AN13593

Using FreeRTOS tickless mode on LPC5500

Rev. 0 — 21 March 2022 Application Note

Contents

1 Introduction

When developing the MCU in bare metal, users can directly configure the relevant functions to make the MCU enter the low-power mode. The low-power mode is also required for the MCU running the real-time operating system (RTOS). Tickless mechanism is a general low-power method adopted by current small RTOSs, such as FreeRTOS, EmbedOS, RTX. This application note describes how to implement the FreeRTOS tickless mode in NXP LPC5500 series MCUs.

2 FreeRTOS tickless principle

The principle of the FreeRTOS tickless mode is to make the MCU enter the low-power mode to save system power consumption when the MCU is performing the idle task. Calling the Wait For Interrupt (WFI) instruction to make the MCU enter the low-power mode and using interrupts to wake up the MCU.

Therefore, the core idea of the FreeRTOS tickless mode is:

- When performing idle tasks, make the MCU enter the low-power mode
- Under appropriate conditions, wake up the MCU through interrupts or external events.

Since FreeRTOS uses the Systick timer to generate system ticks and generate interrupts, this interrupt will also wake up the MCU, so the system tick interrupt must be disabled before entering the low-power mode. Once the system tick is stopped, after exiting the low-power mode, the system tick counter will lose some count values, which is not allowed. For this situation, FreeRTOS also provides a corresponding solution, which is to use a timer to record the time the system is in the low-power mode, and compensate this time to the system tick counter after exiting the low-power mode.

The key steps to implement the tickless mode include:

- Calculate the execution time of the current idle task
- Configure system parameters and enter the low-power mode
- Exit the low-power mode and compensate the system tick counter

The `vPortSuppressTicksAndSleep()` function is defined in FreeRTOS and it can be used to implement the tickless mode. Users can set the `configUSE_TICKLESS_IDLE` macro to 1 to implement the tickless mode. The following explains these three steps with the source code of the `vPortSuppressTicksAndSleep()` function.

2.1 Calculate the duration of the low-power mode

When the tickless mode is enabled (`configUSE_TICKLESS_IDLE !=0`), once the system is idle, the execution time of the current idle task is calculated in the idle task.

```
xExpectedIdleTime = prvGetExpectedIdleTime();
```

This time is also the maximum duration of the low-power mode, because the low-power mode may also be awakened by random external events, such as external key interrupts.

The duration of the low-power mode also has an upper limit. It is because after entering the low-power mode, the user must use a timer to record the time into the low-power mode. The count value of the timer is limited, which determines the maximum length of time the MCU can go into the low-power mode. The timer can be Systick or one of other low-power timers. For example, the system clock frequency is 48 M, the tick frequency is 1 K, and the user uses Systick's 24-bit counter to record the duration of the low-power state, the maximum system tick `xMaximumPossibleSuppressedTicks` that Systick can record is calculated as follows：

```
ulTimerCountsForOneTick = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ );
xMaximumPossibleSuppressedTicks = portMAX_24_BIT_NUMBER / ulTimerCountsForOneTick;
                                = 0XFFFFFF/ (48000000/1000) = 349;
```

If the duration of the idle task xExpectedIdleTime exceeds `xMaximumPossibleSuppressedTicks`, the maximum time of the low-power mode should be `xMaximumPossibleSuppressedTicks`.

Then calculate the initial value of the low-power timer according to the valid `xExpectedIdleTime`.

```
ulReloadValue = portNVIC_SYSTICK_CURRENT_VALUE_REG + ( ulTimerCountsForOneTick * ( xExpectedIdleTime
- 1UL ) );
```

```
/* Make sure the SysTick reload value does not overflow the counter. */
if( xExpectedIdleTime > xMaximumPossibleSuppressedTicks )
{
    xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
}

/* Stop the SysTick momentarily. The time the SysTick is stopped for is
 * accounted for as best it can be, but using the tickless mode will
 * inevitably result in some tiny drift of the time maintained by the
 * kernel with respect to calendar time. */
portNVIC_SYSTICK_CTRL_REG &= ~portNVIC_SYSTICK_ENABLE_BIT;

/* Calculate the reload value required to wait xExpectedIdleTime
 * tick periods. -1 is used because this code will execute part way
 * through one of the tick periods. */
ulReloadValue = portNVIC_SYSTICK_CURRENT_VALUE_REG + ( ulTimerCountsForOneTick * ( xExpectedIdleTime - 1UL ) );

if( ulReloadValue > ulStoppedTimerCompensation )
{
    ulReloadValue -= ulStoppedTimerCompensation;
}
```

Figure 1. Calculate the reload value of the low-power timer

The `ulReloadValue` value is used to configure the low-power timer. When the counter value is reduced to 0, the MCU is awakened.

2.2 Enter low-power mode

Before entering the low-power mode, some preparations must be done, such as closing the SysTick tick interrupt, configuring the wake-up source.

From the description in Chapter 2.1, after closing the tick interrupt of Systick, the user must use a low-power timer, such as Systick or RTC, to record the duration of the low-power state and wake up the MCU.

In the FreeRTOS source code, SysTick is used as a wake-up source of the MCU, and the reload value of SysTick is configured as the `ulReloadValue` value calculated in Chapter 2.1. When the count value of SysTick decreases to 0, the MCU is awakened.

```
/* Set the new reload value. */
portNVIC_SYSTICK_LOAD_REG = ulReloadValue;
```

```
/* Clear the SysTick count flag and set the count value back to zero. */
portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;

/* Restart SysTick. */
portNVIC_SYSTICK_CTRL_REG |= portNVIC_SYSTICK_ENABLE_BIT;
```

If the user wants to use an external interrupt to wake up the MCU, it is also necessary to configure the external interrupt wake-up source before entering the low-power state. After the MCU is awakened by an external interrupt, the duration of the low-power state can be obtained from the initial value and current value of the Systick counter. There is no external interrupt configured as a wake-up source in the source code of FreeRTOS.

After the wake-up source is configured, the MCU can be put into sleep mode. Use the WFI instruction to make the MCU enter the sleep mode. This sleep mode just turns off the core clock and does not affect the work of the peripherals.

```
configPRE_SLEEP_PROCESSING( xModifiableIdleTime );

if( xModifiableIdleTime > 0 )
{
        __asm volatile ( "dsb" ::: "memory" );
        __asm volatile ( "wfi" );
        __asm volatile ( "isb" );
}

configPOST_SLEEP_PROCESSING( xExpectedIdleTime );
```

In addition to the `vPortSuppressTicksAndSleep()` function, FreeRTOS provides users with two other interface functions:

- `configPRE_SLEEP_PROCESSING(xModifiableIdleTime);`
- `configPOST_SLEEP_PROCESSING(xExpectedIdleTime);`

FreeRTOS only provides the function interface, and the user defines the function entity. Before the user can make the MCU enter the low-power mode, `configPRE_SLEEP_PROCESSING()` must be called to configure the system parameters to reduce the system power consumption, such as turning off other peripheral clocks, reducing the system frequency. After exiting the low-power mode, call the `configPOST_SLEEP_PROCESSING()` function to restore the system's main frequency and peripheral functions.

2.3 Exit low-power mode

When the MCU is awakened by a timer interrupt or an external interrupt, the duration in the low-power state must be compensated to the system tick counter. The `vPortSuppressTicksAndSleep()` function also provides the calculation of compensation time, as shown in Figure 2.

```
    * SysTick must have brought the system out of sleep mode). */
    if( ( portNVIC_SYSTICK_CTRL_REG & portNVIC_SYSTICK_COUNT_FLAG_BIT ) != 0 )
    {
        uint32_t ulCalculatedLoadValue;

        /* The tick interrupt is already pending, and the SysTick count
         * reloaded with ulReloadValue.  Reset the
         * portNVIC_SYSTICK_LOAD_REG with whatever remains of this tick
         * period. */
        ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL ) - ( ulReloadValue - portNVIC_SYSTICK_CURRENT_VALUE_REG );

        /* Don't allow a tiny value, or values that have somehow
         * underflowed because the post sleep hook did something
         * that took too long. */
        if( ( ulCalculatedLoadValue < ulStoppedTimerCompensation ) || ( ulCalculatedLoadValue > ulTimerCountsForOneTick ) )
        {
            ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL );
        }

        portNVIC_SYSTICK_LOAD_REG = ulCalculatedLoadValue;

        /* As the pending tick will be processed as soon as this
         * function exits, the tick value maintained by the tick is
         * stepped forward by one less than the time spent waiting. */
        ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
    }
    else
    {
        /* Something other than the tick interrupt ended the sleep.
         * Work out how long the sleep lasted rounded to complete tick
         * periods (not the ulReload value which accounted for part
         * ticks). */
        ulCompletedSysTickDecrements = ( xExpectedIdleTime * ulTimerCountsForOneTick ) - portNVIC_SYSTICK_CURRENT_VALUE_REG;

        /* How many complete tick periods passed while the processor
         * was waiting? */
        ulCompleteTickPeriods = ulCompletedSysTickDecrements / ulTimerCountsForOneTick;

        /* The reload value is set to whatever fraction of a single tick
         * period remains. */
        portNVIC_SYSTICK_LOAD_REG = ( ( ulCompleteTickPeriods + 1UL ) * ulTimerCountsForOneTick ) - ulCompletedSysTickDecrements;
    }
```

Figure 2. Calculate the compensation time for SysTick

When calculating the compensation time, it is necessary to first determine whether the MCU is awakened by a low-power timer or by an external interrupt.

- If it is awakened by SysTick, the compensation time is `xExpectedIdleTime-1UL`.
- If it is awakened by an external interrupt, it means that the value of the SysTick counter has not been reduced to 0. At this moment, it is necessary to calculate the compensation time based on the initial value and current value of the SysTick counter.

Then call the `vTaskStepTick()` function to compensate the calculated value `ulCompleteTickPeriods` to `xTickCount`.

```
vTaskStepTick( ulCompleteTickPeriods );

void vTaskStepTick( const TickType_t xTicksToJump )
{
     /* Correct the tick count value after a period during which the tick
      * was suppressed.  Note this does *not* call the tick hook function for
      * each stepped tick. */
     configASSERT( ( xTickCount + xTicksToJump ) <= xNextTaskUnblockTime );
     xTickCount += xTicksToJump;
     traceINCREASE_TICK_COUNT( xTicksToJump );
}
```

In addition to using SysTick to wake up the MCU, users can also use other low-power timers, such as RTC. Users can also set the value of the `configUSE_TICKLESS_IDLE` macro to 2, so that they can define the `vPortSuppressTicksAndSleep()` function by themselves to configure different wake-up sources.

3 Implement FreeRTOS tickless mode on LPC5500

In Chapter 2, the principle of the FreeRTOS tickless mode is explained based on the source code of FreeRTOS. This chapter describes how to implement the tickless mode on the LPC5500 platform. NXP SDK provides a freertos_tickless example for LPC5500 series MCUs. This application note uses freertos_tickless in LPC55S16 SDK as an example to describe some details of implementing the tickless mode on the LPC55S16 platform. In the LPC55S16 freertos_tickless example, the value of the `configUSE_TICKLESS_IDLE` macro is 2, and the user-defined `vPortSuppressTicksAndSleep` function is used. After entering the low-power mode, the RTC clock is used to record the duration when the system enters the low-power mode and is used to wake up the MCU, as shown in Figure 4.

freertos_tickless.c | FreeRTOSConfig.h

```
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION                    1
#define configUSE_TICKLESS_IDLE                 2
#define configCPU_CLOCK_HZ                      (SystemCoreClock)
#define configTICK_RATE_HZ                      ((TickType_t)1000)
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                ((unsigned short)180)
#define configMAX_TASK_NAME_LEN                 20
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       1
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               8
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configOVERRIDE_DEFAULT_TICK_CONFIGURATION 1
```

Figure 3. Set configUSE_TICKLESS_IDLE macro to 2

```
    else
    {
        /* Set the new reload value. */
        RTC_SetWakeupCount(pxRtcBase, ulReloadValue);

        /* Enable RTC. */
        RTC_StartTimer(pxRtcBase);

        /* Sleep until something happens.  configPRE_SLEEP_PROCESSING() can
        set its parameter to 0 to indicate that its implementation contains
        its own wait for interrupt or wait for event instruction, and so wfi
        should not be executed again.  However, the original expected idle
        time variable must remain unmodified, so a copy is taken. */
        xModifiableIdleTime = xExpectedIdleTime;
        configPRE_SLEEP_PROCESSING(xModifiableIdleTime);
        if (xModifiableIdleTime > 0)
        {
            __DSB();
            __WFI();
            __ISB();
        }
        configPOST_SLEEP_PROCESSING(xExpectedIdleTime);

        ulLPTimerInterruptFired = false;

        /* Re-enable interrupts - see comments above __disable_irq()
        call above. */
        __enable_irq();
        __NOP();
```

Figure 4. Configure RTC as the wake-up source

Before FreeRTOS starts scheduling, an external pin interrupt is also defined to wake up the interrupt, as shown below.

```
/* Initialize PINT */
PINT_Init(PINT);
/* Setup Pin Interrupt 0 for falling edge */
PINT_PinInterruptConfig(PINT, kPINT_PinInt0, kPINT_PinIntEnableFallEdge, pint_intr_callback);
NVIC_SetPriority(BOARD_SW_IRQ, SW_NVIC_PRIO);
EnableIRQ(BOARD_SW_IRQ);
```

Users can press the SW3 button on the LPC55S16-EVK to wake up the MCU.

In this example, two user tasks are defined, tickless task and switch task, as shown below.

```
/*Create tickless task*/
if (xTaskCreate(Tickless_task, "Tickless_task", configMINIMAL_STACK_SIZE + 100, NULL,
tickless_task_PRIORITY, NULL) != pdPASS)
{
    PRINTF("Task creation failed!.\r\n");
    while (1)
    ;
}
if (xTaskCreate(SW_task, "Switch_task", configMINIMAL_STACK_SIZE + 100, NULL, SW_task_PRIORITY,
NULL) != pdPASS)
{
```

```
    PRINTF("Task creation failed!.\r\n");
    while (1)
    ;
}

/* Tickless Task */
static void Tickless_task(void *pvParameters)
{
    for (;;)
    {
        PRINTF("%d\r\n", xTaskGetTickCount());
        vTaskDelay(TIME_DELAY_SLEEP);
    }
}

/* Switch Task */
static void SW_task(void *pvParameters)
{
    xSWSemaphore = xSemaphoreCreateBinary();
    /* Enable callbacks for PINT */
    PINT_EnableCallback(PINT);
    for (;;)
    {
        if (xSemaphoreTake(xSWSemaphore, portMAX_DELAY) == pdTRUE)
        {
            PRINTF("CPU woken up by external interrupt\r\n");
        }
    }
}
```

The tickless task prints the current tick value once, and then calls the `vTaskDelay()` function to delay 5 s. At this time, the idle task is called, and `vPortSuppressTicksAndSleep()` will be called in the idle task to make RTC start a 5 s timer and call WFI instruction that puts the MCU into sleep mode.

When the RTC counter decreases to 0 or the SW3 button is pressed, the MCU is awakened and exits the sleep mode. The user must calculate the compensation time based on the RTC status and the value of the counter, and then compensate the calculated value of `ulCompleteTickPeriods` to xTickCount. If the 5 s delay of the tickless task has been executed, print xTickCount once, and then continue to enter the low-power mode waiting for the next wake-up.

4 Test

4.1 Run freertos_tickless demo

Compile the freertos_tickless project in the SDK package and download the program to the LPC55S16-EVK board, and measure the current consumption on the MCU_VBAT pin, as shown in Figure 6

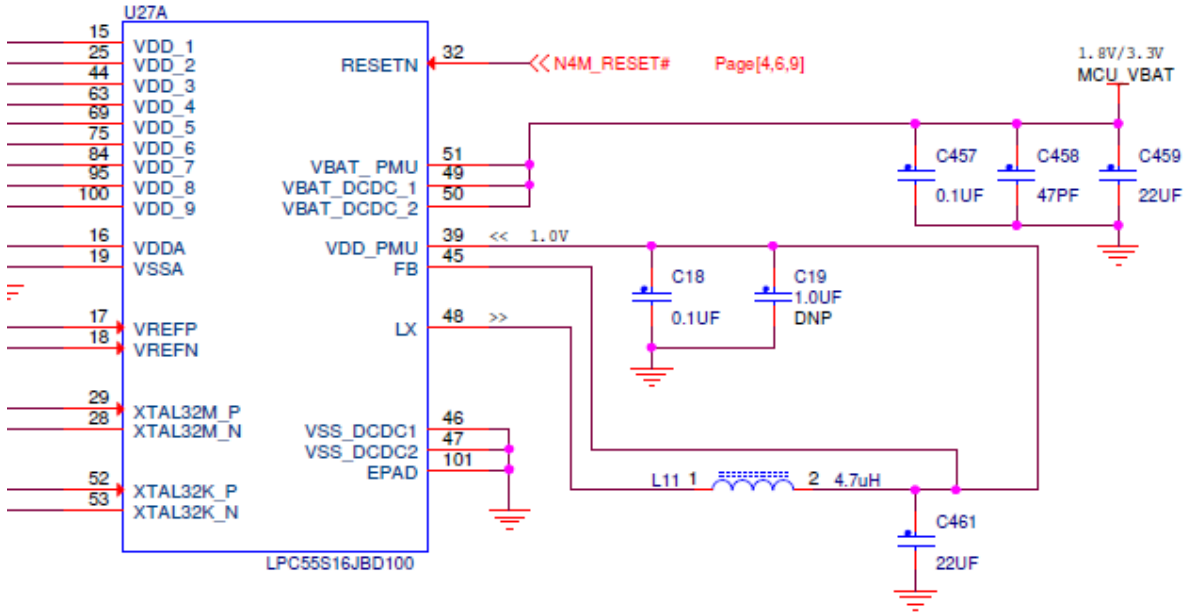

Figure 5. LPC55S16-EVK schematic

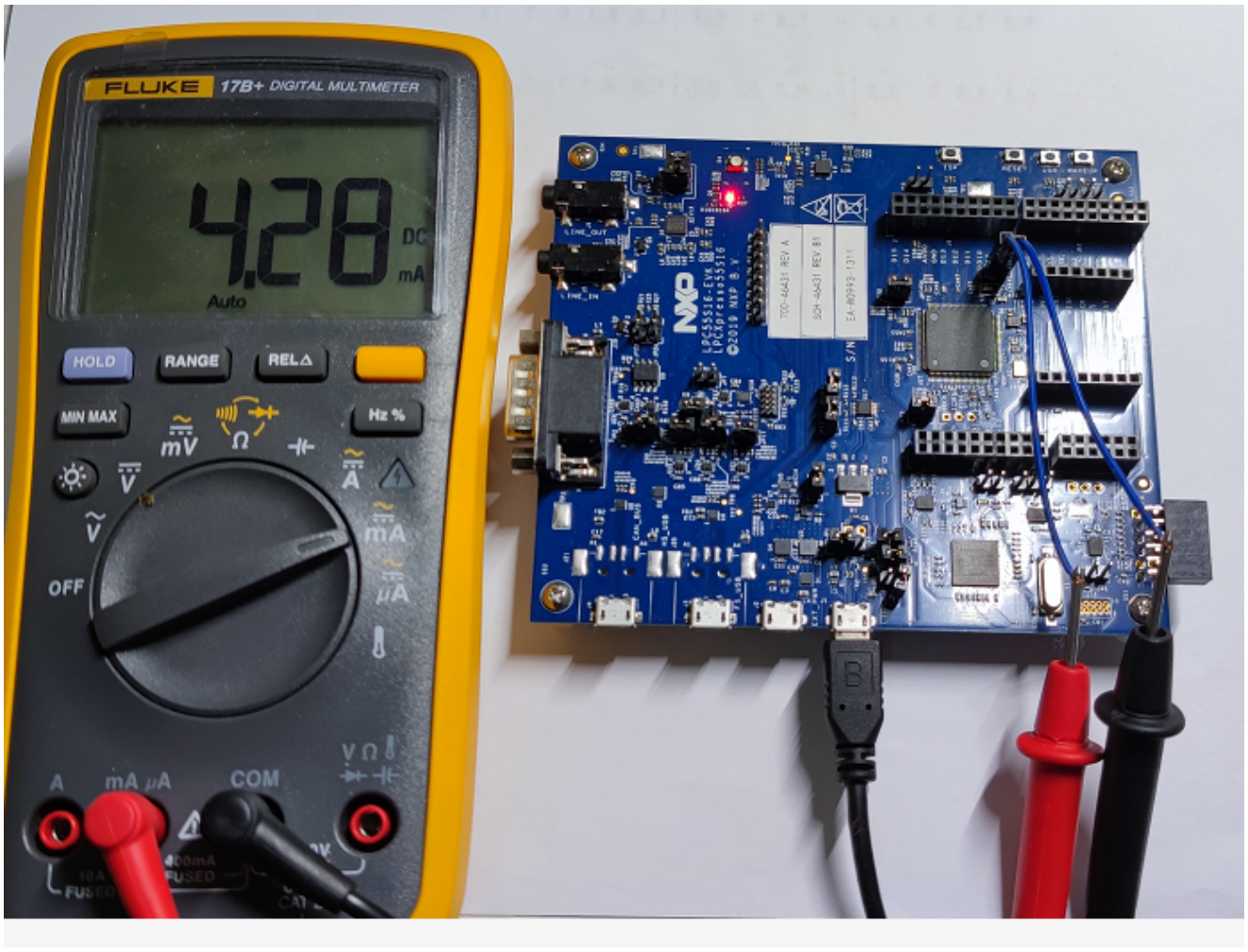

Figure 6. Current consumption of MCU_VBAT pin when the tickles mode is enabled

When the LPC55S16 enters the low-power mode, the current consumption is 4.28 mA.

Modify the value of the `configUSE_TICKLESS_IDLE` macro to 0, that is, disable the tickless mode, recompile the project and download the program to the LPC55S16-EVK board, and test the current on the `MCU_VBAT` pin, as shown in Figure 7.

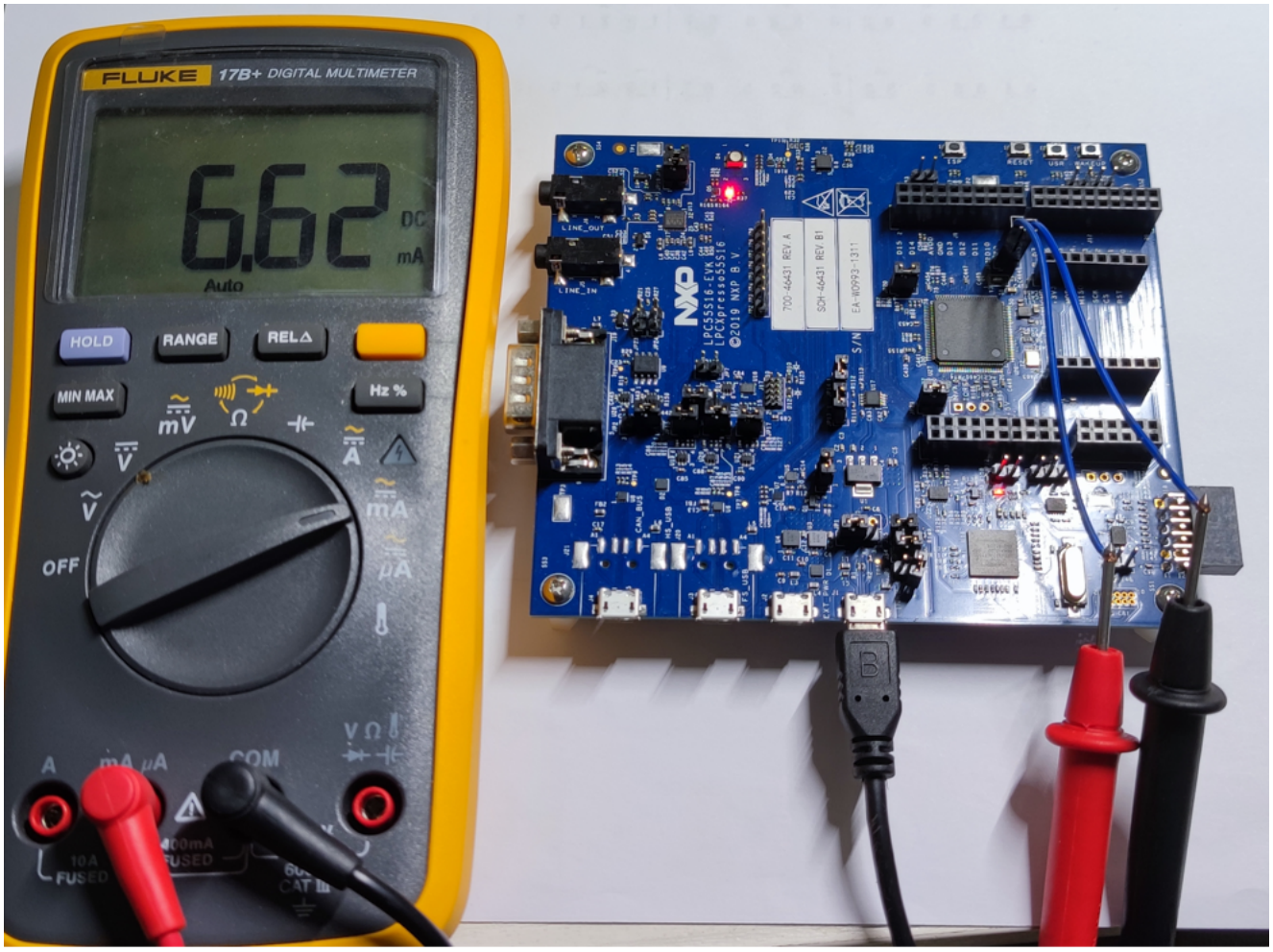

Figure 7. Current consumption of MCU_VBAT pin when the tickles mode is disabled

It can be seen from Figure 6 and Figure 7 that after the tickless mode is enabled, the current consumption on the `MCU_VBAT` pin decreases by 2.34 mA.

4.2 More ways to reduce system power consumption

If users want a lower current consumption, they can make the MCU into the deep sleep mode or the power-down mode when MCU is idle. Users can also use the `configPRE_SLEEP_PROCESSING` and `configPOST_SLEEP_PROCESSING` macro-related functions to perform more power-reducing operations, such as turning off the peripheral clock and lowering the system frequency.

In the `freertos_tickless` example of the LPC55S16 SDK, there are no functions defined for the `configPRE_SLEEP_PROCESSING` and `configPOST_SLEEP_PROCESSING` macros. This section describes how to achieve lower current consumption.

Define `LOW_POWER_MODE` macro to indicate different low-power modes:

LOW_POWER_MODE = 1 : Sleep mode

LOW_POWER_MODE = 2 : Deep sleep mode

LOW_POWER_MODE = 3 : Power-down mode

The current consumption of these modes measured on the LPC55S16-EVK board is shown in Table 1.

Table 1. Current consumption in different power modes

Low-power mode	Current
Normal mode	6.62 mA
Sleep/USART enabled	4.28 mA
Sleep/USART disabled	4.15 mA
Deep sleep	55.8 uA
Power-down	3.6 uA

The following describes how to implement these low-power modes.

4.2.1 Sleep mode

As described in Chapter 2, the `configUSE_TICKLESS_IDLE` macro can be set to 1 or 2 to enable tickless mode, which makes MCU enter the sleep mode. However, in the sleep mode, only the core is stopped and other peripherals may still be working, so users can also use the `configPRE_SLEEP_PROCESSING` and `configPOST_SLEEP_PROCESSING` macros to turn off some peripheral clocks, such as the USART peripherals, before entering the sleep mode.

The implementation method is as follows:

1. Define corresponding functions to disable/enable the USART peripheral for the `configPRE_SLEEP_PROCESSING` and `configPOST_SLEEP_PROCESSING` macros.

```
#ifndef configPRE_SLEEP_PROCESSING
    #define configPRE_SLEEP_PROCESSING( x )   vPortConfigPreSleepProcessing( x )
#endif

#ifndef configPOST_SLEEP_PROCESSING
    #define configPOST_SLEEP_PROCESSING( x )  vPortConfigPostSleepProcessing( x )
#endif

void vPortConfigPreSleepProcessing(TickType_t xExpectedIdleTime)
{
#if LOW_POWER_MODE == 1
    /* Disable USART0 peripheral */
    DbgConsole_Deinit();
    CLOCK_DisableClock(kCLOCK_FlexComm0);
#endif
}

void vPortConfigPostSleepProcessing(TickType_t xExpectedIdleTime)
{
#if (LOW_POWER_MODE == 1) || (LOW_POWER_MODE == 3)
    /* Enable USART0 peripheral */
    BOARD_InitDebugConsole();
#endif
}
```

2. Call the functions corresponding to `configPRE_SLEEP_PROCESSING` and `configPOST_SLEEP_PROCESSING` macros to enable/disable the USART peripheral.

```
xModifiableIdleTime = xExpectedIdleTime;
configPRE_SLEEP_PROCESSING(xModifiableIdleTime);
if (xModifiableIdleTime > 0)
```

```
{
#if LOW_POWER_MODE == 1
    __DSB();
    __WFI();
    __ISB();
#elif LOW_POWER_MODE == 2
    POWER_EnterDeepSleep(APP_EXCLUDE_FROM_DEEPSLEEP, 0x7FFF, WAKEUP_RTC_LITE_ALARM_WAKEUP,
0x0);
#elif LOW_POWER_MODE == 3
    POWER_EnterPowerDown(APP_EXCLUDE_FROM_POWERDOWN, 0x7FFF, WAKEUP_RTC_LITE_ALARM_WAKEUP, 1);
#endif
}
configPOST_SLEEP_PROCESSING(xExpectedIdleTime);
ulLPTimerInterruptFired = false;
```

When the `LOW_POWER_MODE` macro is 1 and the MCU is idle, it enters the sleep mode. Before entering the sleep mode, the USART peripheral is disabled. In this case, the current consumption is 4.15 mA, as shown in Figure 8. After the USART0 peripheral is disabled, the current decreases by 130 uA.

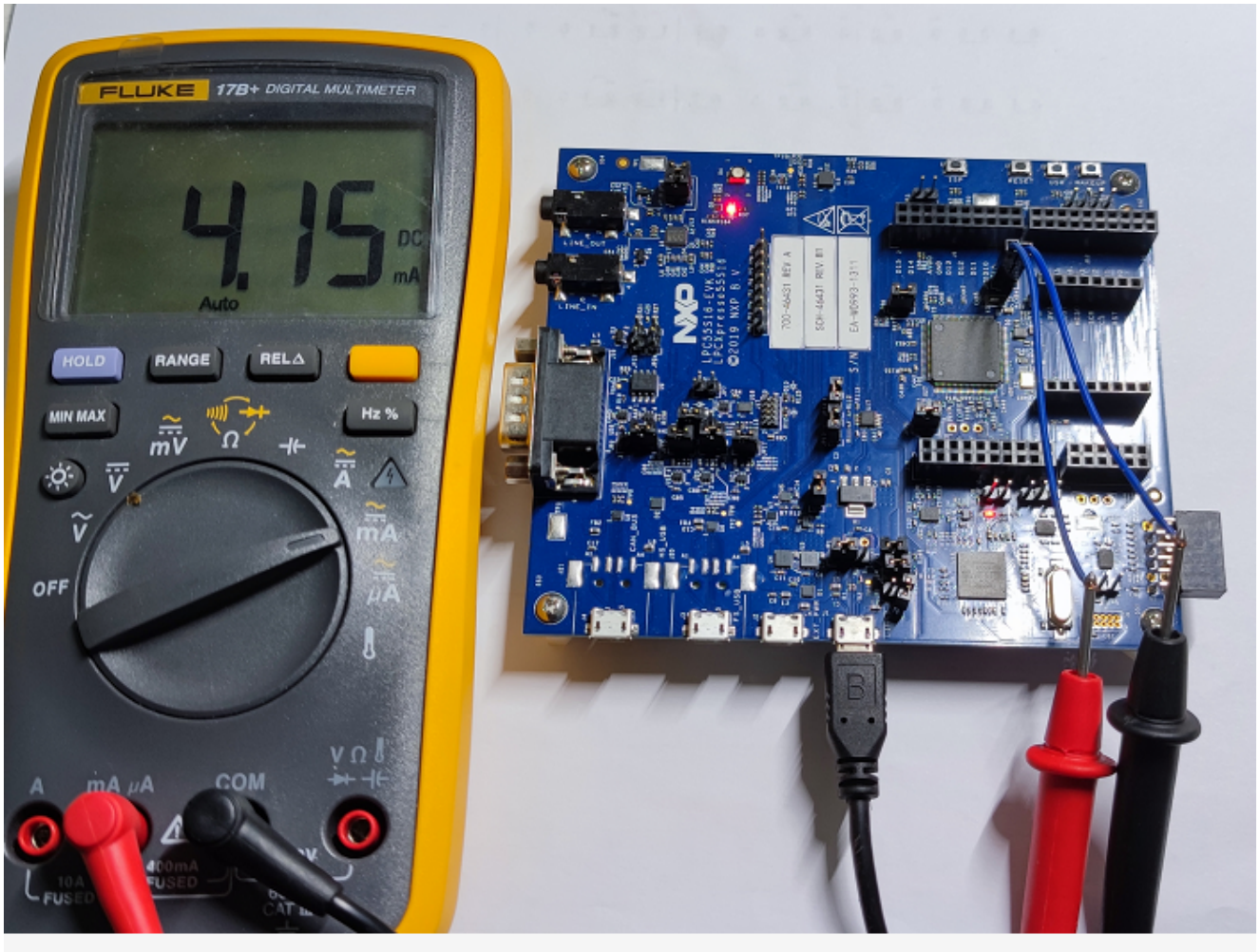

Figure 8. Current consumption on MCU_BVAT pin after disabling the USART0 peripheral

4.2.2 Deep sleep mode

In addition to make the LPC55S16 enter the sleep mode, the user can also make it enter the deep sleep mode when it is idle, as shown below.

```
#define APP_EXCLUDE_FROM_DEEPSLEEP (kPDRUNCFG_PD_XTAL32K)

POWER_EnterDeepSleep(APP_EXCLUDE_FROM_DEEPSLEEP, 0x7FFF,
                     WAKEUP_RTC_LITE_ALARM_WAKEUP, 0x0);
```

The current consumption in deep sleep mode is 55.8 uA, as shown in Figure 9.

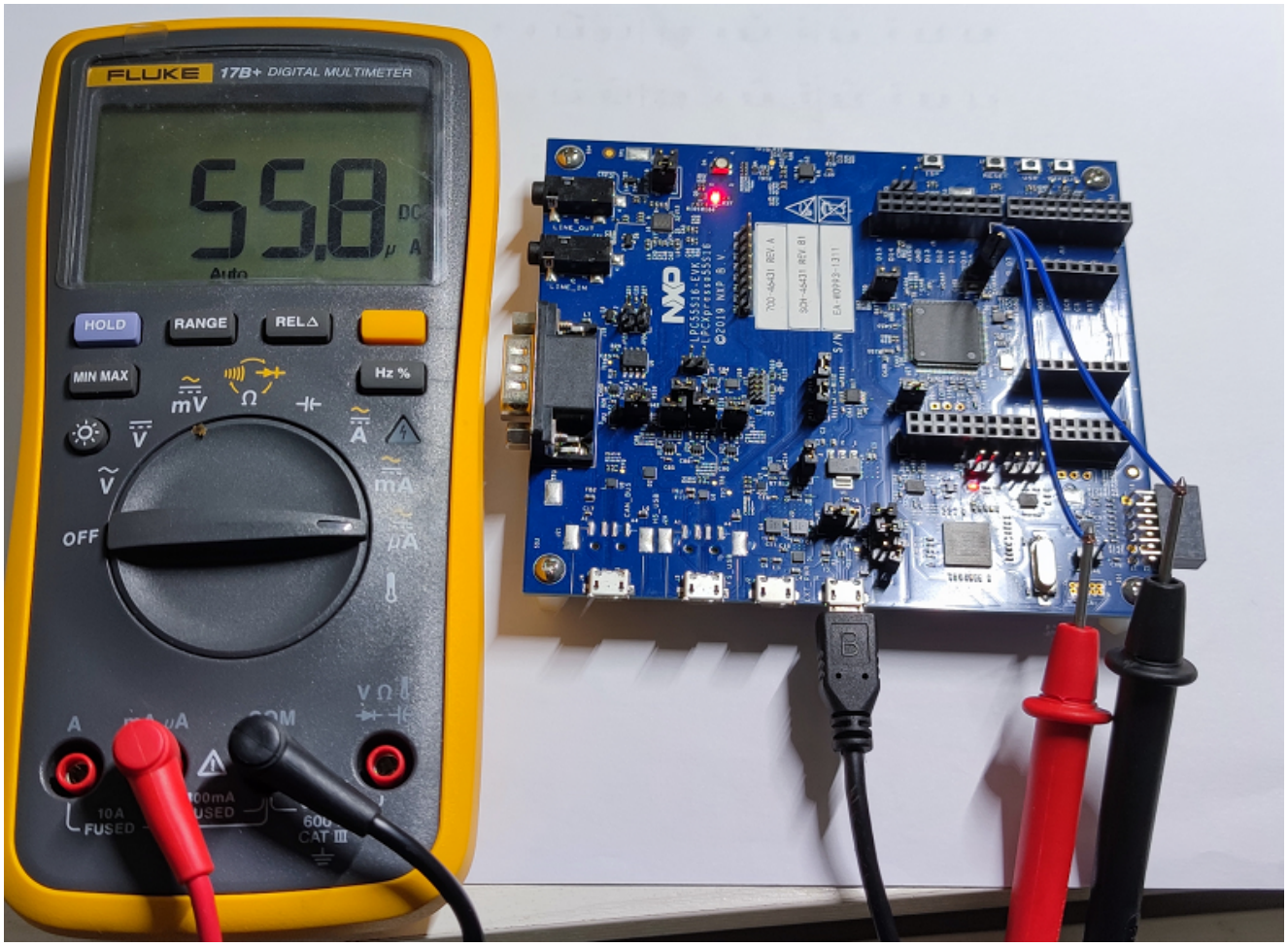

Figure 9. Current consumption in the deep sleep mode

4.2.3 Power-down mode

Users can also call `POWER_EnterPowerDown()` function to make the LPC55S16 enter the power-down mode, as shown below.

```
#define APP_EXCLUDE_FROM_POWERDOWN (kPDRUNCFG_PD_XTAL32K)

POWER_EnterPowerDown(APP_EXCLUDE_FROM_POWERDOWN, 0x7FFF,
                     WAKEUP_RTC_LITE_ALARM_WAKEUP, 1);
```

The current consumption in the power-down mode is 3.6 uA, as shown in Figure 10.

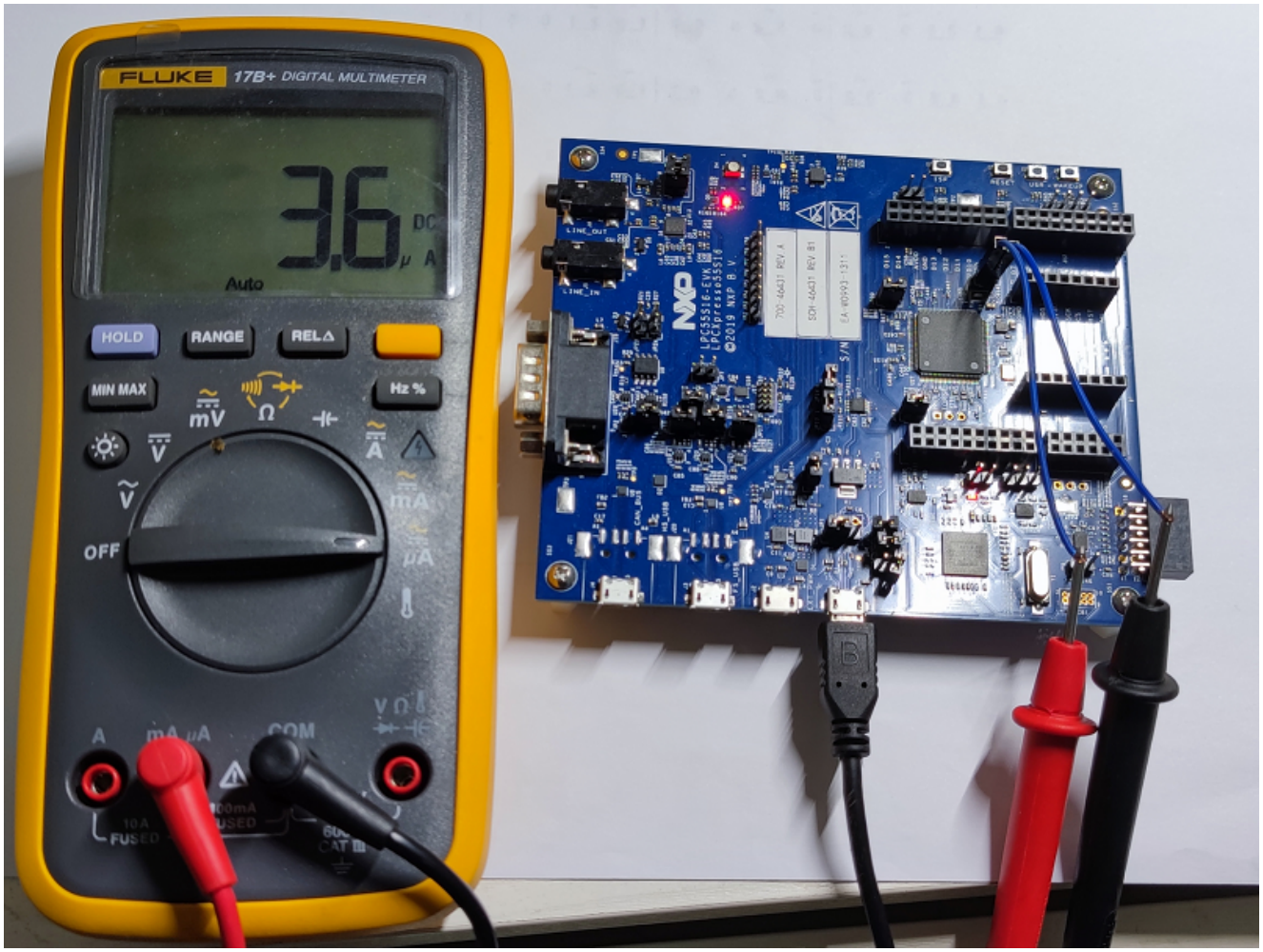

Figure 10. Current consumption in power-down mode

USART0 (Flexcomm0) is still disabled after the MCU is awakened from the power-down mode and must be enabled again, as shown below.

```
void vPortConfigPostSleepProcessing(TickType_t xExpectedIdleTime)
{
#if (LOW_POWER_MODE == 1) || (LOW_POWER_MODE == 3)
    /* Enable USART0 peripheral */
    BOARD_InitDebugConsole();
#endif
}
```

The changes described in Chapter 4.2 are defined in the `fsl_tickless_rtc.c` file, and users can copy the changes to the same location in their `fsl_tickless_rtc.c` file to implement these functions.

5 Conclusion

This application note explains the implementation principle of FreeRTOS's tickless mode and introduces the details of implementing tickless mode on NXP LPC5500 series MCUs. Users can refer to this application note to use tickless mode in their applications to reduce system power consumption.

6 Reference

1). LPC55S1x/LPC551x User Manual

2) LPC55S3x UM.

7 Revision history

Table 2. Revision history

Revision number	Date	Substantive changes
0	21 March 2022	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX,EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. M, M Mobileye and other Mobileye trademarks or logos appearing herein are trademarks of Mobileye Vision Technologies Ltd. in the United States, the EU and/or other jurisdictions.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 21 March 2022
Document identifier: AN13593