HT MICRON SEMICONDUTORES S.A.
Av. Unisinos, 1550 | 93022-750 | São Leopoldo – RS | Brasil
www.htmicron.com.br

HT MICRON semicondutores

iMCP HTNB32L-XXX – PMU and Sleep Modes

PMU and Sleep Modes for iMCP HTNB32L-XXX System-in-Package

Classification: PUBLIC

Doc. Type: USER MANUAL

Revision: v.02

Date: 13/09/2023

Code: HTNB32L-XXX-UM-0005

SUMMARY

DOCUMENT INFO

This document provides a user manual for the Power Management Unit and the Sleep Modes available in the iMCP HTNB32L-XXX System-in-Package. It is intended to present a comprehensive technical guide for developing low-power applications on the HTNB32L, using the related examples available in its SDK.

1. PMU AND SLEEP MANAGER

1.1. WORKFLOW

1.1.1. Idle State Flow

When the program is idle, the software attempts to enter the Idle state automatically. After waking up from the Idle state, the program (PC value) starts executing from the same place from where the Idle state occurred.

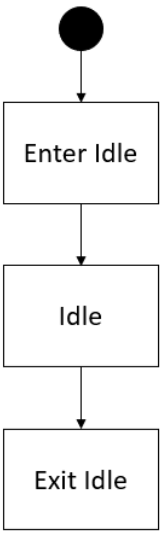

Figure 1: Idle state block diagram.

1.1.2. Sleep1 State Flow

In the Sleep1 state, as all peripherals are powered down, it is necessary to execute a Pre-Sleep1 callback function before entering Sleep1 and also execute a Post-Sleep1 callback function after waking up to restore the state before sleep. Users can use an API to register the callback function themselves. The callback function registered by the user should be as short as possible.

NOTE

Do not use operations that suspend the system, such as sending and receiving operating system messages, waiting for the task queue, calling operating system delay OsDelay, and so on. Also, do not use time-consuming operations, such as flash operation, waiting for external interrupts, waiting for serial data, and so on.

To ensure the PMU efficiency and execution, all pre-sleep callback functions and all Post-Sleep callback functions registered by the user should not exceed 3 ms. The storage and restoration of the driver-level registers are done by the SDK, which do not need users to take separate consideration.

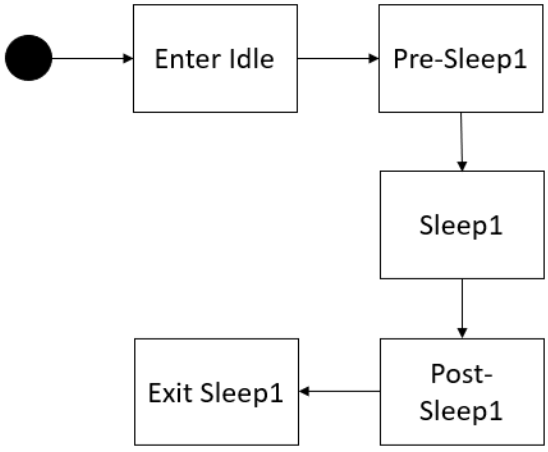

Figure 2: Sleep1 block diagram.

1.1.3. Sleep2 State Flow

In the Sleep2 state, the 256 KB SRAM is powered down and only the 16 KB SRAM is kept in retention. If the sleep is successful, the program runs from the beginning after the system wakes up. After completing an NB process, the user gets control in the "*main_entry*" function. At this time, the user can query the status through API functions to check whether the program is running for the first time after power-on or wake-up from the sleep process. If the sleep fails, the program is executed sequentially.

It is impossible to know whether sleep will succeed or fail before actually going to sleep. Therefore, users must be prepared for failure, registering the corresponding callback function to Post-Sleep2. Similarly, the callback function execution time of Pre-Sleep2 and Post-Sleep2 must be strictly controlled.

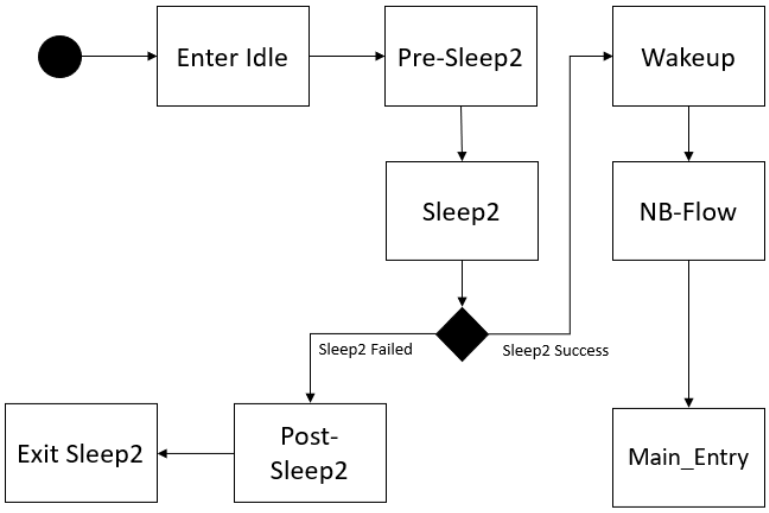

Figure 3: Sleep2 block diagram.

1.1.4. Hibernate State Flow

In the Hibernate state, the 16 KB SRAM is also powered down. The software process is basically the same as Sleep2. Similarly, users can also register the Post-Hib callback function to prepare for hibernate failure:

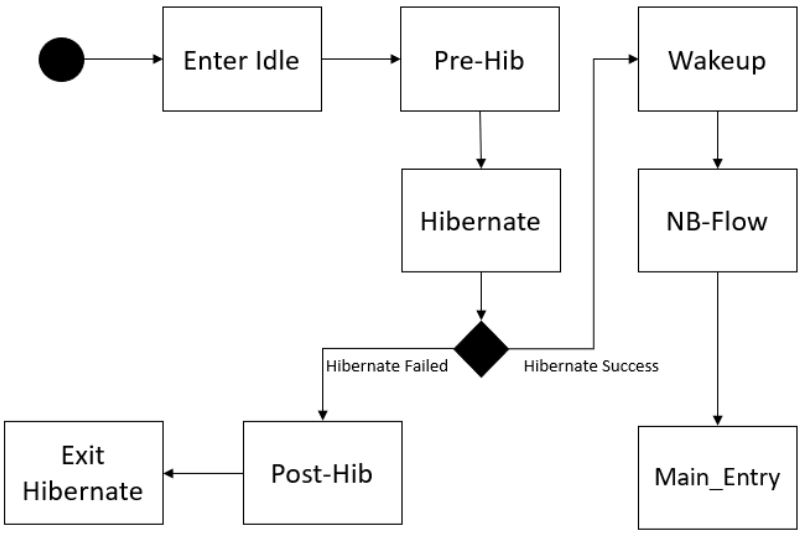

Figure 4: Hibernate block diagram.

1.2. Application Layer Sleep Depth Control

As a low-power NB-MCU, the HTNB32L-XXX device depends on many factors to determine the depth of sleep. A certain level of sleep depth can be entered by voting through an API function, and ultimately which sleep depth the HTNB32L enters depends on the underlying Narrowband (NB) program. Users vote through the API to determine the maximum sleep depth allowed by the application layer. The SDK considers and ensures that the next sleep depth is not higher than the user-specified depth. The mechanism of sleep voting is as follows:

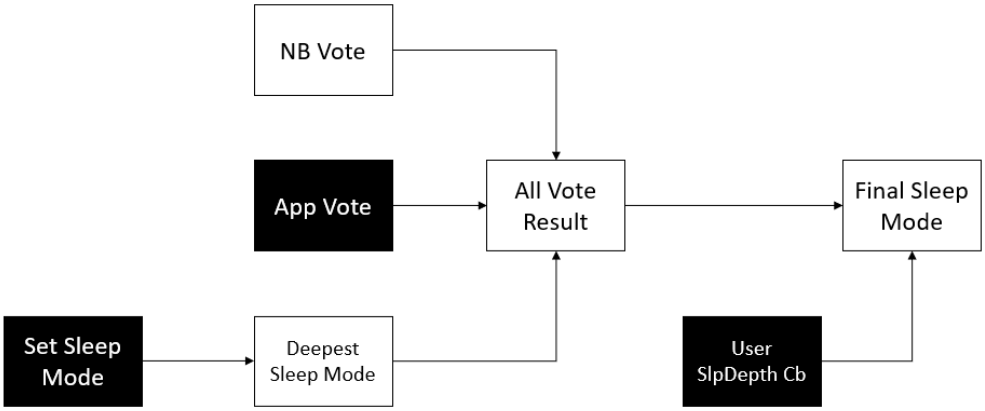

Figure 5: Voting decision diagram.

The black blocks of Figure 5 are the points open to the user to control the depth of sleep. These blocks can be summarized as follows:

- **Set Sleep Mode:** Sets the maximum sleep depth.
- **App Vote:** Controls the sleep depth by applying a voting handle and voting to the handle.
- **User SlpDepth Cb:** Sleep Depth Callback function of the registered user. This function is called before actually entering the sleep process.

The functions available on the SDK to operate these control points are described in Table 1.

Table 1: API functions for the three control points.

API Function	Description
slpManRet_t slpManApplyPlatVoteHandle(const char name, uint8_t *handle)*	Applies for a voting handle and records the handle name. The handle name is saved (maximum length is 9 bytes), and then the handle can be queried according to the handle name. Users should guarantee the uniqueness of the handle name. The function running status can be queried according to the return value of the function.
slpManRet_t slpManGivebackPlatVoteHandle(uint8_t handle)	Returns a voting handle back to the system. After the return, all information of the handle is cleared from the system. Users should clear the vote on the handle before the handle is destroyed. That is, if the *slpManPlatVoteDisableSleep* function is called once for a handle, *slpManPlatVoteEnableSleep* must be called before destruction to ensure the matching of votes.
slpManRet_t slpManFindPlatVoteHandle(const char name, uint8_t *handle)*	The corresponding handle can be retrieved based on the name. Ensure that the name is unique when applying for a handle
slpManRet_t slpManPlatVoteDisableSleep(uint8_t handle, slpManSlpState_t status)	Cancels a certain depth of sleep for a certain handle. For example: • *slpManPlatVoteDisableSleep (handle1, SLP_SLP2_STATE);* // Sleep2 is prohibited, that is, you can only enter IDLE, SLEEP1 • *slpManPlatVoteDisableSleep (handle2, SLP_HIB_STATE);* // Disable HIBERNATE, which means you can only enter IDLE, SLEEP1, SLEEP2
slpManRet_t slpManPlatVoteEnableSleep(uint8_t handle, slpManSlpState_t status)	Enables a certain depth of sleep for a certain handle. For example: • *slpManPlatVoteEnableSleep (handle1, SLP_SLP2_STATE);* // Cancel the previously prohibited operation of Sleep2 • *slpManPlatVoteEnableSleep (handle2, SLP_HIB_STATE);* // Cancel the previously prohibited operation of HIBERNATE

1.3. Other Sleep Manager APIs

1.3.1. Set Sleep Mode

API Function:

void slpManSetPmuSleepMode(bool pmu_enable, slpManSlpState_t mode, bool save2flash)

Description:
Sets PMU to Enable, sets the maximum sleep depth, and sets whether to write this configuration into the flash. After the system enters Hibernate or Sleep2 and wakes up, the variables used by the system are lost. If the application layer does not want to configure it again, this configuration can be written to flash when setting the sleep depth for the first time. The SDK is responsible for reading out the configuration during the system initialization and setting the settings to take effect. To reduce the number of flash writes as much as possible, in OPENCPU mode, it is not recommended to save the settings to flash. Instead, it is advisable to call the function again to set the PMU to Enable and control the sleep depth after the system wakes up.

1.3.2. User Sleep Depth Callback API

API Function:

void slpManRegisterUsrSlpDepthCb(pmuUserdefSleepCb_Func callback)

Description:
The callback function registered here is called before finally going to sleep. This is not a time-consuming operation, so suspension of the system or waiting for messages is expected. For most sleep control, voting decisions are used, but for situations where it is not convenient to participate in voting, the method of registering callbacks to achieve sleep control can be used. For example, in a system, when a pin of NB is high, the system wakes up. When this pin is low, the system enters sleep. At this time, a callback function can be registered, the pin level can be read in the function, and the sleep depth can be determined according to the pin level.

1.3.3. Get Last Sleep State

API Function:

slpManSlpState_t slpManGetLastSlpState(void)

Description:
After the iMCP HTNB32L-XXX system wakes up, the previous sleep state can be obtained through an API. As Sleep2 and Hibernate wake up, the system starts running from the beginning, so getting the previous sleep state becomes particularly important. In the user *main_entry* function for the Sleep2 and Hibernate cases, users can customize different processes to achieve different wake-up processing. The function returns the following states: *SLP_SLP1_STATE; SLP_SLP2_STATE; SLP_HIB_STATE; SLP_ACTIVE_STATE.*

1.3.4. Get Last Wakeup Source

API Function:

slpManWakeSrc_e slpManGetWakeupSrc(void)

Description:
The wake-up source of the last wake-up can be queried through this API. The wake-up source can be POR (indicating that it is reset or powered on, but has not yet entered sleep), RTC, PAD, or USART.

1.3.5. Miscellaneous

The iMCP HTNB32L-XX PMU SDK also provides the following status query functions:

Table 2: Status query functions.

Status query function	Description
bool slpManDrvSafeToSleep(void)	Checks if there is a peripheral device in working state and cannot go to sleep. When peripherals are working, the HTNB32L device can only enter the Idle state
*void slpManGetDrvBitmap(uint32_t *bitmap)*	Obtains the voting status of the peripheral driver. Use this interface to see which peripheral is currently working because of which the system cannot sleep.
*void slpManGetPlatBitmap(uint32_t *slp_bitmap, uint32_t *slp2_bitmap, uint32_t *hib_bitmap)*	Gets the voting results of Sleep1, Sleep2, and Hibernate from the platform layer. The returned result is a bitmap. Bit 0 indicates that the current level of sleep can be entered, and bit 1 indicates that the current level of sleep cannot be entered. The bit position corresponds to the handle applied by *slpManApplyPlatVoteHandle*. For example, if *slp_bitmap = 0x02* is queried, it indicates that handle 1 has not voted for Sleep1 at this time
slpManRet_t slpManFindPlatVoteHandle(const char name, uint8_t *handle)*	Queries the corresponding handle name according to the voting handle.
slpManSlpState_t slpManPlatGetSlpState(void)	Obtains platform layer voting results.
slpManRet_t slpManDrvVoteSleep(slpDrvVoteModule_t module, slpManSlpState_t status)	Peripheral driver voting API, usually the driver in this function SDK is responsible for calling. Users do not require any attention.

1.4. Voting Procedure

1. After applying for a handle, the same depth of sleep can be voted on the same handle. For example, only *slpManPlatVoteDisableSleep(handle1,SLP_SLP2_STATE)* or *slpManPlatVoteEnableSleep(handle1, SLP_SLP2_STATE)*. After executing *slpManPlatVoteDisableSleep(handle1,SLP_SLP2_STATE)*, *slpManPlatVoteEnableSleep(handle1, SLP_HIB_STATE)* cannot be executed.

2. The vote for the handle can be called in multiple threads, and the vote count is introduced. After n times of the *slpManPlatVoteDisableSleep* function, call *N* times of the *slpManPlatVoteEnableSleep* function to clear the vote.

It is recommended to check the return value to ensure that each operation is successful, that is, the return value is RET_TRUE. The enumerated type corresponding to the return type *slpManRet_t* can be found in the header files of the HTNB32L-XXX SDK. If the return value is not checked and the vote is successful, the actual vote fails, making it difficult to track the problem.

1.5. Understanding the Voting Function Names

Disabling SLP_SLP2_STATE is considered as adding a lock before the Sleep2 state. When the PMU encounters the Sleep2 lock while trying to sleep, it cannot enter a deeper sleep state.

Enabling SLP_SLP2_STATE is considered as canceling a lock in the Sleep2 state; whether Hibernate can be entered later or not depends on whether a lock is added in Hibernate. If there is no lock, Hibernate can be entered successfully.
Disabling SLP_SLP2_STATE twice is considered as adding two locks to Sleep2. Therefore, only enable SLP_SLP2_STATE twice, that is, removing two locks can clean up the locks on Sleep2.

If SLP2 is not locked, go to Enable SLP_SLP2_STATE. The result is that there is no lock that can be removed and RET_VOTE_SLP_UNDERFLOW is returned. The error given in this case is only a warning to the user, and there is no problem with enabling SLP2 sleep.

1.6. USER NVM

To perform the recovery of the software process after waking up from deep sleep next time, users often need to save some data on the flash before sleeping. Therefore, a piece of SRAM is reserved for the user, and the data for SRAM area is written into the flash by the SDK before entering deep sleep and is restored from the flash to the SRAM after wake-up. To reduce the frequency of flash erasure, this area should be updated as little as possible. The principle of implementation is as follows:

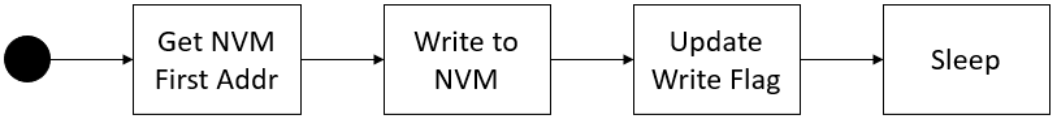

Figure 6: NVM writing procedure.

The following procedure describes how data is saved on the flash (NVM) before the device goes to sleep:

1. User NVM is located in a certain space in the SRAM. The user obtains the first address of this space through the *slpManGetUsrNVMem* function.
2. After obtaining the first address, users can directly operate this address; the available space is 2016 byte. Pointers, arrays, structures, and so on can be specified directly to this space. To prevent cross-border access, an MPU read-only protection is made for the last 32 bytes in this area, so the actual accessible space is 2016 bytes.
3. After the user writes data to this space, call *slpManUpdateUserNVMem* to update the Write flag.
4. When the SDK finally judges that it can enter Sleep2 or Hibernate, query whether it is necessary to write this user data area to the flash.
5. After the system wakes up from deep sleep, the SDK restores the data in this area from the flash during the initialization phase.

Table 3: User NVM API.

API	Description
void slpManUpdateUserNVMem(void)	Tells the SDK that the content in the user data area has just been updated.
*uint8_t * slpManGetUsrNVMem(void)*	Obtains the first address pointer of the user data area. The user can use the (2048-32) byte length space. The variables that the user must keep in deep sleep can be written directly into this area. After writing, call the *slpManUpdateUserNVMem* function to inform the SDK that there is an update in this area, so that the user can write before deep sleep.

void slpManFlushUsrNVMem(void)	Immediately writes the content in this user data area to the flash instead of waiting for it to update to the flash before sleeping. It is recommended that users use this function as little as possible. The automatic writing by the SDK before deep sleep greatly reduces the frequency of flash writing.

2. ALWAYS ON PIN

The iMCP HTNB32L-XXX features an Always On (AON) IO pin (AON_GPIO1), designed to retain its IO level both before entering sleep mode and after waking up from sleep. Users can configure this AON pin using the GPIO driver while the device is in an active state. However, it is important to note that the AON IO pin must be powered on for the GPIO configuration in order to work properly.

Table 4: Always ON pin API.

API	Description
void slpManAONIOLatchEn (IOLatchEn en)	Enables the AON IO. When enabled, the Retention IO maintains its level state during sleep; otherwise, it is powered off. Users can set the Retention IO level using the GPIO API driver. The latch operation only occurs immediately before entering sleep. If the AON IO level changes after calling this function, the pin level will be latched before entering sleep
iOLatchEn slpManAONIOGetLatchCfg (void)	Gets the Enabled state of the AON IO, that is, reads the configuration result of *slpManAONIOLatchEn*.
void slpManAONIOPowerOn(void)	By default, the AON IO is not powered. Call this function to power on the AON IO, and then use the GPIO driver to operate the AON IO after power-on.
void slpManAONIOPowerOff (void)	Powers off all AON IOs. After power-off, all AON IOs are powered off. Then, AON IO cannot latch in sleep.
void slpManAONIORelease(void)	If the latch AON IO is configured before sleep, the GPIO should be configured as soon as possible after the system wakes up and the latch of AON IO should be removed. Normally, the user should complete the latch release within *BSP_CustomInit*.

3. APPLICATION LAYER PMU SOFTWARE

3.1. STARTUP FLOW

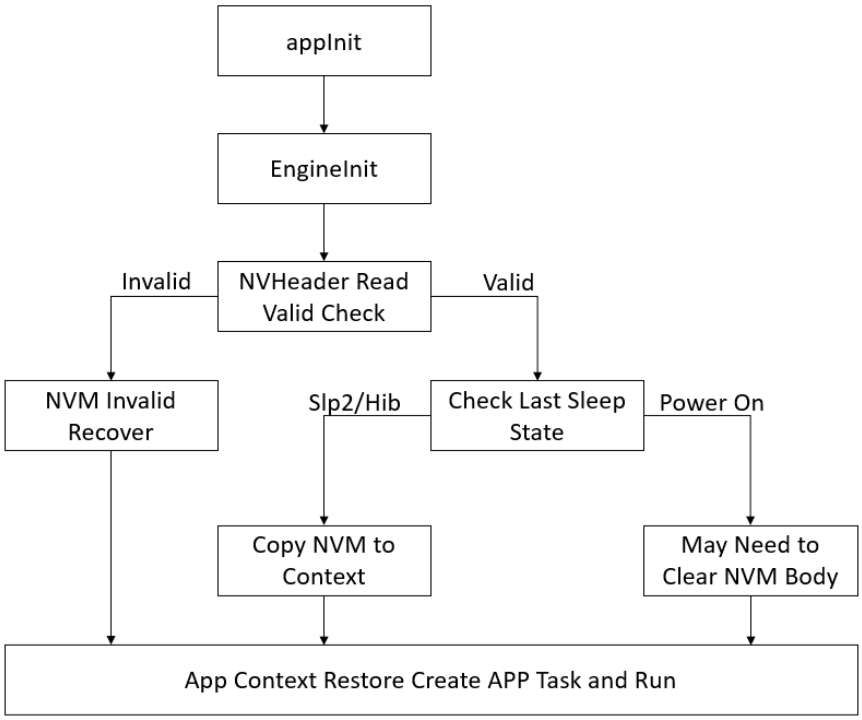

Figure 7: Flowchart for context restore when application starts up.

The context restoration of low-power applications is recommended to be executed in the *appInit* function. Each application corresponds to a low-power entry function *xxxEngineInit()*, and each application corresponds to a context global variable *gXXXContext* and an NVM data file *xxx_config.nvm*.

The format of the NVM data file can be in the following format (for reference only); NVHeader contains Body Length, the Body Empty flag, and other mark information:

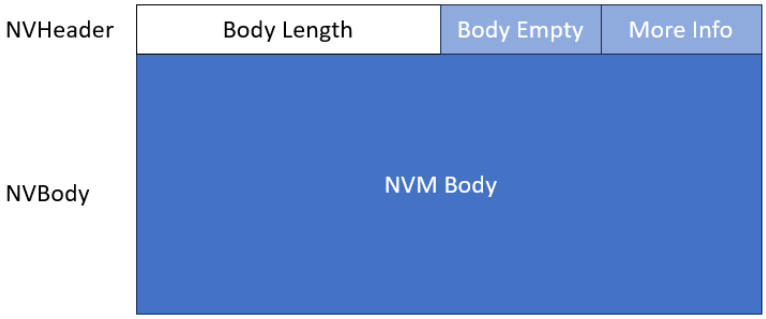

Figure 8: NVM data file format.

- **Body Length:** Marks the actual length of the NVM data area, so that NVM Body can be read according to the length.
- **Body Empty**: Marks whether data has been written in the NVM Body area; if there is no data written, the recovery or erase operation must not be performed.

- More Info: If required, users can add additional information, such as MagicWord, CheckSum, Version, and so on to ensure the integrity of the file.

Users must ensure the integrity of the NVM data file. If the file is detected to be incomplete, the default value must be restored. There may be the following situations:

- To update a software version for the first time, the NVM data is erased, or the flash is empty. The file does not exist, and it must be initialized and written once.

- When the file length is changed or the storage area offset changes after the program upgrade, a recovery mechanism should be considered. Consider using checksum and other methods to check for file abnormalities and erase the file to re-initialize.

The purpose of using NVHeader is to reduce the possibility of reading a large amount of flash, and only read it when it is judged that there is data in the data area. In addition, it can help achieve the integrity check of NVM Body.

After completing the integrity check, check which sleep depth the wake-up occurred. If it is Power-on, the user can decide whether to erase the file, if required.

If wake-up is from the Sleep2/Hibernate state, the Body Empty flag is determined first. If the subsequent NVM is valid, then read and restore to RAM.

After the context is restored, create an APP task, and then execute the process in the APP task. The *appInit* function is called in a very high priority task and is used to perform global initialization before the entire system. Therefore, the context initialization time of the application must be as short as possible. Time-consuming operations, such as waiting for network connections and waiting for messages should be placed in their own application tasks. In AT Command execution, the context is restored first, and then a decision is taken on whether to restart the application task based on the context information. If the application task always runs, the restoration of the context can also be directly put into the application task to execute. The general principle is not to block too long in *appInit*.

3.2. Application Context Recovery-Related API

The application context information is recommended to be stored in the file system. It is easy to use and has an erasure balancing algorithm. The file system is written using the following function:

- *OSAFILE OsaFopen(const char *fileName, const char* mode);*
- *INT32 OsaFclose(OSAFILE fp); UINT32 OsaFread(void *buf, UINT32 size, UINT32 count, OSAFILE fp);*
- *UINT32 OsaFwrite(void *buf, UINT32 size, UINT32 count, OSAFILE fp);*
- *INT32 OsaFseek(OSAFILE fp, INT32 offset, UINT8 seekType);*
- *INT32 OsaFsize(OSAFILE fp);*
- *UINT32 OsaFremove(const char *fileName);*

3.3. APP PMU RECOVERY PROCESS

If the user application is related to the network, wake up and wait for the network to get ready before initiating data services. The reference PMU recovery process is shown as an example in the following figure:

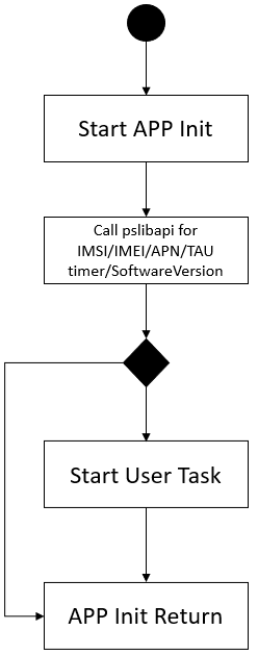

Figure 9: PMU recovery process workflow.

4. SLEEP MODE TEST FUNCTIONS (SLEEP_EXAMPLE)

In addition to the Sleep Manager, the HTNB32L SDK also provides a Sleep_Example, that utilizes test functions that can be used to validate the device's power consumption. Even though these functions are easier to understand than those in Sleep Manager, it is strongly recommended to use the Sleep Manager protocol when developing the final application.

4.1. API

Table 5 presents the sleep mode test functions available in the Sleep_Example. All these functions set the device to a specific sleep mode, without considering the OS or the NB state.

Table 5: Sleep Mode test functions API.

API Function	Description
void pmuSlpTestExtWakeupSleep1(void)	Sets the HTNB32L device to the Sleep1 state and waits for an external wakeup event.
void pmuSlpTestExtWakeupSleep2(void)	Sets the HTNB32L device to the Sleep2 state and waits for an external wakeup event
void pmuSlpTestExtWakeupHibernate1(void)	Sets the HTNB32L device to the Hibernate1 state and waits for an external wakeup event
void pmuSlpTestExtWakeupHibernate2(void)	Sets the HTNB32L device to the Hibernate2 state and waits for an external wakeup event

5. OPENCPU MODE

5.1. FSM EXAMPLE AND TIPS

Initiating the data connection between the chip and the network brings greater power consumption. For the characteristics of NB-IoT applications that are not sensitive to delay, in most cases, the data can be collected first and buffered on the device side. Send all the buffered data to the cloud at one time, when needed. In this working method, the data collection work is handed over to the iMCP HTNB32L-XXX to avoid the increase of power consumption caused by connecting to the network when collecting data.

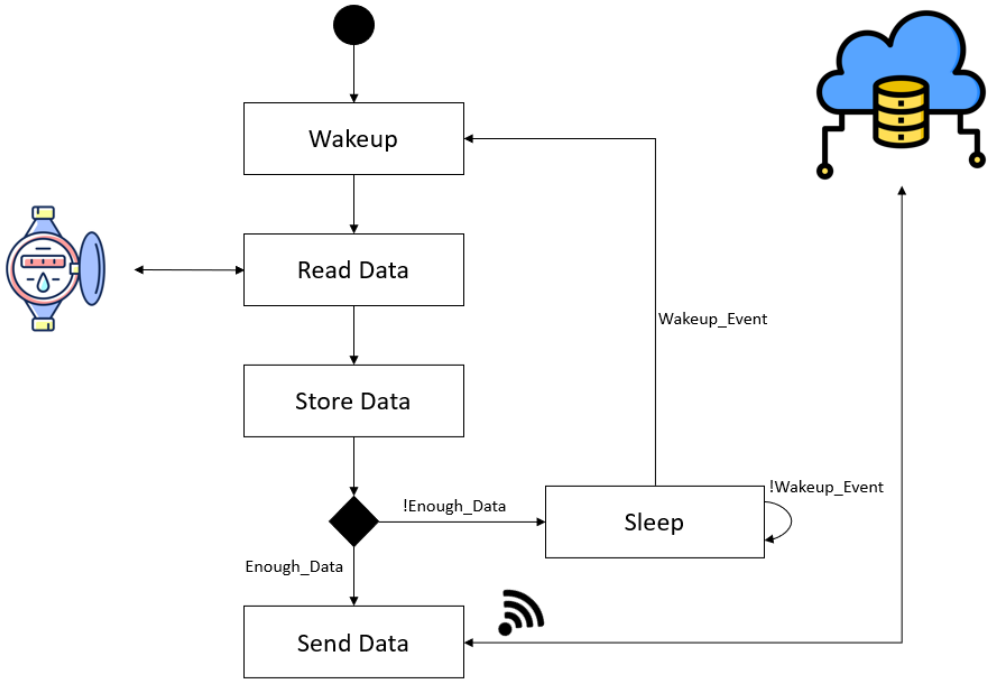

Figure 10: OpenCPU finite state machine example.

6. WAKING UP FROM DUAL-CHIP SOLUTION

When the HTNB32L operates as a module controlled by AT Commands, it typically operates in the Low-Power mode. In this mode, the external MCU is required to follow a wakeup protocol in order to establish a communication thought UART with the NB-IoT device.

In a scenario where an external MCU configures the HTNB32L device to a specific sleep mode, this configuration will be stored in the NVM. As a result, the device will enter the same sleep mode almost immediately after a wakeup event. The iMCP HTNB32L-XXX offers two wakeup schemes to properly manage this behavior, each with different characteristics.

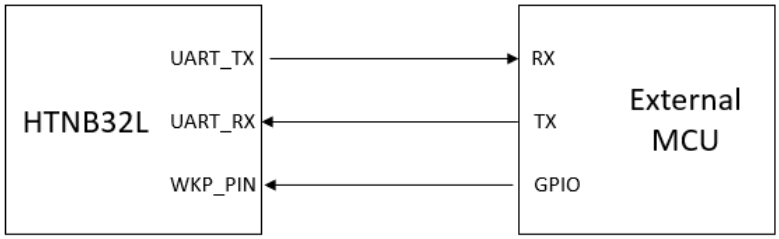

Figure 11: Connection between HTNB32L and external MCU through UART interface and WAKE_UP pin.

6.1. SCHEME 1

Since the HTNB32L generates a wakeup event when detecting the rising edge of the WKP_PIN, it is a good practice to keep this WKP_PIN in low logic-level during the transmission of all necessary commands, preventing the device from entering sleep mode. Figure 12 demonstrates how this procedure can be done:

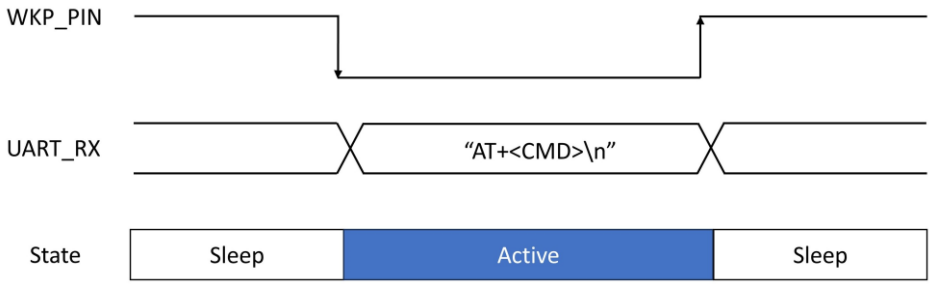

Figure 12: Scheme 1 communication protocol.

6.2. SCHEME 2

Another possible solution to prevent premature sleeping is to add a delay that specifies the amount of time the system must remain awake to receive all commands. After setting the delay time, this configuration is written to the flash memory and takes effect after the system is powered on again. The following command must be used in order to set the desired delay time:

AT$QCPCFG="slpWaitTime",<TIME_IN_MS>

Example with 4000 ms:

AT$QCPCFG="slpWaitTime",4000

More details can be found in the *HTNB32L-XXX-UM0002-AT_Commands* user manual.

7. FAQ

1. What are the differences between the Sleep_Example and Slpman_Example?

R: The main difference lies in the OS. While the Sleep_Example utilizes low-level functions to set the device to a specific low-power mode, the Slpman considers the state of each task running in the FreeRTOS environment. In other words, the Slpman_Example respects the deadline of every NB task, either waiting for its end or taking it into account in to determine the most appropriate sleep mode level.

2. What type of functions should I use in my application? The Sleep Test Functions or Slpman?

R: Users can employ both types of functions in their application, but it is recommended to use the Slpman routines.

ABBREVIATIONS

Table 6: Abbreviations

Acronym	Description
GPIO	General Purpose Input Output
UART	Universal Asynchronous Receiver-Transmitter
ROM	Read-only Memory
RST	Reset
RAM	Random-Access Memory
NVM	Non-volatile Memory
PMU	Power Management Unit
OS	Operating System
Slpman	Sleep Manager
RTOS	Real Time Operating System
WKP	Wakeup
MCU	Microcontroller
NB	Narrowband
FSM	Finite State Machine
APP	Application
API	Application Programming Interface

LIST OF FIGURES

LIST OF TABLES

REVISION HISTORY

Version	Date	Changes	Authors
00	13/09/2023	- Initial draft	HBG

CONTACT

HT MICRON SEMICONDUTORES S.A.
Av. Unisinos, 1550 | 93022-750 | São Leopoldo | RS | Brasil
www.htmicron.com.br

DOCUMENT INFORMATION

Document Title:	iMCP HTNB32L-XXX – PMU and Sleep Modes
Document Subtitle:	PMU and Sleep Modes for iMCP HTNB32L-XXX System-in-Package
Classification:	PUBLIC
Doc. Type:	USER MANUAL
Revision:	v.02
Date:	13/09/2023
Code:	HTNB32L-XXX-UM-0005

DISCLAIMER