how to configure UART port so that both Tx and Rx operations are executed in callback mode.

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

Figure 6-1 on page 6-2 illustrates corresponding sequence diagram for UART deployed in callback mode.

Figure 6-1. UART data exchange in callback mode.



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

The UART is able to receive data if rxBuffer and rxBufferLength fields of the corresponding UART descriptor are not NULL and 0 respectively. For callback mode rxCallback field shall point to a function which will be executed every time data is arrived to UART's rxBuffer with number of received bytes as an argument. Knowing this number application shall retrieve received data from UART rxBuffer to application buffer using HAL_ReadUart() function.

6.1.2 UART polling mode

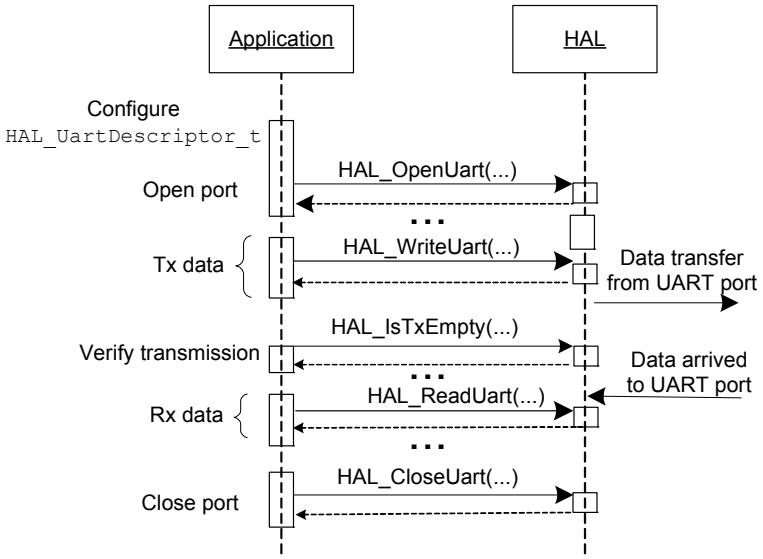
In the polling mode UART Tx/Rx operations utilize corresponding cyclic buffers of UART descriptor. So buffer pointer as well as buffer length shall be set to non-zero for direction to be deployed in polling mode, while corresponding callback function shall be NULL.

Figure 6-2 on page 6-3 illustrates sequence diagram for Tx/Rx in polling mode. The main difference in Tx operation between callback and polling modes is that in latter one after calling HAL_WriteUart() all data submitted for transmission as argument is cyclically moved to the txBuffer of the UART descriptor. Hence application can right away reuse memory occupied by the data. HAL_IsTxEmpty() function can be used in order to verify whether there is enough space in the txBuffer as well as to verify how many bytes were actually transmitted.

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

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

Figure 6-2. UART data exchange in polling mode.



6.1.3 CTS / RTS / DTR management

In addition to data read/write operations BitCloud provides API for managing CTS / RTS / DTR lines of the UART port that supports HW flow control (depends on the platform). See API documentation [5] for detailed description of corresponding functions.

6.2 Two-wire Serial Interface Bus

Note: TWI bus is not supported in HAL for the current release of BitCloud for ATAVRRZRAVEN.

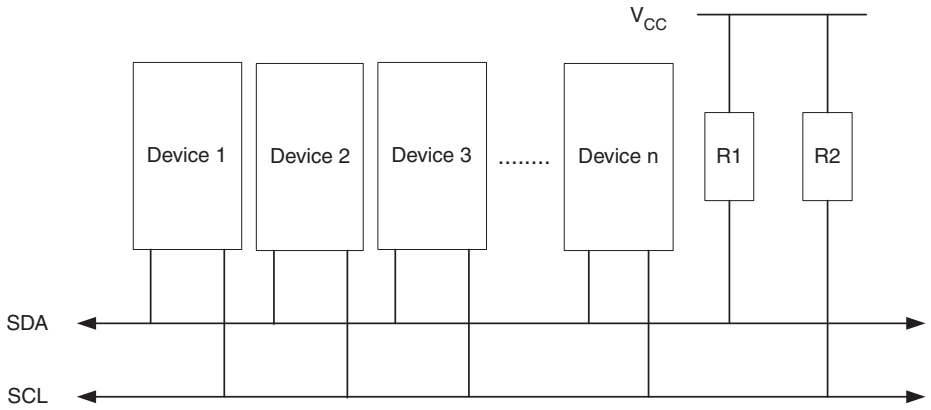
For more details about Using TWI modules as I^2C slave, please refer to:

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

6.2.1 Two-wire Serial Interface Bus Definition

The Two-Wire serial Interface (TWI) is compatible with Philips' I^2C protocol. The Two-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

Figure 6-3. TWI Bus Interconnection



6.2.2 TWI Terminology

The following definitions are frequently encountered in this section.

Figure 6-4. TWI Terminology

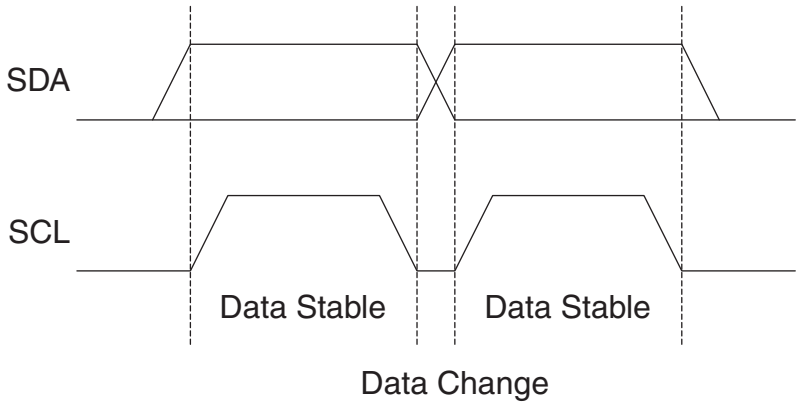
Term	Description
Master	The device that initiates and terminates a transmission. The Master also generates the SCL clock.
Slave	The device addressed by a Master.
Transmitter	The device placing data on the bus.
Receiver	The device reading data from the bus.

6.2.3 Data Transfer and Frame Format

6.2.3.1 Transferring Bits

Each data bit transferred on the TWI bus is accompanied by a pulse on the clock line. The level of the data line must be stable when the clock line is high. The only exception to this rule is for generating start and stop conditions.

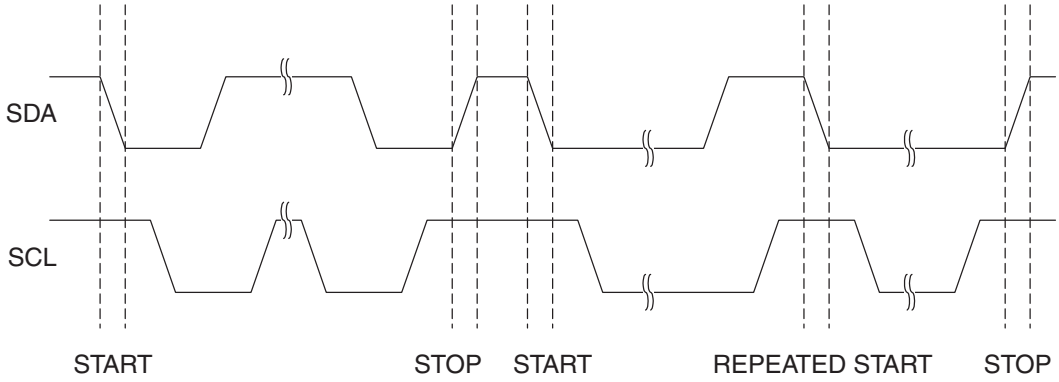
Figure 6-5. Data Validity



6.2.3.2 START and STOP Conditions

The Master initiates and terminates a data transmission. The transmission is initiated when the Master issues a START condition on the bus, and it is terminated when the Master issues a STOP condition. Between a START and a STOP condition, the bus is considered busy, and no other master should try to seize control of the bus. A special case occurs when a new START condition is issued between a START and STOP condition. This is referred to as a REPEATED START condition, and is used when the Master wishes to initiate a new transfer without relinquishing control of the bus. After a REPEATED START, the bus is considered busy until the next STOP. This is identical to the START behavior, and therefore START is used to describe both START and REPEATED START for the remainder of this datasheet, unless otherwise noted. As depicted below, START and STOP conditions are signalled by changing the level of the SDA line when the SCL line is high.

Figure 6-6. START, REPEATED START and STOP conditions



6.2.3.3 Address Packet Format

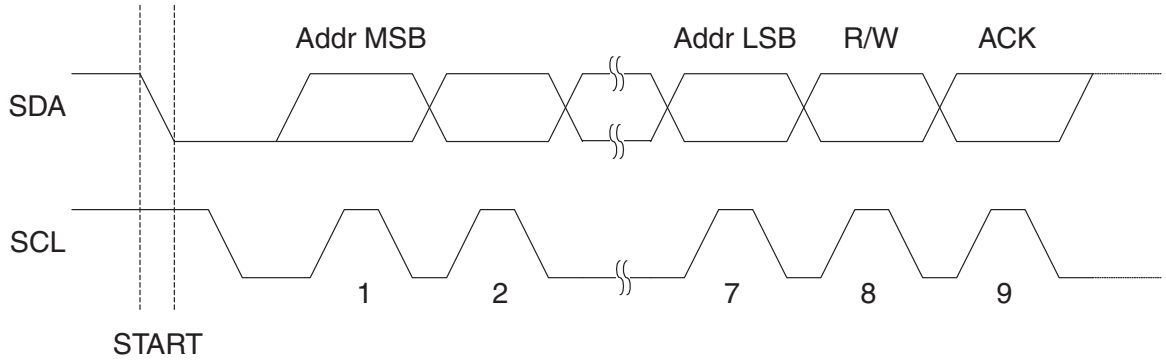
All address packets transmitted on the TWI bus are 9 bits long, consisting of 7 address bits, one READ/WRITE control bit and an acknowledge bit. If the READ/WRITE bit is set, a read operation is to be performed, otherwise a write operation should be performed. When a Slave recognizes that it is being addressed, it should acknowledge by pulling SDA low in the ninth SCL (ACK) cycle. If the addressed Slave is busy, or for some other reason can not service the Master's request, the SDA line should be left high in the ACK clock cycle. The Master can then transmit a STOP condition, or a REPEATED START condition to initiate a new transmission. An address packet consisting of a slave address and a READ or a WRITE bit is called SLA+R or SLA+W, respectively.

The MSB of the address byte is transmitted first. Slave addresses can freely be allocated by the designer, but the address 0000 000 is reserved for a general call.

When a general call is issued, all slaves should respond by pulling the SDA line low in the ACK cycle. A general call is used when a Master wishes to transmit the same message to several slaves in the system. When the general call address followed by a Write bit is transmitted on the bus, all slaves set up to acknowledge the general call will pull the SDA line low in the ack cycle. The following data packets will then be received by all the slaves that acknowledged the general call. Note that transmitting the general call address followed by a Read bit is meaningless, as this would cause contention if several slaves started transmitting different data.

All addresses of the format 1111 xxx should be reserved for future purposes.

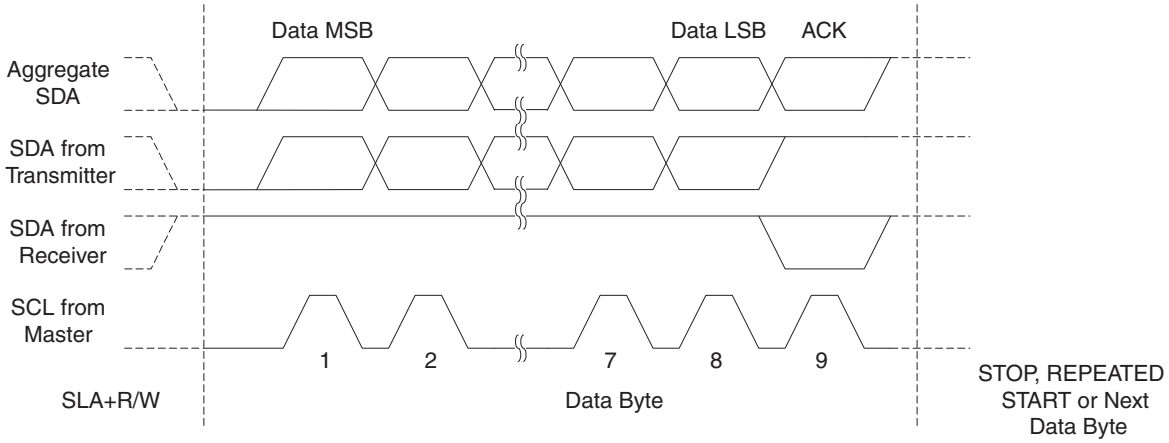
Figure 6-7. Address Packet Format



6.2.3.4 Data Packet Format

All data packets transmitted on the TWI bus are nine bits long, consisting of one data byte and an acknowledge bit. During a data transfer, the Master generates the clock and the START and STOP conditions, while the Receiver is responsible for acknowledging the reception. An Acknowledge (ACK) is signalled by the Receiver pulling the SDA line low during the ninth SCL cycle. If the Receiver leaves the SDA line high, a NACK is signalled. When the Receiver has received the last byte, or for some reason cannot receive any more bytes, it should inform the Transmitter by sending a NACK after the final byte. The MSB of the data byte is transmitted first.

Figure 6-8. Data Packet Format

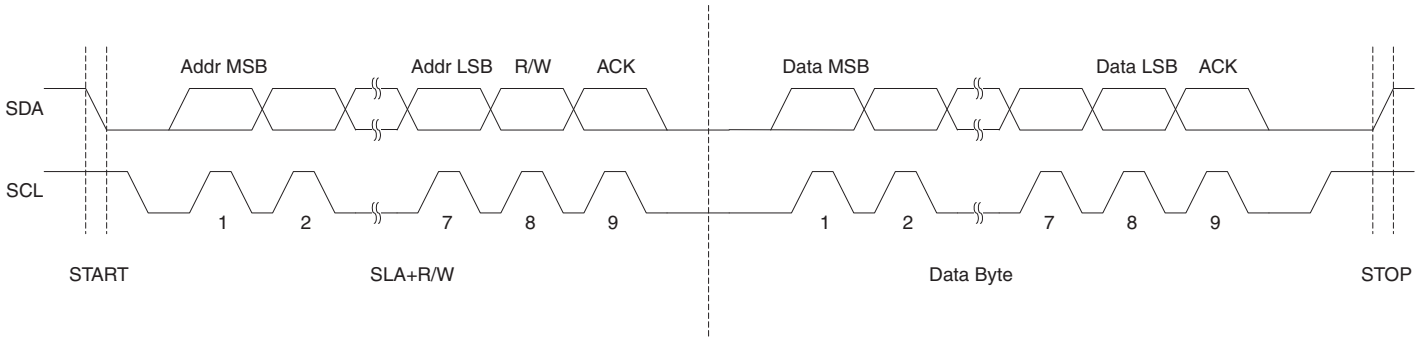


6.2.3.5 Combining Address and Data Packets into a Transmission

A transmission basically consists of a START condition, a SLA+R/W, one or more data packets and a STOP condition. An empty message, consisting of a START followed by a STOP condition, is illegal. Note that the Wired-ANDing of the SCL line can be used to implement handshaking between the Master and the Slave. The Slave can extend the SCL low period by pulling the SCL line low. This is useful if the clock speed set up by the Master is too fast for the Slave, or the Slave needs extra time for processing between the data transmissions. The Slave extending the SCL low period will not affect the SCL high period, which is determined by the Master. As a consequence, the Slave can reduce the TWI data transfer speed by prolonging the SCL duty cycle.

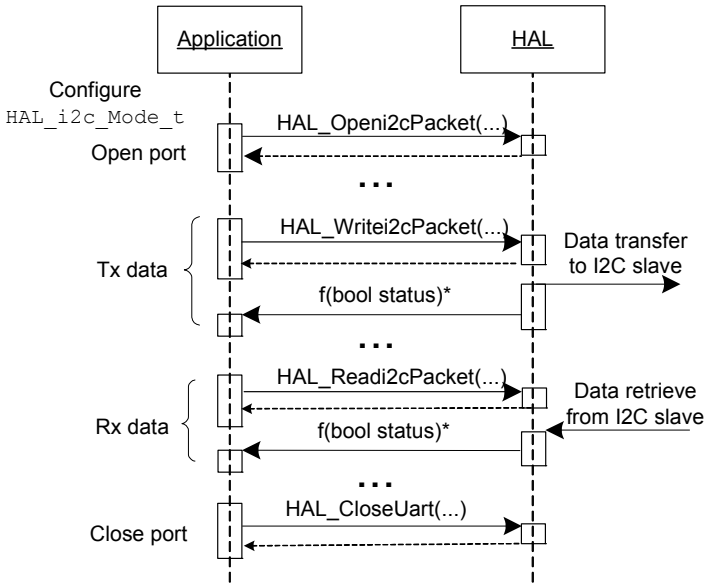
Figure 6-9 on page 6-7 shows a typical data transmission. Note that several data bytes can be transmitted between the SLA+R/W and the STOP condition, depending on the software protocol implemented by the application software.

Figure 6-9. Typical Data Transmission



The BitCloud application can perform only master device functionality of TWI (Two-Wire Interface) protocol. Similar as with any other HW interface, Two-Wire interface shall be first configured and enabled for communication. After that actual data read/write procedures can be performed to a remote TWI slave device. Figure 6-10 on page 6-7 gives a reference for TWI data exchange.

Figure 6-10. Data exchange over TWI bus.



As it is shown in the figure above, TWI write/read operations are executed in asynchronous manner (see "Event-driven programming" on page 3-1). I.e. after calling HAL_WriteI2cPacket() or HAL_ReadI2cPacket() application shall not perform any actions with TWI bus as well as with the memory allocated to the argument of HAL_i2cParams_t type until registered callback function is returned.

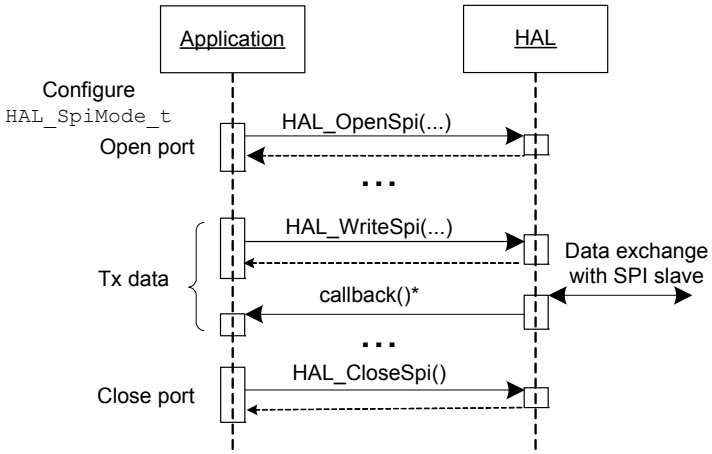
6.3 SPI bus

Note: SPI bus is not supported in HAL for the current release of BitCloud for ATAVRRZRAVEN.

Depending on the MCU platform the HAL component of BitCloud implements SPI protocol either on USART bus (AVR platform) or on SPI1 bus (ARM platform). However corresponding API calls defined in spi.h file and are independent on the underlying platform and the only difference appears in SPI configurations, namely in the fields of static variable of HAL_SpiMode_t type.

The BitCloud application supports only SPI master mode. Figure 14 illustrates a sequence diagram of SPI-related API calls.

Figure 6-11. Data exchange over SPI bus.



Mechanism common for other HW interfaces is applied for SPI bus as well. First it shall be configured and successfully opened (by executing HAL_OpenSpi() with arguments pointing to a variable of HAL_SpiMode_t type and to callback function. Then data transmission can be performed using asynchronous HAL_WriteSpi() request. If callback function is not set to NULL, application leaves the function were HAL_WriteSpi() is called and after that callback function informs application that SPI transaction is finished. If callback argument in HAL_OpenSpi() was NULL, SPI transaction will be performed before actually leaving application function which initiated the transaction. Finally if data exchange is not expected anymore SPI bus shall be closed in order to free occupied resources.

6.4 GPIO interface

Note: The GPIO interface bus is not supported in HAL for the current release of BitCloud for ATAVRRZRAVEN.

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

GPIO related functions and macros names are defined in gpio.h file of HAL_HWD component. Function call has form "GPIO_#pin_name#_#function_name#()". So in order to execute desired function for a cer-

tain pin, corresponding macros for this pin shall be used in the function body. Macros mapping to pin names is given in gpio.h file as well. Following manipulations can be performed with GPIO-enabled pins:

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

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

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

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

- Enable internal pull-up register. Example for GPIO0 pin:

```
GPIO_0_make_pullup();
```

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

```
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 current logical level on input pin. Example for TWI_CLK pin:

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

6.5 Other HAL functionality

Note: Other interfaces are not supported in HAL for the current release of BitCloud for ATAVRRZRAVEN.

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

- ADC
- 1-Wire
- IRQ
- Application timer, system time
- Watchdog, warm reset

The BitCloud Stack Documentation [5] provides information about API functions that shall be used to manage interfaces listed above.

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

- HAL_ReadEeprom/HAL_WriteEeprom
- Sleep Timer

Section 7

Memory and resource allocation

7.1 RAM

RAM is a critical system resource used by BitCloud to store runtime parameters like neighbor tables, routing tables, children tables, etc. As sometimes required by the stack, the values of certain parameters stored in EEPROM are cached in RAM for easier subsequent retrieval. The call stack also resides in RAM, and is shared by the stack and the user application. To conserve RAM, the user must refrain from 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, i.e. 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 is limited to 10, each function having no more than 2 pointer parameters each.

Overall, the RAM demands of the stack must be reconciled with that of the user application. Fortunately, the amount of RAM used by the stack data is a user-configurable parameter. The configuration server, ConfigServer or CS, is distributed in source code (see Table 2-1 on page 2-3), to allow the user to redefine the key runtime parameters of the stack. Among these parameters are parameters directly affecting the size of runtime tables stored in RAM. The table below lists such parameters and provides an easy formula to compute RAM consumption based on the value of a parameter (p):

Table 7-1. Configuration server parameters and RAM consumption

Parameter name	Description	RAM consumption in bytes
CS_NEIB_TABLE_SIZE	Number of entries in neighbor table	51 * p
CS_ROUTE_TABLE_SIZE	Number of entries in routing table	6 * p
CS_DUPLICATE_REJECTION_TABLE_SIZE	Number of entries in duplicate filtering table	7 * p
CS_APS_GROUP_TABLE_ENDPOINTS_AMOUNT	Number of endpoints in a group	(3+CS_APS_GROUP_TABLE_ENDPOINTS_AMOUNT) * CS_APS_ GROUP_TABLE_GROUPS_AMOUNT
CS_APS_GROUP_TABLE_GROUPS_AMOUNT	Number of groups	
CS_ADDRESS_MAP_TABLE_SIZE	Number of entries in address map table	11 * p
CS_ROUTE_DISCOVERY_TABLE_SIZE	Number of entries in temporary table for route discovery	12 * p

Table 7-1. Configuration server parameters and RAM consumption (Continued)

Parameter name	Description	RAM consumption in bytes
CS_NWK_DATA_REQ_BUFFER_SIZE	Number of entries in NWK data requests buffer	90 * p
CS_NWK_DATA_IND_BUFFER_SIZE	Number of entries in NWK data indications buffer	165 * p
CS_APS_DATA_REQ_BUFFER_SIZE	Number of entries in APS data requests buffer	34 * p
CS_APS_ACK_FRAME_BUFFER_SIZE	Number of entries in APS acknowledgements buffer	76 * p

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

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

avr-size -d app.elf

The result will be presented as below:

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

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

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

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

7.2 Flash storage

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

avr-size -d app.elf

The result will be presented as below:

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

The number of bytes consumed will be **text + data.** Unlike with RAM, the user has little control of how much flash space is consumed by the underlying stack. Since the stack libraries are delivered as binary object files, at link time (part of application building process) the linker ensures that mutual dependencies are satisfied, i.e. 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.

7.3 EEPROM

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

PDS also detects any CRC errors by automatically storing a checksum alongside every parameter stored in EEPROM. When a parameter is read, the checksum is computed and compared to the one stored in EEPROM.

7.4 Other resources

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

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

Headquarters

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

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

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

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

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature