life.augmented

Application note

Developing Zigbee® sleepy end devices on STM32WB Series

Introduction

This application note describes the necessary considerations when developing Zigbee® sleepy end devices (SED) on STM32 Series microcontrollers.

Zigbee® sleepy end devices are a special class of nodes capable of operating in very-low power modes, and are often battery powered. In order to conserve power, they manage their power consumption in a low-power state with the radio networking off most of the time. These devices have their `RxOnWhenIdle` property set to false.

In terms of the IEEE 802.15.4 MAC standard, these devices are RFD (reduced functionality devices). In low-power wireless technology, Zigbee sensors are called sleepy end devices.

Note: Parts of this document are under Copyright © 2021 Exegin Technologies Limited. Reproduced with permission.

1 General information

This document applies to the STM32WB Series Arm®-based microcontrollers.

Note: Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Glossary

Table 1. Glossary

Term	Definition
APS	Application support sublayer
PAN	Personal area network
TCSO	Trust center swap out
ZCL	Zigbee® cluster library
OTA	Over the air
FFD	Full function device
RFD	Reduced functionality device
FP	Frame pending
BO	Beacon order

Reference documents

Table 2. Reference documents

Reference	Document
[1]	05-3474-22 Zigbee Specification R22(1)
[2]	AN5500: *ZSDK API implementation for Zigbee® on STM32WB Series*

1. *This URL belongs to a third party. It is active at document publication, however STMicroelectronics shall not be liable for any change, move or inactivation of the URL or the referenced material.*

2 Application design

Sleepy end devices conserve power when set in an off-state mode most of the time. This means that sleepy end devices are mostly inaccessible to other devices in the network. This limits the applications of sleepy end devices compared to devices that collect and push data on the network.

Sensors are an interesting example. A sensor device periodically wakes up, takes a reading, and pushes the results to an always-on concentrator or gateway device. Sleepy end devices work correctly in this context as the sensor data is small and fits into small waking transactions. Moreover, sleepy end devices enable the transfer of large amounts of data.

Sleepy end devices can use the over-the-air (OTA) firmware cluster to download an entire firmware image. The OTA cluster is designed to be flexible to ensure that the device is in control of the download. This has no impact from the server's side if the client takes several days to download a complete image.

There are low-power devices that slowly charge a capacitor. These low-power devices are designed to wake up when the capacitor is charged. They only stay on when the charge of the capacitor permits this. When the charge is exhausted the device sleeps. Therefore, it is important that applications are designed to be flexible and driven by the sleepy end device.

If other devices on the network need to communicate with sleepy devices, smart energy is used through a two-way mirror and notification flags. In smart energy, an always-on-device maintains a copy of the attributes that reside on the sleepy device. This copy is called a mirror.

When remote devices need to read the data from the sleepy device, they can read it from the mirror. The sleepy device's role is to maintain the data on the mirror during the brief intervals in its awake state. Two-way communication is achieved with a two-way mirror. The data sent to the sleepy device allows remote devices to write to the mirror.

Additionally when the mirror is changed, a notification flag is set. When the sleepy device wakes up it checks the notification flags to fetch an update if necessary and, to reset the notification flag when completed.

The following sections are examples of how sleepy end devices are supported by specific design aspects in the application.

3 Creating a sleepy end device

The configuration of a device is done through the config structure passed to `ZbStartup()`.

`ZbStartupT` is a rich structure with many options. Conventionally an application starts by initializing the structure using `ZbStartupConfigGetProDefaults()` or `ZbStartupConfigGetProSeDefaults()` for a smart energy network.

As an end device it cannot form a network, it can only join. The `ZbStartupT config.startupControl` cannot be a `ZbStartType` of `ZbStartTypeForm`, instead it must be either `ZbStartTypeJoin`, or `ZbStartTypeRejoin` when rejoining.

`ZbStartupConfigGetProDefaults()` configures the capabilities as an IEEE 802.15.4 full function device, which is a router, `RxOnWhenIdle` (non-sleepy), mains-powered device.

- Listing 1: example of a device configuration:

```
/* Reduced Capability Device */
config.capability &= ~MCP_ASSOC_CAP_DEV_TYPE;

/* disable RxOnWhenIdle */
config.capability &= ~MCP_ASSOC_CAP_RXONIDLE;

/* battery powered (i.e. not mains powered) */
config.capability &= ~MCP_ASSOC_CAP_PWR_SRC;
```

End devices are intended to be sleepy: `RxOnWhenIdle=false`. This means that the commissioning process is similar for a sleepy end device and a non-sleepy end device. Non-sleepy end devices are rare, if a device’s radio is always on, it is more practical to make the device a router instead. Generally non-sleepy end devices are only used in testing situations.

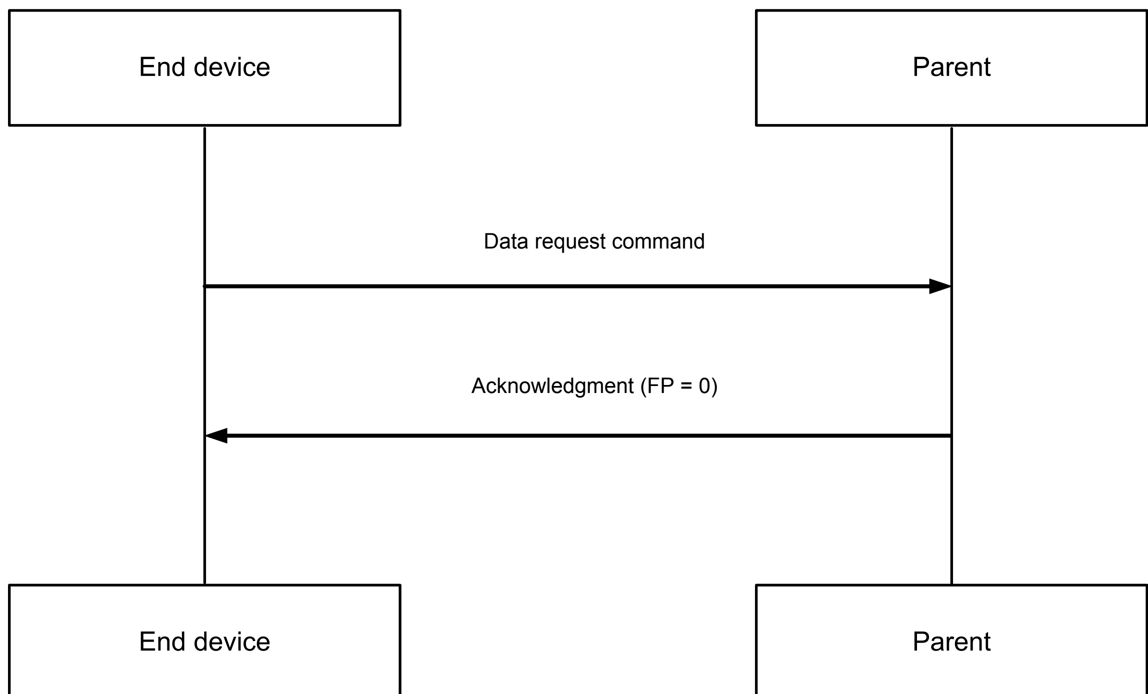
4 Zigbee network behavior in sleepy end devices

A sleepy end device must always be paired with another node. It must be IEEE 802.15.4 FFD (full function device). It can be a router or the coordinator, which is known as the sleepy end device’s parent and the sleepy end device's child.

The sleepy end device does not communicate directly to arbitrary nodes on the network. All communication takes place through the parent. The parent maintains what is known as an indirect message queue for each child. During its wake-up cycle, the child performs a MAC data poll to its parent. When there is no data in the indirect message queue, the parent responds with a MAC Ack with the FP (frame pending) bit clear. To conserve battery the child may choose to return immediately to sleep.

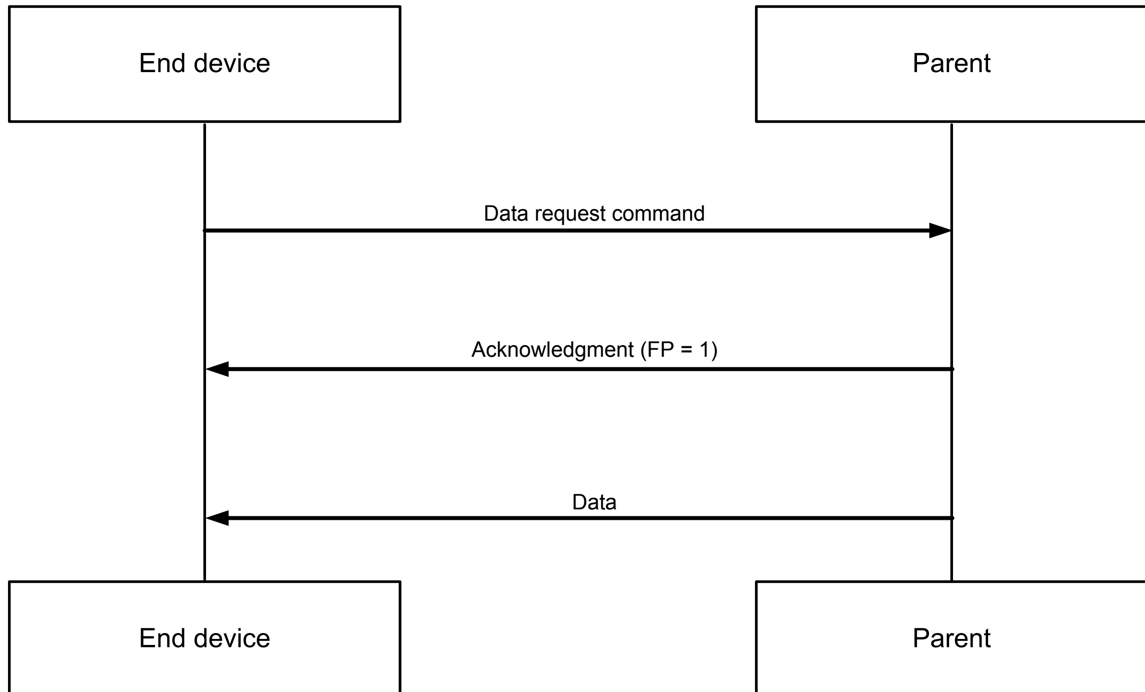
The figure below shows the requesting data without frame pending.

Figure 1. Requesting data without frame pending



The figure below show the requesting data with frame pending.

Figure 2. Requesting data with frame pending



The amount of time that parents retain messages for sleepy children is denoted by the `nwkTransactionPersistenceTime`, which for a 2.4 GHz O-QPSK PHY is 7.68 seconds.

The indirect message queue mechanism is not designed to be a general-purpose mechanism for arbitrary network nodes to communicate with sleepy end nodes. Its purpose is to allow sleepy end nodes to associate with their parent during the joining process.

In general, sleepy end devices need to use various applications with specific pull mechanisms, to determine if there is data waiting for them so that they can choose when to retrieve this data and to balance this, with energy management considerations. The MAC data poll mechanism is a pull mechanism that is used when using association.

Smart energy uses a different application-level mechanism, that allows sleepy end devices to pull data. A smart energy two-way mirror allows arbitrary nodes to retrieve arbitrary data from the mirror at any time, especially when the sleepy node is asleep. It also provides event flags. The sleepy node reads during its powered-on cycle, which tells the sleepy node there is data waiting for on the mirror. The sleepy end node pulls the event flags and decides when it is best to pull the mirrored data, while at the same time it manages its power consumption.

The two-way mirror is specific to smart energy. There is no general-purpose pull mechanism. It is left to the applications to determine how to advertise about the availability of data to the sleepy end device. It is then up to the sleepy end device to handle this.

5 Polling slow and fast rates

There are two rates of MAC data poll, slow and fast. The slow rate is determined entirely by the device. It can be as slow as once every 30 minutes or even once per day, depending on the application. It is important to understand the slow poll rate from the child aging perspective as the parent determines that a child is no longer present and can be aged out. This relates to the keepalive mechanism described in Section 7 End device keepalive. Sleepy end devices go to sleep between slow polls.

While in fast poll mode the end device stays awake and polls at the fast poll rate which is a value less than `nwkTransactionPersistenceTime` (7.68 seconds). Commonly a fast poll value of 7.5 seconds is used. While polling at a fast poll rate the end device is able to receive messages. In general, fast polling is enabled and disabled internally as needed by the stack, without the involvement of the application. The main use of fast polling is to ensure that the end device can receive messages from the trust center during joining. Allowance is made for the trust center to be several hops away.

When the end device needs to receive data from a remote device such as pull data, it enables fast polling by calling `ZbNwkFastPollRequest()`. It is up to the device to determine when and how long to enable the fast polling This is determined based on energy management considerations. For each fast poll request there must be a matching fast poll release when the application calls `ZbNwkFastPollRelease()`.

6 Poll control cluster

End devices support the poll control cluster. The poll control cluster enables or disables fast polling by another device. To use the poll control cluster, the remote application must verify if the device to communicate with is an end device. The application must support the poll control client cluster and the end device must support the poll control server.

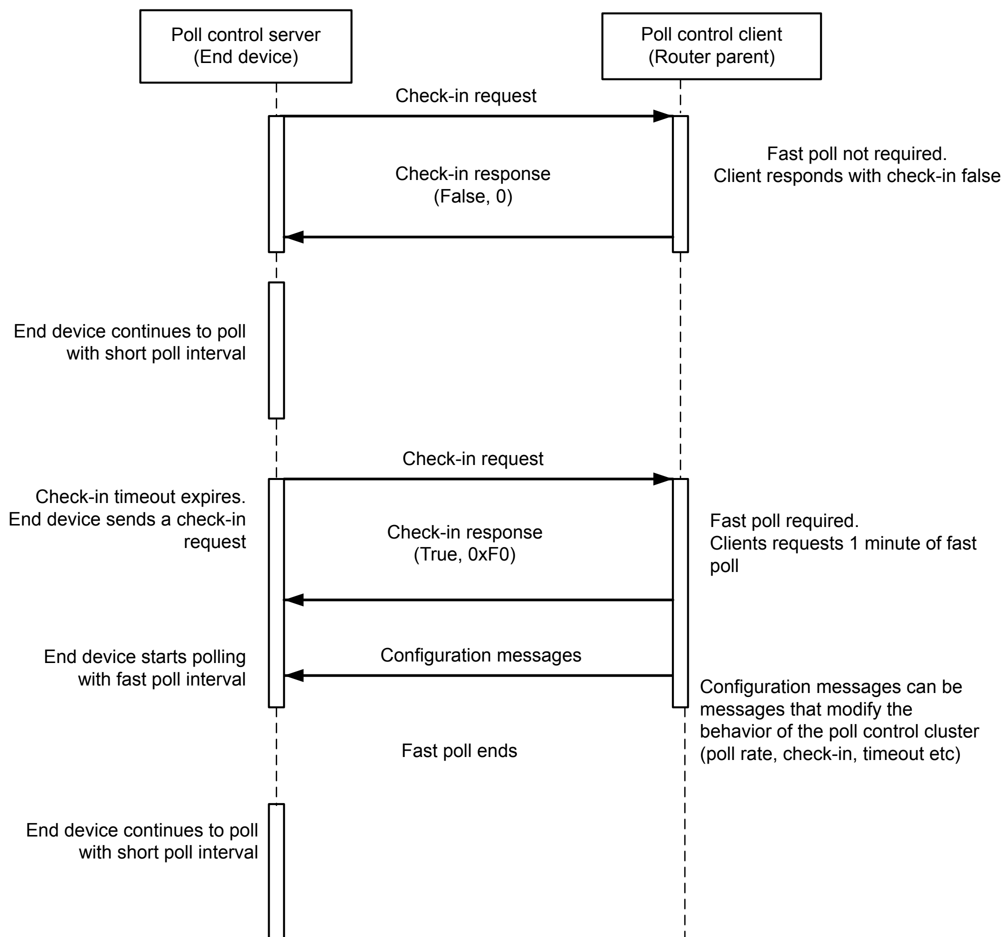
This mechanism must not be supported on sleepy end devices that require tight control over their power usage. The remote application has no acknowledgment of this requirement and can rapidly exhaust the end device battery by forcing it into fast-poll mode for a too long interval or too often. This is particularly important when battery replacement is difficult or impossible.

In order to support the poll control cluster before startup, the end device must add an instance of the poll control cluster to an endpoint by calling `zcl_poll_server_alloc()`. The end device must set the long poll interval using `zcl_poll_server_write_long_poll_intvl()`.

The finding and binding mechanism is used to establish a binding between the server and all poll control clients on the network. Finding and binding must be enabled by setting the `BDB_COMMISSION_MODE_FIND_BIND` bit of `ZB_BDB_CommissioningMode` BDB attribute (set using `ZbBdbSet()`). When set in the BDB commissioning mode, `ZbStartup()` automatically initiates finding and binding after joining the network. Finding and binding can also be initiated at a later time by calling `ZbStartupFindBindStart()` or `ZbStartupFindBindStartEndpoint()` to create bindings to clients that have joined after this end device.

During the operation, a sleepy end device must call `zcl_poll_server_send_checkin()` every time it wakes-up to send a check-in request, which may initiate a fast poll mode.

Figure 3. Poll control flowchart



7 End device keepalive

Networks are dynamic and routers must be able to remove entries for devices that have aged out, meaning that they have no neighbor, routing or other association. However, an end device relies on its parent to access the network. In order to prevent sleepy end devices with long-sleep intervals from being aged out, they support the end device keepalive method.

The support for the end device keepalive method is built into the stack. End devices automatically send end device timeout request messages to their parent at an interval determined by the `nwkEndDeviceTimeoutDefault` IB parameter. By default, this is set to eight minutes. The end device timeout request messages are sent every `nwkEndDeviceTimeoutDefault/4` to ensure that at least three end device timeout request messages are sent within the `nwkEndDeviceTimeoutDefault` period. `nwkEndDeviceTimeoutDefault` is set on startup by configuring the `ZbStartupT` configuration parameter `endDeviceTimeout` to n which give a timeout of 2^n minutes n=1…14 (2,4,8,…16384 minutes). With special values 0 (10 seconds) and 0xFF (disabled) used primarily for testing. After startup, the end device timeout is also available to the application as a NIB parameter, via `ZbNlmeGetReq()` using id `ZB_NWK_NIB_ID_EndDeviceTimeoutDefault`. It is uncommon but it is also possible to reset this value, in such case, the next end device timeout request to the parent requests the new value.

The zigbee stack also supports the router parent side of the end device keepalive. There are no application considerations on routers.

8 ZCL response handling

Most ZCL commands have a response for each request. However, when there is no result from the request, some commands do not define a response. In this case, a generic default response is defined. The primary purpose of a default response is to convey an error status. In order to cut down on unnecessary over-the-air traffic, ZCL commands have a flag that suppresses the default response when the command is successful. When no default response is received, the command is considered successful.

Most sleepy end devices must request a ZCL default response to be sent when the ZCL request does not have a default response defined. In the worst case, on a multi-hop path with APS retries, an error can be received up to 10 seconds after the request is sent if an APS Ack is requested, or three seconds when an APS Ack is not requested. On successful requests, devices may need to wait up to 10 seconds (or three seconds without APS Ack) to know that the request was successful.

When a ZCL response is not specified and expected, requesting that a default response is always sent, even on success, eliminates this timeout delay. To implement this, when no standard ZCL response is defined, the applications should set `noDefaultResp`, field of the ZCL `ZbZclClusterCommandReqT` to a value of `ZCL_NO_DEFAULT_RESPONSE_FALSE`. This results in a default response to always be sent in most cases. This also means that the ZCL transaction ends in much less than 10 seconds or three seconds timeout which is especially important for sleepy devices that want to return to sleep as soon as possible.

9 Persistence and the sleep wake cycle

For some application designs it is not always possible for the device to stay awake during the full interval. The application may choose to power down. This results in missed communication and lost data, however it can be an acceptable compromise in some situations. In general, applications are designed to avoid such circumstances.

When the device wakes up, it must call `ZbStartupPersist()` instead of `ZbStartup()`.`ZbStartupPersist()` restores the stack state before sleep. However, before using `ZbStartupPersist()`, the application must add support for persistence.

In order to enable persistence, the application must call `ZbPersistNotifyRegister()` both at original startup and when waking up from sleep mode. It can provide a callback function, for example:

```
MyPersistenceCallback(structZigBeeT *zb, void
        *cbarg)
```

The application is responsible in this callback to save the persistence data to the Flash memory. The current snapshot of the data to persist is obtained from Zigbee stack by calling `ZbPersistGet()`. The application must not alter the block of persistence data, it must be persisted and later restored.

Some Flash memory systems have limitations such as the number of writes persisted. The application can adjust the rate at which the callback is called through the BDB parameter `ZB_BDB_PersistTimeoutMs`, which is the minimum delay between persistence updates. The application can also temporarily disable or re-enable the persistence callback using `ZbPersistNotifyControl()`.

Revision history

Table 3. Document revision history

Date	Version	Changes
14-Jan-2022	1	Initial release.

Contents

1 General information . . . 2
2 Application design . . . 3
3 Creating a sleepy end device . . . 4
4 Zigbee network behavior in sleepy end devices . . . 5
5 Polling slow and fast rates . . . 7
6 Poll control cluster . . . 8
7 End device keepalive . . . 9
8 ZCL response handling . . . 10
9 Persistence and the sleep wake cycle . . . 11
Revision history . . . 12
Contents . . . 13
List of tables . . . 14
List of figures . . . 15

List of tables

Table 1. Glossary . . . 2
Table 2. Reference documents . . . 2
Table 3. Document revision history . . . 12

List of figures

Figure 1. Requesting data without frame pending 5
Figure 2. Requesting data with frame pending 6
Figure 3. Poll control flowchart 8

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2022 STMicroelectronics – All rights reserved