Programmer's Guide - Power Saving

Ingchips Technology Co., Ltd.

Web: http://www.ingchips.com

E-mail: service@ingchips.com

Tel: 010-85160285

Address: Room 803, Building #3, Zijin Digital Park, Haidian District, *Beijing*

Room 316, Tower A, Juxin Building, Xiangke Road #58, *Shanghai*

Room 1009, Shuguang Building, Science Park, Nanshan District, *Shenzhen*

Contents

List of Figures

List of Tables

Revision History

Version	Note	Data
0.9	Initial	2023-02-23

Chapter 1

Introduction

Welcome to use *INGCHIPS* 918/916 Software Development Kit.

Power saving is important in Bluetooth LE products, especially for those using batteries. This document is a guide for developing low power products from the perspective of software.

Roughly speaking, the goal of power saving is, when there is something to do, to design the system carefully to reduce power consumption; when there is nothing to do, to put the system into special modes to reduce power consumption. The latter one is covered by the power saving Framework, while the former one is briefly discussed in Tips on Low Power System.

1.1 Abbreviations & Terminology

Table 1.1: Abbreviations

Abbreviation	Notes
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
IRQ	Interrupt Request
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
ROM	Read Only Memory
RTOS	Real-time Operating System kernel
SDK	Software Development Kit

Table 1.2: Terminology

Terminology	Notes
Cache	A small but fast memory that stores copies of data from frequently used main memory locations
Flash	An electronic non-volatile computer storage medium
FreeRTOS	A real-time operating system kernel
ISR	Interrupt Service Routine
KiB	1024 Bytes
System tick	A system timer that can be used to generate periodic interrupts or measure time

1.2 References

1. FreeRTOS[1]
2. Mastering the FreeRTOS™ Real Time Kernel[2]

[1] https://freertos.org
[2] https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

Chapter 2

Framework

The implementation of power saving in SDK is designed to be easy to use. Generally, power saving relies on the idle process in RTOS, which is usually a process[1] having the lowest priority. It's a common practice for OS(es) to have an idle process, for example, the idle process with PID 0 in Windows and Linux.

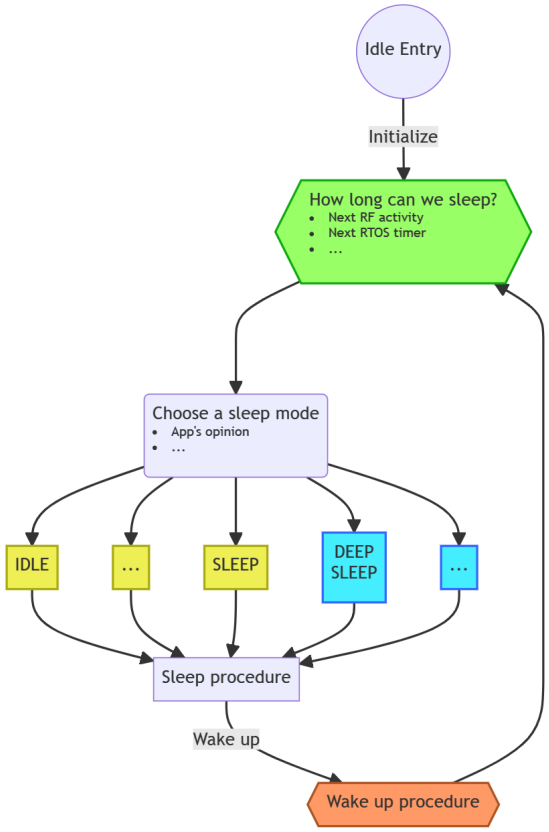

Figure 2.1: Idle Process

Figure 2.1 is an overview of the idle process. Once initialized, the idle process

[1]In embedding systems, it may also be called a *task* or *thread*.

keeps trying to put the system into sleep modes to save power. It first checks how long the system could sleep, i.e. sleep time, which is generally determined by:

- When the next BLE sub-system activity occurs;
- When the next RTOS timer expires.

Note that, peripherals, such as hardware timers, UART, are not taken into account here.

Then, the idle process makes choice on sleep modes. Following factors are considered:

- Sleep time
- App's opinion;
- Current system state.

Different sleep modes have their own hard time requirements, because it needs extra time to bring the system into sleep modes and wake it up again. App's opinion tells the idle process a subset of sleep modes which are allowed to be chosen from. Current system state is also considered.

After the most suitable sleep mode which consumes lowest power, is decided, the sleep procedure stops RTOS and takes the system into the chosen sleep mode.

After waken up, the wake up procedure performs post processing to make sure everything is normal, for example, the system tick is taken care of here. This framework knows nothing about peripherals and their configurations, and it just notifies App that the system has waken up. App could take this opportunity to configure peripherals. After everything is ready, the wake up procedure allows RTOS to continue. And the whole loop is started all over again.

2.1 Sleep Modes

There are a variety of sleep modes available (supported by SoC and power saving framework) summarized in Table 2.1.

Table 2.1: Sleep modes

Mode	ING918	ING916	Notes
IDLE	✓	✓	Everything is active; CPU is waiting for interrupts.
SNOOZE	✓		Same as IDLE except RF is shutdown.
SLEEP	✓		Same as IDLE except BLE sub-system is shutdown.

Mode	ING918	ING916	Notes
DEEP SLEEP	✓	✓	CPU and peripherals are all shutdown. Memory is retained.
DEEPER SLEEP		✓	Like DEEP SLEEP, but memory for BLE is shutdown.
BLE ONLY SLEEP		✓	Like DEEP SLEEP, but BLE sub-system fully functional

IDLE mode is the slightest sleep mode, backed by the underlying CPU while other modes are defined and backed by dedicated logic in the SoC out of CPU itself. There are subtle differences between different sleep modes besides the notes in Table 2.1:

- For DEEP SLEEP on ING916, some peripherals, such as GPIO, can be optionally put into special low power modes, rather than shutdown, in which case, its output could be retained or acting as wake up sources;
- For DEEPER SLEEP, comparing with DEEP SLEEP:
  - Less peripherals are allowed to be put into special low power modes;
  - Internal $1\mu s$ timer becomes less accurate.
- For BLE ONLY SLEEP, the system can be waken up only by BLE sub-system.

Different sleep modes consume different amount of current, since different components in SoC consume different amount of current. Here is a very rough quantitative summary, providing a qualitative analysis:

- CPU + Flash

  Current consumption heavily depends on working frequency. It may vary from about $1mA$ to more than $10mA$.

  For ING918, CPU clock is fixed at $48MHz$, consuming $\sim 5mA$.

- RAM retention

  More memory retained, more current is needed. It costs roughly $\sim 0.02\mu A$ per KiB.

- BLE Activity

  It costs from several mA to more than $10mA$.

2.2 Wake up Sources

There are two types of sources that can cause the system to wake up:

- Internal sources

  There is only one internal source: sleep timer. It is a dedicated hardware timer used only by the idle process, configured to sleep time, and cause the system to wake up when expired.

  This timer has 32 bits, and is driven by the 32K clock.

- External sources

  External sources are given in Table 2.2 and Table 2.3[2].

Table 2.2: External wake up sources of ING918

Mode	Sources
IDLE	All interrupts.
SNOOZE	All interrupts.
SLEEP	All interrupts.
DEEP SLEEP	EXT_INT.

Table 2.3: External wake up sources of ING916

Mode	Sources
IDLE	All interrupts.
DEEP SLEEP	GPIO/RTC/Comparator. Configurable.
DEEPER SLEEP	GPIO. Configurable.
BLE ONLY SLEEP	BLE sub-system.

– No matter which mode SoC is in, RESET pin always works, but we don't name it as an external wake up source;

– Watchdog is an ordinary peripheral, and powered off in DEEP SLEEP, DEEPER SLEEP, and BLE ONLY SLEEP modes.

[2]Refer to *Programmer's Guide - ING916XX Peripheral* for the configuration of wake up sources

2.3 Involvement of Apps

Sleep modes can be classified into two categories, Category A that do not need the involvements of Apps (IDLE, SNOOZE, and SLEEP), and Category B that do need the involvements of Apps (DEEP SLEEP, DEEPER SLEEP, and BLE ONLY SLEEP).

This framework tried its best to minimize the involvements of Apps, and there are just two things to be handled by Apps:

1. Tell the idle process which modes in Category B are allowed to be used;

   For ING916, Apps can take this opportunity to configure external wake up sources as well.

2. (Re-)Configure peripherals after waken up from Category B sleep modes.

These two things are done in two platform event handlers.

How to determine if a mode in Category B is allowed? Here are some suggestions.

- DEEP SLEEP
  - If peripherals can't be powered off now, for example, UART is running or FIFO not empty, then NO;
  - Hardware timer is used, which generates interrupts very frequently (such as $1000Hz$), then NO.
- DEEPER SLEEP
  - If DEEP SLEEP is not allowed, then NO;
  - If external wake up source does not fit the need, for example, comparator is required as a wake up source, then NO;
  - If memory allocated from Link Layer's heap is in use[3], then NO;
  - If internal $1\mu s$ timer should be as accurate as in DEEP SLEEP, then NO.
- BLE ONLY SLEEP
  - If DEEP SLEEP is not allowed, then NO;
  - If other external wake up source are needed, then NO.

[3]This heap is powered off in DEEPER SLEEP mode.

2.4 Shutdown

The idle process implements an automatic and passive power saving mechanism. A proactive mechanism also exists, *Shutdown*, where developers can optionally specify a time after which the system is powered on again, and a portion of memory to be retained during shutdown. The time can be set to 0, which means to disable the power on timer, and stay in shutdown mode unless external power on signal is asserted or RESET.

- For ING918

  Shutdown is based on DEEP SLEEP, except that most of if not all memory is powered off too. As in DEEP SLEEP, the system can be powered on again by EXT_INT.

- For ING916

  Shutdown is based on DEEPER SLEEP. The system can be powered on again by external wake up source for DEEPER SLEEP too.

Chapter 3

API

There is a global switch for power saving, *Enable* or *Disable*, which can be configured by `platform_config` at any time:

```
// To enable power saving
platform_config(PLATFORM_CFG_POWER_SAVING, PLATFORM_CFG_ENABLE);

// To disable power saving
platform_config(PLATFORM_CFG_POWER_SAVING, PLATFORM_CFG_DISABLE);
```

3.1 Callback events

As discussed in Involvement of Apps, there are two events that should be handled by Apps:

- **PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED**

  Handler of this event returns bit combination of allowed sleep modes in Category B. A value of 0 means that no mode in Category B is allowed at present.

  ```
  #define PLATFORM_ALLOW_DEEP_SLEEP      ...
  #define PLATFORM_ALLOW_DEEPER_SLEEP    ...
  #define PLATFORM_ALLOW_BLE_ONLY_SLEEP  ...
  ```

  For example, if both DEEP and DEEPER SLEEP are allowed, then:

```
return PLATFORM_ALLOW_DEEP_SLEEP + PLATFORM_ALLOW_DEEPER_SLEEP;
```

For ING916, Apps can also configure external wake up sources in the handle.

- **PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP**

  In the handler of this event, Apps can re-initialize peripherals and do other jobs according to the needs.

  Input parameter `void *data` is casted from `platform_wakeup_call_reason_t`:

```
typedef enum
{
    PLATFORM_WAKEUP_REASON_NORMAL = ...,
    PLATFORM_WAKEUP_REASON_ABORTED = ...,
} platform_wakeup_call_reason_t;
```

  PLATFORM_WAKEUP_REASON_NORMAL means that this event is emitted for a normal (successful) wake up procedure, i.e. the sleep procedure is completed successfully; *PLATFORM_WAKEUP_REASON_ABORTED* means that the previous sleep procedure is aborted, i.e. sleep had not happened at all. For example, trying to go to DEEP SLEEP while EXT_INT is asserted will be a failure, and sleep will not happen.

  Event with *PLATFORM_WAKEUP_REASON_NORMAL* is always emitted; while event with *PLATFORM_WAKEUP_REASON_ABORTED* is emitted when `PLATFORM_CFG_ALWAYS_CALL_WAKEUP` is *Enabled*. For ING918, `PLATFORM_CFG_ALWAYS_CALL_WAKEUP` is default to *Disabled*; while for ING916, default to *Enabled*. This is because for ING916, external wake up sources such GPIO might need to be reconfigured if sleep procedure is aborted. Without this event, Apps will not be able to do such reconfiguration. Since external sources are not configurable in the case of ING918, so it is default to *Disabled*.

  If `PLATFORM_CFG_ALWAYS_CALL_WAKEUP` is *Enabled*, then there is *definitely* a *PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP* event for each *PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED* event that returns a non-0 value.

Prototype of event handles should be compatible of:

```
typedef uint32_t (*f_platform_evt_cb)(void *data, void *user_data);
```

Handlers are registered by:

```
void platform_set_evt_callback(
  // the event
  platform_evt_callback_type_t type,
  // the callback function
  f_platform_evt_cb f,
  // user data that will be passed into callback function `f`
  void *user_data
);
```

3.2 Interrupts

Since CPU is powered off in Category B sleep modes, after waken up, from CPU's point of view, interrupts should be re-enabled.

For example, an UART port is used the App with both Rx and Tx interrupts are enabled. Then, in the waken up event handle, not only the peripheral itself should be re-initialized, its Rx and Tx interrupts enabled, but also the corresponding position in interrupt vector table. In the case ARM Cortex-M processors, it is called NVIC[1].

If the ISR is registered by `platform_set_irq_callback`, then there are two options to re-enable the interrupt:

1. Call `platform_set_irq_callback` again, since this API will enable the interrupt;
2. Use `platform_enable_irq`:

```
void platform_enable_irq(
  // corresponding interrupt request type
  platform_irq_callback_type_t type,
  // flag = 1 for enable; flag = 0 for disable
  uint8_t flag);
```

If the ISR is in the table registered by `platform_set_irq_callback_table`, then use `platform_enable_irq`, too.

3.3 Shutdown

Shutdown is initiated by calling `platform_shutdown`:

[1] Nested Vectored Interrupt Controller

```
void platform_shutdown(
  // Duration before power on again (measured in cycles of 32k clock)
  const uint32_t duration_cycles,
  // Pointer to the start of data to be retained
  const void *p_retention_data,
  // Size of the data to be retained
  const uint32_t data_size);
```

When `duration_cycles` is zero, power on when external wake up source is asserted. When `duration_cycles` is not zero, it must be larger than a minimum value of 825 cycles (about $25.18ms$) reserved for hardware. `data_size` can be zero too if no data is needed to be retained. Only part of SYS memory starting from 0x2000000 can be retained.

If this function fails, it will return. If this function successes, CPU is powered off, so it will not return.

3.4 32k Clock

32k clock is crucial for power saving. There are two sources for this clock, one is internal RC 32k clock, and the other is 32k crystal oscillator which needs an external crystal. So, platform provides APIs to select clock source, and change settings:

- **PLATFORM_CFG_RC32K_EN**

  Enable/Disable the internal RC 32k clock. Default: Enabled.

- **PLATFORM_CFG_OSC32K_EN**

  Enable/Disable 32k crystal oscillator. Default: Enable.

- **PLATFORM_CFG_32K_CLK**

  32k clock selection. Flag is `platform_32k_clk_src_t` with default value `PLATFORM_32K_RC`:

```
typedef enum
{
  // external 32k crystal oscillator
  PLATFORM_32K_OSC,
  // internal RC 32k clock
  PLATFORM_32K_RC
} platform_32k_clk_src_t;
```

When modifying this configuration, both RC32K and OSC32K should be **enabled**. For ING918: both clocks must be running; To ensure a clock is running:

  - **OSC32K**: wait until status of OSC32K is OK;
  - **RC32K**: wait $100\mu s$ after enabled.

  It's recommended to wait another $100\mu s$ before disabling the unused one.

- **PLATFORM_CFG_32K_CLK_ACC**

  Configure 32k clock accuracy in ppm.

- **PLATFORM_CFG_32K_CALI_PERIOD**

  32K clock auto-calibration[2] period in seconds. Default: $3600 \times 2 = 2(hours)$.

3.5 Timers

Platform also provides a few timer APIs coupled with power saving.

There is an internal counter increased by 1 per $1\mu s$, which wraps after $7953.64days(\approx 21.79years)$. Therefore, this timer can be treated as never wrapping practically. This counter is carefully recovered during waking up[3]. It is reset to zero when power up again from shutdown. Use `platform_get_us_time` to read this counter:

```
uint64_t platform_get_us_time(void);
```

System tick is disabled and restarted when entering and leaving sleep modes. It's difficult to properly reconfigure it accurately. Platform provides a configure item `PLATFORM_CFG_RTOS_ENH_TICK`. When enabled, accuracy of system ticks is improved but less power efficiency.

Platform also provides another timer that survives in all sleep modes and is more accurate than system ticks. Such timers can be created by calling `platform_set_timer`:

```
void platform_set_timer(
  // callback function when timer expires
  void (* callback)(void),
  // timer expires after this delay from now
  // Unit: 625us
  uint32_t delay);
```

This timer has some unique features:

[2] *Calibration* means to measure the frequency of this clock with another high frequency and high precision clock.

[3] For DEEPER SLEEP mode of ING916, it is coarsely recovered comparing to other modes.

- Comparing to hardware timers, this timer can be thought as "running" during power saving mode;
- Comparing to RTOS software timers, this timer is software + hardware too;
- Comparing to RTOS software timers, this timer may be more accurate in some circumstance;
- This function always succeed, and only fails when running out of memory.

`callback` is also the identifier of the timer. So below two lines defines only one timer expiring after 200 units but not two separate timers:

```
platform_set_timer(f, 100);
platform_set_timer(f, 200);
```

The callback function is called in a RTOS task (if existing), but not an ISR.

Range of `delay` is $0 \sim 0x7fffffff$. When `delay` is zero, the timer associated with the callback function is cleared. Since `delay` is in $625\mu s$, the maximum delay is $372.827hours(\approx 15.5345days)$.

3.6 IDLE Procedure

As mentioned in Sleep Modes, IDLE mode is base on CPU's feature solely. It's common for CPU to have an instruction that instruct CPU to stop and enter a slight sleep mode. It can wake up (continue to execute) *quickly* when there is any interrupt/exception, or other signals. For x86 processors, the instruction is *HLT* (halt), while for ARM Cortex-M processors, it is *wfi* (wait for interrupts) or *wfe* (wait for events). When there is only one ARM Cortex-M processor, the IDLE procedure is simply a *wfi* guarded by a pair of synchronization barriers:

```
__DSB();
__wfi();
__ISB();
```

Since there is a PLL in ING916, CPU and other components may run at a range of frequencies, event `PLATFORM_CB_EVT_IDLE_PROC` is defined to provide flexibility of customizing the IDLE procedure. This event is emitted each time IDLE mode is entered, and default IDLE procedure is replaced by the callback function. For example, Apps can lower clock frequencies to enjoy much low power consumption in IDLE mode if the slower response in waking up is acceptable:

```
static uint32_t idle_proc(void *dummy, void *user_data)
{
    SYSCTRL_ConfigPLLClk(DIV_PRE, LOOP, 63);
    __DSB();
    __WFI();
    __ISB();
    SYSCTRL_ConfigPLLClk(DIV_PRE, LOOP, DIV_OUTPUT);
    return 0;
}

platform_set_evt_callback(PLATFORM_CB_EVT_IDLE_PROC,
    idle_proc, NULL);
```

Chapter 4

Porting

When using NoOS variants of platform bundles, power saving framework needs to be ported to other RTOS(es). The main work is to port the idle process to the targeting RTOS. The idle process[1] looks like this:

```
void idle_task(void)
{
    for (;;)
    {
        timeout_tick = rtos_next_busy_tick();
        if (platform_pre_suppress_ticks_and_sleep_processing(
          timeout_tick) > some_limit)
        {
            rtos_stop_scheduler();
            platform_pre_sleep_processing();
            platform_post_sleep_processing();
            tick_cnt = how_long_has_I_slept();
            rtos_compensate_system_ticks(tick_cnt);
            restart_system_ticks();
            rtos_resume_scheduler();
        }
        platform_os_idle_resumed_hook();
    }
}
```

What needs to be ported is exactly those `rtos_...` functionalities:

- `rtos_next_busy_tick` returns how many ticks the CPU can be put in sleep modes;

[1] Since system tick interrupts are disabled during sleep, it is called tick-less low power feature in some RTOS(es).

- `rtos_stop_scheduler` stops the scheduler
- `rtos_compensate_system_tick` compensates ticks counter of RTOS
- `rtos_resume_scheduler` resumes the scheduler

`how_long_has_I_slept` calculates how long the last sleep was in ticks which is used to compensates ticks counter of RTOS. The result can be deduced by check system tick registers of CPU, or use other real-time timer including `platform_get_us_time`. `restart_system_tick` restarts system ticks.

Besides, optionally, similar function of `PLATFORM_CFG_RTOS_ENH_TICK` needs to be ported too.

The idle process must be the **only** process having the lowest priority. Some RTOS(es) supported multiple processes have the same lowest priority. Developers should ensure that no other processes are having the lowest priority.

4.1 API

NoOS variants of platform bundles provide a collection of APIs to cooperate the porting.

```
// Pre-suppress ticks and sleep processing
// @return  adjusted ticks to sleep
uint32_t platform_pre_suppress_ticks_and_sleep_processing(
  // expected ticks to sleep
  uint32_t expected_ticks
);

// Preprocessing for tick-less sleep
void platform_pre_sleep_processing(void);

// Postprocessing for tick-less sleep
void platform_post_sleep_processing(void);

// Hook for idle task got resumed
void platform_os_idle_resumed_hook(void);
```

4.2 Port to FreeRTOS

FreeRTOS[2] is a real-time operating system for micro controllers and small microprocessors. It is a cross-platform RTOS kernel that is distributed freely under the MIT open source license.

Platform binaries that has a built-in RTOS bundles FreeRTOS. NoOS variants are certainly portable to FreeRTOS. The portable can be done by just defining several macros to feed above APIs into FreeRTOS, such as:

```
#define configPRE_SUPPRESS_TICKS_AND_SLEEP_PROCESSING(xExpectedIdleTime)  \
  do {                                                                    \
    xExpectedIdleTime =                                                   \
      platform_pre_suppress_ticks_and_sleep_processing(xExpectedIdleTime);\
  } while (0)
```

A complete example can be found in SDK repository[3].

4.3 Port to RT-Thread

RT-Thread[4] is an open source embedded real-time operating system for IoT devices. It has a small size, rich features, high performance and scalability.

`rtos_...` functionalities can be mapped to relevant RT-Thread APIs as shown in Table 4.1.

Table 4.1: Relevant RT-Thread APIs

`rtos_...` Functionalities	RT-Thread APIs
`rtos_next_busy_tick`	`rt_timer_next_timeout_tick`
`rtos_stop_scheduler`	`rt_enter_critical`, `rt_hw_interrupt_disable`
`rtos_compensate_system_tick`	`rt_tick_set`
`rtos_resume_scheduler`	`rt_exit_critical`, `rt_hw_interrupt_enable`

A complete example can be found in SDK repository[5].

[2] https://www.freertos.org/

[3] https://github.com/ingchips/ING918XX_SDK_SOURCE/tree/master/examples/peripheral_console_freertos

[4] https://www.rt-thread.io/

[5] https://github.com/ingchips/ING918XX_SDK_SOURCE/tree/master/examples/peripheral_console_rt-thread

4.4 Port to Huawei LiteOS

Huawei LiteOS[6] was a lightweight real-time operating system for IoT devices developed by Huawei. It was open source, POSIX compliant, and has been incorporated into HarmonyOS.

`rtos_...` functionalities can be mapped to relevant LiteOS APIs as shown in Table 4.2.

Table 4.2: Relevant LiteOS APIs

`rtos_...` Functionalities	LiteOS APIs
`rtos_next_busy_tick`	`OsSleepTicksGet`
`rtos_stop_scheduler`	`LOS_IntLock`
`rtos_compensate_system_tick`	`OsSysTimeUpdate`
`rtos_resume_scheduler`	`LOS_IntRestore`

A complete example can be found in SDK repository[7].

4.5 Port to Azure RTOS ThreadX

Azure RTOS ThreadX[8] is Microsoft's Real-Time Operating System (RTOS) for deeply embedded, real-time, and IoT applications. It has a small footprint, fast execution speed, and advanced features such as preemption-threshold scheduling, event chaining.

`rtos_...` functionalities can be mapped to relevant ThreadX APIs as shown in Table 4.3.

Table 4.3: Relevant ThreadX APIs

`rtos_...` Functionalities	ThreadX APIs
`rtos_next_busy_tick`	`tx_timer_get_next`
`rtos_stop_scheduler`	`TX_DISABLE`
`rtos_compensate_system_tick`	`tx_time_increment`
`rtos_resume_scheduler`	`TX_RESTORE`

A detailed explanation can be found online[9].

[6] https://gitee.com/LiteOS/LiteOS

[7] https://github.com/ingchips/ING918XX_SDK_SOURCE/tree/master/examples-gcc/peripheral_console_liteos

[8] http://docs.microsoft.com/azure/rtos/threadx

[9] https://ingchips.github.io/blog/2022-03-11-threadx-porting/

4.6 Truly No OS

It's possible to run NoOS bundles without any RTOS, including full functional power saving. A complete example can be found in SDK repository[10].

In the example, because there are no software timers, and no next busy tick, the whole idle process couldn't be simpler:

```
static void idle_process(void)
{
    uint32_t ticks =
      platform_pre_suppress_ticks_and_sleep_processing(0xffffff);
    if (ticks < 5) return;
    enter_critical();
    platform_pre_sleep_processing();
    platform_post_sleep_processing();
    leave_critical();
    platform_os_idle_resumed_hook();
}
```

[10] https://github.com/ingchips/ING918XX_SDK_SOURCE/tree/master/examples-gcc/peripheral_console_realtime

Chapter 5

Tips on Low Power Products

Here are some tips for building low power products.

5.1 Clock Gating

Make sure that clock is gated for those peripherals that are not used. Since clock of all peripherals (except SYSCTRL and the default UART[1]) are defaults to gated, so, make sure do not release the gating for those peripherals that are not used. If default UART is not used, then gate its clock.

5.2 Choose Best Frequency

For SoC that has complex clock configurations like ING916, clock frequencies of each peripheral as well as CPU itself, should be chosen with care.

5.2.1 CPU

The lower frequency, the less current is consumed. But, with a lower frequency, processing time becomes longer. Do not only measure current, but also execution time.

Table 5.1 shows current consumption[2] of CPU running at varies frequencies under room temperature.

[1]It is UART0 if there are multiple UART peripherals.
[2]Refer to *ING916X BLE5.3 SoC Test Report*.

Table 5.1: CPU Current Consumption (mA)

Frequency (MHz)	$V_{bat} = 3.3V$	$V_{bat} = 2.5V$	$V_{bat} = 1.8V$
112	8.95	11.32	14.26
64	5.97	7.17	9.26
24	1.71	2.06	2.69
8	0.90	1.09	1.47

Better ways to measure power efficiency is to measure current per MHz (Table 5.2) or power per MHz (Table 5.3).

Table 5.2: CPU Current Consumption Per MHz ($\mu A/MHz$)

Frequency (MHz)	$V_{bat} = 3.3V$	$V_{bat} = 2.5V$	$V_{bat} = 1.8V$
112	89.9	101.1	127.3
64	93.3	112.0	144.7
24	71.3	85.8	112.1
8	112.6	136.3	183.8

Table 5.3: CPU Power Consumption Per MHz ($\mu W/MHz$)

Frequency (MHz)	$V_{bat} = 3.3V$	$V_{bat} = 2.5V$	$V_{bat} = 1.8V$
112	263.7	252.7	229.2
64	307.8	280.1	260.4
24	235.1	214.6	201.8
8	371.3	340.6	330.8

From these numbers we can see that:

- It's not power efficient to do computation at a very low frequency, for example, under the same condition, running at $8MHz$ consumes at least 40% more power than $112MHz$;
- $24MHz$ looks like a sweet spot if the processing delay is acceptable;
- If frequency is fixed, always try to use lower V_{bat} voltage.

5.2.2 CPU + Flash

Flash embedded in ING916 SoC interfacing CPU through QSPI, which might become a bottleneck without cache. Since cache is shutdown in Category B sleep modes, in a short duration after waking up, the system works like without cache. Therefore, it is important to measurement current consumption in such cases.

For example, create a function filled with 8192 NOP(s)(no operation) that is larger than I-Cache, loop the function for 200 times, and measure the used electric charge. Figure 5.1 shows the result. It can be found that when lowering Flash frequency, much more electric charge is used. To put it simple, run Flash as fast as possible to save power.

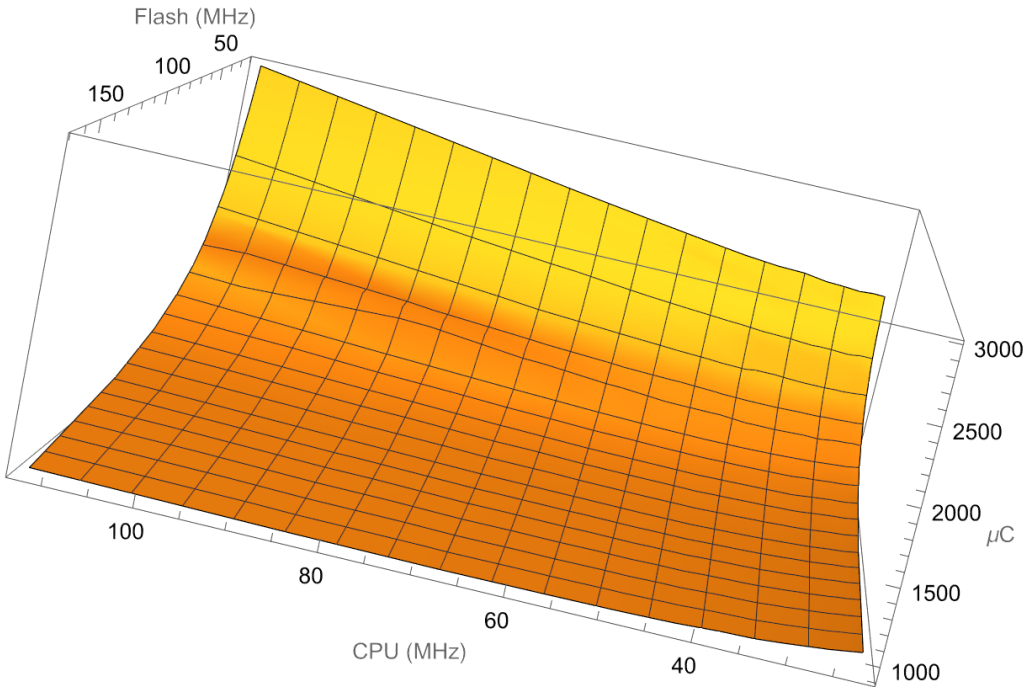

Figure 5.1: Idle Process

Further more, focusing on Flash running at $168MHz$, the best frequency for CPU is $\sim 67MHz$, as shown in Figure 5.2. If Flash running at $192MHz$, the best frequency for CPU is $\sim 77MHz$. These two groups of numbers show that in the case of no cache, it's better to set the frequency of CPU to be 40% of that of Flash, proving that Flash is really the bottleneck.

On the other handle, when cache becomes hot and has a high hit rate, it would be better to run CPU as fast as possible and the impaction of the slower Flash can be neglected.

5.2.3 Computational Intensive vs Hardware Timing

There are computational intensive jobs, and there are slow hardware operations that need quite some time to complete. Different strategies shall be considered.

For computational intensive jobs, check section CPU on how to let it run more efficiently. For slow hardware operations, check if sleep modes can be utilized; if not, consider changing to a lower clock frequency for CPU, Flash, and notably *hclk*.

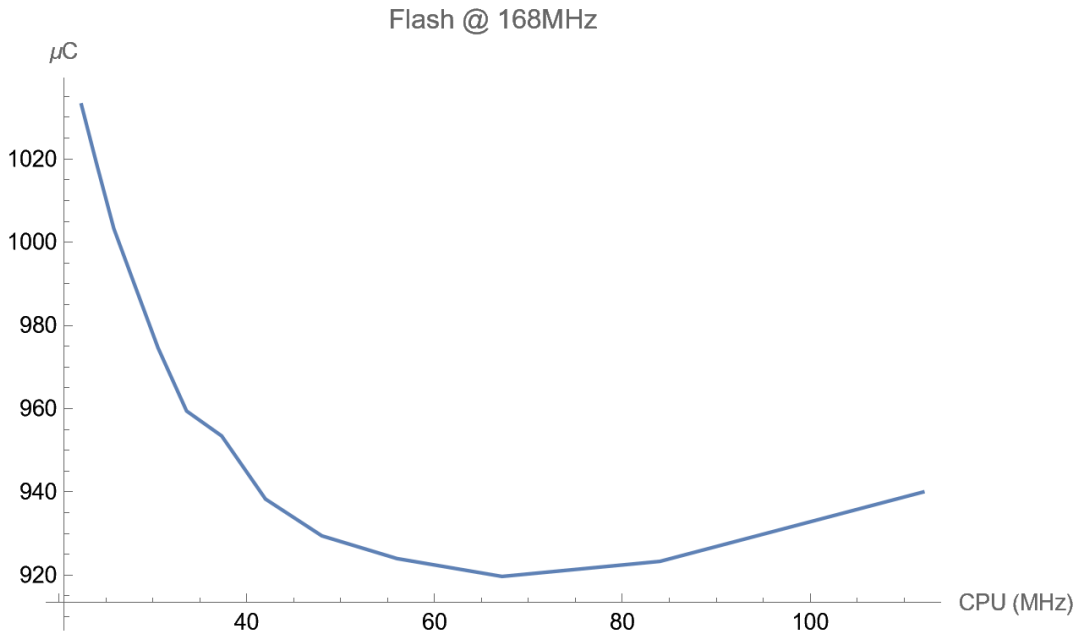

Figure 5.2: Idle Process

Chapter 6

FAQ

- **Q**: Why do the reconfiguration in event handler of waking up? The framework can remember all peripheral configurations and reconfigure them.
- **A**: Configure a peripheral is not an easy job like writing a bulk of registers. It's impractical to handle all peripherals one by one. On the other hand, it's easy for Apps to reconfigure them, as the code is ready there.

- **Q**: What happened to CPU in sleep modes of Category B?
- **A**: It's powered off.

- **Q**: How to debug programs when power saving is enabled?
- **A**: It depends on which part of programs need to be debugged.

  For the part between PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED and PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP, it's generally not debug-able.

  For other parts, firstly, disable power saving and debug it as usual. After everything is OK, re-enable power saving. Or, in the handler of PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED, check if debugger is attached, and if so, disallow sleep modes of Category B:

```
uint32_t query_deep_sleep_allowed(void *dummy,
  void *user_data)
{
  if (IS_DEBUGGER_ATTACHED())
```

```
    return 0;

  // ....
}
```