User manual

DA14580 Sleep mode configuration

UM-B-006

Abstract
This document contains information about the sleep mode configuration of the DA14580 and the software architecture that has been defined to support these modes of operation.

Contents

Figures

Tables

1 Terms and definitions

AON	Always ON
API	Application Programming Interface
ARM	Acorn RISC Machines
BLE	Bluetooth Low Energy
GPIO	Generic Purpose Input Output pin
I2C	Inter-Integrated Circuit interface
ISR	Interrupt Service Routine
OS	Operating System
RAM	Random Access Memory
RC16M	16MHz clock generated by internal oscillator
ROM	Read Only Memory
SPI	Serial Peripheral Interface
SysRAM	System RAM
UART	Universal Asynchronous Receiver Transmitter
WFI	Wait For Interrupt instruction
XTAL16M	16MHz crystal

2 References

1. DA14580 DataSheet, Dialog Semiconductor
2. UM-B-002, Software Architecture, Dialog Semiconductor

3 Introduction

3.1 Sleep modes

This document describes the software architecture of the sleep modes of DA14580. The various modes of operation of the chip are:

1. Active mode,
2. Extended sleep mode,
3. Deep sleep mode.

In active mode, the system domain (ARM processor, SysRAM, ROM, etc.), the radio (including Radio and Bluetooth Low Energy – BLE core) and the peripheral domains (UART1/2, I2C, SPI, etc.) are active.

In Extended sleep mode, the system domain except the SysRAM, the radio domain and the peripheral domain are powered down and the XTAL16M clock is stopped. The SysRAM is still powered to retain data but is not accessible. The Always ON (AON) power domain is on to keep data in the retention RAMs and to supply power to the blocks that can wake the system up, i.e. wakeup timer, quadrature decoder and the BLE timer.

In Deep sleep mode, to reduce the power consumption even further, the SysRAM is also powered off. The status of the other power domains is the same as in Extended sleep mode.

3.2 Wake up events

When in any of the above mentioned sleep modes, DA14580 can be woken up in two ways:

1. Synchronously, via the BLE timer which can be programmed to wake up the system in order to serve a BLE event and
2. Asynchronously, via the Wakeup Timer and Quadrature Decoder if triggered by an external event (input).

In a BLE application, DA14580 could be set to either of the above mentioned sleep modes. For an advertising event, connection event or other wireless communication event, DA14580 needs to be woken up and go to active mode in order to send/receive packets over a BLE wireless link. Since these events are time based events, the BLE timer is used to wake up the system, including the BLE core, the radio, the ARM processor and the rest of the blocks. In this case, the following convention is used: "*the system is woken up synchronously with the BLE core*".

When in Extended / Deep sleep mode, DA14580 can also be woken up by an external event and after waking up the ARM processor can perform some functions. However, at that moment it may not yet be the time for a BLE communication event, e.g. a connection event and thus the BLE and the radio can remain in power off state. In this case, the following convention is used: "*the system is woken up asynchronously with the BLE core*".

If the system is woken up asynchronously then any requests for transmission of messages to kernel tasks (and, eventually, over the Bluetooth wireless link) cannot be performed immediately and must be synchronized to the BLE core. This is because the stack is built in such a way that the handling of message events requires the BLE core to be active so that timing information from the BLE core is available. This timing information is not available when the BLE core is powered down.

4 Description of sleep mode concept

In this section, a step-by-step description is given of the way a kernel's main loop can be modified in order to incorporate power saving features.

The simplest form of the main loop is shown in Figure 1.

```
void main_loop(void)
{
    while(1)
    {
        schedule();
    }
}
```

Figure 1: Simple main loop

This simple loop executes the kernel's scheduling function all the time. The main processor is never idle even when nothing needs to be done.

An evolution of this simple loop is shown below.

```
void main_loop(void)
{
    while(1)
    {
        schedule();

        WFI();
    }
}
```

Figure 2: Main loop that allows main processor to be idle

With the addition of a call to `WFI()`, the loop executes `schedule()`, which serves all pending events and tasks, and, then, the main processor is halted, waiting for an interrupt to occur. This approach can be used in interrupt driven systems like DA14580[1]. It introduces a power saving feature since the main processor consumes lower power when it is halted than when it is active.

This approach can be extended further to take into account that specific blocks of the system can be powered down. In DA14580, the BLE core and the radio can be turned off, if not needed. In order to support this feature, the following main loop can be used, assuming that the system always wakes up synchronously to the BLE core. This is a constraint that comes from the fact that, in DA14580, the kernel's timing reference is taken from the BLE core. So, in order to able to call `schedule()`, the BLE core and the radio must be active.

```
void main_loop(void)
{
    while(1)
    {
        schedule();

        GLOBAL_INT_STOP();

        if (ble_sleep() == true)
        {
            set_radio_off();
        }

        WFI();

        GLOBAL_INT_START();
    }
}
```

Figure 3: Main loop that allows the BLE core and the radio to be powered down

[1] Actually, the same approach is used even in non-interrupt driven systems. In this case, a System Timer is set to hit at every i.e. 10ms which triggers the execution of the main loop.

With the above approach, not only the main processor is halted when all processing has finished but the BLE core and the radio are turned off when they are not needed, reducing the power consumption even further. Apart from controlling the power status of the BLE core and the radio, the DA14580 offers the ability to control the state of all the power domains except for the AON domain.

In order to exploit the power management capabilities of DA14580 to the highest level, the main loop can be modified as shown below.

```
void main_loop(void)
{
    while(1)
    {
        schedule();

        GLOBAL_INT_STOP();

        sleep_mode = ble_sleep();

        if (sleep_mode == mode_ext_sleep || sleep_mode == mode_deep_sleep)
        {
            set_radio_off();

            SCB->SCR |= 1<<2;         // turn-off System PD with the next WFI() call

            if (sleep_mode == mode_ext_sleep)
            {
                set_pd_ext_sleep();  // turn-off PDs for Extended sleep
            }
            else
            {
                set_pd_deep_sleep(); // turn-off PDs for Deep sleep
            }

            WFI();

            SCB->SCR &= ~(1<<2);
        }
        else
        {
            WFI();
        }

        GLOBAL_INT_START();
    }
}
```

Figure 4: Main loop that implements Deep and Extended sleep modes

In this main loop, after the call to `schedule()`, the possibility to sleep is investigated by calling `ble_sleep()`. If sleeping is allowed then the BLE core and the radio are turned off as in the previous case. Apart from this, though, the other power domains are set to their proper state according to the selected mode of operation (Extended or Deep sleep). If sleeping is not allowed because of pending BLE activity, then a simple `WFI()` call is issued to halt the main processor.

This main loop suffices for applications that wake up synchronously. This way it is guaranteed that whenever the main loop is executed, the BLE core and the radio are active and `schedule()` can be called.

As mentioned before, the DA14580 can also wake up asynchronously. So, the main loop will have to be executed while the BLE core and the radio are turned off. In this case, a different structure is needed for the main loop. This is described in detail in section 6.

5 Requirements

Based on the above, a SW architecture needs to be devised that will accommodate the following:

- Handling of entry to and exit from the sleep modes of operation.
- Handling of asynchronous wake up events / requests to send messages to kernel tasks and transmit data over the Bluetooth link.

Also, it has to provide an API so that the application can:

- dynamically modify the mode of operation (Active, Extended sleep, Deep sleep) according to its needs and
- add application handlers
  - for asynchronous events (note that the term "*asynchronous*" characterizes all events that their execution is not handled by kernel tasks and, thus, are not synchronous to the operation of the BLE core and the stack),
  - for the synchronization of requests for transmission made by asynchronous events and
  - for actions the application needs to take just before the system enters into the sleep mode and immediately after exiting.

6 SW architecture

The following pseudocode describes the functionality of that part of the SW that is responsible for scheduling the various messages generated by the various tasks, for serving the BLE core's requests and handling the asynchronous events' processing and their synchronization and of the entry to / exit from the sleep modes.

```
    while(1)
    {
    ------------------------------------------------------------ Part 1: Scheduling (synchronous)
           IF BLE is ON {
              DO scheduling

              DO synchronization of asynchronous events' requests                    (Hook #1)
           }

    ------------------------------------------------------------ Part 2: Processing of asynchronous events
           DO asynchronous events processing                                      (Hook #2)

    ------------------------------------------------------------------ Part 3: Sleep entry / exit
#if SLEEP_ENABLED
           GLOBAL_INT_STOP();

           DO asynchronous events sleep processing                                (Hook #3)

           // if app has turned sleep off, rwip_sleep() will act accordingly
           SET sleep_mode based on rwip_sleep() result

           // update mode according to app sleep flags
           IF sleep_mode IS sleeping // app defines the mode
              IF App has selected deep sleep
                 SET sleep_mode to deep sleep
              ELSE
                 SET sleep_mode to extended sleep

           IF (sleep_mode IS extended sleep) OR (sleep_mode IS deep sleep) {
              SET radio off

              DO descriptors and non-ret Heap checks

              DO asynchronous events sleep preparation processing                    (Hook #4)

              IF (sleep_mode IS extended sleep) OR (sleep_mode IS deep sleep) {
                 SET SCB->SCR[2] to 1
                 SET Pad Latches to ON
```

```
                SET Peripheral power domain to OFF

                IF sleep_mode IS extended sleep {
                    SET SRAM as retained
                    SET OTP copy to disabled
                } else { // deep sleep
#if DEVELOPMENT__NO_OTP
                    SET SRAM as retained
#else
                    SET SRAM to OFF
#endif
                    DO Prepare OTP
                }
            }

            DO App specific tasks just before sleeping                    (Hook #5)


            WFI();

            DO App specific tasks immediately after wake up               (Hook #6)

            SET SCB->SCR[2] to 0
        }
        ELSE IF sleep_mode IS idle
            WFI();

        GLOBAL_INT_START();
#endif
    }
```

Note that the asynchronous events are generated by external interrupts. It is strongly advisable that the corresponding ISRs are kept as short as possible so that the system exits with minimal delay and returns to user mode. Any processing has to be done in **Hook #2**. If the outcome of this processing results in sending one or more data packets over the BLE wireless link then this must be handled in **Hook #1**. Since inter-process communication (task messaging) requires timing information that is available only when the BLE core is active, **Hook #1** is where any application task message shall be sent to the lower layers of the stack (be "*synchronized to the BLE*").

Since this point is critical, a few examples follow that clarify how these two hooks are used.

Assume that DA14580 is sleeping and an interrupt is issued by the Wakeup Timer and Quadrature Controller block. This interrupt wakes up the system. In the corresponding ISR, after acknowledging the interrupt, a flag is set (i.e. `event1_flag`). The intention is to send a Notification over the BLE wireless link at the next connection event. Thus, the developer needs only to synchronize the `event1_flag`. So, code has to be added in **Hook #1** that checks this flag and sends the appropriate message to the stack. The `event1_flag` must be placed in the Retention RAM since DA14580 may go to sleep until the BLE core eventually wakes up.

If the flag is used to trigger asynchronous processing of data then code has to be added in **Hook #2** that checks the `event1_flag`. If the processing results into having to send a Notification over the BLE link then another flag must be set (i.e. `event1_sync_flag`). This flag must be placed at the Retention RAM because the DA14580 may fall again into sleep until the BLE core eventually wakes up. The `event1_flag` does not have to be in the Retention RAM in this case. In **Hook #1** code must be added to check the `event1_sync_flag` and send the appropriate message to the stack.

Finally, there is a case where connection latency is used and the Notification must be sent at the next latency anchor point. In this case, the code in **Hook #2** must additionally force a wake up of the BLE core. An API function is defined for this purpose. The `event1_sync_flag` needs not be in the Retention Ram in this case since the DA14580 will not go to sleep until the BLE core is up and scheduling is done.

Hook #3 is provided so that the application can make sure that no asynchronous events are missed before deciding whether it should fall in sleep or not. For example, after examining a flag that was set by an asynchronous event's ISR, the application may alter its state at this point. In order to

demonstrate the necessity of this hook, consider an application that has programmed the Wakeup Timer and Quadrature Decoder block to monitor three external inputs and wake up the system whenever one of them is triggered. The application uses three flags to indicate the occurrence of any of these events (i.e. `event1_flag`, `event2_flag` and `event3_flag`). All events trigger asynchronous processing in **Hook #2**. It is obvious that the code in this hook will examine these flags one after the other and execute the actions specified for each one of them. Assume a case where DA14580 wakes up because the third input was triggered and that the `event3_flag` is set. In **Hook #2**, the code will check the first two flags and find them to be not set so it will move on to the third one, which is set, and start executing any actions programmed for this case. Finally, assume that during this processing the first input triggers an interrupt. The processing will stop, the ISR will be executed and the `event1_flag` will be set. When the ISR exits the code will resume execution from the point it was interrupted. The problem is that **Hook #2** has already checked `event1_flag` and found it was not set. So, if the code execution continues without checking this flag again and the chip goes to sleep then this event will be lost. Thus, in **Hook #3**, which is in a section that is protected from interrupts, all events' flags can be checked again to make sure that no asynchronous events are lost.

Hook #4 is provided so that the application may dynamically alter any previously taken decision to go to sleep, based on its current state. So, continuing the previous example, the application would check its state at this point and if it does not allow sleeping then it would change sleep_mode to idle. This way no power domains would be shut off and a simple `WFI()` would be executed. Note that the BLE and the Radio are still powered off in this case. In other words, this hook offers the ability to use a "hybrid sleep mode" where the BLE core and the radio are turned off but the other power domains are on.

Hook #5 and **Hook #6** are provided so that the application can execute specific functionality just before entering and immediately after exiting the low power mode. For example, an application may need to program the Wakeup Timer and Quadrature Decoder block to wake up the system while it is sleeping. Or, it may choose to lower the RC16M frequency when in "hybrid sleep mode" with peripherals active to reduce the power consumption even further during the WFI period. After waking up, the application should restore the RC16M frequency again.

The application can modify the mode of operation (Active, Extended or Deep sleep) at any Hook or synchronously. However, if this is done in **Hook #5**, then any modification will be taken into consideration after the system sleeps once and wakes up.

In the following table the various Hooks are listed together with their corresponding timing constraints.

Table 1: Description of Hooks

#	Function name	Functionality	Timing Constraints
1	`app_asynch_trm()`	Used for sending messages to kernel tasks generated from asynchronous events that have been processed in **Hook #2**.	Medium
2	`app_asynch_proc()`	Used for processing of asynchronous events at "*user*" level. The corresponding ISRs should be kept as short as possible and the remaining processing should be done at this point.	Medium
3	`app_asynch_sleep_proc()`	Used for updating the state of the application based on the latest status just before sleep checking starts.	Medium

4	app_sleep_prepare_proc()	Used to allow cancelling the entry to Extended or Deep sleep based on the current application state. The BLE and the Radio are still powered off but the other of the power domains stay active.	Hard
5	app_sleep_entry_proc()	Used for application specific tasks just before entering the low power mode.	Hard
6	app_sleep_exit_proc()	Used for application specific tasks immediately after exiting the low power mode.	Hard

As already mentioned, **Hook #1** is used for synchronizing asynchronously generated requests for transmission of messages to OS tasks (note that this is also the flow followed for sending messages over the Bluetooth link). The synchronization is done following the requirements posed by the stack. These stack requirements translate to a fixed structure of this function which is described in pseudocode below.

```
------------------------------ synchronization of asynchronous events' requests (Hook #1)
FUNCTION app_asynch_trm(void) RETURNS bool
{
    IF Condition A is true {
        DO something that results in sending a message to a kernel task

        // Force rescheduling
        RETURN true
    }

    IF Condition B is true {
        DO something that results in sending a message to a kernel task

        // Force rescheduling
        RETURN true
    }

    IF Condition C is true {
        DO something that does not involve sending a message to a kernel task

        // Do not force rescheduling
    }

    RETURN false
}
```

In the example above, the function checks three conditions and acts accordingly. Conditions A and B result, either directly or after processing, in sending a message to another task. In order for this message to be served, scheduling must be executed. Thus, this function stops any further execution and returns true. This forces the execution of scheduling. After that, this function will be called again. So, if all conditions (A, B and C) are true then the code execution will follow this path:

1. DO scheduling.
2. CALL `app_async_trm()`.
3. Condition A is true. Requested action for A is taken. A message is sent to another task. `app_async_trm()` stops returning true.
4. DO scheduling. The message is served by the receiving task. Any other messages that are generated are also served immediately.
5. CALL `app_async_trm()`.

6. Condition A is false. Condition B is true. Requested action for B is taken. A message is sent to another task. `app_async_trm()` stops returning true.
7. DO scheduling. The message is served by the receiving task. Any other messages that are generated are also served immediately.
8. CALL `app_async_trm()`.
9. Condition A is false. Condition B is false. Condition C is true. Requested action for C is taken. `app_async_trm()` stops returning false.
10. Code continues to **Hook #2**.

7 Application Programming Interface

The following API is provided to the application to modify the mode of operation (Active, Extended sleep, Deep sleep). This API is available only for single processor solutions.

Table 2: Sleep API

API Functions

`void app_disable_sleep(void);`
 	
`Disables all sleep modes. The system is either in idle or active state.`

`void app_set_extended_sleep(void);`
 	
`Activates extended sleep mode.`

`void app_set_deep_sleep(void);`
 	
`Activates deep sleep mode.`

`uint8_t app_get_sleep_mode(void);`
 	
`Returns the current mode of operation.`
`0: sleep is disabled`
`1: extended sleep`
`2: deep sleep`

`void app_force_active_mode(void)`
 	
`If sleep is on then it is disabled. The current sleep mode (before setting it to Active) is stored in order to be able to restore it, if needed.`

`void app_restore_sleep_mode(void)`
 	
`Restores the previous sleep mode (if any) that was changed with a call to app_force_active_mode(). The application must not have modified the sleep mode in the meantime.`

```
void app_ble_ext_wakeup_on(void)

      Put BLE into permanent sleep waiting a forced wakeup. After waking up from
an external event, if the system has to wake BLE up then it must restore the
default mode of operation by calling app_ble_ext_wakeup_off() or the BLE won't be
able to wake up in order to serve BLE events!
```

```
void app_ble_ext_wakeup_off(void)

      Restore BLE cores' operation to default mode. In this mode, the BLE core
will wake up every 10sec even if no BLE events are scheduled. If an event has been
scheduled earlier, then the BLE core will wake up sooner to serve it.
```

```
bool app_ble_ext_wakeup_get(void)

      Returns the current mode of operation of the BLE core:
                  false: default mode
                  true: permanent sleep, external wake-up is required.
```

```
bool app_ble_force_wakeup(void)

      If the BLE core is sleeping (permanently or not), this function wakes it up.
A call to app_ble_ext_wakeup_off() should follow in case of permanent sleep.
```

Note that the module monitors the calls to `app_force_active_mode()` (a counter is incremented at each call) and to `app_restore_sleep_mode()` (the counter is decremented). In order for `app_restore_sleep_mode()` to actually enable sleep, the counter must be zero! In other words, the application must ensure that `app_restore_sleep_mode()` is called (at least) as many times as `app_force_active_mode()`. If the application is split into different modules then this rule applies to each module separately.

Finally, note that the Debugger cannot be used in any of the sleep modes because it has to be turned off in order to allow powering down the System power domain.

7.1 Setting sleep mode via da14580_config.h

The file `da14580_config.h` includes two definitions that can be used to set the sleep mode statically. These are:

- `CFG_EXT_SLEEP`: if defined then the application is configured to use Extended sleep mode,
- `CFG_DEEP_SLEEP`: if defined and `CFG_EXT_SLEEP` is not defined, then the application is configured to use Deep sleep mode,
- if both `CFG_EXT_SLEEP` and `CFG_DEEP_SLEEP` are undefined then the application is configured to use Active mode only.

External processor solutions have no other means to define the sleep mode since the sleep API is not available in this case.

In case of single processor solutions these settings may be bypassed by the application code, if the sleep API is used since it has higher priority.

Note that these settings are also used when the chosen memory map is checked against the chosen sleep mode to verify that the application will be built correctly. Thus, even if the application chooses to bypass the sleep mode settings in `da14580_config.h` and use the API, which is strongly suggested, these settings must be correct and reflect the deepest sleep mode that the application uses.

For example, an application may use the sleep API to switch between all modes of operation (Active, Extended sleep and Deep sleep) depending on its current state. In this case, the settings in the file `da14580_config.h` should be:

```
#undef CFG_EXT_SLEEP
#define CFG_DEEP_SLEEP
```

If the application used only Active and Extended sleep modes then these settings should be:

```
#define CFG_EXT_SLEEP
#undef CFG_DEEP_SLEEP
```

8 Revision history

Revision	Date	Description
1.0	27-Mar-2014	Initial version for DA14580-01

Status definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

Disclaimer

RoHS Compliance

Dialog Semiconductor complies to European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2).
Dialog Semiconductor's statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email:
enquiry@diasemi.com

North America
Dialog Semiconductor Inc.
Phone: +1 408 845 8500

Japan
Dialog Semiconductor K. K.
Phone: +81 3 5425 4567

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Singapore
Dialog Semiconductor Singapore
Phone: +65 64 849929

China
Dialog Semiconductor China
Phone: +86 21 5178 2561

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8291