JN-UG-3133

JN518x and K32W041/K32W061/K32W1 Core Utilities User Guide

Rev. 2.2 — 3 March 2023 User guide

Document information

Information	Content
Keywords	JNUG3133, ZigBee, JN518x and K32W041/K32W061/K32W1, Core Utilities (JCU), JCU modules, Wireless network applications
Abstract	This document provides information for implementing Core Utilities on the NXP K32W041, K32W061, K32W1, and JN518x family of wireless microcontrollers

About this manual

This manual provides a single point of reference for information relating to the Core Utilities (JCU), for use with the NXP K32W041, K32W061, K32W1, and JN518x family of wireless microcontrollers. The manual provides both conceptual and practical information concerning the JCU, and provides guidance on use of the JCU Application Programming Interfaces (APIs). The API resources (functions and structures) are fully detailed.

The Core Utilities described in this user guide are legacy modules still supported on K32W061, K32W041, K32W1, or JN518x devices for users transitioning from previous JN devices or SDKs to provide a level of backward compatibility.

For new developments, users should consider the modules described in the Connectivity Framework Reference Manual.

Organization

This manual is divided into two parts:

- Part I: Concept and Operational Information consists of five chapters:
  - Chapter 1 introduces the Core Utilities and associated APIs.
  - Chapter 2 describes how to use the Flash-based PDM.
  - Chapter 3 describes how to use the Power Manager (PWRM).
  - Chapter 4 describes how to use the Protocol Data Unit Manager (PDUM).
  - Chapter 5 describes how to use the Debug (DBG) module.
- Part II: Reference Information consists of five chapters:
  - Chapter 6 describes the functions of the PDM API for EEPROM.
  - Chapter 7 describes the functions of the PWRM API.
  - Chapter 8 describes the functions of the PDUM API.
  - Chapter 9 describes the functions of the DBG API.
  - Chapter 10 details the structures used by the JCU.
  - Chapter 11 lists the revisions made to this document.

Conventions

The below conventions are used in this document:

- Files, folders, functions, and parameter types are represented in **bold** type.
- Function parameters are represented in *italics* type.
- Code fragments are represented in the `Courier New` font.

Note: *This convention is a* ***Note****. It highlights important additional information.*

Acronyms and abbreviations

Table 1. Acronyms and abbreviations

Acronym	Description
API	Application Programming Interface
CCA	Clear Channel Assessment
FCF	Frame Control Field
FCS	Frame Check Sequence
GP	Green Power

Table 1. Acronyms and abbreviations...*continued*

Acronym	Description
GPD	Green Power Device
MAC	Medium Access Control
PAN	Personal Area Network
PIB	PAN Information Base
SDK	Software Developer's Kit
ZGPD	ZigBee Green Power Device
ZGPP	ZigBee Green Power Proxy
ZGPS	ZigBee Green Power Sink

Related documents

1. MCUXSDKJN5189APIRM (MCUXpresso SDK API Reference Manual_JN518x)
2. MCUXSDKK32W041APIRM (SDK API Reference Manual_K32W061/K32W041)
3. JN-UG-3130 (ZigBee 3.0 Stack User Guide)
4. JN-UG-3131 (ZigBee 3.0 Devices User Guide)
5. JN-UG-3132 (ZigBee Cluster Library (for ZigBee 3.0) User Guide)
6. JN-UG-3134 (ZigBee Green Power User Guide)
7. JN518x Data Sheet (JN518x Datasheet)
8. K32W061/41 Data Sheet (K32W041/K32W061 Data Sheet)
9. K32W1480 Data Sheet (K32W1480 Data Sheet)
10. CONNFWRM (Connectivity Framework Reference Manual)

Support Resources

To access online support resources such as SDKs, Application Notes, and User Guides, visit the Wireless Connectivity area of the NXP website:

www.nxp.com/products/interface-and-connectivity/wireless-connectivity

All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

The software described in this manual can currently be used on the NXP K32W061, K32W1, K32W041, and JN518x family of wireless microcontrollers.

Part I: Concept and operational information

1 Introduction

The device Core Utilities (JCU) are designed for use in wireless network applications for the NXP K32W041, K32W061, K32W1, and JN518x devices. These utilities provide an interface which simplifies the programming of a range of operations that are not specific to wireless networking.

1.1 Modules and architecture

The Core Utilities consist of four utilities/modules, each with a dedicated Application Programming Interface (API) to facilitate easy interaction between the application and the corresponding JCU module. API of each module consists of a set of C functions and associated resources.

1.1.1 JCU modules

The JCU modules are briefly described below:

- **Persistent Data Manager (PDM):** This module handles the storage of context and application data in Non-Volatile Memory (NVM), and the retrieval of this data. It provides a mechanism by which the device can resume operation without loss of continuity following a power loss. For the K32W041, K32W061, K32W1, and JN518x device, this NVM uses internal flash memory. The PDM module is described in Chapter 2
- **Power Manager (PWRM):** This module manages the transitions of the device into and out of low-power modes, such as Sleep mode. The PWRM module is described in Chapter 3.
- **Protocol Data Unit Manager (PDUM):** This module is concerned with managing memory, as well as inserting data into messages to be transmitted and extracting data from messages that have been received. The PDUM module is described in Chapter 4.
- **Debug (DBG):** This module allows diagnostic messages to be output when the application runs, as an aid to debugging the application code. The DBG module is described in Chapter 5.

Note:

1. *The JCU modules are supplied in the NXP Software Developer's Kit (SDK) for the wireless networking protocols. Not all of the JCU modules are provided in every SDK - for details of the supplied modules, refer to the Release Notes of your SDK.*
2. *Not all of the supplied JCU modules must be used in an application. Enable the modules individually for use by the application - for details, refer to the chapters for the modules.*

1.1.2 Software architecture

On a node in a wireless network, the JCU interacts with the following software blocks:

- User application (through use of the JCU APIs in the application code)
- Wireless networking stack (for example, ZigBee PRO stack)
- SDK peripheral APIs

The JCU can be envisaged as sitting alongside the wireless networking stack and the SDK peripheral API, as depicted in Figure 1.

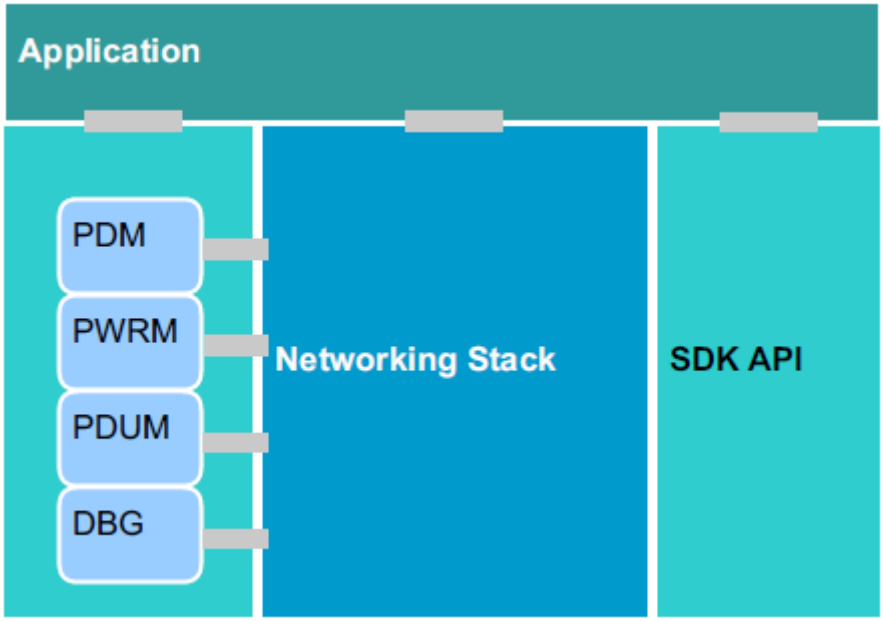

Figure 1. Basic software architecture

2 Persistent Data Manager

This chapter describes the Persistent Data Manager (PDM) module which handles the storage of stack context data and application data in Non-Volatile Memory (NVM). For the K32W041, K32W061, K32W1, and JN518x devices this memory is implemented in the internal flash memory. This chapter refers to this memory as NVM.

Note: *The PDM functions mentioned in this chapter are detailed in Chapter 6.*

Tip: *In this chapter, a cold start refers to either a first-time start or a restart without memory (RAM) held. A warm start refers to a restart with memory held (for example following sleep with memory held).*

2.1 Overview

If the operational data of the network node is solely stored in on-chip RAM, the node only keeps this data while it is powered. However, if the power supply is interrupted, it loses the data (for example, power failure or battery replacement). This data includes context data for the network stack and application data.

To recover the node from a power interruption with continuity of service, provision must be made for storing essential operational data in Non-Volatile Memory (NVM), which is held in flash memory. This data can then be recovered during a reboot following power loss, allowing the node to resume its role in the network.

The storage and recovery of operational data in NVM can be handled using the Persistent Data Manager (PDM) module described in this chapter and covers the following topics:

- Initializing the PDM module - see Section 2.2
- Managing data in NVM - see Section 2.3
- Storing counters in NVM - see Section 2.4
- PDM features including mutexes, NVM wear counts, and event handling - see Section 2.4

The PDM can be used with ZigBee PRO and IEEE802.15.4 wireless networking protocols.

2.2 Initializing the PDM and building a file system

The application must initialize the PDM module following a cold or warm start, irrespective of the PDM functionality used. For example, context data storage or counter implementation. PDM initialization is performed using the function **PDM_eInitialise()**.

This function requires the following information to be specified:

- The number of Flash segments and the first segment to be allocated to PDM use, are provided.

Once the **PDM_eInitialise()** function is called, the PDM module builds a file system in RAM containing information about the segments that it manages in Flash. The PDM reads the header data from each NVM segment and builds the file system.

The file system allows the PDM to do the following functions:

- Perform efficient searches when operating on data
- Track the occupation of all the segments in the NVM
- Count the number of segments available for data allocation at any time

It also helps to even out the wear across NVM segments - for more information on NVM segment wear, refer to Section 2.4.4.

2.2.1 Building applications that use PDM

To use the PDM in applications developed, the flag PDM_NO_RTOS must be defined in the makefile, as follows:

```
CFLAGS+=-DPDM_NO_RTOS
```

This means that the application does not need to define a mutex in order for the PDM to function and the relevant parameter is removed from the **PDM_eInitialise()** function.

2.3 Managing data in non-volatile memory

This section describes the use of the PDM module to persist data in NVM in order to provide continuity of service when the device resumes operation after a cold start or a warm start without memory held.

Data is stored in NVM in terms of ‘records’. A record occupies at least one NVM segment but may be larger than a segment and occupy multiple segments. Any number of records of different lengths can be created, ensuring that they do not exceed the NVM capacity. The records are created automatically for stack context data and by the application (as indicated in Section 2.3.1) for application data. A unique 16-bit value is assigned to each record to identify it, when the record is created - for application data, this identifier is user-defined.

The stack context data which is stored in NVM includes the following:

- Application layer data:
  - AIB members, such as the EPID and ZDO state
  - Group Address table
  - Binding table
  - Application key-pair descriptor
  - Trust Centre device table
- Network layer data:
  - NIB members, such as PAN ID and radio channel
  - Neighbor table
  - Network keys

– Address Map table

On performing a device cold start or warm start without RAM held, the PDM must be initialized in the application as described in Section 2.2.

- If it is the first ever cold start, there is no stack context data or application data preserved in the NVM.
- If it is a cold or warm start following previous use (such as after a reset), there should be stack context data and application data preserved in the NVM.

On startup, the PDM builds a file system in RAM and scans the NVM for valid data. If any data is found, it is incorporated in the file system.

The PDM saves a Cyclic Redundancy Code (CRC) for each segment of a record. Any failure results in the data being unrecoverable and the record becoming invalid.

Saving, recovering, and deleting application data in NVM are described in the following subsections.

2.3.1 Saving data to non-volatile memory sectors

Application data and stack context data are saved from RAM to Non-Volatile Memory (NVM) as described in this section.

Note: *If the NVM must be defragmented and purged during a data save, it is done automatically, resulting all records to be resaved.*

Application data

Save application data to NVM when important changes have been made to the data in RAM. Use the function **PDM_eSaveRecordData()** to save Application data in RAM to an individual record in NVM. A buffer of data in RAM is saved to a single record in NVM (a record may span multiple NVM segments).

The first time that a record is saved using **PDM_eSaveRecordData()**, the record is created. The data is written in its entirety, provided there is enough free space to hold the data. It is recommended to first find out how many segments are available using the function **PDM_u8GetSegmentCapacity()**. When a record is first created, the application must assign a unique 16-bit identifier to the record. This identifier is then used to reference the record. The value used must not clash with the ones used by the NXP libraries - the ZigBee PRO stack libraries use values above 0x8000.

Then, in performing resave to the same record (specified by its 16-bit identifier), the original NVM segments associated with the record is over-written but only one or more segments containing data changes is altered. If no data has changed, no write is performed. This method of only making incremental saves, improves the occupancy level of the size-restricted NVM.

If a save fails, the function **PDM_eSaveRecordData()** returns the code PDM_E_STATUS_NOT_SAVED. Alternatively, the callback event E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED can be used to notify the application of a save failure. For this, you should register a PDM callback function during the initialization of the PDM using the function **PDM_eInitialise()** or the function **PDM_vRegisterSystemCallback()**, as described in Section 2.4.2.

Stack context data

The NXP ZigBee PRO stack automatically saves its own context data from RAM to NVM when certain data items change. This data is not encrypted.

2.3.2 Recovering data from NVM

Application data and stack context data are loaded from the NVM to RAM as described in this section.

Application data

Application reads the data records in NVM using the function **PDM_eReadDataFromRecord()**. The record to be read is specified using its 16-bit identifier. Also, specify a data buffer in RAM in which the read data is stored.

Before calling **PDM_eReadDataFromRecord()**, it may be useful to call the function **PDM_bDoesDataExist()** to determine whether a record with the specified identifier exists in the NVM and, if it does, to obtain its size and therefore the length of the required RAM buffer.

Once the PDM module has been initialized (see Section 2.2) during a cold start or a warm start without memory held, **PDM_eReadDataFromRecord()** must be called. This function must be called for each record of application data in NVM that must be copied to RAM.

Stack Context Data

The function **PDM_eReadDataFromRecord()**, described above, is not used for records of stack context data. Stack automatically handles loading of this data from the NVM to RAM, ensuring that the PDM has been initialized.

2.3.3 Deleting data in NVM

An individual record of application data in the NVM can be deleted using the function **PDM_vDeleteDataRecord**. The record to be deleted is specified using its 16-bit identifier. Alternatively, all records (application data and stack context data) in the NVM can be deleted using the function **PDM_vDeleteAllDataRecords()**.

CAUTION: *You are not recommended to delete records of ZigBee PRO stack context data by calling **PDM_vDeleteAllDataRecords()** before rejoining the same secured network. If these records are deleted, the destination node rejects the data sent by the node after rejoining since the frame counter has been reset on the source node. For more information and advice, refer to the "Application Design Notes" appendix in the ZigBee 3.0 Stack User Guide (JN-UG-3130).*

2.4 PDM features

2.4.1 Mutex in PDM

PDM functions are not re-entrant and a mutex is implemented within the PDM to enforce this. It works by disabling interrupts during any critical operations.

2.4.2 PDM event and error handler

The internal PDM library allows a handler to be called to alert the application of events and error conditions in the devices internal NVM. This callback function is registered either during the initialization of the PDM using the function **PDM_eInitialise()** or by calling the function **PDM_vRegisterSystemCallback()**. The PDM events/error conditions are listed and described in Section 10.1.3.

An application must trap E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_SPACE and E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED callback errors during testing. The ZigBee PRO stack uses multiple records. Once an ‘out of space’ error has occurred, the records would be in an inconsistent state. The software must be altered to use smaller record sizes or an external SPI Flash device, or more space is allocated at PDM initialization (if available). The PDM record sizes for the ZigBee PRO stack are dependent on table sizes set in the ZPS Configuration Editor.

The registered callback function may also be designed to handle a Wear Count event E_PDM_SYSTEM_EVENT_WEAR_COUNT_TRIGGER_VALUE_REACHED, which indicates that the Wear Count for an NVM segment has reached the configured trigger level (see Section 2.4.4).

2.4.3 NVM capacity

The NVM consists of 512 byte segments. The number of segments allotted to NVM is application-dependent. It depends on the number of records to be saved and their size as well as its value. The call to PDM_eInitialize sets its value, which is up to 110. A typical value of 63 is sufficient.

63 segments are used in ZigBee 3.0 applications. This allocation is set in the application code so that the user has control of it. The segments allotted to NVM must be balanced with the firmware size and possibly another firmware image if OTA is enabled.

The internal PDM library can store no more than one data record in each segment, although a large record may be stored across multiple segments. The PDM library must store some system information in each segment, so in practice each segment can hold only up to 502 bytes of record data. This means that a PDM record that has a single byte of information requires the same space as a 502 byte record and a 503 byte record requires two segments (the same as a 1004 byte record).

The function **PDM_u8GetSegmentCapacity()** returns the number of segments that are free for PDM. The function **PDM_u8GetSegmentOccupancy()** returns the number of segments that are in use. One of these functions may be called after all the records have been created and saved (including records in the ZigBee PRO stack). When updating a record, the PDM saves the new data before deleting the old data (to ensure that data is retained over any unexpected power cycles). Therefore, there must be sufficient capacity in the NVM to store another copy of a record before the old copy is deleted. To allow for the worst-case scenario, the value returned by **PDM_u8GetSegmentCapacity()** must be greater than the number of segments required to store the largest record.

2.4.4 NVM wear count

An NVM device supports a limited number of data writes to each byte before the storage medium begins to fail. For the K32W041, K32W061, K32W1, or JN518x Flash, at least 100000 writes are guaranteed and a million writes must be possible. See the devices Datasheet. For each NVM segment, a record of the number of writes made to the segment so far is kept. This is the ‘Wear Count’, which is stored and maintained in the segment header. The PDM manages the use of NVM segments in a way that minimizes wear and attempts to spread the wear evenly across the segments.

The function **PDM_eGetSegmentWearCount()** allows the current value of the Wear Count of a particular segment to be obtained. It is also possible to set up the generation of an event when the Wear Count of any segment reaches a certain trigger level. This trigger level can be configured (for all segments) using the function **PDM_vSetWearCountTriggerLevel()**. The Wear Count event is E_PDM_SYSTEM_EVENT_WEAR_COUNT_TRIGGER_VALUE_REACHED and the user-defined PDM callback function (see Section 2.4.2) should be designed to process this Wear Count event.

2.4.5 Ensuring consistency of PDM records

The data in the PDM may differ in structure from what the application anticipates. The structures stored by the ZigBee PRO libraries can change due to altering table sizes in the ZPS Configuration Editor, as well as between releases of the ZigBee PRO stack libraries. Inconsistency can occur under the following circumstances:

- The internal NVM used by the PDM on a device is not erased when programming an application with the devices Production Flash Programmer. If multiple applications are run on the same hardware, it is unlikely that the structures are consistent between the applications.

- When a ZigBee Over-The-Air (OTA) software update is performed, the PDM data is not erased. This update is normally a benefit because it allows the application to rejoin the network. However, if any of the PDM structures change, a factory reset must be performed by calling **PDM_vDeleteAllDataRecords().**

Applications normally contain a way to perform a factory reset of the PDM module. For example, by calling **PDM_vDeleteAllDataRecords()**, if a button is held down during reset.

The application can automatically check for PDM consistency by storing an application-specific 'magic number' in a record. A new magic number should be used if the application software or ZigBee PRO libraries PDM usage is inconsistent with the previous version of the software. Immediately after calling **PDM_eInitialise()**, the application should call **PDM_eReadDataFromRecord()**. If the magic number does not match, the application should call **PDM_vDeleteAllDataRecords()** to erase all records before attempting to start the ZigBee PRO stack. If the call to **PDM_eReadDataFromRecord()** indicates that the record has not been found, the application should also call **PDM_vDeleteAllDataRecords()**. This function must be called because another application might be running that does not use the same record ID but has written inconsistent ZigBee PRO records to the PDM module.

3 Power Manager

This chapter describes the Power Manager (PWRM) module, which manages the transitions of the device into and out of low-power modes.

PWRM module uses wake up timer 1 for activity scheduling, the application shall not use wake up timer 1 directly. If the application must use the second wake-up timer (wake up timer 0) independently of the PWRM, the WTIMER APIs should be used. WTIMER APIs allow the PWRM to enable a wake from wake-up timer 0 prior to entering Sleep mode.

Low-power modes are used to prolong the battery life of a node by reducing the power consumption of the device during periods when the node does not need to receive, transmit, or perform any other activities. Therefore, low-power modes only apply to End Devices, as the Coordinator and Routers must always remain fully alert for routing purposes.

3.1 Low-Power modes

A number of low-power modes are available on the device. In descending order of power consumption, the modes are:

- Doze mode
- Sleep modes:
  - Sleep with memory held
    - Sleep with memory held, 32 kHz Oscillator running
    - Sleep with memory held, 32 kHz Oscillator not running
  - Sleep without memory held, 32 kHz Oscillator running
- Deep Sleep mode without memory held, 32 kHz Oscillator not running

When the node is inactive, the Power Manager puts the device into the lowest power mode possible.

The above low-power modes are described in the subsections below. For further information on the low-power modes of a device, refer to the relevant device data sheet.

3.1.1 Doze Mode

In Doze mode, the CPU of the chip pauses (the CPU clock is stopped) but all other parts of the device continue to run. Any interrupt causes Doze mode to terminate and the application program continues running from the

next instruction. To prevent the Watchdog firing when in Doze mode, the application must ensure that a timer is running at a higher frequency than the Watchdog expiry period.

3.1.2 Sleep mode with memory held

During Sleep mode with memory held, the contents of on-chip RAM are maintained, including stack context data and application data. Therefore, on waking, the device can recover from sleep very quickly to continue normal operation from the next instruction.

In this mode, all power domains are powered down except the ones for the on-chip RAM, LDO always ON, and LDO MEM supplies. In addition, the 32 kHz on-chip oscillator can optionally be left running, which allows the device to be woken from sleep using Wake Up Timers. Otherwise, the device can only be woken by changes on the DIO pins replace by, or NTAG FD (field detect (JN518xT or K32W061 only)) or comparator.

Although memory contents are retained, it is still necessary to reconfigure the IEEE 802.15.4 stack layers and re-initialize most of the on-chip peripherals upon waking. Wake callback functions can be registered for this purpose:

- Do not re-initialize the DIOs, Wake Up Timers.
- Re-initialize everything else, including all other on-chip peripherals, the IEEE 802.15.4 MAC layer. If using callbacks, the Programmable Interrupt Controller (PIC), the callback functions must be re-registered.
- Reconfigure any DIOs that were reconfigured prior to going to sleep in order to minimize current drain. This reconfiguration may include setting the IO mux and pull-up/pull-down settings for those DIOs.

3.1.3 Sleep mode without memory held

During Sleep mode without memory held, on-chip RAM is powered down, and therefore stack context data and application data are not preserved on-chip. Normally, this data must be saved to NVM before the chip enters Sleep mode, and then recovered from NVM on waking (see Chapter 2).

On waking, the application program must be reloaded from flash memory before the node can resume operation. All variables and peripherals must be reinitialized, except the ones used as wake sources and the DIO lines.

3.1.4 Deep Sleep mode

In Deep Sleep mode, all switchable power domains are powered down and the 32 kHz oscillator is stopped. The device can be woken from deep sleep either via a hardware reset (RESETN pin), a DIO line change, the NTAG Field Detect (JN518xT or K32W061 only), or the analog comparator event.

On waking, the application program must be reloaded from flash memory before the node can resume operation. All variables and peripherals must be reinitialized, including the DIO lines.

3.2 Wake-up source from Low-Power modes

Wake up source is selected both by the requested low-power mode when calling *PWRM_vInit()* and *PWRM_vWakeUpConfig()* API.

3.2.1 Timer wake-up

In Doze mode, the CPU is in WFI, all interrupt sources are sources of wake-up provided the interrupt line is activated.

In sleep modes, there are limited sources of wake-up.

If the 32 kHz oscillator remains active during low-power mode, the chip can wake up from low-power mode on a scheduled activity timer or wake-up Timer 0. For further information, see *PWRM_eScheduleActivity() API*.

Note:

The Wake Up Timer 1 is reserved for PWRM). It is the application responsibility to program the Wake Up Timer 0 correctly and enable the interrupt for wake-up as done during active. If no timer is programmed, the PWRM does not enter low-power mode while 32 kHz oscillator is kept ON in low-power mode. It switches to Doze mode instead.

If the 32 kHz oscillator is switched OFF during low-power mode, no wake-up by Timer from low-power mode is possible.

3.2.2 DIO wake-up

DIO wake-up is allowed in all sleep modes even in deep Sleep mode. *PWRM_vWakeUpConfig()* configures the DIO wake-up. This API should be called after *PWRM_vInit()*.

Note: *PWRM_vWakeUpConfig() deprecates PWRM_vWakeUpIO().*

3.2.3 NTAG FD wake-up

TAG Field detect wake-up is allowed in all sleep modes even in deep Sleep mode. *PWRM_vWakeUpConfig()* configures it. This API should be called after *PWRM_vInit()*.

The application ensures that the NTAG Field detect interrupt is properly configured before going to Sleep mode.

3.2.4 Analog Comparator wake-up

Analog Comparator wake-up is allowed in all sleep modes except deep Sleep mode. *PWRM_vWakeUpConfig()* configures it. The application ensures that the Analog comparator is correctly set up and interrupt is enabled properly before going to Sleep mode.

3.3 Callback functions for Power Manager

If you intend to use the Power Manager, a number of callback functions must be available for the Power Manager to call in order to:

- Start the application (see Section 3.3.1)
- Perform housekeeping tasks when entering and leaving low-power mode (see Section 3.3.2)
- Handle interrupts from Wake Up Timer 1 (see Section "7.2.4 PWRM_vWakeInterruptCallback").

3.3.1 Essential callback function

For cold start (Sleep modes without RAM held), call AppColdStart() from hardware_init() function after the BOARD_InitHardware() initializes peripherals. An example of the hardware_init() function looks as shown below:

```
void hardware_init(void)
{
BOARD_InitHardware();
AppColdStart();
}
```

Note: *If the OSAbstraction component is integrated, then the application code is implemented from main_start() function.*

For Warm start (Sleep modes with RAM held), implement int WarmMain (void). The AppWarmStart() function should be called after the BOARD_InitHardware() initializes the peripherals.

```
intWarmMain(void)
{
BOARD_InitHardware();
AppWarmStart();
}
```

3.3.1.1 PWRM initialization

PWRM_vInit()

Initialization should be done on cold start (Sleep without RAM held) only. It is safe to call **PWRM_vInit()** after **AppColdStart()** have been called. There is no need to call on Warm start since the PWRM context is held. The only exception is for recalibrating the 32 kHz oscillator that is performed in the PWRM_vInit() function.

See Section 3.4, "Initializing and Starting the Power Manager".

3.3.2 Pre-sleep and post-sleep callback functions

To implement low-power modes, you must provide the Power Manager with user-defined callback functions to perform housekeeping tasks when the node enters and leaves low-power mode. Registration functions are provided for these callback functions, where the registration functions must be called in the user-defined callback function **vAppRegisterPWRMCallbacks()**.

- The pre-sleep callback function is called by the Power Manager just before putting the device into low-power mode. This callback function is registered in your code through the API function **PWRM_vRegisterPreSleepCallback()**.
- The post-sleep callback function is called by the Power Manager just after the device leaves low-power mode (irrespective of how the device was woken from sleep). This callback function is registered in your code through the API function **PWRM_vRegisterWakeupCallback()**.
- If not done in the Board_InitHardware() function, typical use is to restore the DIO lines to its primary function before sleep, or to restore some power domains, radio, Zigbee Power domain.

***Note:** Note: If a post-sleep function is registered in the **Board_InitHardware()** function before **AppColdStart()**, the registered callback is called by **AppColdStart()** when executed.*

vAppRegisterPWRMCallbacks() is called by the application as part of a cold start.

The pre-sleep and post-sleep callback function themselves must each be declared in the code using the macro:

PWRM_CALLBACK(*fn_name***)** where *fn_name* is the name of the callback function.

Each of these callback functions must also have a descriptor. This structure is used in the above registering functions to specify the callback function to register.

The callback descriptor must be declared using the macro:

PWRM_DECLARE_CALLBACK_DESCRIPTOR(*desc_name***, *fn_name*)** where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vPreSleepCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(pscb1_desc,vPreSleepCB1);
```

3.4 Initializing and starting the Power Manager

The Power Manager is initialized and started using the function **PWRM_vInit()**. This function requires one of five possible low-power configurations to be specified:

- Sleep with 32 kHz oscillator running and memory held.
- Sleep with 32 kHz oscillator running and memory not held.
- Sleep with 32 kHz oscillator not running and memory held.
- Sleep with 32 kHz oscillator not running and memory not held. This mode is also called deep-sleep mode.

The specified configuration is the low-power mode in which the Power Manager attempts to put the device during inactive periods.

Note: *Doze mode cannot be explicitly specified. However, the Power Manager may put the device into Doze mode at times when the specified mode cannot be entered (see Section 3.9.1).*

The criteria for selecting a Sleep mode are as follows:

- **Oscillator setting:**
  - If the 32 kHz (chip-dependent) oscillator is left running during sleep, a wake point can be scheduled using **PWRM_vScheduleActivity()** - see Section 3.7.
  - Otherwise, an external event can also be used to wake the device.
- **Memory setting:**
  - If memory is held during sleep, stack context data and application data are preserved in memory. This setting allows the device to resume operation quickly through a warm restart following sleep.
  - Sleep without memory held provides a greater power saving. However, stack context data and application data must be saved to the file system before entering Sleep mode. Also, ensure to restore stack context data and application data into on-chip memory during a cold restart on exiting sleep (see Chapter 2).

3.5 Enabling Power-Saving

To enable the Power Manager to put the device into low-power mode at appropriate times, you must call the function **PWRM_vManagePower()**, normally from an idle loop. When possible, the Power Manager puts the device into the Sleep mode specified through **PWRM_vInit()** once this function is called (or, alternatively, into Doze mode - see Section 3.9.1).

3.6 Non-interruptible activities

In order to enter Sleep mode, no activity must be running that must not be interrupted by sleep. This condition for entering Sleep mode is monitored using an activity counter - Sleep mode can only be entered when this counter is zero. The application is responsible for maintaining the activity counter, as follows:

- Whenever an activity is started that must not be interrupted by sleep, the application must notify the Power Manager using the function **PWRM_eStartActivity()**, which increments the activity counter.
- Whenever such an activity is completed, the application must notify the Power Manager using the function **PWRM_eFinishActivity()**, which decrements the activity counter.

CAUTION: *Application must only call the **PWRM_eFinishActivity()** following a matching call to **PWRM_eStartActivity()**. The ZigBee PRO stack also uses the activity counter, so calling **PWRM_eFinishActivity()** inappropriately can leave the ZigBee PRO stack in an inconsistent state.*

You can obtain the current value of the activity counter using the function **PWRM_u16GetActivityCount()**.

3.7 Scheduling wake events

Note: *This section is only applicable to the Sleep mode in which the 32 kHz oscillator is left running and memory is held.*

- For wake from DIO events, the application checks the SDK API **POWER_GetIoWakeStatus()** API to establish the DIO responsible for the wake-up event.
- For wake-up from the analogue comparator or NTAG Field Detect (JN518xT or K32W061 only) the application checks the interrupt lines in the CPU Interrupt Controller (NVIC).

In **PWRM_vInit()**, if you have selected the Sleep mode with the 32 kHz oscillator running and memory held, you can schedule wake events, which ensure that the device is awoken at certain times. This implies that if the device is sleeping, it is woken at the scheduled time. This scheduling uses Wake Up Timer 1 of the device, which operates at 32 kHz.

A wake event can be scheduled using the function **PWRM_eScheduleActivity()**. This function requires you to specify the number of mSec of the Wake Up Timer until the wake event. Also ensure to specify the user-defined callback function that must be called when the wake event occurs.

When the Wake Up Timer expires for a scheduled wake event, an interrupt is generated. The interrupt handler of the application then calls the pre-defined callback function **PWRM_WakeInterruptCallback()**. This function maintains the list of scheduled wake events and, if necessary, restarts the Wake Up Timer for the next scheduled wake event. The function also calls the user-defined callback function specified through **PWRM_eScheduleActivity()**.

Note: *In addition, when the device wakes from sleep, the user-defined callback function registered through **PWRM_vRegisterWakeupCallback()** is also called. However, this function is a general-purpose wake-up function which is called irrespective of how the device has been woken. (It is called not only for scheduled wake events, but also called for external wake events.)*

3.8 Terminating Low-Power mode

Low-power modes can be terminated in a number of ways:

- **Any Interrupt:** When in Doze mode, any interrupt can wake the device.
- **Wake Up Timer:** When in Sleep mode in which the 32 kHz oscillator runs, a scheduled wake event configured using the function **PWRM_vScheduleActivity()** can wake the device. For more information on scheduled wake events, refer to Section 3.9.
- **External Wake Event:** The following external wake events are available:
  - **DIO:** When in Sleep and Deep Sleep modes, a change of state of a DIO line can wake the device.
  - **NTAG Field detect** from the internal NTAG (JN518xT or K32W061 only).
  - **Analog comparator event**.

    The functions of the SDK API can control the above external wake events.

The valid wake sources for the different low-power modes are summarized in Table 2 below.

On leaving low-power mode, the Power Manager calls the user-defined callback function that has been registered using **PWRM_vRegisterWakeupCallback()**.

Table 2. Valid wake sources for Low-Power modes

Low-Power Mode	Wake Source					
	Any Interrupt	Wake Up Timer	DIO	Hardware Reset	NTAG FD	Analog Comparator
Doze mode	Yes	Yes	Yes	Yes	Yes	Yes

Table 2. Valid wake sources for Low-Power modes...*continued*

Low-Power Mode	Wake Source					
	Any Interrupt	Wake Up Timer	DIO	Hardware Reset	NTAG FD	Analog Comparator
Sleep mode with oscillator running	No	Yes	Yes	Yes	Yes	Yes
Sleep mode without oscillator running (deep-sleep mode)	No	No	Yes	Yes	Yes	Yes

3.9 Doze mode

Doze mode is a lighter power-saving mode than the sleep modes, as all elements of the device remain powered but the CPU is paused. CPU clock is stopped.

This low-power mode cannot be explicitly selected in **PWRM_vInit()**. The Power Manager puts the device into Doze mode only in certain circumstances, described in Section 3.9.1 below. However, to enter Doze mode, the Power Manager must have been initialized using **PWRM_vInit()** and the power-saving modes must have been enabled using **PWRM_vManagePower()**.

3.9.1 Circumstances that lead to Doze mode

Although Sleep and Deep Sleep modes cannot be entered while there are activities running that must not be interrupted by sleep (see Section 3.6), the Power Manager can put the device into Doze mode while the activity counter is non-zero.

Even when the activity counter is zero, if a Sleep mode has been configured with the 32 kHz oscillator running (see Section 3.4) but no wake event has been scheduled (see Section 3.7), the Power Manager puts the device into Doze mode instead of Sleep mode.

The decision to put a device into a Sleep mode or Doze mode is illustrated in the flowchart in Figure 2.

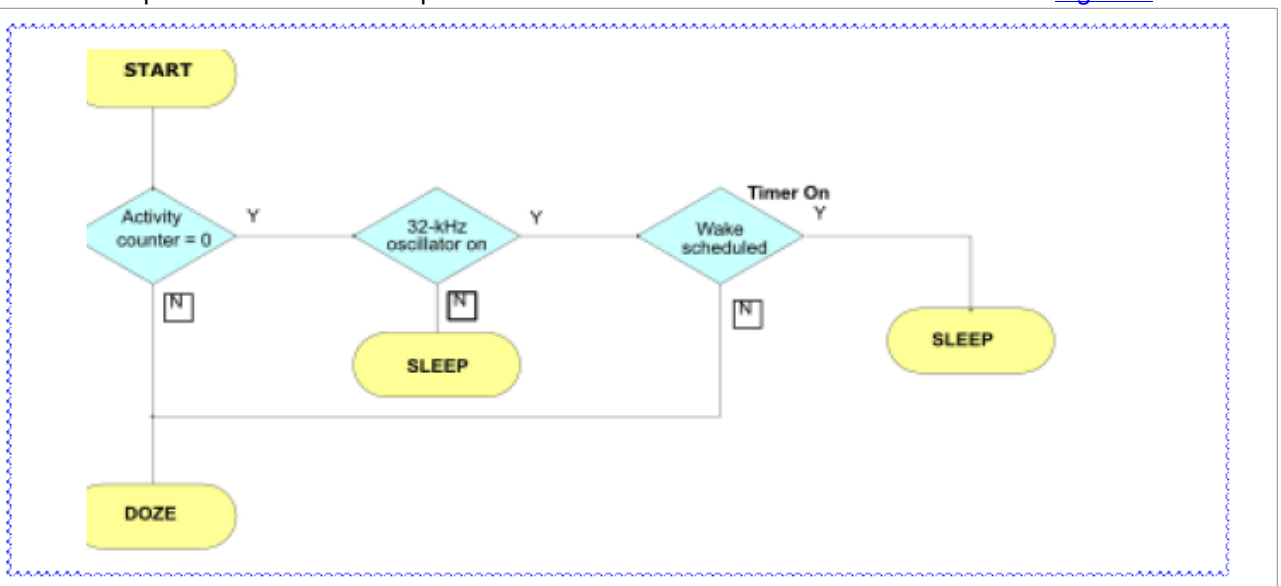

Figure 2. Flowchart of decision to enter Doze mode

3.9.2 Doze mode monitoring during development

Depending on the circumstances described in Section 3.9.1, the device may spend a significant proportion of its time in Doze mode. Power Manager API provides a function that allows you to investigate the fraction of time that the device spends in Doze mode for a given application. The function provides a doze monitoring output on the DIO1 pin of the device. This functionality can be used when the application is running in Debug mode.

The function **PWRM_vSetupDozeMonitor()** must be called to start a monitoring session. The state of the DIO1 pin then reflects the doze state of the device, allowing you to make doze state measurements using external equipment. The fraction of time that the device spends in Doze mode can then be estimated as: *Total time in Doze mode during session / Elapsed time of session*, shown in Figure 3

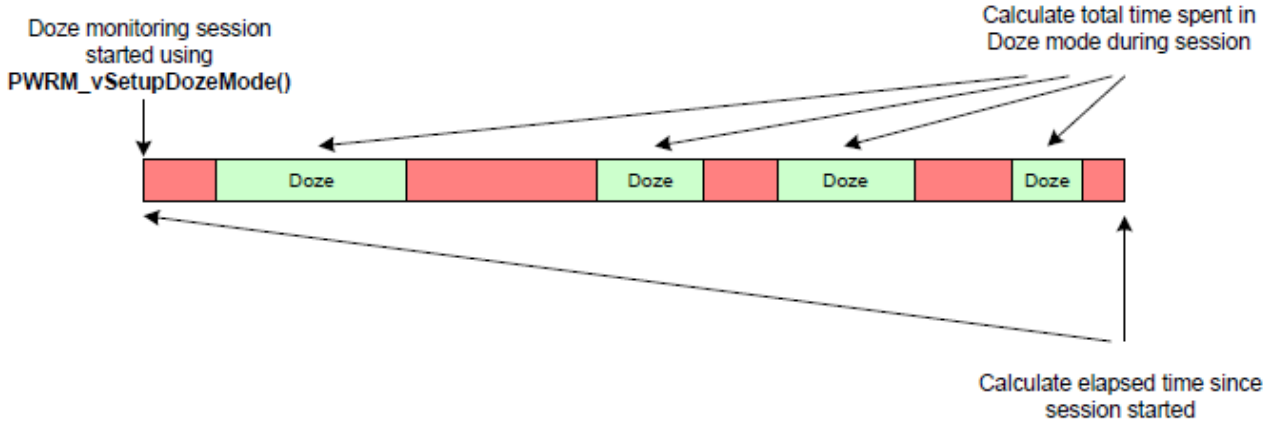

Figure 3. Doze monitoring

To obtain sensible results, doze monitoring should be allowed to run for a significant time.

4 Protocol Data Unit Manager

Communication between nodes in a wireless network is implemented using messages which contain application data. The part of a message which contains this data is called the Application Protocol Data Unit (APDU). The Protocol Data Unit Manager (PDUM) is concerned with the following:

- APDU memory management.
- Assembling and disassembling APDUs, that is, inserting data into APDUs to be transmitted and extracting data from received APDUs.

The PDUM is intended for use with ZigBee PRO applications.

4.1 Message assembly and disassembly

A message travels over a wireless network as a packet (usually an 802.15.4 packet). This message contains application data surrounded by header and footer information relating to the different layers of the protocol stack.

A message to be sent is prepared at the application level, at the top of the protocol stack, by creating an APDU containing the application data to be included in the message. This APDU is then passed down the layers of the stack, with each layer adding its own protocol information to the header and footer. On reaching the ‘physical’ layer at the bottom of the stack, the message is complete and ready to be transmitted.

For transmission, the message is converted to a Network Protocol Data Unit (NPDU). If the length of the message is greater than the packet size used in network communication (for example, 802.15.4 packet size), the message is divided up and transmitted in multiple NPDUs. You must be aware of this when using a sniffer to detect transmitted packets.

Note: *Data is stored in memory in the device in big-endian byte order but is transmitted over the network in little-endian byte order.*

A received message is passed up the protocol stack, with each stack layer stripping out the corresponding protocol information from the header and footer. On reaching the application level, only the APDU remains. The application data can then be extracted from this APDU.

The assembly and disassembly of a message, described above, are illustrated in Figure 4 n which the lower stack layers (MAC and Physical) are provided by the IEEE 802.15.4 protocol.

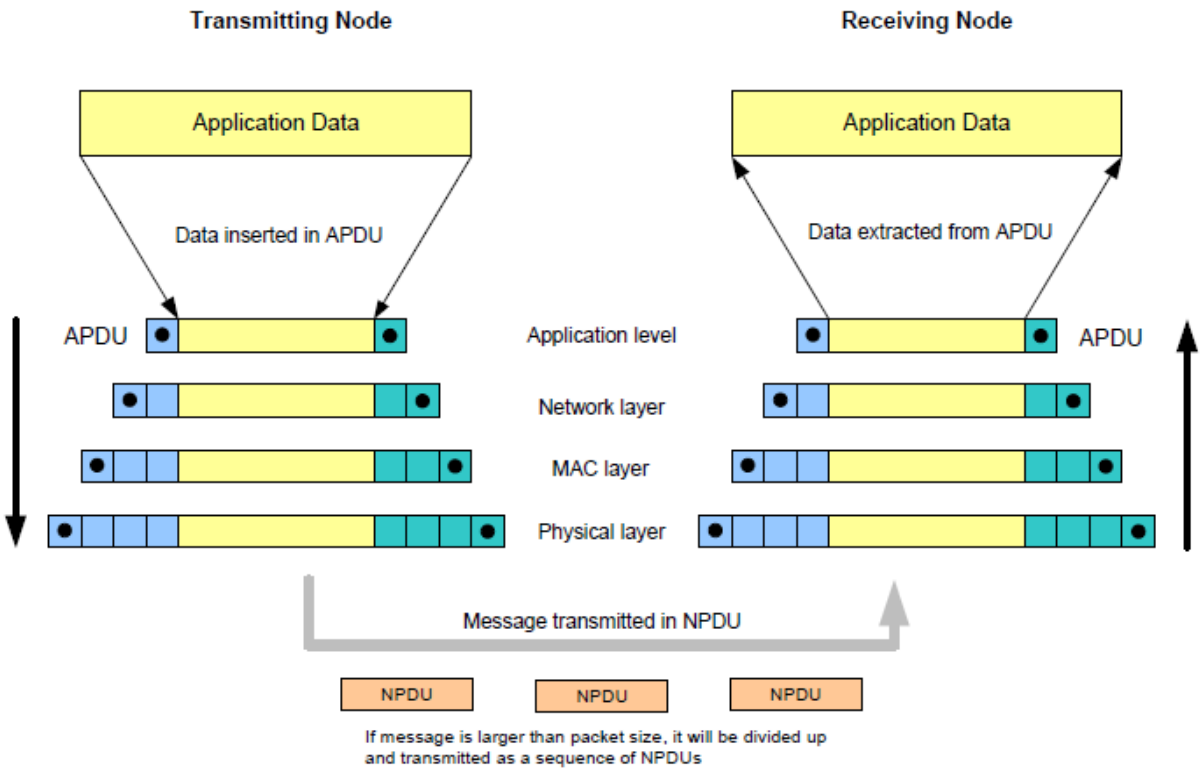

Figure 4. Message assembly and disassembly

4.2 Preparing the PDU Manager

In order to use the PDU Manager:

- Ensure to define the required APDUs statically using the ZPS Configuration Editor (described in the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*). Each APDU is given a unique handle. While the data payload of an APDU can be of arbitrary length, a maximum length is set for an APDU.
- Before calling any other PDUM functions in your code, you must call the function **PDUM_vInit()** to initialize the PDU Manager.

4.3 Inserting data into outgoing message

When sending a message to another node, you must first create an APDU containing the application data to be sent. First allocate an APDU instance by calling the function **PDUM_hAPduAllocateAPduInstance()** and then populate the APDU instance with data using **PDUM_u16APduInstanceWriteNBO()**, in which you must specify:

- The handle of the APDU instance in which data is to be inserted (**PDUM_hAPduAllocateAPduInstance()** returns this handle).
- The starting position of the data in the APDU - that is, the position of the least significant data byte.
- The format of the data payload - the data can be made up of a sequence of data values of different types.
- The data values to be inserted in the data payload.

Alternatively, the function **PDUM_u16APduInstanceWriteStrNBO()** can be used to populate the APDU instance - this function allows a data structure to be inserted into the APDU.

Then, ensure to use the relevant ZigBee PRO API function to send the message. Refer to the *ZigBee 3.0 Stack User Guide (JN-UG-3130)* for details. Once the message has been sent, the ZigBee PRO stack automatically de-allocates the memory-space used for the APDU instance.

***Note:** **PDUM_u16APduInstanceWriteNBO()** performs the necessary data conversion from big-endian byte order to little-endian byte order for transmission.*

Alternatively, you can produce your own code to insert data into the payload of an APDU. To help you, two functions are provided:

- **PDUM_pvAPduInstanceGetPayload()**: This function returns a pointer to the start of the payload section of the APDU instance.
- **PDUM_eAPduInstanceSetPayloadSize()**: This function sets the size, in bytes, of the data payload. This function is needed to provide the data size to the APDU instance, after having populated the APDU instance with data.

***CAUTION:** Data must be stored in memory in big-endian order but is transmitted over the network in little-endian byte order. Therefore, if you use your own code to insert data into an APDU, you must reverse the byte order of the data before inserting it. Failure to change the endianness of the data results in an alignment exception.*

4.4 Extracting data from incoming message

The function **PDUM_u16APduInstanceReadNBO()** provides an easy way of extracting the data payload from an incoming message. The **PDUM_u16APduInstanceReadNBO()** function requires the following to be specified:

- The handle of the APDU instance containing the data to be extracted (this handle is contained in the ZPS_EVENT_APS_DATA_INDICATION stack event that notified the application of the arrival of the data message).
- The starting position of the data in the APDU - that is, the position of the least significant data byte.
- The format of the data payload - the data can be made up of a sequence of data values of different types.
- A pointer to a structure in which the extracted data is stored.

Once the data has been extracted, you should de-allocate the memory space used for the APDU instance by calling the function **PDUM_eAPduFreeAPduInstance()**.

***Note:** **PDUM_u16APduInstanceReadNBO()** performs the necessary data conversion from little-endian byte order to big-endian byte order for storage.*

Alternatively, you can produce your own code to extract the payload data from an APDU. To help you, two functions are provided:

- **PDUM_pvAPduInstanceGetPayload()**: This function returns a pointer to the start of the payload data in the APDU instance.
- **PDUM_u16APduInstanceGetPayloadSize()**: This function returns the size, in bytes, of the data payload.

CAUTION: *Data is received from the network in little-endian byte order, but must be stored in memory in big-endian order. Therefore, if you use your own code to extract data from an APDU, you must reverse the byte order of the data before storing it. Failure to change the endianness of the data results in an alignment exception.*

5 Debug (DBG) Module

This chapter describes the Debug (DBG) module which allows application code to be debugged with diagnostic messages that are output to a display device.

5.1 Overview

The Debug module comprises an API containing diagnostic functions that can be embedded in your application code. Application debugging using the Debug module requires the device to be connected to a display device (such as a PC) via an IO interface, such as one of the on-chip UARTs. The display device must provide a dumb terminal through which output from the device can be viewed. A typical implementation is illustrated in Figure 5.

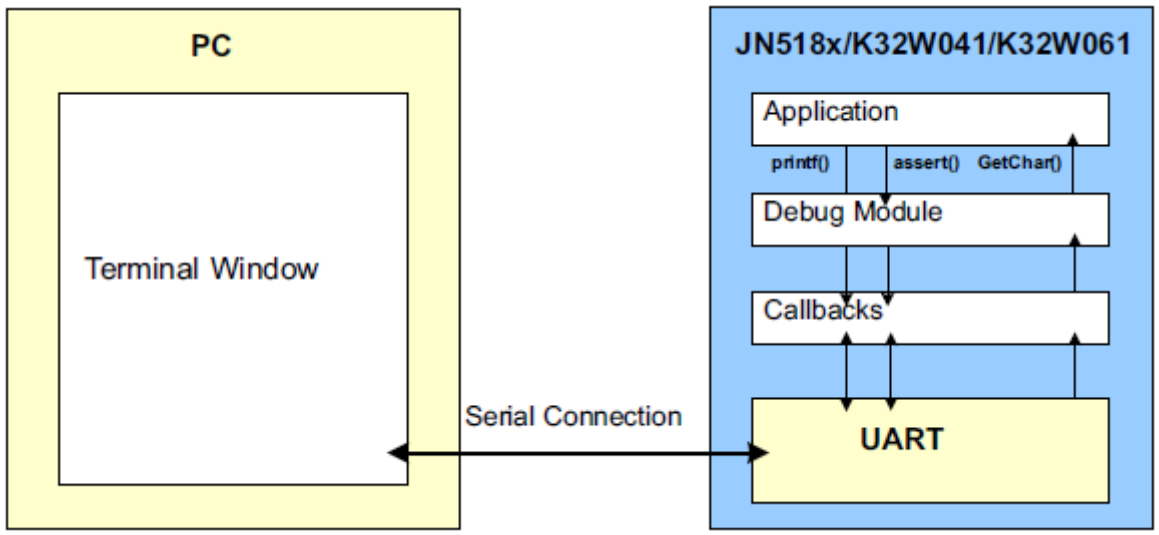

Figure 5. Typical hardware set-up for debugging

The API provides the essential printf- and assert-style debug functions, which can be strategically placed in your code:

- **DBG_vPrintf()** is used to output formatted strings and data values at an appropriate point during program execution, in order to indicate progress.
- **DBG_vAssert()** is used to test a logical condition, and to stop program execution when the test fails (condition is FALSE).

User-specified callback functions are used by the Debug module to control the IO interface (see Section 5.3).

The terminal on the PC can also supply input to the devices UART. The application uses the function **DBG_iGetChar()** to obtain a character from this input source. The application then handles this input.

5.2 Enabling the Debug module

The Debug module API is defined in the header file **DBG.h**, which must be included in your code.

The functions **DBG_vPrintf()** and **DBG_vAssert(),** each include a Boolean parameter that can be used to enable/disable individual instances of these functions. Two or more instances of these functions can be grouped to form a 'stream' for which this Boolean parameter is a common constant used to enable/disable the whole function group. This constant can be set at build time (see Section 5.4).

The Debug Module is built upon the SDK Debug Console functionality. See the SDK Reference Manual for further information on this.

5.3 Initializing and Configuring the Debug Module

The Debug Module does not require specific configuration. However, the SDK Debug Console must be initialized by calling DbgConsole_Init(). See the SDK Reference Manual for further information on this.

- If a device UART is to be used for output, the required initialization/configuration is as described in Section 5.3.1. This option is taken by most users.
- If any other serial IO interface is to be used for output, the required initialization/configuration is as described in Section 5.3.2.

Flags are provided in the global variable *DBG_u32Flags* for configuring certain aspects of the Debug module.

5.3.1 Using UART Input/Output

When the devices UART is to be used for the input/output of debug information, the configuration and initialization of the Debug module is accomplished with a single call to the function **DBG_vUartInit()**, which allows selection of the UART (0 or 1) and the baud-rate to be used. This function is used both during a cold start of the device and during a warm start (where the latter is a device re-start with memory contents retained).

5.3.2 Using Alternative Serial Output

When an alternative to an on-chip UART is to be used for the output of debug information, the required IO interface must first be configured and enabled (using the relevant functions from the JN51xx Integrated Peripherals API).

The Debug module must then be initialized using the function **DBG_vInit()**. This function is used both during a cold start of the device and during a warm start (where the latter is a device re-start with memory contents retained). The function takes as input a structure which contains pointers to four callback functions needed for debugging:

```
typedef struct
{
   void (*prInitHardwareCb)(void);
   void (*prPutchCb) (char c);
   void (*prFlushCb) (void);
   void (*prFailedAssertCb)(void);
} tsDBG_FunctionTbl;
```

The callback functions are user-defined and are described in the table below.

Table 3. Callback Functions Specified in DBG_vInit()

Pointer	Callback Function
**prInitHardwareCb*	Function which re-initializes the IO interface after a warm start, e.g. when device wakes from sleep.

Table 3. Callback Functions Specified in DBG_vInit()*...continued*

Pointer	Callback Function
**prPutchCb*	Function used by **DBG_vPrintf()** to output a single character to the IO interface.
**prFlushCb*	Function used by **DBG_vPrintf()** to flush the IO interface buffer to allow buffered output characters to be displayed. If the output is unbuffered, this function should do nothing or wait for the last character output using the **putch()** function to be made available. Note that the function should not append a newline character, as this should be handled by the format-ting string passed to **DBG_vPrintf()**.
**prFailedAssertCb*	Function which is called when **DBG_vAssert()** fails. The function should stop execution and may reset the device.

5.4 Example Diagnostic Code

The following code fragment illustrates use of the Debug module API. The devices UART 0 is used. Two debug 'streams' (1 and 2) are used to separately enable/disable two groups of debug lines.

```
#include <jendefs.h>
#include "app.h"
#include "dbg.h"
#ifndef DBG_STREAM_1
#define DBG_STREAM_1 FALSE
#endif
#ifndef DBG_STREAM_2
#define DBG_STREAM_2 FALSE
#endif
 int main(void)
 {
 int i = 0;
 /* Standard board pin, clock, debug console init (calls DbgConsole_Init)
 */
 BOARD_InitHardware();
 /* Now we can use DBG_vPrintf() and DBG_vAssert() to output characters
 to the UART device */
 DBG_vPrintf(DBG_STREAM_1, "Printing to stream 1\n");
 DBG_vPrintf(DBG_STREAM_2, "Printing an integer %i to stream 2\n", 10);
 DBG_vAssert(DBG_STREAM_1, i == 1);
}
```

When building this application, you have the following options:

- Debug disabled (the default)
- Debug enabled only for stream 1 - build with:

```
-DDBG_ENABLE-DDBG_STREAM_1=TRUE
```

- Debug enabled only for stream 2 - build with:

```
-DDBG_ENABLE-DDBG_STREAM_2=TRUE
```

- DBG enabled for both streams - build with:

```
-DDBG_ENABLE-DDBG_STREAM_1=TRUE-DDBG_STREAM_2=TRUE
```

Part II: Reference information

6 Persistent Data Manager API

This chapter details the functions of the Persistent Data Manager (PDM) API that supports context data and application data. The data is saved in Non-Volatile Memory (NVM) on the K32W041, K32W061, K32W1, or JN518x device which is held in the flash.

The API is defined in the header file `pdm.h` and is divided into the following categories:

- Internal NVM functions - see Section 6.1.
- Internal NVM PDM miscellaneous functions - see Section 6.2.

Note: ***Note:*** *For more information on how to use the functions described in this chapter, refer to Chapter 2.*

6.1 Internal NVM PDM functions

The PDM functions are listed below:

1. PDM_eInitialise
2. PDM_eSaveRecordData
3. PDM_eReadDataFromRecord
4. PDM_eDeleteData
5. PDM_eDeleteAllData
6. PDM_u8GetSegmentCapacity
7. PDM_u8GetSegmentOccupancy
8. PDM_bDoesDataExist

Note: *For the description of how to use these functions, refer to Section 2.3.*

6.1.1 PDM_eInitialise

```
PDM_teStatusPDM_eInitialise(
            uint16 u16StartSegment,
            uint8 u8NumberOfSegments,
PDM_tpfvSystemEventCallback
fpvPDM_SystemEventCallback;
```

Note: *The function prototype in the header file includes two additional parameters as conditional build options. These are not present in the library as provided in the SDK.*

Description

This function initializes the PDM module and registers the required PDM functions. It must be called during both a warm start and a cold start.

The function initializes the PDM environment and builds the underlying Flash file system. A RAM-based file system is created to allow the PDM to map data to/from the Flash. The Flash sectors are scanned for evidence of any valid user data, which is mapped into the RAM file system. This routine handles any write errors that may have occurred if the Flash was powered down whilst data was being written to the PDM system. Once the file system has been constructed, you can then write data to and read data from the Flash via PDM.

The region of flash to use for the PDM is specified by *u16StartSegment* and *u8NumberOfSegments*. A segment is 512 bytes. The PDM can operate within any number of Flash segments, as specified through the parameter

u8NumberOfSegments. A zero value results in an error code of PDM_E_STATUS_INVLD_PARAM being returned. For the device SDK, there is no need for the application to specify a mutex to control access to the PDM. The mutex is implemented within the PDM itself. However, it is necessary for the application build process to define the build token PDM_NO_RTOS so that the PDM header file is parsed correctly.

Parameters

- *u16StartSegment*: First segment in flash to use for PDM data storage
- *u8NumberOfSegments*: Number of contiguous Flash sectors to be managed.
- *fpvPDM_SystemEventCallback* : Function in the application to be called when a PDM system event has occurred. This function can also be set or changed by calling *PDM_vRegisterSystemCallback* (see Section 6.2).

Returns

PDM_E_STATUS_INVLD_PARAM

6.1.2 PDM_eSaveRecordData

```
PDM_teStatusPDM_eSaveRecordData(
    uint16 u16IdValue,
    uint8 *pu8DataBuffer,
    uint16 u16Datalength);
```

Description

This function saves the specified application data from RAM to the specified record in NVM. The record is identified by means of a 16-bit user-defined value.

CAUTION: *The application software must not use record identifier values that would clash with those used by the NXP libraries used with the application. The ZigBee PRO stack libraries use values above 0x8000.*

When a data record is saved to the NVM for the first time, the data is written provided there are enough NVM segments available to hold the data. Upon subsequent save requests, if there was a change between the RAM-based and NVM-based data buffers, then the PDM attempts to re-save only the segments that have changed. Consequently, if no data has changed, no save is performed. This is advantageous due to the restricted size of the NVM and the constraint that old data must be preserved while saving changed data to the NVM.

Provided that you have registered a callback function with the PDM (see Section 6.2), the callback mechanism signals when a save has failed. Upon failure, the callback function is invoked and passes the event `E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED` to the application.

Parameters

- *u16IdValue*: User-defined ID of the record to be saved (see Caution above)
- **pu8DataBuffer*: Pointer to data buffer to be saved in the record in NVM
- *u16Datalength*: Length of data to be saved, in bytes

Returns

- PDM_E_STATUS_OK (success)
- PDM_E_STATUS_INVLD_PARAM (specified record ID is invalid)
- PDM_E_STATUS_NOT_SAVED (save to NVM failed)

6.1.3 PDM_eReadDataFromRecord

```
PDM_teStatusPDM_eReadDataFromRecord(
  uint16 u16IdValue,
  void *pvDataBuffer,
  uint16 u16DataBufferLength,
  uint16* pu16DataBytesRead);
```

Description

This function reads the specified record of application data from the NVM and stores the read data in the supplied data buffer in RAM. The record is specified using its unique 16-bit identifier.

Before calling this function, it may be useful to call **PDM_bDoesDataExist()** in order to determine whether a record with the specified identifier exists in the NVM and, if it does, to obtain its size.

Parameters

- *u16IdValue*: User-defined ID of the record to be read
- **pvDataBuffer*: Pointer to the data buffer in RAM where the read data is to be stored
- *u16DataBufferLength*: Length of the data buffer, in bytes
- **pu16DataBytesRead*: Pointer to a location to receive the number of data bytes read

Returns

- PDM_E_STATUS_OK (success)
- PDM_E_STATUS_INVLD_PARAM (specified record ID is invalid)

6.1.4 PDM_eDeleteData

```
PDM_vDeleteAllDataRecords(uint16 u16IdValue);
```

Description

This function deletes the specified record of application data in NVM.

Alternatively, all records in NVM can be deleted using the function **PDM_vDeleteAllDataRecords()**.

Parameters

u16IdValue: User-defined ID of the record to be deleted

Returns

None

6.1.5 PDM_eDeleteAllData

```
void PDM_vDeleteAllDataRecords(void);
```

Description

This function deletes all records in NVM, including both application data and stack context data, resulting in an empty PDM file system. The NVM segment Wear Count values are preserved (and incremented) throughout this function call.

CAUTION: *It is not recommended to delete records of stack context data before a rejoin of the same secured network. If these records are deleted, data sent by the node after the rejoin will be rejected by the destination node since the frame counter has been reset on the source node. For more details, refer to "Application Design Notes" Appendix in the ZigBee 3.0 Stack User Guide (JN-UG-3130).*

Alternatively, an individual record of application data can be deleted using the function **PDM_vDeleteDataRecord()**.

Parameters

None

Returns

None

6.1.6 PDM_u8GetSegmentCapacity

```
uint8 PDM_u8GetSegmentCapacity(void);
```

Description

This function returns the number of unused segments that remain in the NVM.

Parameters

None

Returns

Number of PDM NVM segments free

6.1.7 PDM_u8GetSegmentOccupancy

```
uint8 PDM_u8GetSegmentOccupancy(void);
```

Description

This function returns the number of used segments in the NVM.

Parameters

None

Returns

Number of NVM segments used

6.1.8 PDM_bDoesDataExist

```
bool_t PDM_bDoesDataExist(uint16 u16IdValue,
                          uint16 *pu16DataLength);
```

Description

This function checks whether data associated with the specified record ID exists in the NVM. If the data record exists, the function returns the data length, in bytes, in a location to which a pointer must be provided.

Parameters

- *u16IdValue*: User-defined ID of the record to be found
- ***pu16DataLength*: Pointer to location to receive length, in bytes, of data record (if any) associated with specified record ID

Returns

TRUE if data record found, FALSE otherwise.

6.2 Internal NVM PDM Miscellaneous Functions

The PDM miscellaneous functions include a function for registering a user-defined PDM system callback function and functions related to the Wear Counts of NVM segments. The functions are listed below:

1. PDM_vRegisterSystemCallback
2. PDM_vSetWearCountTriggerLevel
3. PDM_eGetSegmentWearCount

Note: *For a description of how to use these functions, refer to Section 2.4.2 and Section 2.4.4.*

6.2.1 PDM_vRegisterSystemCallback

```
void PDM_vRegisterSystemCallback(
          PDM_tpfvSystemEventCallback
          fpvPDM_SystemEventCallback);
```

Description

This function registers a user-defined callback function to handle PDM events and errors.

Parameters

fpvPDM_SystemEventCallback: Pointer to the application callback function. The function type **PDM_tpfvSystemEventCallback** is documented in Section 10.1.1. The events generated by the PDM library are documented in Section 10.1.3.

Returns

None

6.2.2 PDM_vSetWearCountTriggerLevel

```
void PDM_vSetWearCountTriggerLevel(
      uint32 u32WearCountTriggerLevel);
```

Description

This function sets the Wear Count value of an NVM segment at which a Wear Count event is triggered and the PDM callback function is activated. The invoked callback function is user-defined and is registered using the function **PDM_vRegisterSystemCallback()**.

The callback function is invoked only once for a particular segment, when the specified Wear Count value occurs. It is not invoked for every occurrence afterwards, when the segment Wear Count exceeds the trigger value).

Parameters

u32WearCountTriggerLevel: Wear Count value that triggers a Wear Count event

Returns

None

6.2.3 PDM_eGetSegmentWearCount

```
PDM_teStatusPDM_eGetSegmentWearCount(
           uint8 u8SegmentIndex,
           uint32 *pu32WearCount);
```

Description

This function obtains the current Wear Count value of the specified NVM segment.

Parameters

- *u8SegmentIndex*: Index of Flash segment for which Wear Count needed
- **pu32WearCount*: Pointer to location to receive obtained Wear Count value

Returns

- PDM_E_STATUS_OK (success)
- PDM_E_STATUS_INVLD_PARAM (an invalid parameter value was supplied)

7 PWRM API

This chapter describes the functions of the Power Manager (PWRM) API. The API is defined in the header file **pwrm.h**.

CAUTION: *The Power Manager uses Wake Up Timer 1 of the device if scheduled wake events are configured. In this case, do not use this Wake Up Timer for any other purpose in your application.*

The PWRM API functions are divided into the following categories:

- 'Core' functions, described in Section 7.1.
- 'Callback Set-up' functions, described in Section 7.2.

Note: *For more information on the API, refer to the `pwrm.h` file.*

7.1 Core Functions

The PWRM core functions are listed below:

1. PWRM_vColdStart
2. PWRM_vInit
3. PWRM_eStartActivity
4. PWRM_eFinishActivity
5. PWRM_u16GetActivityCount
6. PWRM_eScheduleActivity
7. PWRM_vManagePower
8. PWRM_vWakeUpConfig
9. PWRM_GetFro32KCalibrationValue

7.1.1 PWRM_vColdStart

```
void PWRM_vColdStart(void);
```

Description

This function should be called from **hardware_init()** after the hardware is initialized by **APP_vSetUpHardware()**. **PWRM_vColdStart()** does the following actions:

- call vAppRegisterPWRMCallbacks
- reset the Wake Up Timer IP if the reset cause if different from the power down reset
- Stop the PWRM timer if the reset cause is the power down reset
- call the post wake-up callbacks

PWRM_vColdStart() does not call the **PWRM_eScheduleActivity()** callback since the RAM is OFF

7.1.1.1

Parameters

None

Returns

None

7.1.2 PWRM_vInit

```
void PWRM_vInit(PWRM_tePowerMode ePowerMode);
```

Description

This function is used to initialize the Power Manager and specify the low-power mode in which the device should be put when inactive.

There are five possible low-power modes that can be specified:

- Sleep with 32-kHz oscillator running and memory held
- Sleep with 32-kHz oscillator running and memory not held
- Sleep with 32-kHz oscillator not running and memory held
- Sleep with 32-kHz oscillator not running and memory not held
- Deep Sleep (32-kHz oscillator not running and memory not held)

The enumerations for the above power modes are listed below and described in Section 10.2.1. For further information on these low-power modes and how to wake from them, refer to Section 3.1.

Note that if the Power Manager is unable to put the device into the specified low-power mode, it will put the device into Doze mode instead - see description of **PWRM_vManagePower()**.

If the 32-kHz oscillator is run, the device's Wake Up Timer 1 is calibrated and made available (and then must not be used for any other purpose).

Parameters

ePowerMode: The power mode to be used during sleep, one of:

- E_AHI_SLEEP_OSCON_RAMON
- E_AHI_SLEEP_OSCON_RAMOFF
- E_AHI_SLEEP_OSCOFF_RAMON
- E_AHI_SLEEP_OSCOFF_RAMOFF
- E_AHI_SLEEP_DEEP

Returns

- PWRM_E_OK
- PWRM_E_MODE_INVALID

7.1.3 PWRM_eStartActivity

```
PWRM_teStatus PWRM_eStartActivity(void);
```

Description

This function is used to notify the Power Manager that an activity has been started which must not be interrupted by sleep. Thus, while such an activity is running, the device does not enter sleep mode.

The function **PWRM_eFinishActivity()** must then be called when the activity has completed. However, if **PWRM_eStartActivity()** has also been called for other activities that have not yet finished, the device will not be able to enter sleep mode until **PWRM_eFinishActivity()** has been called for all such activities.

The activity for which **PWRM_eStartActivity()** is called does not need to be identified, since the function simply increments a counter of running activities that must not be interrupted by sleep.

There is an upper limit of 64K to the value of this counter. If this limit is exceeded, an overflow error is returned.

Parameters

None

Returns

- PWRM_E_OK (success)
- PWRM_E_ACTIVITY_OVERFLOW (activity counter limit exceeded)

7.1.4 PWRM_eFinishActivity

```
PWRM_teStatus PWRM_eFinishActivity(void);
```

Description

This function is used to notify the Power Manager that an activity has completed which was not to be interrupted by sleep.

The function call must be paired with a previous call to **PWRM_eStartActivity()**. Sleep mode cannot be entered until **PWRM_eFinishActivity()** has been called for all activities for which **PWRM_eStartActivity()** has been previously called.

The activity for which **PWRM_eFinishActivity()** is called does not need to be identified, since the function simply decrements a counter of running activities that must not be interrupted by sleep. Sleep mode must not be entered until this counter reaches zero. If this function is called when the counter is already zero, an underflow error is returned.

Parameters

None

Returns

PWRM_E_OK (success)
PWRM_E_ACTIVITY_UNDERFLOW (activity counter already zero)

7.1.5 PWRM_u16GetActivityCount

```
uint16 PWRM_u16GetActivityCount(void);
```

Description

This function obtains the current value of the activity counter which indicates the number of activities currently running that must not be interrupted by sleep. Sleep mode cannot be entered until the value of this counter is zero.

Parameters

None

Returns

Current value of activity counter

7.1.6 PWRM_eScheduleActivity

```
PWRM_teStatus PWRM_eScheduleActivity(
          pwrm_tsWakeTimerEvent *psWake,
          uint32 u32Ticks,
          void (*prCallbackfn)(void));
```

Description

This function can be used to add a wake point and associated callback function to a list of scheduled wake points and callbacks. The new wake point is linked to an exclusive 32-kHz software Wake Up Timer, through the specified structure.

The function takes as input the number of ticks of the Wake Up Timer until the scheduled wake point. When the Wake Up Timer expires, the device will be woken from sleep and the specified callback function will be called.

To use this function, the Power Manager must be configured through **PWRM_vInit()** to implement a low-power mode in which the 32-kHz oscillator is running and memory is held. If this is not ensured, the list of scheduled wake points will be lost when the device enters sleep mode.

The function returns an error (see below) if the 32-kHz oscillator has not been configured to run during sleep or the software Wake Up Timer is already running for another wake point.

Parameters

- **psWake:* Pointer to a structure to be populated with the wake point and callback function (see below).
- *u32Ticks:* The number of ticks of the 32-kHz Wake Up Timer until wake point.
- **prCallbackfn*: Pointer to callback function associated with wake point.

Returns

- PWRM_E_OK (Wake Up Timer started successfully)
- PWRM_E_TIMER_RUNNING (Wake Up Timer already running for another wake point)
- PWRM_E_TIMER_INVALID (oscillator not configured to run during sleep)

7.1.7 PWRM_vManagePower

```
void PWRM_vManagePower(void);
```

Description

This function instructs the Power Manager to manage the power state of the device. The device must be idle when this function is called, i.e. the function is typically called from the OS idle task.

Once this function has been called, whenever appropriate, the Power Manager will put the device into the low-power mode specified through the function **PWRM_vInit()**. To allow the device to enter sleep mode:

- No activities that are uninterruptable by sleep must be running - that is, the activity counter must be zero.
- If the 32-kHz oscillator will run during sleep, a wake point must have been scheduled using **PWRM_vScheduleActivity()** (this condition does not apply when the oscillator is not used)

If the above two conditions are not satisfied, the function puts the device into Doze mode instead of sleep mode. Doze mode simply pauses the on-chip CPU, leaving all components powered (example. radio), and requires an interrupt to be configured to wake the device.

Before putting the device into sleep mode, this function calls any user-defined callback functions that have been registered using the function **PWRM_vRegisterPreSleepCallback()**.

Parameters

None

Returns

None

7.1.8 PWRM_vWakeUpConfig

```
PWRM_teStatus PWRM_vWakeUpConfig(uint32_tio_mask);
```

Description

This function instructs the power manager the wake-up event from sleep mode excluding the Timer event (which is done byPWRM_eScheduleActivity) the wake-up event can be:

- any IO or set of IO
- NTAG field detect
- Analog Comparator

Parameters

In case of an IO wake-up is programmed, on Wakeup, the application shall check the IO status in the post wake-up callback to see if the wake-up source is IO (`POWER_GetIoWakeStatus()` from `fsl_power.h`). `PWRM_vWakeUpConfig()` does not apply for doze mode. To disable the wake-up sources list, set the `pwrm_config` to `0`.

Returns

- PWRM_E_OK if the bit field is valid
- PWRM_E_IO_INVALID if the pwrm_config is incorrect.

7.1.9 PWRM_GetFro32KCalibrationValue

```
uint32_tPWRM_GetFro32KCalibrationValue(void);
```

Description

This function get the 32KHz FRO clock frequency. If the calibration has already been done in PWRM_vInit(), the function returns immediately with the calibration value. Otherwise, the calibration will be performed.

Parameters

The application shall enable the FRO32K prior calling this function. If FRO32K is disabled,

- Then the function does nothing and returns 0.
- If the XTAL32MHz is not enabled, the function enables it for FRO32K calibration.
- The XTAL32M is not disabled on the return of the function.

Returns

Number of FRO 32KHz cycles in one second / 32KHz frequency. Optimal value if FRO32KHz is accurate is 32768.

7.2 Callback Set-up Functions

The PWRM callback set-up functions are used to introduce user-defined callback functions that must be defined when using the Power Manager.

The functions are listed below:

1. PWRM_vRegisterPreSleepCallback
2. PWRM_vRegisterWakeupCallback
3. vAppRegisterPWRMCallbacks
4. PWRM_vWakeInterruptCallback

7.2.1 PWRM_vRegisterPreSleepCallback

```
void PWRM_vRegisterPreSleepCallback(
        tsCallbackDescriptor *psCBDesc);
```

Description

This function is used to register a user-defined callback function that is called by the Power Manager before the device enters sleep mode. You must specify a pointer to a structure containing a descriptor for your callback function.

The callback function must have been declared using the macro **PWRM_CALLBACK(***fn_name***)**, where *fn_name* is the name of the callback function.

The callback descriptor must have been declared using the macro **PWRM_DECLARE_CALLBACK_DESCRIPTOR(***desc_name***,** *fn_name***)**, where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vPreSleepCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(pscb1_desc,vPreSleepCB1);
```

The callback function should perform any housekeeping tasks that are necessary before the device enters sleep mode.

Note that this registration function is normally called within the user-defined function **vAppRegisterPWRMCallbacks()**. This ensures that the callback is registered during a cold start.

Parameters

In Pre-sleep call back, the Application shall turned OFF the clock it does not use in sleep mode, and configure the pin muxing to avoid pad leakage. Also, if a debug console is used, it shall deinitialize the console before going to sleep.

**psCBDesc*: Pointer to callback descriptor structure

Returns

None

7.2.2 PWRM_vRegisterWakeupCallback

```
void PWRM_vRegisterWakeupCallback(
        tsCallbackDescriptor *psCBDesc);
```

Description

This function is used to register a user-defined callback function that will be called by the Power Manager when the device wakes from sleep (this may be due to a change on a DIO line or comparator input, or the expiry of a Wake Up Timer). You must specify a pointer to a structure containing a descriptor for your callback function.

The callback function must have been declared using the macro **PWRM_CALLBACK(***fn_name***)**, where *fn_name* is the name of the callback function.

The callback descriptor must have been declared using the macro **PWRM_DECLARE_CALLBACK_DESCRIPTOR(***desc_name***,** *fn_name***)**, where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vWakeUpCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(wucb1_desc, vWakeUpCB1);
```

The callback function should perform any housekeeping tasks that are necessary after the device wakes from sleep.

Note that this registration function is normally called within the user-defined function **vAppRegisterPWRMCallbacks()**. This ensures that the callback is registered during a cold start.

Parameters

In case of Warm Start, the application reconfigures all the Hardware peripherals in the wake-up call back (by calling **APP_vSetUpHardware()**).

**psCBDesc*: Pointer to callback descriptor structure

Returns

None

7.2.3 vAppRegisterPWRMCallbacks

```
void vAppRegisterPWRMCallbacks(void);
```

Description

This is a user-defined function to register pre-sleep and post-sleep callback functions, if required.

The function definition must itself use **PWRM_vRegisterPreSleepCallback()** and **PWRM_vRegisterWakeupCallback()** to register the required callbacks.

Parameters

None

Returns

None

7.2.4 PWRM_vWakeInterruptCallback

```
void PWRM_vWakeInterruptCallback(void);
```

Description

This function is a pre-defined callback function which must be called from the application's interrupt handler to deal with interrupts from Wake Up Timer 1 on the device.

The function is needed to maintain the scheduled wake points list, by restarting the Wake Up Timer for the next wake-up event (if any) when the previous one has just completed. The function also calls the user-defined callback function specified through **PWRM_vScheduleActivity()**.

Parameters

None

Returns

None

8 PDUM API

This chapter describes the functions of the Protocol Data Unit Manager (PDUM) API. The API is defined in the header file **pdum.h**.

The PDUM API functions are listed below:

1. PDUM_vInit
2. PDUM_hAPduAllocateAPduInstance
3. PDUM_eAPduFreeAPduInstance
4. PDUM_u16APduInstanceReadNBO
5. PDUM_u16APduInstanceWriteNBO
6. PDUM_u16APduInstanceWriteStrNBO
7. PDUM_u16SizeNBO
8. PDUM_u16APduGetSize
9. PDUM_pvAPduInstanceGetPayload
10. PDUM_u16APduInstanceGetPayloadSize
11. PDUM_eAPduInstanceSetPayloadSize
12. PDUM_vDBGPrintAPduInstance

Note: *In ZigBee PRO, the APDUs used by the application must be pre-defined (before building the application) using the ZPS Configuration Editor. This tool is detailed in the ZigBee 3.0 Stack User Guide (JN-UG-3130).*

8.1 PDUM_vInit

```
void PDUM_vInit();
```

Description

This function initializes the PDU Manager and must therefore be the first PDUM function called.

Parameters

None

Returns

None

8.2 PDUM_hAPduAllocateAPduInstance

```
PDUM_thAPduInstance
 PDUM_hAPduAllocateAPduInstance(
               PDUM_thAPdu hAPdu);
```

Description

This function allocates an instance of an Application Protocol Data Unit (APDU) - that is, memory space is allocated to the APDU instance.

The available APDUs (types and their handles) are pre-defined using the ZPS Configuration Editor (refer to the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*).

The allocated APDU instance can subsequently be populated with data and sent to another node.

Parameters

hAPdu: Handle of APDU (type)

Returns

- Handle of allocated APDU instance.
- PDUM_INVALID_HANDLE if no APDU instances are free.

8.3 PDUM_eAPduFreeAPduInstance

```
PDUM_teStatusPDUM_eAPduFreeAPduInstance(
              PDUM_thAPduInstance hAPduInst);
```

Description

This function de-allocates the specified APDU instance, thus freeing the associated memory space.

Parameters

hAPduInstance: Handle of APDU instance

Returns

PDUM_E_INTERNAL_ERROR

8.4 PDUM_u16APduInstanceReadNBO

```
uint16 PDUM_u16APduInstanceReadNBO(
        PDUM_thAPduInstance hAPduInst,
        uint16 u16Pos,
        constchar *szFormat,
        void *pvStruct);
```

Description

This function reads data from the specified APDU instance and inserts the data into a C structure. The byte position of the start (least significant byte) of the data in the APDU instance must be specified, as well as the format of the data.

Data is read from the APDU instance in packed network byte order (little-endian) and translated into unpacked host byte order for the C structure (big-endian).

Parameters

- *hAPduInst:* Handle of APDU instance to read the data from.
- *u16Pos:* The starting position (least significant byte) of the data within the APDU:
  - **szFormat:* Format string of the data:
  - b: 8-bit byte
  - h: 16-bit half-word (short integer)
  - w: 32-bit word
  - l: 64-bit long-word (long integer)
  - a\xnn: nn (hex) bytes of data (array)
  - p\xnn: nn (hex) bytes of packing
- ***pvStruct:* Pointer to C structure to receive the data.

Note: *Please note that the compiler does not correctly interpret the format string "a\xnnb" for a data array followed by a single byte, for example, "a\b". In this case, to ensure that the 'b' (for byte) is not interpreted as a hex value, use the format "a\xnn" "b". For example, "a\x0a" "b".*

Returns

- Total number of data bytes read from the APDU instance.

8.5 PDUM_u16APduInstanceWriteNBO

```
uint16 PDUM_u16APduInstanceWriteNBO(
           PDUM_thAPduInstance hAPduInst,
           uint16 u16Pos,
           constchar *szFormat,...);
```

Description

This function writes the specified data values into the specified APDU instance. The byte position of the start of the data (least significant byte) in the APDU instance must be specified, as well as the format of the data.

The data values are written into the APDU instance at the specified position in packed network byte order (little-endian). The input data values should be in host byte order (big-endian).

Parameters

- *hAPduInst:* Handle of the APDU instance to write the data into
- *u16Pos:* The starting position (least significant byte) of the data within the APDU instance:
  - **szFormat:* Format string of the data:
  - b: 8-bit byte
  - h: 16-bit half-word (short integer)
  - w: 32-bit word
  - l: 64-bit long-word (long integer)
  - a\xnn: nn (hex) bytes of data (array)
  - p\xnn: nn (hex) bytes of packing
- ...: Variable list of data values described by the format string

Note:

Please note that the compiler will not correctly interpret the format string "a\xnnb" for a data array followed by a single byte, for example: "a\b".

- *In this case, to ensure that the 'b' (for byte) is not interpreted as a hex value, use the format* ***"a\xnn" "b"****, such as* ***"a\x0a" "b"****.*

Returns

Total number of bytes written to the APDU instance.

8.6 PDUM_u16APduInstanceWriteStrNBO

```
uint16PDUM_u16APduInstanceWriteStrNBO(
        PDUM_thAPduInstance hAPduInst,
        uint16 u16Pos,
        constchar *szFormat,
        void *pvStruct);
```

Description

This function writes data from the specified structure into the specified APDU instance. The byte position of the start of the data (least significant byte) in the APDU instance must be specified, as well as the format of the data.

The data values are written into the APDU instance at the specified position in packed network byte order (little-endian). The input data values should be in host byte order (big-endian).

Parameters

- *hAPduInst*: Handle of the APDU instance to write the data into
- *u16Pos*: The starting position (least significant byte) of the data within the APDU instance
- **szFormat*: Format string of the data:
  - b8-bit byte
  - h16-bit half-word (short integer)
  - w32-bit word
  - l64-bit long-word (long integer)
  - a\xnn: nn (hex) bytes of data (array)
  - p\xnn: nn (hex) bytes of packing
- **pvStruct*: Pointer to C structure to containing data

Note:

Please note that the compiler does not correctly interpret the format string "a\xnnb" for a data array followed by a single byte (example: "a\b"). In this case, to ensure that the 'b' (for byte) is not interpreted as a hex value, use the format ***"a\xnn" "b"****, for example:* ***"a\x0a" "b"****.*

Returns

Total number of bytes written to the APDU instance.

8.7 PDUM_u16SizeNBO

```
uint16 PDUM_u16SizeNBO(constchar *szFormat);
```

Description

This function obtains the size, in bytes, of an APDU data payload, given the format of the data.

Parameters

- **szFormat*Format string of the data:
  - b8-bit byte
  - h16-bit half-word (short integer)
  - w32-bit word
  - l64-bit long-word (long integer)
  - a\xnn: nn (hex) bytes of data (array)
  - p\xnn: nn (hex) bytes of packing

Note:

Please note that the compiler does not correctly interpret the format string "a\xnnb" for a data array followed by a single byte, such as "a\b". In this case, to ensure that the 'b' (for byte) is not interpreted as a hex value, use the format "a\xnn" "b". For example, "a\x0a".

Returns

Number of bytes in data payload

8.8 PDUM_u16APduGetSize

```
uint16 PDUM_u16APduGetSize(PDUM_thAPdu hAPdu);
```

Description

This function obtains the maximum size, in bytes, of the specified APDU (type).

Parameters

hAPdu: Handle of APDU

Returns

Number of bytes in APDU

8.9 PDUM_pvAPduInstanceGetPayload

```
void *PDUM_pvAPduInstanceGetPayload(
       PDUM_thAPduInstance hAPduInst);
```

Description

This function obtains a pointer to the payload data of the specified APDU instance.

Parameters

hAPduInst: Handle of APDU instance to access

Returns

Pointer to data as an array of bytes.

8.10 PDUM_u16APduInstanceGetPayloadSize

```
uint16 PDUM_u16APduInstanceGetPayloadSize(
             PDUM_thAPduInstance hAPduInst);
```

Description

This function obtains the size, in bytes, of the payload data of the specified APDU instance.

Parameters

hAPduInst: Handle of APDU instance to access

Returns

Size of the payload data, in bytes.

8.11 PDUM_eAPduInstanceSetPayloadSize

```
PDUM_teStatusPDUM_eAPduInstanceSetPayloadSize(
        PDUM_thAPduInstance hAPduInst,
        uint16 u16Size);
```

Description

This function sets the size, in bytes, of the payload of the specified APDU instance. This function is needed to provide the data size to the APDU instance, after having populated the APDU instance with data.

Parameters

hAPduInst: Handle of APDU instance
u16Size: Size of payload to set, in bytes

Returns

- PDUM_OK.
- PDUM_E_APDU_INSTANCE_TOO_BIG.

8.12 PDUM_vDBGPrintAPduInstance

```
void PDUM_vDBGPrintAPduInstance(
            PDUM_thAPduInstance hAPduInst);
```

Description

This function can be used to output the specified APDU instance via the Debug (DBG) module.

For details of the DBG functions, refer to Chapter 9.

Parameters

hAPdu: Handle of APDU instance to output

Returns

None.

9 DBG API

The chapter describes the functions of the Debug (DBG) module API. The API is defined in the header file **dbg.h**.

To use the Debug module, it must be enabled at build-time by defining `DBG_ENABLE` in the build. This can be done, for example, by adding the `-DDBG_ENABLE` option to the compiler.

By default, the Debug module just displays each line as passed. However, if DBG_VERBOSE is defined at build-time, then each line displayed is prefixed with the file name and line number of the debug statement.

Note: *Compiling with the DBG option results in a larger application size, requiring a lot more space in RAM.*

The DBG API functions are listed below:

- DBG_vPrintf
- DBG_vAssert

9.1 DBG_vPrintf

```
void DBG_vPrintf(bool_t bStreamEnabled,
                 constchar *pcFormat,...);
```

Description

This function is an adapted **printf()** function, allowing a formatted string to be output (for example, via the UART) for display.

The function contains a parameter that allows the output of the string to be enabled or disabled. The value of this Boolean parameter must be a literal. If disabled, the compiler optimizes out this function, but its parameters are still evaluated.

The supported output formats are as follows:

Table 4. Supported output formats of printf() function

Format Specifier	Purpose
Flags	
-	Left align
0	Pad with zeroes
+	Sign with plus
‘ ‘ (space)	Sign with space
Width	
<integer>	Field width
Length	
l	Long
ll	Long long
h	Short
Type	
i	Signed integer
d	Signed integer
u	Unsigned integer
x	Unsigned integer as hexadecimal
p	Pointer
c	Character
s	String

Table 4. Supported output formats of printf() function*...continued*

Format Specifier	Purpose
Escape sequence	
\n	Newline/carriage return

Parameters

- *bStreamEnabled:* Boolean which determines whether string will be output:
  - TRUE: Output string
  - FALSE: Do not output string (compile out function)
- **pcFormat:* Pointer to printf-style formatting string.
  For supported output formats, see the above table.

Returns

None

9.2 DBG_vAssert

```
voidDBG_vAssert(bool_t bStreamEnabled,
                bool_t bAssertion);
```

Description

This function is an adapted **assert()** function, allowing a Boolean condition to be tested.

The function contains a parameter which allows the test to be enabled or disabled. The value of this Boolean parameter must be a literal. If disabled, the compiler will optimize out this function.

The Boolean condition to be tested is specified as a parameter:

- If the condition is TRUE, program execution continues.
- If the condition is FALSE, an error message is output and execution is passed to a callback function, which stops execution. This callback function is specified when **DGB_vInit()** is called for a cold start.

Parameters

- *bStreamEnabled*: Boolean which determines whether test will be performed:
  - TRUE: Perform test.
  - FALSE: Do not perform test.
- bAssertion: Boolean expression to be tested.

Returns

None.

10 JCU Structures

This chapter describes the structures (including enumerations) used by the JCU modules:

- PDM structures are detailed in Section 10.1
- PWRM structures are detailed in Section 10.2

- DBG structures are detailed in Section 10.3

10.1 PDM Structures

10.1.1 PDM_tpfvSystemEventCallback

This type defines the callback function that receives PDM events.

```
typedef void (*PDM_tpfvSystemEventCallback) (
      uint32                 u32eventNumber,
     PDM_eSystemEventCode   eSystemEventCode);
```

where:

- `u32eventNumber` gives further information about the event depending on the event code, as detailed in Section 10.1.3
- `eSystemEventCode` identifies the type of event that triggered the callback.

10.1.2 tsReg128

This is a constant structure which contains a 128-bit encryption key used by the PDM module - the key is passed into the module via the **PDM_vInit()** function.

```
typedefstruct
{
  uint32u32register0;
  uint32u32register1;
  uint32u32register2;
  uint32u32register3;
} tsReg128;
```

In the above structure, `u32register0` contains the 32 least significant bits and `u32register3` contains the 32 most significant bits of the key.

10.1.3 PDM_eSystemEventCode

This structure contains enumerations for the events generated by the PDM library.

```
typedefenum
{
    E_PDM_SYSTEM_EVENT_WEAR_COUNT_TRIGGER_VALUE_REACHED=0,
    E_PDM_SYSTEM_EVENT_SAVE_FAILED,
    E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_SPACE,
    E_PDM_SYSTEM_EVENT_LARGEST_RECORD_FULL_SAVE_NO_LONGER_POSSIBLE,
    E_PDM_SYSTEM_EVENT_SEGMENT_DATA_CHECKSUM_FAIL,
//Debug event codes
    E_PDM_SYSTEM_EVENT_NVM_SEGMENT_HEADER_REPAIRED,
    E_PDM_SYSTEM_EVENT_SYSTEM_INTERNAL_BUFFER_WEAR_COUNT_SWAP,
    E_PDM_SYSTEM_EVENT_SYSTEM_DUPLICATE_FILE_SEGMENT_DETECTED,
    E_PDM_SYSTEM_EVENT_SYSTEM_ERROR,
} PDM_eSystemEventCode;
```

The events are outlined in Table 5 below.

Table 5. PDM Event Codes (Flash)

Event Enumeration	Description
E_PDM_SYSTEM_EVENT_WEAR_COUNT_ TRIGGER_VALUE_REACHED	An NVM segment has reached a set Wear Count (set by the user or left at the manufacturer stated maximum value). `u32EventNumber` carries the NVM segment number.
E_PDM_SYSTEM_EVENT_SAVE_FAILED	A save has failed. `u32eventNumber` contains the `u16IdValue` of the record that failed to save. This is a fatal error as the stack records may be inconsistent. Test software should log this error and halt. Production software may need to perform a factory reset.
E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_ SPACE	There is not enough space to hold all the PDM records. `u32event Number` contains the `u16IdValue` of the record that was being processed. This is a fatal error as the stack records may be inconsistent. Test software should log this error and halt. Production software may need to perform a factory reset.
E_PDM_SYSTEM_EVENT_LARGEST_RECORD_ FULL_SAVE_NO_LONGER_POSSIBLE	The NVM occupancy is such that the largest record in the PDM can no longer be fully saved. `u32EventNumber` carries the `u16IdValue` of the record that was being processed.
E_PDM_SYSTEM_EVENT_SEGMENT_DATA_ CHECKSUM_FAIL	The calculated checksum for the data in an NVM segment does not match the stored checksum value. `u32EventNumber` carries the number of the segment.
E_PDM_SYSTEM_EVENT_NVM_ SEGMENT_ HEADER_REPAIRED	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_SYSTEM_INTERNAL_ BUFFER_WEAR_COUNT_SWAP	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_SYSTEM_ DUPLICATE_FILE_SEGMENT_DETECTED	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_SYSTEM_ERROR	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.

10.1.4 PDM_teStatus

This structure contains enumerations for the status codes generated by the PDM.

```
typedefenum
{
    PDM_E_STATUS_OK,
    PDM_E_STATUS_INVLD_PARAM,
//NVM based PDM codes
    PDM_E_STATUS_PDM_FULL,
    PDM_E_STATUS_NOT_SAVED,
    PDM_E_STATUS_RECOVERED,
    PDM_E_STATUS_PDM_RECOVERED_NOT_SAVED,
    PDM_E_STATUS_USER_BUFFER_SIZE,
    PDM_E_STATUS_BITMAP_SATURATED_NO_INCREMENT,
    PDM_E_STATUS_BITMAP_SATURATED_OK,
    PDM_E_STATUS_IMAGE_BITMAP_COMPLETE,
    PDM_E_STATUS_IMAGE_BITMAP_INCOMPLETE,
    PDM_E_STATUS_INTERNAL_ERROR
} PDM_teStatus;
```

The status codes are described in Table 6 below.

Table 6. PDM Status Codes

Event Enumeration	Description
PDM_E_STATUS_OK	The function completed without error.
PDM_E_STATUS_INVLD_PARAM	An invalid parameter value was supplied.
PDM_E_STATUS_PDM_FULL	There is no available Flash space for PDM.
PDM_E_STATUS_NOT_SAVED	A PDM save to Flash failed.
PDM_E_STATUS_RECOVERED	The record was recovered from a previous save to NVM.
PDM_E_STATUS_PDM_RECOVERED_NOT_SAVED	The record was not recovered from a previous save to NVM.
PDM_E_STATUS_USER_BUFFER_SIZE	Not used.
PDM_E_STATUS_BITMAP_SATURATED_NO_INCREMENT	Counter increment not made because the NVM segment is saturated.
PDM_E_STATUS_BITMAP_SATURATED_OK	Counter increment made but the NVM segment is now saturated.
PDM_E_STATUS_IMAGE_BITMAP_COMPLETE	For internal use.
PDM_E_STATUS_IMAGE_BITMAP_INCOMPLETE	For internal use.
PDM_E_STATUS_INTERNAL_ERROR	An unspecified internal PDM error has occurred.

10.1.5 PDM_tsHwFncTable

This structure is used in the function **PDM_vInit()** to specify a set of user-defined functions used to interact with a custom NVM device.

```
typedef struct
{
    /* This function is called after a cold or warm start */
      void (*prInitHwCb)(void);
   /* This function is called to erase the given sector */
      void (*prEraseCb) (uint8 u8Sector);
  /*This function is called to write data to an address
    * within a given sector. Address zero is the start of the
    * given sector */
      void (*prWriteCb) (uint8 u8Sector,
            uint16 u16Addr,
            uint16 u16Len,
            uint8 *pu8Data);
   /* This function is called to read data from an address
   * within a given sector. Address zero is the start of the
   * given sector */
     void (*prReadCb) (uint8 u8Sector,
            uint16 u16Addr,
            uint16 u16Len,
            uint8 *pu8Data);
} PDM_tsHwFncTable;
```

10.2 PWRM Structures

10.2.1 PWRM_teSleepMode

This structure contains the enumerations used to set the power mode of the device during sleep.

```
typedefenum
    {
    PWRM_E_SLEEP_OSCON_RAMON,      /*32-kHzOsconandRAMon*/
    PWRM_E_SLEEP_OSCON_RAMOFF,     /*32-kHzOsconandRAMoff*/
    PWRM_E_SLEEP_OSCOFF_RAMON,     /*32-kHzOscoffandRAMon*/
    PWRM_E_SLEEP_OSCOFF_RAMOFF,    /*32-kHzOscoffandRAMoff*/
    PWRM_E_SLEEP_DEEP,             /*DeepSleep*/
    }
PWRM_teSleepMode;
```

10.3 DBG Structures

10.3.1 DBG_tsFunctionTbl

This structure contains callback functions used by the Debug (DBG) module to interact with the output interface.

```
typedefstruct
{
    void(*prInitHardwareCb)(void);
    void(*prPutchCb)(charc);
    void(*prFlushCb)(void);
    void(*prFailedAssertCb)(void);
}  DBG_tsFunctionTbl;
```

For details of the callback functions, refer to the description of Section 9.

11 Revision History

The Table 7 summarizes the revisions to this document.

Document revision history

Version	Date	Comments
2.2	03 March 2023	Added support for K32W1 devices
2.1	25-May-2022	Updated to standard NXP template format and other minor updates
2.0	19-Nov-2019	Updated for K32W\JN5189
1.0	12-June-2018	First release

12 Legal information

12.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

12.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

12.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Matter, Zigbee — are developed by the Connectivity Standards Alliance. The Alliance's Brands and all goodwill associated therewith, are the exclusive property of the Alliance.

NTAG — is a trademark of NXP B.V.

Contents

1 Organization 2
2 Conventions 2
3 Acronyms and abbreviations 2
4 Related documents 3
5 Support Resources 3
6 Trademarks 3
7 Chip Compatibility 3
1 Introduction 4
1.1 Modules and architecture 4
1.1.1 JCU modules 4
1.1.2 Software architecture 4
2 Persistent Data Manager 5
2.1 Overview 5
2.2 Initializing the PDM and building a file system 6
2.2.1 Building applications that use PDM 6
2.3 Managing data in non-volatile memory 6
2.3.1 Saving data to non-volatile memory sectors 7
2.3.2 Recovering data from NVM 7
2.3.3 Deleting data in NVM 8
2.4 PDM features 8
2.4.1 Mutex in PDM 8
2.4.2 PDM event and error handler 8
2.4.3 NVM capacity 9
2.4.4 NVM wear count 9
2.4.5 Ensuring consistency of PDM records 9
3 Power Manager 10
3.1 Low-Power modes 10
3.1.1 Doze Mode 10
3.1.2 Sleep mode with memory held 11
3.1.3 Sleep mode without memory held 11
3.1.4 Deep Sleep mode 11
3.2 Wake-up source from Low-Power modes 11
3.2.1 Timer wake-up 11
3.2.2 DIO wake-up 12
3.2.3 NTAG FD wake-up 12
3.2.4 Analog Comparator wake-up 12
3.3 Callback functions for Power Manager 12
3.3.1 Essential callback function 12
3.3.1.1 PWRM initialization 13
3.3.2 Pre-sleep and post-sleep callback functions 13
3.4 Initializing and starting the Power Manager 14
3.5 Enabling Power-Saving 14
3.6 Non-interruptible activities 14
3.7 Scheduling wake events 15
3.8 Terminating Low-Power mode 15
3.9 Doze mode 16
3.9.1 Circumstances that lead to Doze mode 16
3.9.2 Doze mode monitoring during development 17
4 Protocol Data Unit Manager 17
4.1 Message assembly and disassembly 17
4.2 Preparing the PDU Manager 18
4.3 Inserting data into outgoing message 19
4.4 Extracting data from incoming message 19
5 Debug (DBG) Module 20
5.1 Overview 20
5.2 Enabling the Debug module 21
5.3 Initializing and Configuring the Debug Module 21
5.3.1 Using UART Input/Output 21
5.3.2 Using Alternative Serial Output 21
5.4 Example Diagnostic Code 22
6 Persistent Data Manager API 23
6.1 Internal NVM PDM functions 23
6.1.1 PDM_eInitialise 23
6.1.2 PDM_eSaveRecordData 24
6.1.3 PDM_eReadDataFromRecord 25
6.1.4 PDM_eDeleteData 25
6.1.5 PDM_eDeleteAllData 25
6.1.6 PDM_u8GetSegmentCapacity 26
6.1.7 PDM_u8GetSegmentOccupancy 26
6.1.8 PDM_bDoesDataExist 27
6.2 Internal NVM PDM Miscellaneous Functions 27
6.2.1 PDM_vRegisterSystemCallback 27
6.2.2 PDM_vSetWearCountTriggerLevel 28
6.2.3 PDM_eGetSegmentWearCount 28
7 PWRM API 28
7.1 Core Functions 29
7.1.1 PWRM_vColdStart 29
7.1.1.1 29
7.1.2 PWRM_vInit 29
7.1.3 PWRM_eStartActivity 30
7.1.4 PWRM_eFinishActivity 31
7.1.5 PWRM_u16GetActivityCount 31
7.1.6 PWRM_eScheduleActivity 32
7.1.7 PWRM_vManagePower 32
7.1.8 PWRM_vWakeUpConfig 33
7.1.9 PWRM_GetFro32KCalibrationValue 33
7.2 Callback Set-up Functions 34
7.2.1 PWRM_vRegisterPreSleepCallback 34
7.2.2 PWRM_vRegisterWakeupCallback 35
7.2.3 vAppRegisterPWRMCallbacks 35
7.2.4 PWRM_vWakeInterruptCallback 36
8 PDUM API 36
8.1 PDUM_vInit 37
8.2 PDUM_hAPduAllocateAPduInstance 37
8.3 PDUM_eAPduFreeAPduInstance 37
8.4 PDUM_u16APduInstanceReadNBO 38
8.5 PDUM_u16APduInstanceWriteNBO 39
8.6 PDUM_u16APduInstanceWriteStrNBO 39
8.7 PDUM_u16SizeNBO 40
8.8 PDUM_u16APduGetSize 41
8.9 PDUM_pvAPduInstanceGetPayload 41
8.10 PDUM_u16APduInstanceGetPayloadSize 41
8.11 PDUM_eAPduInstanceSetPayloadSize 42
8.12 PDUM_vDBGPrintAPduInstance 42
9 DBG API 42
9.1 DBG_vPrintf 43
9.2 DBG_vAssert 44
10 JCU Structures 44
10.1 PDM Structures 45

10.1.1 PDM_tpfvSystemEventCallback 45
10.1.2 tsReg128 45
10.1.3 PDM_eSystemEventCode 45
10.1.4 PDM_teStatus 46
10.1.5 PDM_tsHwFncTable 47
10.2 PWRM Structures 47
10.2.1 PWRM_teSleepMode 48
10.3 DBG Structures 48
10.3.1 DBG_tsFunctionTbl 48
11 Revision History 48
12 Legal information 49

For more information, please visit: http://www.nxp.com

Date of release: 3 March 2023
Document identifier: JN-UG-3133