AN5469

Application note

Getting started with BlueNRG-LP/BlueNRG-LPS/STM32WB0 MCUs radio timer module

Introduction

The BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices are very low-power Bluetooth® Low Energy (BLE) single-mode systems-on-chip, compliant with Bluetooth® specifications. The architecture core is a Cortex-M0+ 32-bits.

The BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices are referred as the *devices* in this document.

This document describes the features and functionalities of the software module that manages the *devices* link controller timers. A detailed description of the different hardware timers can be found in the radio controller reference manual.

The radio timer driver (also known as radio timer module) allows the application to program an event that can be related to a wake-up of the device, a user timeout or a preconfigured radio transaction to be triggered.

Therefore, any Bluetooth® LE and radio proprietary application is based on the timer module library.

1 General information

The STM32WB0 series are Arm® Cortex® core-based microcontrollers.

For more information on Bluetooth®, refer to http://www.bluetooth.com.

Note: *Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

2 Radio timer description

The radio timer driver of the BlueNRG-LP and BlueNRG-LPS SW package (STSW-BNRGLP-DK) consists of the following files:

- rf_driver_hal_vtimer.c
- rf_driver_hal_vtimer.h
- rf_driver_ll_timer.c
- rf_driver_ll_timer.h

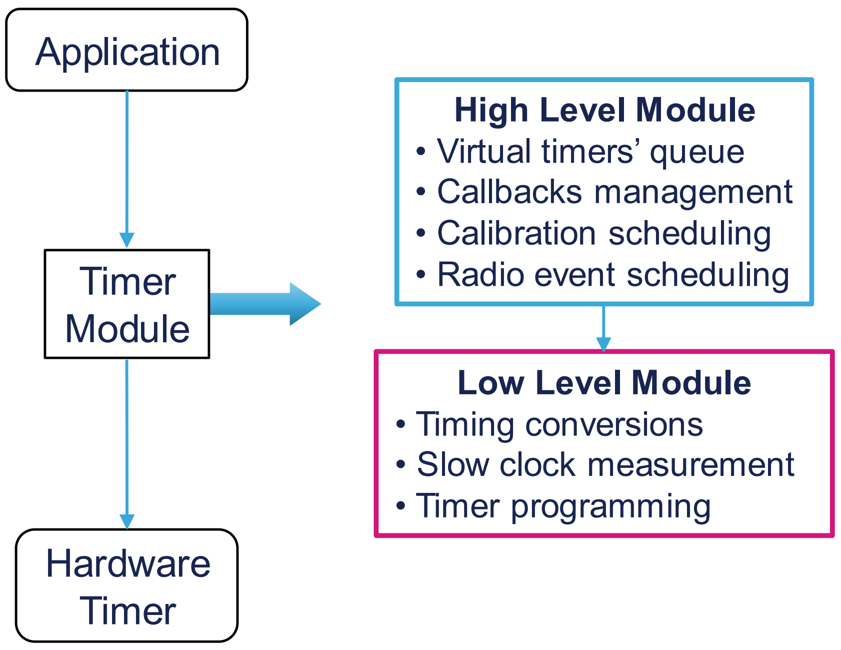
The radio timer driver of the STM32CubeWB0 SW package consists of the following files:

- stm32wb0x_hal_radio_timer.c
- stm32wb0x_hal_radio_timer.h
- stm32wb0x_ll_radio_timer.h

These files represent two layers that separate the application from the hardware.

The first layer abstracts the hardware timers through software structures that allow virtualizing the resources available on the device. The second layer is directly connected to the hardware and converts time expressed in hardware independent units to hardware dependent units and vice versa. These conversions take into account the rate at which the hardware timers are counting.

Figure 1. Timer module components



3 Virtual timer

The *devices* link controller provide a radio timer counter which is used to wake up the device during low-power mode phases. The timer counter can not trigger radio operations.

The radio timer module exploits the hardware resources of this single timer enabling the allocation of many virtual timers.

The only constraint on the number of virtual timers is the memory available on the device.

A virtual timer acts just like a normal timer. For example, the user can program a virtual timer to execute some actions at a certain time.

From the point of view of the application, the virtual timer is a software timer structure that contains the pointers to some user data, callback, and the expiration time.

The software abstraction enables the hardware radio timer to share its capabilities to the virtual timers of the application.

When a virtual timer is started, its instance is placed in a queue ordered by the expiration time. If it elapses before the other events in the queue, the virtual timer is placed at the top and the hardware timer is programmed. Otherwise, it takes place between the other already started timers when its turn comes.

After the virtual timer expiration, the internal state machine is in charge of executing the callback linked to the just expired virtual timer and reserves the hardware counter for the next timer in the queue.

The timeout of a virtual timer is treated as an absolute time. It means that it occurs like an event on the calendar at a specific time.

3.1 Virtual time base

Inside the radio timer module, the time is measured according to a special unit called system time unit (STU). It is independent to the hardware oscillator variations and it is directly exposed to the user. Every timeout event is expressed in STUs. One STU is equal to 625/256 µs (about 2.4414 µs). This unit makes the timings dictated by the Bluetooth protocol easier to read. Only before programming the real counter, the time expressed in STUs is converted in the hardware timer counting unit.

The time in STUs is accumulated on a global variable that is 64-bits long. If a digital watch wraps every 24 hours, then the radio timer module time base needs more than one million years to wrap.

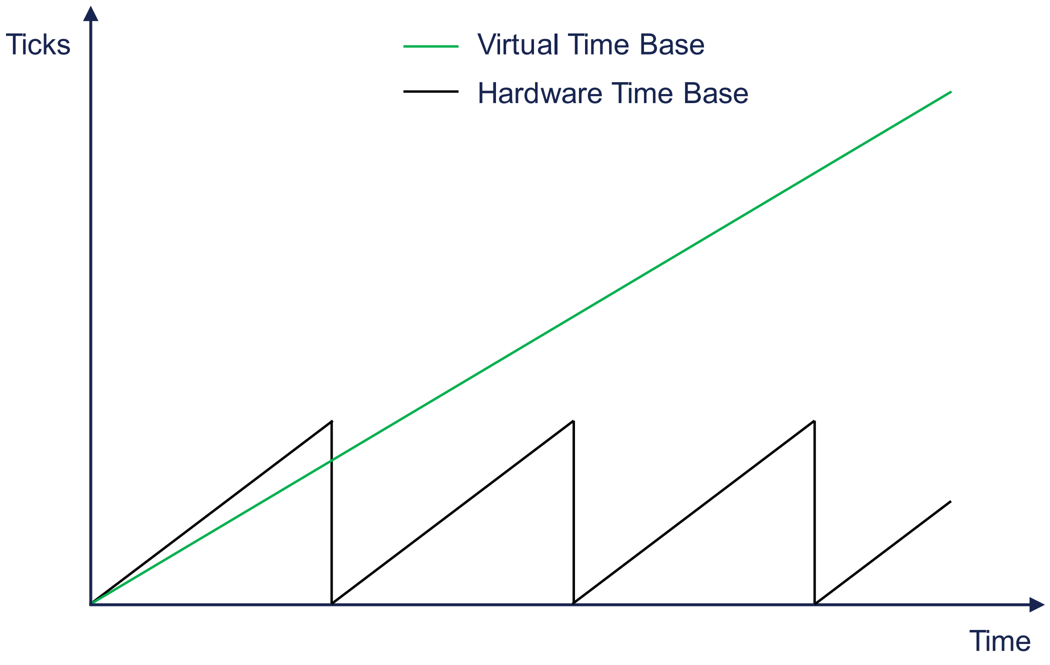
A never wrapping time base is very useful since it always knows if an event is before or after another one or whether it is in the past or not.

However, this time base must deal with the finite length of the hardware timer and for this reason it is called virtual as well. In order to accumulate the time correctly, the virtual time base variable must be updated at least one time before every hardware timer wrapping.

This important mechanism is internal and is not in charge of the user. The module automatically arms a virtual timer that is dedicated to this task during the initialization phase.

This special virtual timer is periodically programmed with the maximum possible value admitted by the hardware capacity of the timer. It means that if the device is in low-power mode, it wakes up periodically to execute the time base maintenance. The *devices* wake up about every 138 minutes.

Figure 2. **Virtual and hardware time bases**



4 Low speed oscillator and calibration procedure

In addition to the timer shared by the virtual timers, another timer is able to trigger a radio activity. The timing of the radio transactions is a critical aspect in most cases and a certain accuracy must be guaranteed.

The low speed oscillator feeds the *devices* link controller timers. According to the configuration, the low speed oscillator source can be the external XO or the internal RO. Unlike the external one, the speed of the internal oscillator can change depending on the temperature. This behavior implies that, if clocked by the internal oscillator, the timers could count according to a period that is not fixed.

Then, for example, the same timeout can actually correspond to different time intervals depending on the frequency of the clock. If the internal oscillator is adopted, in order to guarantee the accuracy of the programmed timeouts, the variations in frequency are periodically taken into account. In this case, the same virtual timer used to maintain the virtual time base is also in charge of starting, at every calibration interval, the calibration procedure that consists of measuring a certain number of periods of the internal oscillator exploiting a stable clock source. Once the frequency is measured, the next timeouts that are always expressed in STUs are converted with a better accuracy in the hardware counter unit called machine time unit (MTU).

Essentially, the measurement of the frequency of the low speed oscillator is used as a conversion factor that allows time expressed in STUs to be translated in a time expressed in an MTU and vice versa.

After the initialization phase and that the user has defined the calibration interval, the firmware as the time base maintenance mechanism manage the low-speed oscillator frequency measurement.

The nominal frequency of the low speed oscillator is 32.768 kHz. If the *devices* are equipped with an external crystal oscillator running to the nominal frequency, no calibration procedure is necessary. The *devices* can also adopt an external oscillator running to different speeds. In this last case, a first calibration only is necessary in order to assess the frequency of the oscillator.

In any case, as much as possible, the user does not need to deal with the counting unit of the hardware timer and with the frequency of the oscillator but only with time expressed in STUs.

4.1 Calibration interval

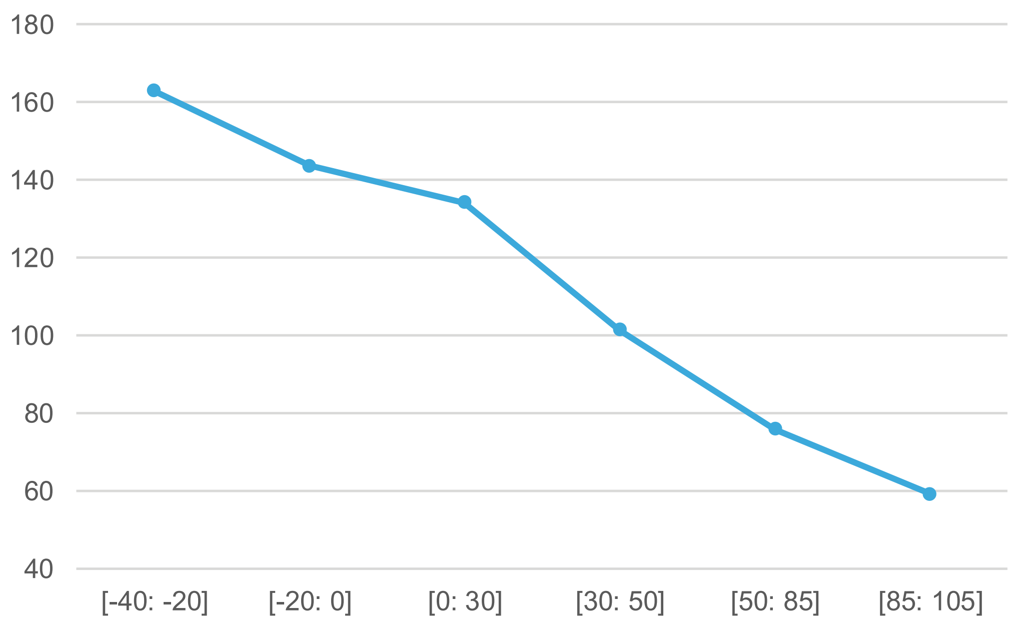
The calibration interval is a parameter that can be set during the initialization phase to decide how often the device has to perform the measurement of the frequency of the internal oscillator.

As previously mentioned, if an external crystal oscillator is adopted, the user can neglect this parameter and set it to zero.

The main reason to measure periodically the frequency of the internal oscillator is to compensate the variation of the frequency due to temperature changes and to guarantee a certain accuracy on the timeouts, in particular during radio operations.

After the initialization phase and that the user has defined the calibration interval, the firmware as the time base maintenance mechanism manage the low speed oscillator frequency measurement. The frequency temperature sensitivity follows the trend below in the operating range [-40 °C: 105 °C].

Figure 3. **Frequency temperature sensitivity (ppm/°C)**



The frequency temperature sensitivity is the error in frequency due to a variation of one degree. Therefore, the determination of the temperature evolution and the maximum error admitted is necessary to compute how often the internal oscillator must be measured.

Suppose that the temperature changes at a rate of 0.1 °C/s and the frequency temperature sensitivity is 160 ppm/°C. In order to guarantee an error below 500 ppm, the frequency of the internal oscillator must be measured at least every 31 seconds.

Once the calibration interval has been set, the firmware autonomously schedules a calibration procedure at each calibration interval. The calibration procedure lasts about 800 µs.

If the device is already in the active state, the calibration procedure is started in advance and the next calibration event is programmed consequently.

5 Radio timer module examples

In the following sections, some typical use cases are described.

Three APIs are always called inside all those applications that exploit the BlueNRG-LP, BlueNRG-LPS radio timer module in the context of the STSW-BNRGLP-DK SW package.

`HAL_VTIMER_Init()`. It initializes the radio timer module according to the kind of low speed oscillator and the high speed clock startup time. Furthermore, it starts the virtual timer that is in charge of triggering the calibration procedure and the virtual time base maintenance operations.

This information is contained inside a dedicated structure defined as follows:

```
typedef struct HAL_VTIMER_InitS {
    /* XTAL startup in 2.44 us unit */
    uint16_t XTAL_StartupTime;
    /* Enable initial estimation of the frequency
    of the Low Speed Oscillator */
    BOOL EnableInitialCalibration;
    /* Periodic calibration interval in ms,
    to disable set to 0 */
    uint32_t PeriodicCalibrationInterval;
    } HAL_VTIMER_InitType;
```

The `XTAL_StartupTime` is the time that the high speed clock needs to be stable. This value is expressed in system time units and it is especially useful for the timing of radio operations.

The flag `EnableInitialCalibration` allow estimating the frequency of the low speed oscillator during the initialization. Generally, if the external crystal oscillator is adopted, the initial estimation can be disabled by putting this flag to zero.

The `PeriodicCalibrationInterval` is expressed in milliseconds and it represents how often the low speed oscillator must be measured according to the temperature variations. Also in this case, if the external crystal oscillator is adopted, choosing a calibration interval equal to zero disables the periodic calibration.

`HAL_VTIMER_Tick()`. It is called inside the application main loop. It manages the virtual timer queue, checks if a virtual timer is expired, manages the sharing mechanism of the hardware timer resources and the execution of the user callbacks of the expired timers.

If `PeriodicCalibrationInterval` is different than zero, it periodically starts the calibration procedure when the calibration timer expires. If the calibration timer is not yet expired but the device is in the active state, the calibration procedure can be started in advance.

`HAL_VTIMER_TimeoutCallback()`. It is called inside the dedicated timer IRQ handler. It is executed when the hardware timer expires, and signals it to the application. ***Note that it is not a user defined callback.***

```
void CPU_WKUP_IRQHandler(void)
{
   HAL_VTIMER_TimeoutCallback();
}
```

Note: *In the context of the STM32CubeWB0 SW package, the equivalent APIs are the following:*

```
void HAL_RADIO_TIMER_Init(RADIO_TIMER_InitTypeDef *RADIO_TIMER_InitStruct);
void HAL_RADIO_TIMER_Tick(void);
void HAL_RADIO_TIMER_TimeoutCallback(void);
```

5.1 Starting and stopping a virtual timer

In the context of the BlueNRG-LP, and BlueNRG-LPS STSW-BNRGLP-DK SW package, a virtual timer is a structure defined as follows:

```
typedef struct VTIMER_HandleTypeS {
 uint64_t expiryTime; /* Absolute timeout expressed in STU */
 VTIMER_CallbackType callback; /* User callback */
 BOOL active; /* Managed by the internal tick */
 struct VTIMER_HandleTypeS *next; /* Managed by the internal tick */
 void *userData; /* Pointer to user data */
 }
```

Once a `VTIMER_HandleType` is declared inside the application, the user can define a callback and pass the desired timeout when the timer is started. If needed, the user can also define some data that are carried on by the timer handle.

In the context of the BlueNRG-LP, and BlueNRG-LPS STSW-BNRGLP-DK SW package, follow down below an applicative example that shows how to start a virtual timer. The expiration time is expressed as a relative time interval in milliseconds using the dedicated API `HAL_VTIMER_StartTimerMs()`..

```
void callback(void *handle)
{
  printf("Timer Callback after one second! \r\n");
}

int main(void)
{
  uint32_t delay = 1000; /* One second delay */
  HAL_VTIMER_InitType VTIMER_InitStruct = {HS_STARTUP_TIME, INITIAL_CALIBRATION,CALIBRATION_INTERVAL};
  /* System initialization function */
  if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M) != SUCCESS) {
    /* Error during system clock configuration take appropriate action */
    while(1);
  }
  HAL_VTIMER_Init(&VTIMER_InitStruct);
  timerHandle.callback = callback;
  HAL_VTIMER_StartTimerMs(&timerHandle, delay);
  while(1) {
    HAL_VTIMER_Tick();
  }
}
```

When the timer finally expires, the callback is triggered inside the `HAL_VTIMER_Tick()`.

If a timer is already started, it cannot be started again. In this case, the API returns an error code and the virtual timer is not inserted in the queue.

If the delay is too small, the timer is considered as already expired and the related callback is executed.

The timeout can be also expressed as an absolute time in STUs. In this case, `HAL_VTIMER_StartTimerSysTime()` is used.

```
static VTIMER_HandleType timerHandle;
void callback(void *handle)
{
  printf("Timer Callback after one second! \r\n");
}

int main(void)
{
  uint32_t delay = 409600; /* number of system units in one second */
  HAL_VTIMER_InitType VTIMER_InitStruct = {HS_STARTUP_TIME, INITIAL_CALIBRATION, CALIBRATION_INTERVAL};
  /* System initialization function */
  if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M) != SUCCESS) {
    /* Error during system clock configuration take appropriate action */
    while(1);
  }
  HAL_VTIMER_Init(&VTIMER_InitStruct);
  timerHandle.callback = callback;
  HAL_VITMER_StartTimerSysTime(&timerHandle, TIMER_GetCurrentSysTime() + delay);
  while(1) {
    HAL_VTIMER_Tick();
  }
}
```

The current time expressed in STUs is gotten and the number of system time units corresponding to one second is added. Therefore, the expiration time becomes an absolute time expressed in STUs.

Once a virtual timer has started, it is possible to call `HAL_VTIMER_StopTimer()` to stop a virtual time that has started before its expiration.

```
HAL_VTIMER_StopTimer(&timerHandle);
```

Note: In the context of the STM32CubeWB0 SW package, an equivalent code is also applicable by referring to the related HAL radio driver stm32wb0x_hal_radio_timer.[ch] APIs:

```
uint32_t HAL_RADIO_TIMER_StartVirtualTimer(VTIMER_HandleType *timerHandle, uint32_t msRelTim
eout);

uint32_t HAL_RADIO_TIMER_StartVirtualTimerSysTime(VTIMER_HandleType *timerHandle, uint64_t t
ime);

void HAL_RADIO_TIMER_StopVirtualTimer(VTIMER_HandleType *timerHandle);

void HAL_RADIO_TIMER_Tick(void);
```

6 Radio timer

The *devices* provide another timer that is dedicated to trigger a radio transaction that can be a transmission or a reception. In particular, the timer module library offers the possibility to program two different events related to a radio operation:

- The first transmitted bit over-the-air.
- The beginning of the receive window.

Note: *The timer module programs the radio timer only and it does not configure the radio for transmissions or receptions. Moreover, the timer module does not program the timeout of back-to-back communications. A dedicated software library accomplishes both tasks. Refer to the radio driver user manual for more details. The radio timer is not virtualized in queue of timers (it is not needed) as with the virtual timers, but even in this case the user has to express the expiration time of one of the two events described before in an STU as an absolute time in the future. In addition, the radio timer is abstracted in a software structure but it does not need any action from the user. If the timeout is too close, the request to program the timer is rejected and an error code is returned.*

6.1 Calibration and radio timer

As previously mentioned, the calibration procedure is necessary only if the low speed internal oscillator is the clock of the *devices* link layer timers. If this is the case, the firmware ensures that the radio timer for the next radio transaction is programmed to exploit the latest low speed frequency measurement improving the accuracy of the timeout.

In other words, if the next radio transaction occurs after the next calibration event, the timer is not immediately programmed but remains pending until the new frequency measurement is available. On the contrary, if the next calibration event occurs after the next radio transaction, the timer is programmed when requested.

Since the calibration values are available in thread mode inside the `HAL_VTIMER_Tick()`, after the low speed oscillator measurement is over, a margin is set in order to give to the module enough time to finalize the pending timer with the new values. If this margin is not respected, a radio event that was previously set in the future could be shifted in the past and therefore not programmed.

Note: *In the context of the STM32CubeWB0 SW package, the equivalent APIs are the following:*

```
void HAL_RADIO_TIMER_Tick(void)
```

6.2 Radio timer programming example

A radio timer can only be programmed after a radio transaction has been previously configured. Then, the radio timer programming APIs can be considered as the last step of the configuration of a radio transaction. Therefore, some specific information is needed to program the radio timer properly:

- The timeout expressed in STU
- The type of the transaction (transmission or reception)
- The PLL calibration for the channel frequency

According to the previous parameters, in order to respect the desired timeout, the timer module compensates the time that the radio needs for its configuration. The different RF set-up times are initialized in some specific structures in RAM during the radio initialization. More details about radio RAM structures are present in the radio controller reference manual.

Therefore, supposing that the radio has been initialized and the transaction has been configured, the radio timer can be programmed through `HAL_VTIMER_SetRadioTimerValue()` in the context of the BlueNRG-LP, and BlueNRG-LPS SW package.

Note: *Note that the timer module initialization occurs as shown in the Section 3: Virtual timer and after the radio initialization.*

The timeout is expressed as an absolute time. For example, a transmission can be programmed to trigger after one second as follows:

```
uint8_t event_type = HAL_VTIMER_TX_EVENT;
uint8_t cal_req = HAL_VTIMER_PLL_CALIB_REQ;
uint32_t timeout = TIMER_GetCurrentSysTime() + 409600;
retVal = HAL_VTIMER_SetRadioTimerValue(timeout,event_type,cal_req);
```

The API returns an error code if the timeout passed is too close. The radio timer can be stopped before its triggering. However, if the timeout is too close when the timer is stopped, it may not be properly cleared. In other terms, the firmware always clears the timer but it could be already triggered and then can no longer be stopped. A dedicated API can be used to stop the radio timer. The same API returns a different value if the timer has been cleared successfully or not.

```
/**
* @brief Clear the last radio activity scheduled disabling the radio timer too.
Furthermore, it returns different values if the timeout is too close
and possibly the radio activity cannot be cleared in time.
@return 0 if the radio activity has been cleared successfully.
@return 1 if it is too late to clear the last radio activity.
@return 2 if it could be not possible to clear the last radio activity.
*/
uint8_t HAL_VTIMER_ClearRadioTimerValue(void);
```

Note: *In the context of the STM32CubeWB0 SW package, an equivalent code is also applicable by referring to the related HAL radio driver stm32wb0x_hal_radio_timer.[ch] APIs:*

```
uint64_t HAL_RADIO_TIMER_GetCurrentSysTime(void);

uint32_t HAL_RADIO_TIMER_SetRadioTimerValue(uint32_t time, uint8_t event_type, uint8_t cal_r
eq);

uint32_t HAL_RADIO_TIMER_ClearRadioTimerValue(void);
```

7 Sleep management

The timer module prevents the device from going to sleep in different conditions:

- A virtual timer triggered but its related callback has not yet been executed.
- A low speed clock measurement is ongoing.
- The next radio transaction is very close.
- The device is in a back-to-back communication.

Note: The timer module autonomously starts the internal virtual timer to perform the calibration procedure end/or the time base maintenance. If the low-power mode with no timer is requested at application level, if there is not a radio timer scheduled and a virtual timer programmed, the timer module also disables the internal virtual timer. In this case, if configured properly, the device is able to wake up only through external sources.

Revision history

Table 1. Document revision history

Date	Version	Changes
13-Jul-2020	1	Initial release.
06-Apr-2022	2	Updated Section Introduction and Section 2: Radio timer description. Added the BlueNRG-LPS references throughout the document.
20-Jun-2024	3	Added the STM32WB0 series reference throughout the document.

Contents