Realtek Ameba-1 Power Modes

Table of Contents

1 Power State

Ameba supports three low power modes which are deep sleep mode, deep standby mode, and sleep mode. Deep sleep mode turn off more power domain than deep standby mode, and deep standby mode turn off more power domain than sleep mode. Various power modes can only switch back to run mode before change to other mode, as shown in Figure 1.1:

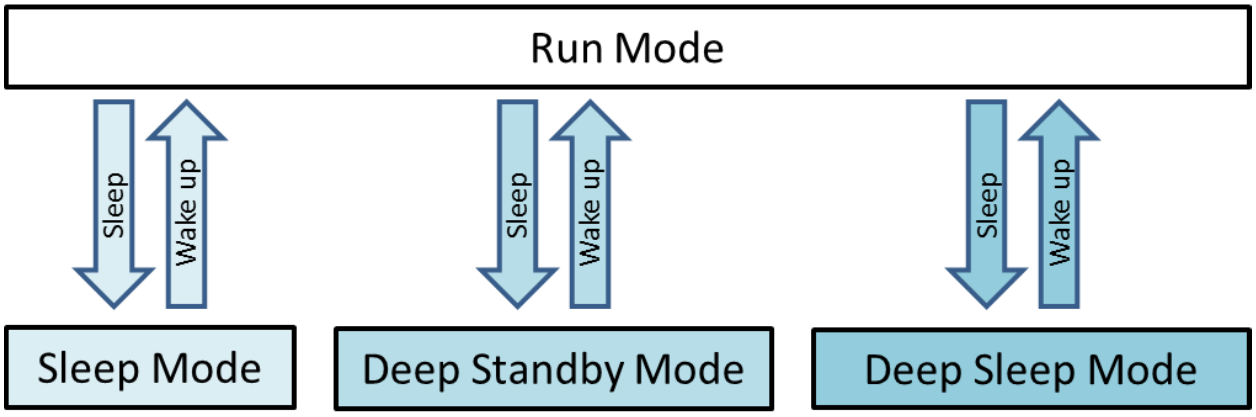


Figure 1.1. Ameba Power Mode

1.1 Deep Sleep Mode

Power Domain: Deep sleep mode turn off power domain including cortex-M3 core, system clock, SRAM, SDRAM, and regulator. Peripherals are turned off except wakeup source which serve one wake-up pin and one low precision timer to wake up system. All of the registers are turned off expect the ones that are used to kept wake-up pin. System restarts after wakeup.

Wakeup Sources:

- GPIOB_1
- General purpose timer

Each wakeup sources can be OR'ed, that means, either one condition fire up triggers wakeup event. (Ex. Both GPIOB_1 and lower precision timer can wakeup device).

1.2 Deep Standby Mode

Power Domain: Deep standby mode turn off power domain including cortex-M3 core, system clock, SRAM, SDRAM, and regulator. Peripherals are turned off except wakeup source which serve 4 GPIO and one timer to wake up system. Only around 200 bytes of registers are kept for wakeup usage, other registers are turned off. System restarts after wakeup.

Wakeup Sources:

- GPIOA_5
- GPIOC_7
- GPIOD_5
- GPIOE_3
- system timer

Each wakeup sources can be OR'ed.

1.3 Sleep Mode

Power Domain: Sleep mode turn off power domain including cortex-M3 core, and system clock. System is not required to restart after wakeup.

Wakeup Sources:

- GPIO interrupt
- system timer
- general purpose timer
- wlan protocol

1.3.1 Wakeup from sleep mode by UART

UART (or log UART) is not in wakeup sources of sleep mode. To support wake from UART, we can treat UART signal as GPIO interrupt signal and then wake system by GPIO interrupt. **Please note that it requires ~3ms for UART_RX to be ready to receive.**

Below are 2 solutions:

(A) Select a UART RX pin which is also an a GPIO interrupt pin
(B) Parallel UART RX with another GPIO interrupt pin.

Solution A costs no extra pin, but not all UART RX pin are also GPIO interrupt pin. So solution 1 needs select specific UART in pinmux.

Solution B costs 1 extra GPIO interrupt pin, but it provides flexible choice on UART pin selection.

Table 1.3.1 shows the choices in different package:

Package	**Solution A**	**Solution B**
8195AM	UART RX PA_0/PD_4 (Other UART RX pin are not GPIO interrupt pin)	Feasible
8711AM	N/A	Feasible
8711AF	UART RX PA_0 (Other UART RX pin are not GPIO interrupt pin)	Feasible

Table 1.3.1 UART wakeup selections on different package

Below sections describe how Solution A and B are implemented.

1.3.1.1 Solution A, select a UART RX pin which is also a GPIO interrupt pin

Let's take PA_0 (UART RX) and PA_4 (UART TX) as example. Please refer Figure 1.3.2:

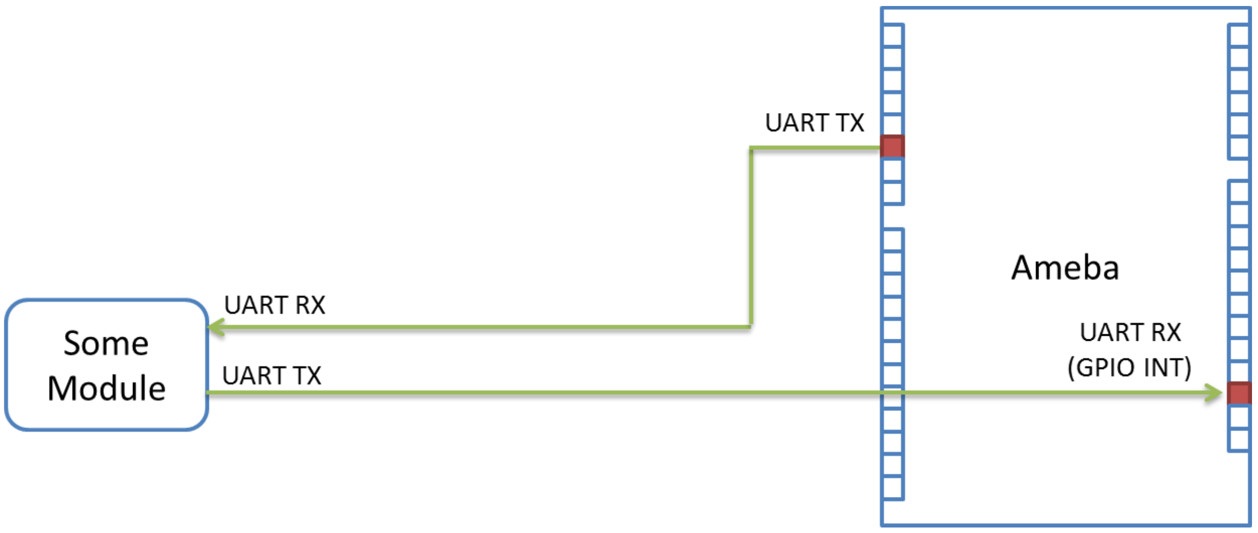


Figure 1.3.2 Use UART to wake Ameba

Please note that if module sends some characters to wakeup Ameba, "it requires **~3ms** for UART_RX to be ready to receive". There are two methods to handle this condition:

Method 1: implement a simple protocol:

(1) Sending a character to Ameba for wakeup usage before sending any data.
(2) Ameba receives this character, make sure system won't enter sleep again and send back a character for acknowledgement usage.
(3) The module receives the acknowledgement. Sending desired UART data with an ending character.
(4) Ameba receives this data and gets back to sleep.

Figure 1.3.3 shows this scenario:

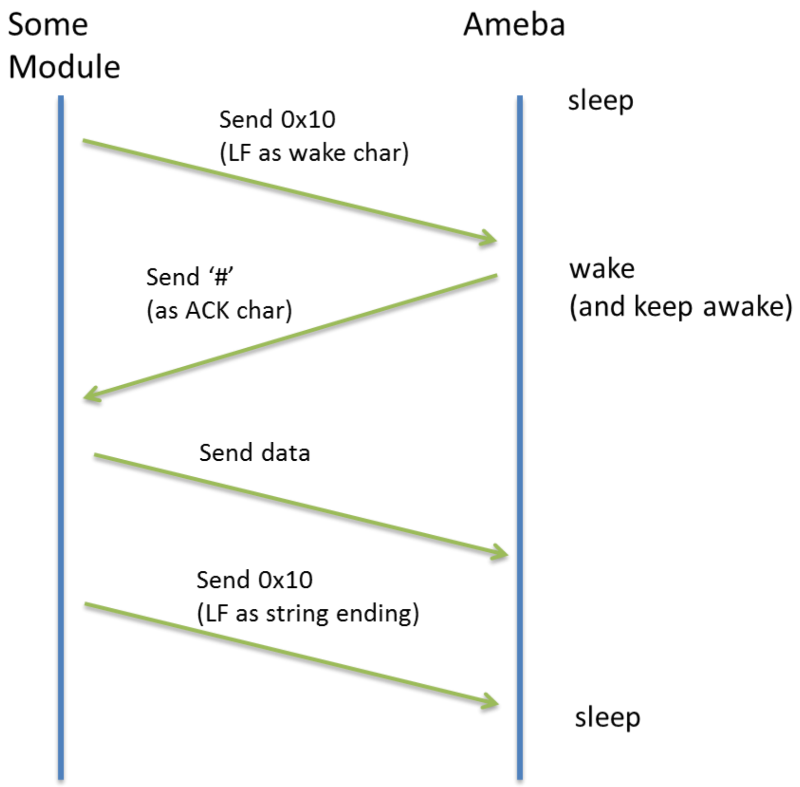


Figure 1.3.3 A simple protocol to send data to Ameba

Method 2: Enter Sleep mode without disabling PLL

If PLL clock is reserved during sleep mode, UART_RX requires no delay to back to ready for receive.

Please set FREERTOS_PMU_TICKLESS_PLL_RESERVED to 1 in platform_opt.h

#define FREERTOS_PMU_TICKLESS_PLL_RESERVED **1**

1.3.1.2 Solution B, Parallel UART RX with another GPIO interrupt pin

Let's take PA_6 (UART RX), PA_7 (UART TX), and PC_1 (GPIO interrupt) as example: Ref Figure 1.3.4:

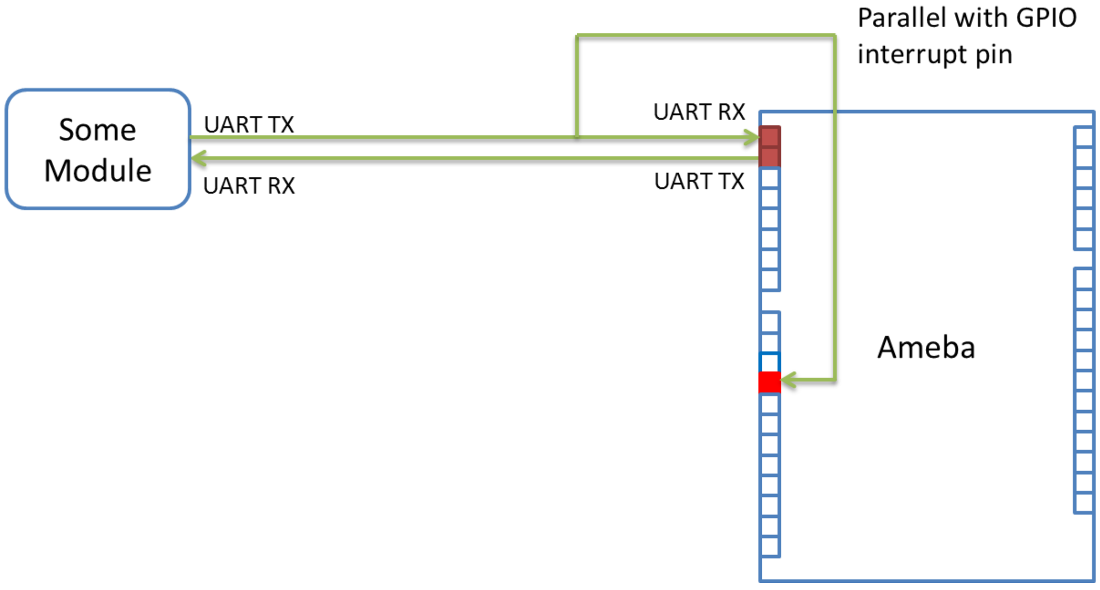


Figure 1.3.4 Use UART and parallel with GPIO interrupt to wake ameba

It's similar to Solution A excepts that it uses an extra GPIO interrupt pin to wake system. It can also use same protocol which described in Solution A to send desired UART data.

Please note that UART_Rx requires ~3ms after system wake up to back to ready to receive. Please refer to previous section in Solution A for suggested methodologies.

1.4 Power domain

Table 1.4.1 shows the comparison of power domain in various power saving mode:

	System Status during Power Save							
	Cortex M3 core	System Clock	Lower Power Clock	SRAM	Register	Regulator	Main digital supply	Peripheral
Deep Sleep	X	X	O	X	X	X	O	Δ
Deep Standby	X	X	O	X	X	X	O	Δ
Sleep	Δ	Δ	O	O	O	O	O	O
Active	O	O	O	O	O	O	O	O

Table 1.4.1 Power domain comparison

Table 1.4.2 shows the comparison of wakeup source and wakeup procedure.

	Wakeup source	Wakeup Procedure Required			
	Wakeup Source	System restart	Wlan init	Wlan connect	Peripheral init
Deep Sleep	1 gpio / general purpose timer	Yes	Yes	Yes	Yes
Deep Standby	4 gpio / system timer	Yes	Yes	Yes	Yes
Sleep	gpio (interrupt) / system timer / general purpose timer / wlan	No	No	No	No
Active	N/A	No	No	No	No

Table 1.4.2. Wake sources comparison

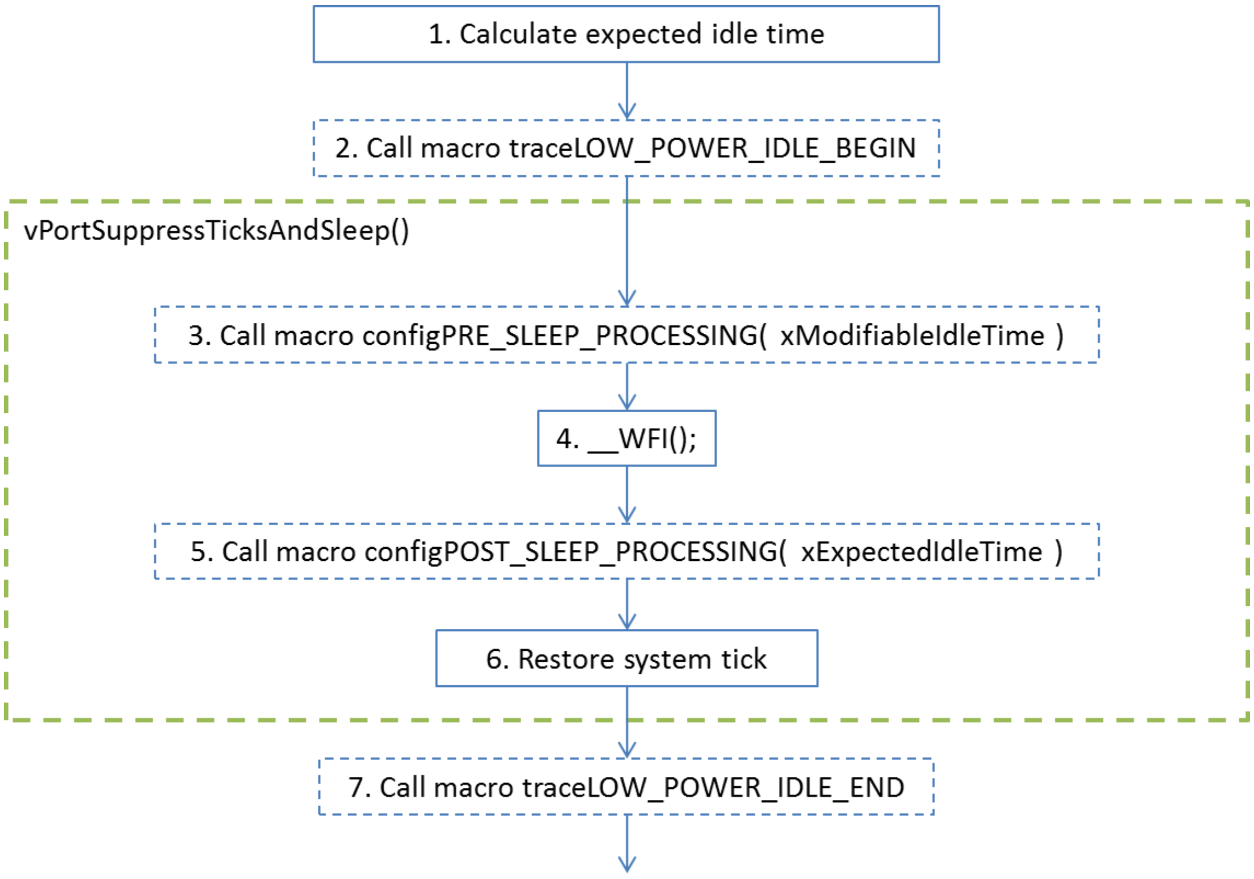
1.5 Pull control

It needs doing I/O pull control when enter deep sleep, deep standby, and sleep mode. Otherwise it result power leakage. For example, UART voltage level is high. If we pull down uart pin or not pull, then power leakage happens. So **we need make sure each pin has proper pull control**. In SDK, example pm_deepsleep and pm_deepstandby provide a reference pull control for EVB board.

2 FreeRTOS Low Power Feature

2.1 FreeRTOS tickless design

FreeRTOS support a low power feature called tickless. It is implemented in idle task which has lowest priority. It means it is invoked when there is no other task under running. Its idea is place the microcontroller into a low power state if it expects there will be no event in nearly future. It calculates expected idle time by looking timer task list. Then it performs suspend action by using ARM instruction "WFI" (Wait For Interrupt) which makes the processor suspend execution (Clock is stopped) until interrupt happened. The interrupts include timer which set by FreeRTOS with value equal to expected idle time. So every time idle task is invoked, it calculates the expected idle time, setup timer and then put process into suspend. Picture 2.1 illustrate the simplified call flow.


1. Calculate expected idle time
2. Call macro traceLOW_POWER_IDLE_BEGIN
vPortSuppressTicksAndSleep()
3. Call macro configPRE_SLEEP_PROCESSING(xModifiableIdleTime)
4. __WFI();
5. Call macro configPOST_SLEEP_PROCESSING(xExpectedIdleTime)
6. Restore system tick
7. Call macro traceLOW_POWER_IDLE_END

Picture 2.1 FreeRTOS Tickless in idle task

At step 1, it calculates expected idle time. Then it does some condition check. For example, it checks if the idle time is larger than configEXPECTED_IDLE_TIME_BEFORE_SLEEP, otherwise it aborts suppressing tick process.

Step 2 and step 7 are un-implemented macro. It is for tracing sleep process before/after vPortSuppressTickAndSleep(). Although it may abort sleep process inside vPortSuppressTickAndSLeep(), so **we can consider Step 2 as First gate** before entering sleep.

After step 2, there are some tasks. It double check sleep conditions, backup timer related registers, and check if some interrupt happens at this moment.

Step 3 and step 5 are un-implemented macro which are actually enter/leave sleep. So **we can consider Step 3 as second gate** before entering sleep.

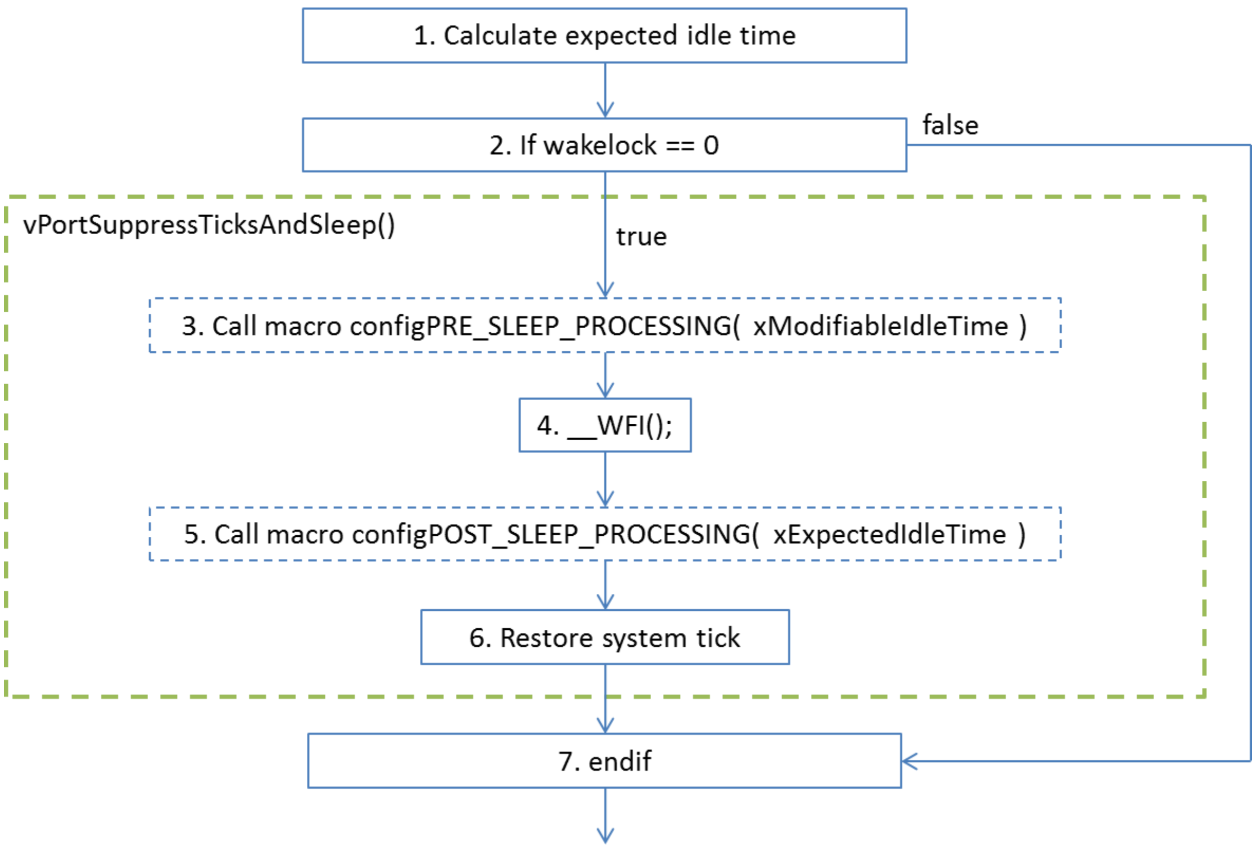
After Step 3, it is enter sleep. FreeRTOS uses WFI which is AMR instruction for sleep.

2.2 Wakelock Feature

In some situations, we need system keep awake to receive certain events. Otherwise event might be missed when system is under sleep. An idea of wakelock is introduced that system cannot sleep if some module holding wakelock.

We implement wakelock api by implementing macro "traceLOW_POWER_IDLE_BEGIN" as a function and check wakelock status. If there is no one holding wakelock, then system is permit to sleep. If there are one or more modules holding wakelock, then we abort sleep.

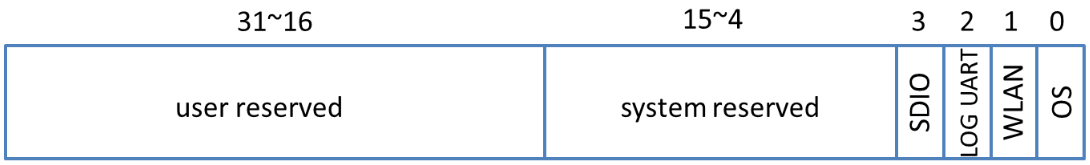
Picture 2.2 illustrate the wakelock design:


1. Calculate expected idle time
2. If wakelock == 0
false
vPortSuppressTicksAndSleep()
true
3. Call macro configPRE_SLEEP_PROCESSING(xModifiableIdleTime)
4. __WFI();
5. Call macro configPOST_SLEEP_PROCESSING(xExpectedIdleTime)
6. Restore system tick
7. endif

Picture 2.2 wakelock feature in FreeRTOS Tickless design

We can see step 2 and step 7 are modified as condition check for wakelock.

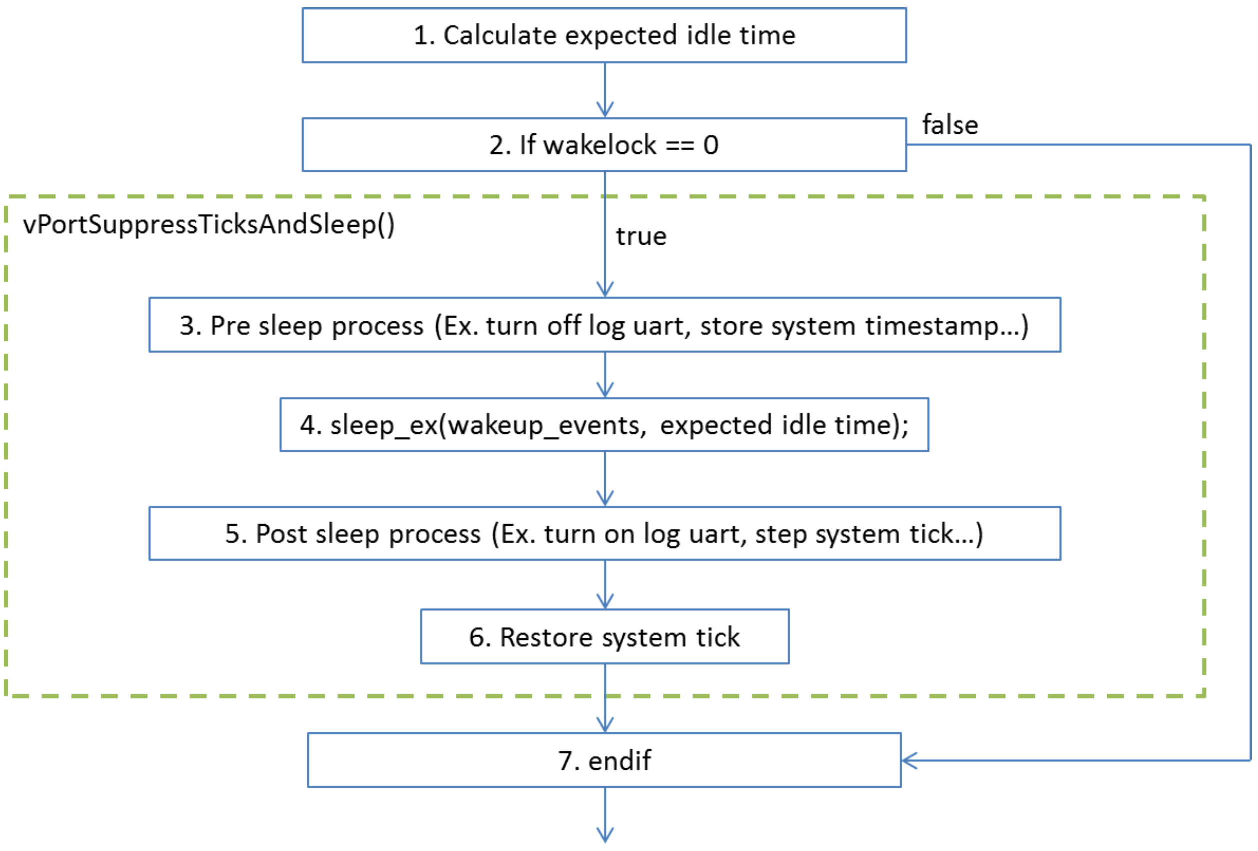
A wakelock bit map is for storing the wakelock status. Each module has its own bit in wakelock bit map. If the wakelock bit map equals to zero means there is no module holding wakelock. If the wakelock bit map larger than zero means there is some module holding wakelock. Bit 0~15 are reserved for ameba system usage, and user can use bit 16~31.



Picture 2.3 wakelock bit map

2.3 Use Ameba sleep in tickless

By default FreeRTOS uses ARM WFI which only suppresses CPU tick. We can save more power consumption if we use Ameba sleep. Picture 2.4 illustrate the modification:


1. Calculate expected idle time
2. If wakelock == 0
false
vPortSuppressTicksAndSleep()
true
3. Pre sleep process (Ex. turn off log uart, store system timestamp…)
4. sleep_ex(wakeup_events, expected idle time);
5. Post sleep process (Ex. turn on log uart, step system tick…)
6. Restore system tick
7. endif

Picture 2.4 wakelock feature in FreeRTOS Tickless design

Step 3, 4, 5 are modified. In Step 3, we turn off peripherals like log uart. Please note that CPU tick won't update during sleep because we close CPU. It means software timer may works abnormal if we don't update system tick. So we store system timestamp which get from "us_ticker_api.h".

Step 4 performs the sleep. System sleeps with expected idle time if there is no other interrupt wake up system. The wakeup events include most events include system timer, gtimer, gpio interrupt, wlan protocol interrupt.

In Step 5 we turn on peripherals like log uart. And we check how much time passes, and update system tick accordingly.

2.4 Wakelock AT command

We provide AT command to use wakelock api. Below are the description

- **A**cquire wakelock
  ATSL=a[bitmap]
  Ex. ATSL=a[00010000], it acquires wakelock at bit 16

- **R**elease wakelock
  ATSL=r[bitmap]
  Ex. ATSL=r[00010000], it releases wakelock at bit 16

- Query wakelock status
  ATSL=?
  It print current wakelock bit map. We can use this command to debug why system doesn't enter sleep

AT command is sent through log uart. However it is hard to enter any command if system is under tickless. So we implement the simple uart protocol (Ref section 1.3.1) for log uart. It means log uart module acquire wakelock if user press "Enter", and release wakelock after user complete a command.

If user want to enter several commands and don't want system enter tickless mode. He can follow below scenario:

1. ATSL=a[00010000]
2. Send desired several AT commands
3. ATSL=r[00010000]

3 Put UART into tickless design

Section 1.3.1 describe how to make uart wake system. We can also put uart simple protocol into tickless design. Figure 3.1 illustrate the modification:

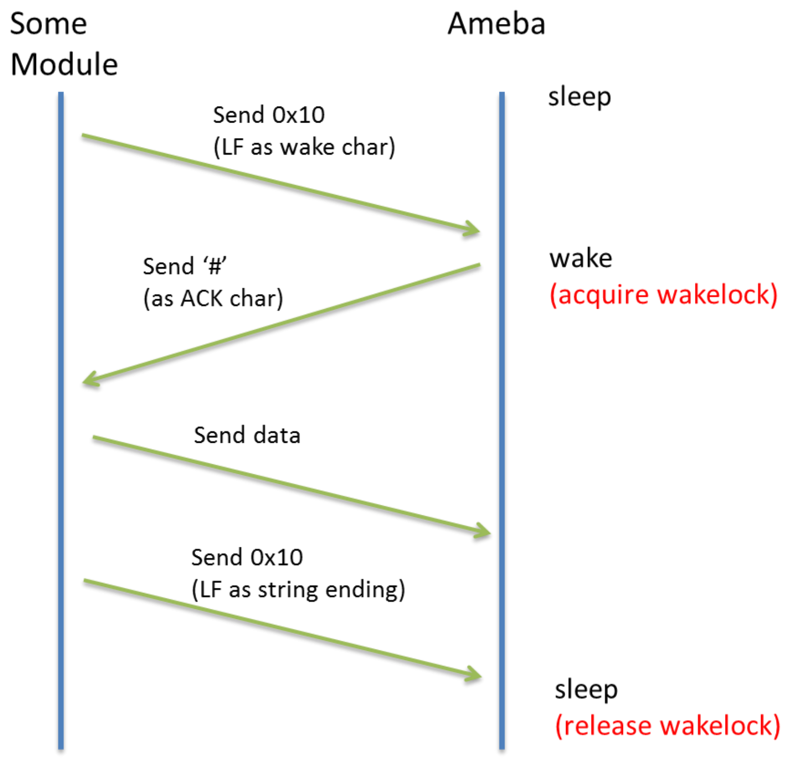


Figure 3.1 UART in tickless design

4 Put WLAN into tickless design

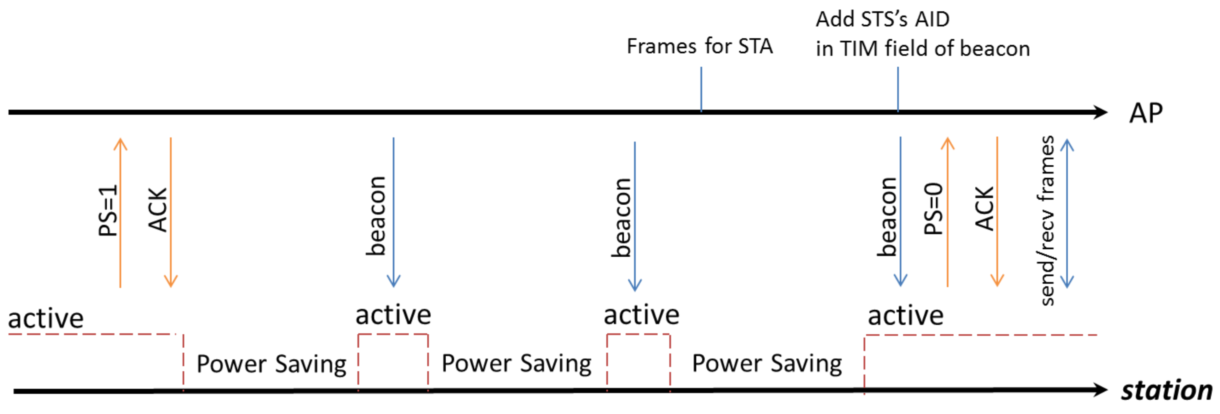
In IEEE 802.11 power save management, it allows station enter their own sleep state. It defines station need keep awake in certain timestamp and may enter sleep state otherwise.

Wlan driver acquire wakelock to avoid system enter sleep in tickless design when wlan need keep awake. And it release wakelock when it is permitted to enter sleep state.

Below sections describe IEEE 802.11 power save, and its implementation on Ameba include tickless design.

4.1 IEEE 802.11 power management

IEEE 802.11 power management allows station enter power saving mode. Station cannot receive any frames during power saving. Thus AP need buffers these frames and requires station periodically wakeup to check beacon which has information of buffered frames. Picture 4.1 illustrates the timeline of power saving.



Picture 4.1 timeline of power saving

4.2 Ameba LPS

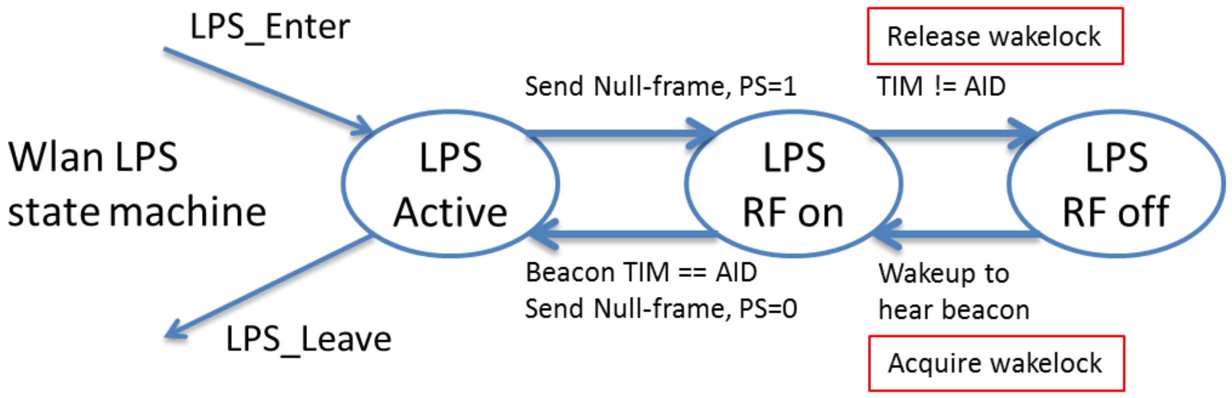
This feature is implemented in wlan driver. wlan driver enters LPS automatically without user application involved.

Ameba LPS (Leisure Power Save) implements IEEE 802.11 power management. Wlan driver enters LPS if flowing criteria meets:

(i) TX + RX packets count <= 8 in 2 seconds

(ii) RX packets count <= 2 in 2 seconds

It is checked in traffic status watch dog. The criteria are to keep high performance while traffic is busy. After enter LPS, there is PMU (Power Management Unit) control state machines. Please refer Picture 4.2:

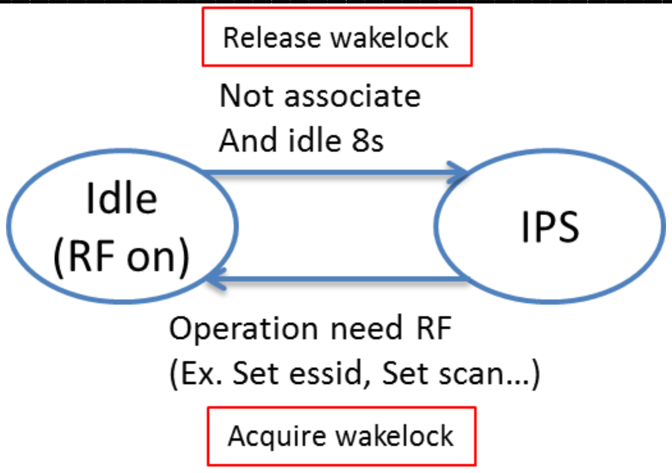


Picture 4.2 LPS state machine

4.3 Ameba IPS

This feature is implemented in wlan driver. Wlan driver enters IPS automatically without user application involved.

Ameba LPS is for situation that Ameba is associate to an AP. If Ameba is not associated to an AP, driver automatically turns off RF and other module to save power. Wlan Driver also releases wlan’s wakelock at this time. When wlan driver needs to use RF related function, it automatically turns RF on and acquire wlan’s wakelock. This scenario is called IPS (Inactive Power Save).



Picture 4.3 IPS state machine

5 Measure Power Consumption

5.1 Hardware preparation

In Ameba-1 reference board 3V0, there are other components that consume power. For example, there are cortex-M0 for DAP usage, LEDs, and capacitances. To measure power consumptions only for Ameba-1, you need remove capacitance at R43. And you can weld wires as Figure 5.1.1:

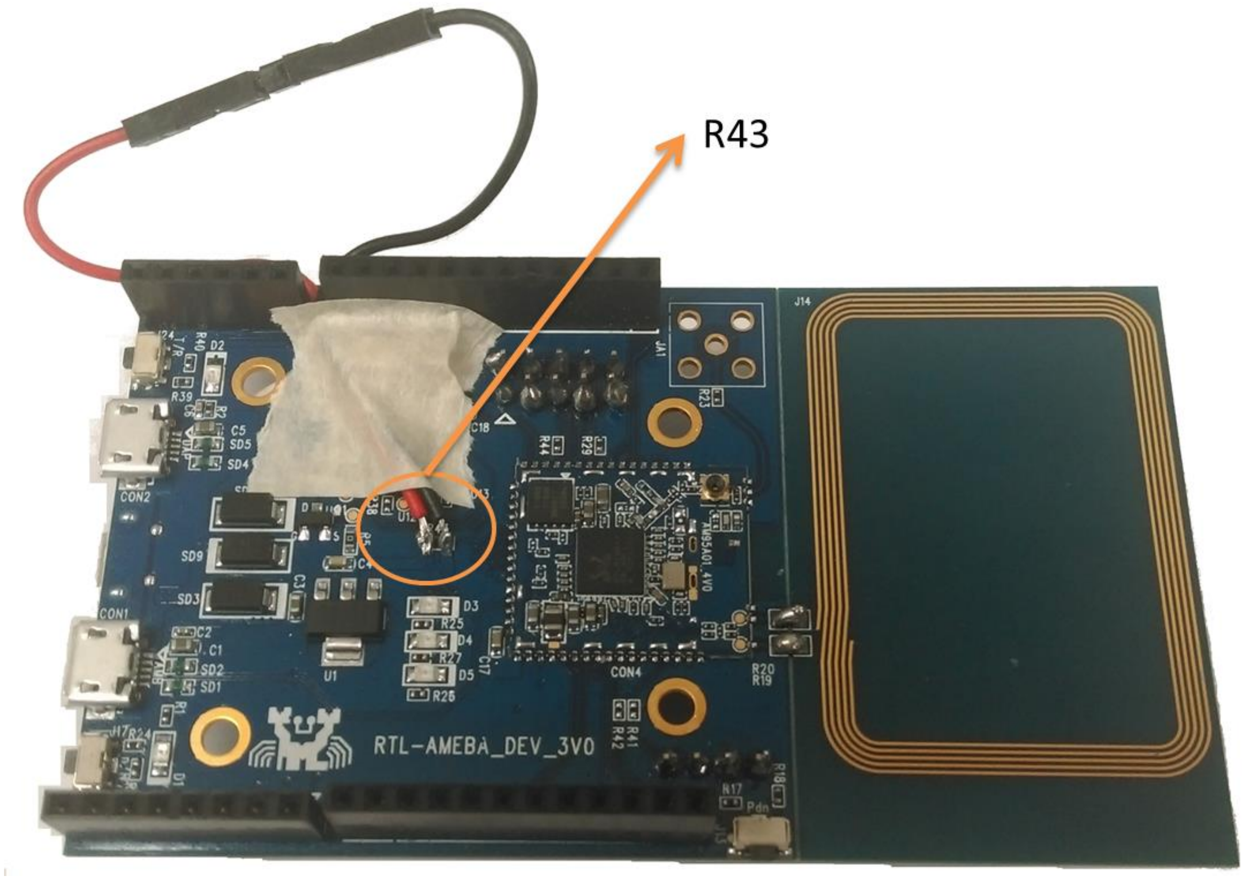


Figure 5.1.1 Power consumption measurement

In the right side of R43 as shown in figure 5.1.1, there is a black wire which is power source of Ameba-1. You can connect this point to current meter and 3V3 power supply to measure power consumption. In this case, it is required to use J-link interface for code loading, and it is also required to provide J-Link power separately. Figure 5.1.2 shows this idea:

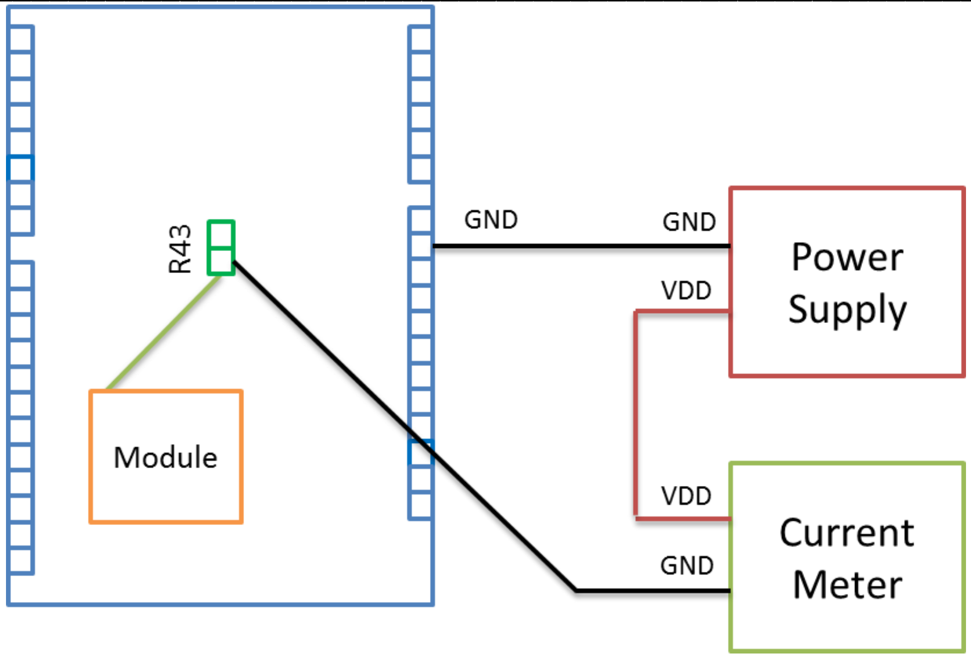


Figure 5.1.2 Measure power consumption from power supply

If you want to use power source from micro usb, you can use the red wire to connect to a current meter and then connect to the black wire. In this way, you have to consider power usage from JTAG component lunched by DAP component. Figure 5.1.3 shows this idea:

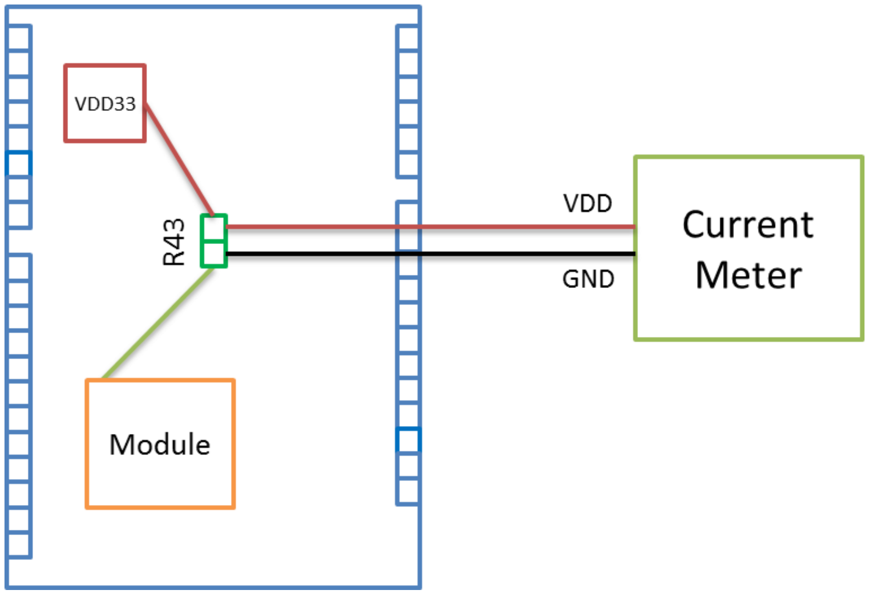


Figure 5.1.3 Measure power consumption from micro usb

5.2 Build SDK example

Example “**pm_tickless**” illustrate how to use wakelock in tickless mode and we can measure power consumption of wlan associate idle. It is similar to example wlan except that (1) it configure UART as wake up source. (Ref Section 1.3.1 and Section 3), and (2) it release OS wakelock in initialize.

Below are suggesting operations to measure power consumption of wlan association idle:

1. When device boot up, **press several “Enter”** in log uart to avoid system enter tickless mode. Log uart module acquire wakelock at this moment.
2. Send AT command:
   ATSL=a[00010000]
   This acquire user defined wakelock and avoid system enter tickless mode. (Ref section 2.4) Log uart module release its wakelock when user completes a command. So we need acquire another user defined wakelock.
   This command is necessary if we want to send several commands.
3. Send AT command:
   ATW0=my_ap_name
   ATW1=my_ap_password
   ATWC
   Wait until wlan associate success.
4. Send AT command:
   ATSL=r[00010000]
   It release user defined wakelock. Now system is under wlan association idle and tickless mode.

You may find it hard to type any command under tickless mode. Please refer section 1.3.1 and section 3 to let log uart has capability to wake system.