AN1065
Application Notes

PY32F030_PY32F003_PY32F002A
Application Notes

Introduction

The PY32F030_PY32F003_PY32F002A series of microcontrollers features a high-performance 32-bit ARM® Cortex®-M0+ core, a wide voltage range MCU. They are embedded with up to 64 Kbytes of flash memory and 8 Kbytes of SRAM memory, operating at a maximum frequency of 48 MHz.

This application note will help users understand the precautions for the application of various modules in PY32F030_PY32F003_PY32F002A and quickly start development.

Table 1 Applicable Products

Type	Product Series
Microcontroller Series	PY32F030、PY32F003、PY32F002A

Contents

1. PWR Configuration

- To ensure system stability, it is essential to enable the watchdog function.
- It is recommended that customers enable the watchdog in the OPTION and set the watchdog overflow time according to actual conditions through software.
- Once the watchdog is enabled, it cannot be turned off by software. Therefore, in low power consumption modes, an LPTIM timer should be used to wake up and feed the watchdog (refer to example routines in Appendix 1).
- When using event wakeup in Sleep mode, if the EXTI module clock and the CPU clock are one clock source, the divider frequency must not be used.

2. GPIO Configuration

- All GPIOs must not have a negative voltage exceeding -0.3V; for example, in the intercom market, it is recommended to use an IO port without AD functionality for channel switching features.
- The input voltage of all GPIO pins must not exceed VCC+0.3V.
- BOOT0 must not be high during any reset (including after being configured as a regular GPIO), otherwise, it will enter the BOOTLOADER.
- Structures such as GPIO initialization must be assigned a value of 0 to avoid undefined initial values.

3. ADC Configuration

3.1 ADC Software Configuration

- Add ADC_FORCE_RESET before ADC initialization to ensure successful initialization.
- ADC channels must be configured before enabling; configuring after enabling will result in failure.
- The ADC clock must be configured to below 16MHz to ensure ADC sampling accuracy.
- An 8 ADC clock cycle delay must be added after ADC enabling before enabling conversion; otherwise, it will affect sampling accuracy.
- Directly driving high-power devices with GPIO can affect ADC sampling results (e.g., digital tube display. It is recommended not to sample ADC during digital tube display, or to insert a 10-100 ohm resistor in series on each data line of the digital tube, which can be adjusted according to actual conditions).
- After ADC is enabled, it cannot be disabled by software; the ADC module needs to be

reset, reinitialized, and then restarted.

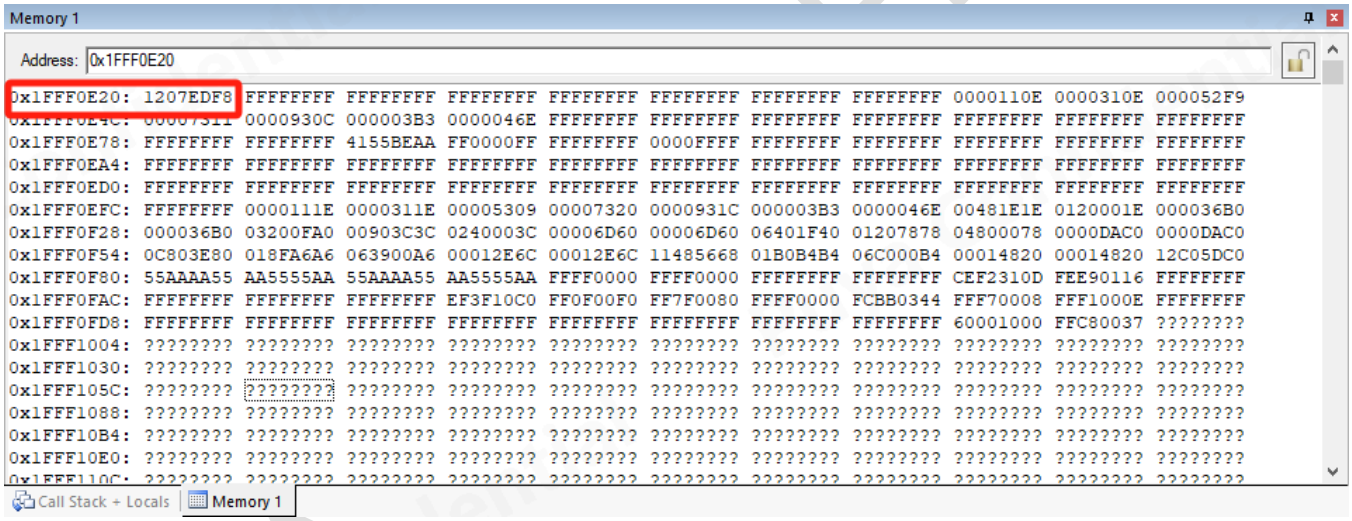
- When ADC is in continuous mode or discontinuous mode, and only channel 0 is used, the scan sequence must be set to go downwards.
- In single mode, after the conversion is completed, the ADC module needs to be re-enabled (ADC_EN = 1) to start the next conversion.

3.2 ADC Hardware Configuration

- The voltage of the ADC channel must not exceed VCC+0.3V (even if the ADC channel is not configured as an AD function), otherwise the ADC sampling will be inaccurate.

3.3 Vreferint 1.2V

- The Vreferint 1.2V actual value of the chip is placed in the information area (0x1FFF0E20) in FLASH. (The high 16 bits are the actual value and the low 16 bits are the inverse code.) The procedure for reading Vreferint 1.2V is shown in Appendix 2:



- When sampling the internal reference voltage of 1.2V, the result calculated through the ADC sampling time conversion formula should be at least 20 microseconds. The methods are as follows:

a) Reduce the resolution;

b) Decrease the ADC clock frequency;

c) Increase the ADC sampling period.

The total conversion time is calculated as follows:

tCONV = Sampling Time 采样时间 + (Conversion Resolution+0.5) x ADC Clock Cycle

For example:

When ADC_CLK = 12MHz, the resolution is 12 bits, and the sampling time is 239.5 ADC clock cycles:

tCONV = (239.5 + 12.5) x ADC Clock Cycle=252 x ADC Clock Cycle= 21 us

4. I2C Configuration

- After initializing pins PF0 and PF1 as SCL and SDA for I2C, the BUSY bit status is affected by the IO port and is set to 1, which affects the use of I2C. Software must reset the I2C module once after IO port initialization to clear the BUSY bit.
- When the I2C slave sends a frame of data, the buffer pointer will increment after the master reissues the address, so the slave needs to reinitialize the buffer pointer in the address interrupt.
- When the I2C slave receives each byte and requires clock stretching, the first two bytes from the I2C master to the slave cannot be clock stretched.
- When operating as an I2C slave and the master reads data from the slave, the master does not send an ACK for the last byte, preventing the slave from entering the STOP interrupt. The NACK interrupt can be used as the STOP interrupt to end the transmission. (Reference code is as follows)

```
void I2C1_IRQHandler(void)
  {
    HAL_I2C_EV_IRQHandler(&I2cHandle);
    HAL_I2C_ER_IRQHandler(&I2cHandle);
  }
```

5. COMP Configuration

- To use Comparator 2, Comparator 1's SCALER_EN must also be enabled. (Reference code is as follows)

```
int main(void)
{
  /* Reset of all peripherals，  Initializes the Systick */
  HAL_Init();

  __HAL_RCC_COMP1_CLK_ENABLE();                    /* Enable COMP1 clock */

  __HAL_RCC_COMP2_CLK_ENABLE();                    /* Enable COMP2 clock */

  SET_BIT(COMP1->CSR，COMP_CSR_SCALER_EN);      /* Enable COMP1 SCALER_EN */

  while (1)
  {
  }
}
```

6. RCC Configuration

- The PLL can only multiply the frequency from 24MHz to 48MHz.
- With the PLL enabled, FLASH_LATENCY needs to be set to 1.
- The PLL needs to be turned off before entering sleep mode, and the clock should be switched to HSI.
- At 48MHz, the PLL should be turned off during IAP jump (refer to Appendix 3 for examples).
- E-series chips do not support frequency division of HSI.
- In environments with strong electromagnetic interference, it is not recommended to use HSE and LSE; for example, in walkie-talkie applications, it is recommended to use HSE in frequency measurement mode and HSI in non-frequency measurement mode.

7. SPI Transmission and Reception

- When SPI acts as a master and receives a series of data, it will have an extra byte, and the software needs to discard the first byte.
- When using SPI as a master for transmission, it is not recommended to use hardware chip select; instead, software chip select is recommended. To release the hardware chip select, SPI needs to be disabled.
- When SPI acts as a master and sends data, an extra 0x00 byte is sent before each byte. Forcing a cast on the DR register (((uint8_t volatile *)&SPI1->DR)) = byte can avoid this issue.
- When SPI acts as a master and sends data by directly writing to the DR register, it is necessary to add four NOP() instructions after writing to DR to ensure successful transmission.
- It is recommended that customers use libraries directly for SPI operations.
- When SPI uses DMA mode, SPI should be started first, and then DMA should be enabled.

8. SPI TFT Display Application

- It is recommended to use simplex mode for SPI, with TX in polling mode and RX in DMA mode.

9. IAP Upgrade

- In PY32F030/PY32F003/PY32F002A, when IAP jumps to the APP, interrupt remapping is required. The APP code and interrupt vectors are located at the address (0x80000000

+ VECT_TAB_OFFSET), which is 0x80001000. Therefore, VECT_TAB_OFFSET is 0x1000, so it is necessary to define VECT_TAB_OFFSET as 0x1000 (#define VECT_TAB_OFFSET 0x1000).

- The BOOT program area needs to have write protection to prevent the BOOT area from being erased. (Write protection needs to be configured in the programmer).
- If there are write operations to FLASH in the program, write protection should be added to the program area to avoid accidental operations in the program area.

10. LCD Configuration

- When using the high current pin alone, it is necessary to enable the LED module clock, set LED_CR_EHS = 1, and configure the GPIO speed as VERY_HIGH. (Reference code is as follows)

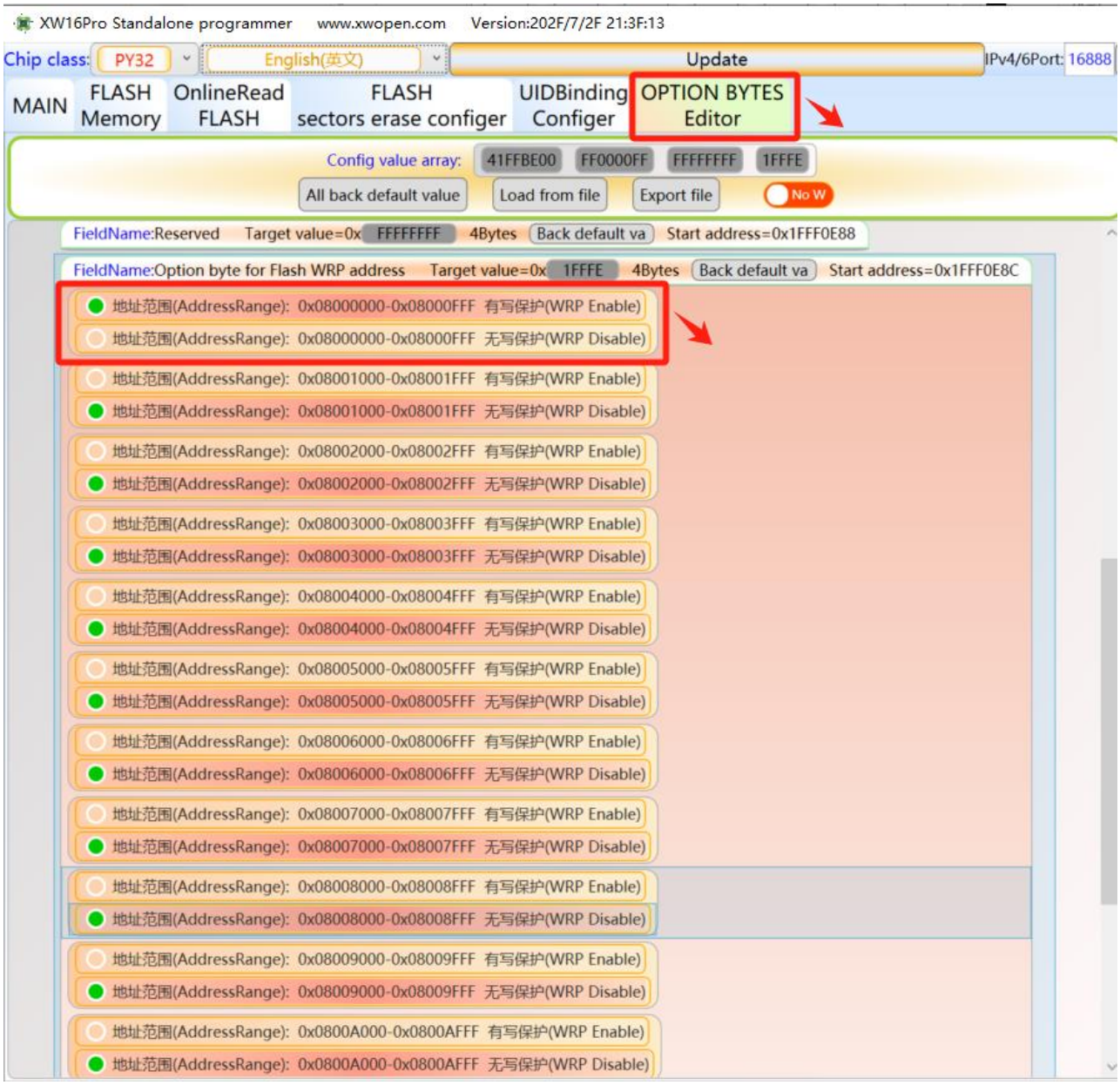
```
int main(void)
{
  /* Reset of all peripherals，  Initializes the Systick */
  HAL_Init();
  SET_BIT(RCC->APBENR2，RCC_APBENR2_LEDEN);
  SET_BIT(LED->CR，LED_CR_EHS);
  APP_GpioConfig();
}

/**
  * @brief  GPIO configuration
  * @param  None
  * @retval None
  */
static void APP_GpioConfig(void)
{
  GPIO_InitTypeDef  GPIO_InitStruct = {0};

  __HAL_RCC_GPIOB_CLK_ENABLE();                          /* Enable GPIOB clock */

  GPIO_InitStruct.Pin = GPIO_PIN_3;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;            /* Push-pull output */
  GPIO_InitStruct.Pull = GPIO_NOPULL;                    /* Enable pull-up */
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;     /* GPIO speed */
  /* GPIO Initialization */
  HAL_GPIO_Init(GPIOB，  &GPIO_InitStruct);
}
```

11. FLASH Configuration

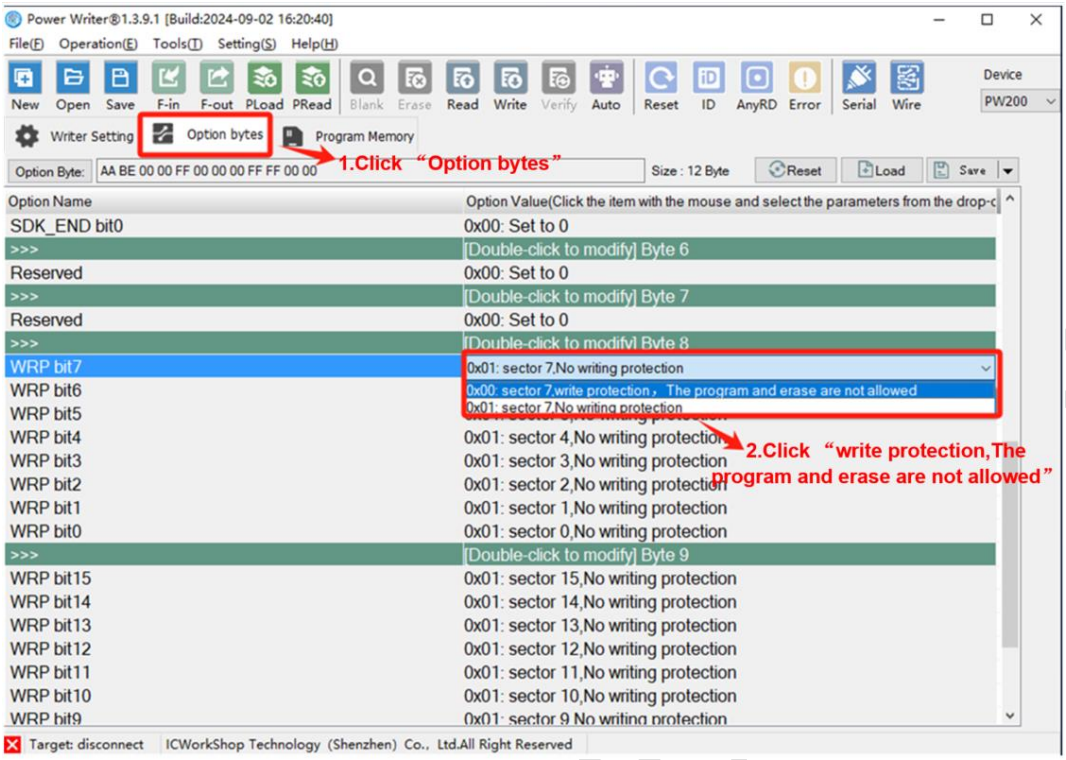
- FLASH only supports Page erase and Page write, a Page is 256 bytes, the start address can only be Page aligned; (e.g. start address 0x08005000, 0x08005100, etc.)
- Every time Page write must be preceded by Page erase.

12. OPTION Configuration

- When mass production, OPTION operation needs to be configured in the burner option byte, and the function that operates OPTION in the programme should be blocked;
- It is recommended that the customer's programme enable write-protection, and the write-protection is set in OPTION, the specific operation steps are shown in the following figure;

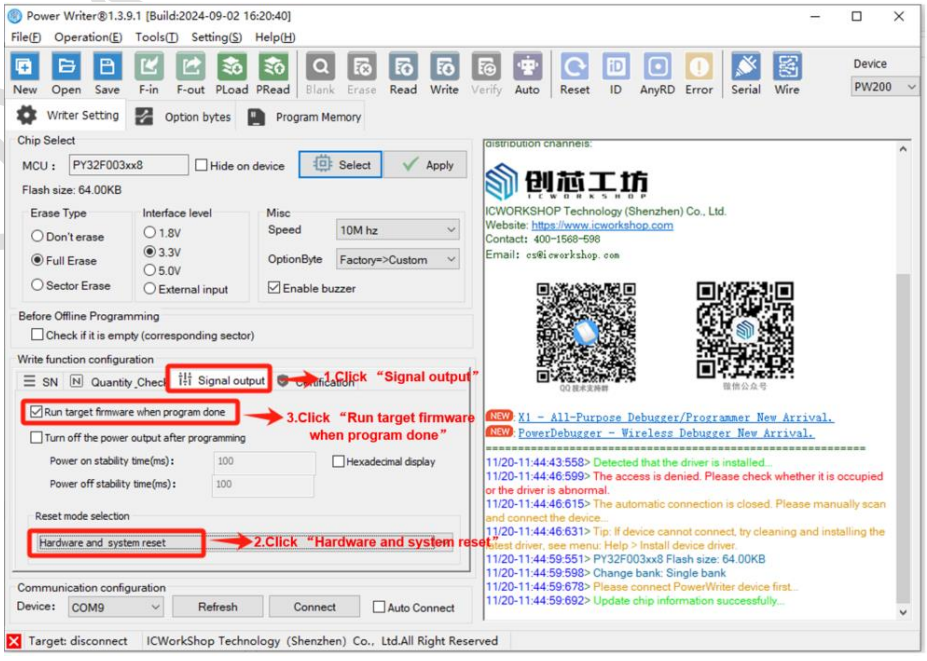


Xuanwei Operation Write Protection

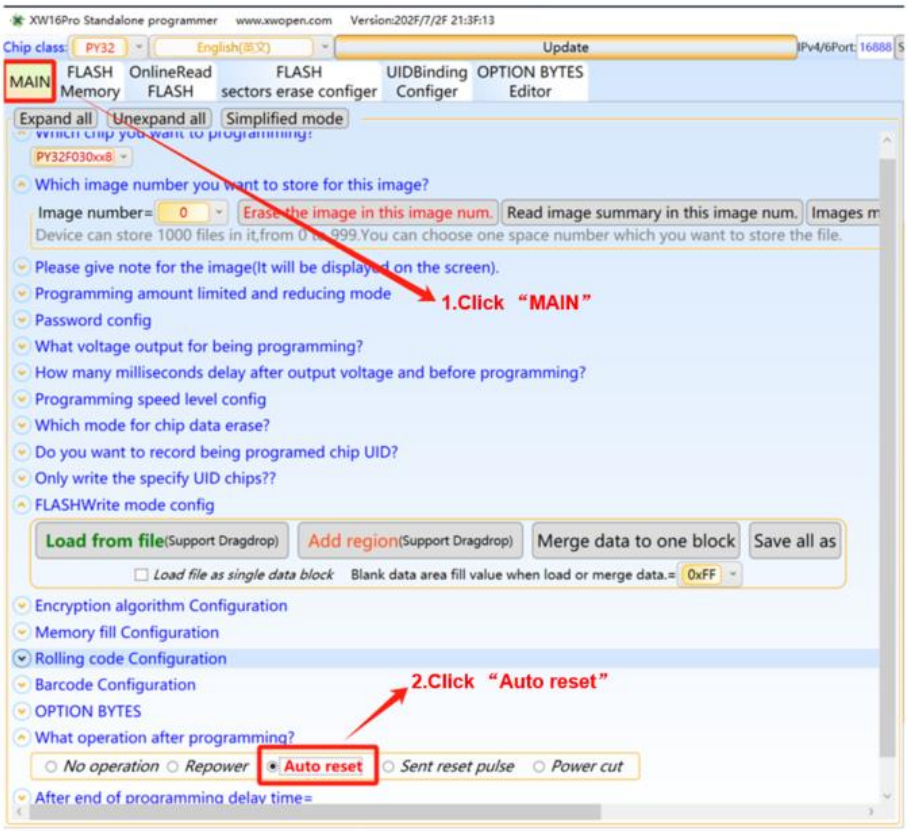


Chip Workshop Operation Write Protection

- When configuring OPTION in the burner, it is necessary to tick the smart reset function/reboot the chip after programming (all the burners have similar options to be ticked), the specific operation steps are shown in the following figure.



Chip Workshop Operation Click "Restart the chip after programming "



Xuanwei Operation Auto Reset

13. LPTIM Configuration

- When the LPTIM clock source is LSE, HSE must be enabled before enabling LSE. (Refer to Appendix 4 for examples; this operation is not required for PY32F030-E/PY32F003-E/PY32F002A-E versions)

13.1 LPTIM Continuous Mode

- LPTIM Continuous Mode must clear ARRMCF each time before entering STOP and wait for 1 LSI clock cycle*PSC factor.(Approx. 40us*PSC including programme execution time)
- Changing the LPTIM reload value requires a wait of 4 LSI clock cycles * PSC factor.(Approx. 160us*PSC including programme execution time)

13.2 LPTIM Single Shot Mode

- LPTIM single mode wake-up from STOP and wait 3 LSI clock cycles before entering STOP again. (Takes about 120us, including programme execution time)

- Wait 4 LSI clock cycles when changing the reload value LPTIM_ARR. (Approx. 160us including programme execution time)

14. TIM1 Configuration

- TIM1_CH1, TIM1_CH2, and TIM1_CH3 will not reset their OCREF (Output Compare Reference) to zero after being disabled and then re-enabled; they will continue counting. TIM1_CH4, however, will reset its OCREF to zero and start counting again after being disabled and then re-enabled. If you need TIM1_CH1/CH2/CH3 signals to be synchronized with TIM1_CH4, you must manually clear the OCREF of all channels after disabling and before re-enabling them.

15. Version History

Version	Date	Update Records
V1.0	2024.03.28	Initial release
V1.1	2024.06.06	Modify the content of chapters 1, 4, 9, and 11.
V1.2	2024.10.15	Add new chapters for FLASH, LPTIM, and TIM1. Combine chapters 3, 4, and 5, and merge chapters 6 and 7. Modify chapters 1, 2, 3, 4, 5, 6, 7, 9, 10, and 12.

Puya Semiconductor Co., Ltd.

Appendix 1

1.1 PY32F030/PY32F003/PY3F002A Low Power Mode, Timed Wake-Up Feed Dog Routine (LL Library)

```
int main(void)
{
  /* Configure system clock */
  APP_SystemClockConfig();

  /* Enable LPTIM and PWR clock */
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_LPTIM1);
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);

  /* Initialize LED and button */
  BSP_LED_Init(LED3);
  BSP_PB_Init(BUTTON_USER,BUTTON_MODE_GPIO);
  /* Configure LPTIM clock source as LSI */
  APP_LPTIMClockconf();
  /* Configure and enable LPTIM */
  APP_ConfigLPTIMOneShot();

  /* Turn on LED */
  BSP_LED_On(LED3);

  /* Wait for button press */
  while(BSP_PB_GetState(BUTTON_USER) != 0)
  {}
  APP_IwdgConfig();

  /* Turn off LED */
  BSP_LED_Off(LED3);

  while (1)
  {
    /* Enable low power run mode */
    LL_PWR_EnableLowPowerRunMode();
    /* Disable LPTIM */
    LL_LPTIM_Disable(LPTIM1);
    APP_uDelay(120);                    //A delay of more than 120 microseconds must be added here
    /* Enable LPTIM */
    LL_LPTIM_Enable(LPTIM1);
    /* Set autoreload value */
    LL_LPTIM_SetAutoReload(LPTIM, 51);
    /* Start LPTIM in one-shot mode */
    LL_LPTIM_StartCounter(LPTIM1,LL_LPTIM_OPERATING_MODE_ONESHOT);

    /* Set SLEEPDEEP bit of Cortex System Control Register */
    LL_LPM_EnableDeepSleep();

    /* Request Wait For Interrupt */
    __WFI();
    LL_IWDG_ReloadCounter(IWDG);
    LL_GPIO_TogglePin(GPIOA,LL_GPIO_PIN_3);
  }
}
```

```
void APP_IwdgConfig(void)
{
  /* Enable LSI */
  LL_RCC_LSI_Enable();
  while (LL_RCC_LSI_IsReady() == 0U) {;}

  /* Enable IWDG */
  LL_IWDG_Enable(IWDG);

  /* Enable write access */
  LL_IWDG_EnableWriteAccess(IWDG);

  /* Set IWDG prescaler */
  LL_IWDG_SetPrescaler(IWDG, LL_IWDG_PRESCALER_32); /* T=1MS */

  /* Set watchdog reload counter */
  LL_IWDG_SetReloadCounter(IWDG, 1000); /* 1ms*1000=1s */

  /* IWDG initialization */
  while (LL_IWDG_IsReady(IWDG) == 0U) {;}

  /* Feed watchdog */
  LL_IWDG_ReloadCounter(IWDG);
}
/**
  * @brief  LPTIM clock configuration
  * @param  None
  * @retval None
  */
static void APP_LPTIMClockconf(void)
{
  /* Enable LSI */
  LL_RCC_LSI_Enable();

  /* Wait for LSI to be ready */
  while(LL_RCC_LSI_IsReady() == 0)
  {}

  /* Configure LSI as LPTIM clock source */
  LL_RCC_SetLPTIMClockSource(LL_RCC_LPTIM1_CLKSOURCE_LSI);
}

/**
  * @brief  Configure LPTIM in one-shot mode
  * @param  None
  * @retval None
  */
static void APP_ConfigLPTIMOneShot(void)
{
  /* Configure LPTIM */
  /* LPTIM prescaler: divide by 128 */
  LL_LPTIM_SetPrescaler(LPTIM1,LL_LPTIM_PRESCALER_DIV128);

  /* Update ARR at the end of LPTIM counting period */
  LL_LPTIM_SetUpdateMode(LPTIM1,LL_LPTIM_UPDATE_MODE_ENDOFPERIOD);

  /* Enable ARR interrupt */
  LL_LPTIM_EnableIT_ARRM(LPTIM1);

  /* Enable NVIC interrupt request */
```

```
    NVIC_EnableIRQ(LPTIM1_IRQn);
    NVIC_SetPriority(LPTIM1_IRQn,0);

    /* Enable LPTIM */
    LL_LPTIM_Enable(LPTIM1);

    /* Configure auto-reload value: 51 */
    LL_LPTIM_SetAutoReload(LPTIM1,51);
}
void APP_LPTIMCallback(void)
{
    /*Toggle LED*/
    BSP_LED_Toggle(LED3);
}
static void APP_uDelay(uint32_t Delay)
{
    uint32_t temp;
    SysTick->LOAD=Delay*(SystemCoreClock/1000000);
    SysTick->VAL=0x00;
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk;
    do
    {
       temp=SysTick->CTRL;
    }
    while((temp&0x01)&&!(temp&(1<<16)));
    SysTick->CTRL=0x00;
    SysTick->VAL =0x00;
}
static void APP_SystemClockConfig(void)
{
    /* Enable HSI */
    LL_RCC_HSI_Enable();
    while(LL_RCC_HSI_IsReady() != 1)
    {
    }

    /* Set AHB prescaler */
    LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);

    /* Configure HSISYS as system clock source */
    LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSISYS);
    while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_HSISYS)
    {
    }

    /* Set APB1 prescaler */
    LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
    LL_Init1msTick(8000000);

    /* Update system clock global variable SystemCoreClock (can also be updated by calling
SystemCoreClockUpdate function) */
    LL_SetSystemCoreClock(8000000);
}
void APP_ErrorHandler(void)
{
    /* Infinite loop */
    while(1)
    {
    }
}
```

1.2 PY32F030/PY32F003/PY3F002A Low Power Mode, Timed Wake-Up Feed Dog Routine (HAL Library)

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Systick */
  HAL_Init();

  /* Clock configuration */
  APP_RCCOscConfig();

  /* Initialize LED */
  BSP_LED_Init(LED3);

  /* Initialize button */
  BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);

  /* LPTIM initialization */
  APP_LPTIMInit();

  /* Enable PWR */
  __HAL_RCC_PWR_CLK_ENABLE();

  /* Turn on LED */
  BSP_LED_On(LED_GREEN);

  /* Wait for button press */
  while (BSP_PB_GetState(BUTTON_USER) != 0)
  {
  }

  /* IWDG initialization */
  APP_IWDGInit();

  /* Turn off LED */
  BSP_LED_Off(LED_GREEN);

  while (1)
  {
    /* Disable LPTIM */
    __HAL_LPTIM_DISABLE(&LPTIMCONF);

    /* Enable LPTIM and interrupt, and start in single count mode */
    APP_LPTIMStart();

    /* Suspend Systick interrupt */
    HAL_SuspendTick();
    /* Enter STOP mode with interrupt wakeup */
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON,PWR_STOPENTRY_WFI);
    /* Resume Systick interrupt */
        HAL_ResumeTick();
    if (HAL_IWDG_Refresh(&IwdgHandle) != HAL_OK)
    {
      Error_Handler();
    }
          /* LED Toggle */
    BSP_LED_Toggle(LED_GREEN);
```

```
    }
}

static void APP_RCCOscConfig(void)
{
    RCC_OscInitTypeDef OSCINIT;
    RCC_PeriphCLKInitTypeDef LPTIM_RCC;

    /* LSI clock configuration */
    OSCINIT.OscillatorType = RCC_OSCILLATORTYPE_LSI;    /* Set the oscillator type to LSI */
    OSCINIT.LSIState = RCC_LSI_ON;                      /* Enable LSI */
    /* Clock initialization */
    if (HAL_RCC_OscConfig(&OSCINIT) != HAL_OK)
    {
        Error_Handler();
    }

    /* LPTIM clock configuration */
    LPTIM_RCC.PeriphClockSelection = RCC_PERIPHCLK_LPTIM;         /* Select peripheral clock: LPTIM */
    LPTIM_RCC.LptimClockSelection = RCC_LPTIMCLKSOURCE_LSI;      /* Select LPTIM clock source: LSI
*/
    /* Peripheral clock initialization */
    if (HAL_RCCEx_PeriphCLKConfig(&LPTIM_RCC) != HAL_OK)
    {
        Error_Handler();
    }

    /* Enable LPTIM clock */
    __HAL_RCC_LPTIM_CLK_ENABLE();
}
static void APP_LPTIMInit(void)
{
    /* LPTIM configuration */
    LPTIMCONF.Instance = LPTIM;                                   /* LPTIM */
    LPTIMCONF.Init.Prescaler = LPTIM_PRESCALER_DIV128;    /* Prescaler: 128 */
    LPTIMCONF.Init.UpdateMode = LPTIM_UPDATE_IMMEDIATE; /* Immediate update mode */
    /* Initialize LPTIM */
    if (HAL_LPTIM_Init(&LPTIMCONF) != HAL_OK)
    {
        Error_Handler();
    }
}

static void APP_LPTIMStart(void)
{
    /* Enable autoreload interrupt */
    __HAL_LPTIM_ENABLE_IT(&LPTIMCONF, LPTIM_IT_ARRM);

    __HAL_LPTIM_DISABLE(&LPTIMCONF);
    /* Delay 120us */
    APP_delay_us(120);                    //A delay of more than 120 microseconds must be added here
    /* Enable LPTIM */
    __HAL_LPTIM_ENABLE(&LPTIMCONF);

    /* Load autoreload value */
    __HAL_LPTIM_AUTORELOAD_SET(&LPTIMCONF, 51);

    /* Start single count mode */
    __HAL_LPTIM_START_SINGLE(&LPTIMCONF);
}
```

```
  static void APP_IWDGInit(void)
  {
    IwdgHandle.Instance = IWDG;                              /* Select IWDG */
    IwdgHandle.Init.Prescaler = IWDG_PRESCALER_32;    /* Configure prescaler to 32 */
    IwdgHandle.Init.Reload = (1000);                        /* Set IWDG counter reload value to 1000, 1s */
    /* Initialize IWDG */
    if (HAL_IWDG_Init(&IwdgHandle) != HAL_OK)
    {
      APP_ErrorHandler();
    }
  }

  static void APP_delay_us(int us)
 {
       unsigned t1, t2, count, delta, sysclk; sysclk = 24 ; //Modify this according to the system clock

       t1 = SysTick->VAL;
       while(1)
       {
            t2 = SysTick->VAL;
            delta = t2<t1?(t1-t2):(SysTick->LOAD - t2 + t1);
            if(delta >= us * sysclk)
            break;
       }
 }

 void Error_Handler(void)
{
  /* Infinite circulation */
  while (1)
  {
  }
}
```

Appendix 2

2. PY32F030/PY32F003/PY32F002A reads the Vreferint 1.2V measured value stored in the information area (see 1.3.3 for specific address).

```
#define HAL_VREF_INT                    (*(uint8_t *)(0x1fff0E23))
#define HAL_VREF_DEC                    (*(uint8_t *)(0x1fff0E22))
#define vref_int            (*(uint8_t *)(HAL_VREF_INT))
// Store the integer part of the reference voltage
#define vref_dec            (*(uint8_t *)(HAL_VREF_DEC))
// Store the fractional part of the reference voltage
float vref;                     //reference voltage

static uint8_t Bcd2ToByte(uint8_t Value)
{
      uint32_t tmp = 0U;
      tmp = ((uint8_t)(Value & (uint8_t)0xF0) >> (uint8_t)0x4) * 10U;
      return (tmp + (Value & (uint8_t)0x0F));
}

float read_1_2V(void)
{
       uint8_t data_vref_int,data_vref_dec;
    data_vref_int = Bcd2ToByte(HAL_VREF_INT);
    data_vref_dec = Bcd2ToByte(HAL_VREF_DEC);

       //Initialise all peripherals, flash interface, systick
      vref = data_vref_int/10;        //Calculating the reference voltage
      vref = vref + ((data_vref_int%10)*0.1 + data_vref_dec*0.001);
         return vref;
}
```

Appendix 3

3. When using PLL48M as the system clock, IAP jump should disable the PLL

```
LL_UTILS_ClkInitTypeDef UTILS_ClkInitStruct;
static void SYSCLK(void);
/**
   * @brief    The application entry point.
   * @param    None
   * @retval None
   */
int main(void)
{
      SYSCLK();
#ifdef JUMP_TO_APP_BY_USER_BUTTON
   /* Configure user Button */
   BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);

   /* Check if the USER Button is pressed */
   if (BSP_PB_GetState(BUTTON_USER) == 0x00)
   {
      JumpToAddress(APP_ADDR);
   }
#endif

   APP_SystemClockConfig(LL_RCC_HSICALIBRATION_24MHz, 24000000);

   Bootloader_Init();

   /* Infinite loop */
   while (1)
   {
      Bootloader_ProtocolDetection();
   }
}
static void SYSCLK(void)
{
   /* Enable and initialize HSI */
   LL_RCC_HSI_Enable();
   LL_RCC_HSI_SetCalibFreq(LL_RCC_HSICALIBRATION_24MHz);
   while(LL_RCC_HSI_IsReady() != 1)
   {
   }

   LL_PLL_ConfigSystemClock_HSI(&UTILS_ClkInitStruct);

   /* Set AHB prescaler */
   LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);

   /* Configure HSISYS as system clock and initialize it */
   LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);
   while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL)
   {
   }

   /* Set APB1 prescaler and initialize it */
   LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
```

```
    /* Update system clock global variable SystemCoreClock (can also be updated by calling
SystemCoreClockUpdate function) */
    LL_SetSystemCoreClock(48000000);
}
void APP_SystemClockConfig(uint32_t Value, uint32_t HCLKFrequency)
{
    /* HSI Enable and Initialization */
    LL_RCC_HSI_Enable();
    LL_RCC_HSI_SetCalibFreq(Value);
    while(LL_RCC_HSI_IsReady() != 1)
    {
    }

    /*Setting AHB Clock Divider */
    LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);

    /* Configuring HSISYS as System Clock and Initialization */
    LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSISYS);
    while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_HSISYS)
    {
    }

    LL_FLASH_SetLatency(LL_FLASH_LATENCY_0);

    /* Setting APB1 Clock Divider and Initialization */
    LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
    /*Updating the global variable SystemCoreClock for the system clock (which can also be updated through
debugging the SystemCoreClockUpdate function) */
    LL_SetSystemCoreClock(HCLKFrequency);
}
```

Appendix 4

4.1 PY32F030/PY32F003/PY3F002A Low Power Mode, LSE as LPTIM Clock Source Timed Wakeup Feeder Dog Routine (LL Library)

```
int main(void)
{
  /* Configure system clock */
    APP_SystemClockConfig();

    /* Enable LPTIM and PWR clock */
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_LPTIM1);
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);

    /* Initialize LED and button */
    BSP_LED_Init(LED3);
    BSP_PB_Init(BUTTON_USER,BUTTON_MODE_GPIO);
      /* Configure LPTIM clock source as LSI */
    APP_LPTIMClockconf();

    /* Configure and enable LPTIM */
    APP_ConfigLPTIMOneShot();

    /* Turn on LED */
    BSP_LED_On(LED3);

    /* Wait for button press */
    while(BSP_PB_GetState(BUTTON_USER) != 0)
    {}
    APP_IwdgConfig();
    /* Turn off LED */
    BSP_LED_Off(LED3);

    while (1)
    {
       /* Enable low power run mode */
       LL_PWR_EnableLowPowerRunMode();
       /* Disable LPTIM */
       LL_LPTIM_Disable(LPTIM1);
             APP_uDelay(120);
       /* Enable LPTIM */
       LL_LPTIM_Enable(LPTIM1);
          /* Set autoreload value */
       LL_LPTIM_SetAutoReload(LPTIM, 51);

       /* Start LPTIM in one-shot mode */
       LL_LPTIM_StartCounter(LPTIM1,LL_LPTIM_OPERATING_MODE_ONESHOT);

       /* Set SLEEPDEEP bit of Cortex System Control Register */
       LL_LPM_EnableDeepSleep();

       /* Request Wait For Interrupt */
       __WFI();
             LL_IWDG_ReloadCounter(IWDG);
             LL_GPIO_TogglePin(GPIOA,LL_GPIO_PIN_3);
    }
}
void APP_IwdgConfig(void)
```

```
{
  /* Enable LSI */
  LL_RCC_LSI_Enable();
  while (LL_RCC_LSI_IsReady() == 0U) {;}

  /* Enable IWDG */
  LL_IWDG_Enable(IWDG);

  /* Enable write access */
  LL_IWDG_EnableWriteAccess(IWDG);

  /* Set IWDG prescaler */
  LL_IWDG_SetPrescaler(IWDG, LL_IWDG_PRESCALER_32); /* T=1MS */

  /* Set watchdog reload counter */
  LL_IWDG_SetReloadCounter(IWDG, 1000); /* 1ms*1000=1s */

  /* IWDG initialization */
  while (LL_IWDG_IsReady(IWDG) == 0U) {;}

  /* Feed watchdog */
  LL_IWDG_ReloadCounter(IWDG);
}
/**
  * @brief  LPTIM clock configuration
  * @param  None
  * @retval None
  */
static void APP_LPTIMClockconf(void)
{
  LL_PWR_EnableBkUpAccess();
  while(LL_PWR_IsEnabledBkUpAccess() == 0)
  {
  }
  LL_RCC_LSE_Enable();
  while (LL_RCC_LSE_IsReady() != 1)
  {
  }
  LL_PWR_DisableBkUpAccess();
  LL_APB1_GRP1_DisableClock(LL_APB1_GRP1_PERIPH_PWR);

  /* Configure LSI as LPTIM clock source */
  LL_RCC_SetLPTIMClockSource(LL_RCC_LPTIM1_CLKSOURCE_LSE);
}

/**
  * @brief  Configure LPTIM in one-shot mode
  * @param  None
  * @retval None
  */
static void APP_ConfigLPTIMOneShot(void)
{
  /* Configure LPTIM */
  /* LPTIM prescaler: divide by 128 */
  LL_LPTIM_SetPrescaler(LPTIM1,LL_LPTIM_PRESCALER_DIV128);

  /* Update ARR at the end of LPTIM counting period */
  LL_LPTIM_SetUpdateMode(LPTIM1,LL_LPTIM_UPDATE_MODE_ENDOFPERIOD);

  /* Enable ARR interrupt */
```

```
    LL_LPTIM_EnableIT_ARRM(LPTIM1);

    /* Enable NVIC interrupt request */
    NVIC_EnableIRQ(LPTIM1_IRQn);
    NVIC_SetPriority(LPTIM1_IRQn,0);

    /* Enable LPTIM */
    LL_LPTIM_Enable(LPTIM1);

    /* Configure auto-reload value: 51 */
    LL_LPTIM_SetAutoReload(LPTIM1,51);
}

/**
    * @brief    LPTIM ARR interrupt callback function
    * @param    None
    * @retval None
    */
void APP_LPTIMCallback(void)
{
    /*Toggle LED*/
    BSP_LED_Toggle(LED3);
}

/**
    * @brief    Microsecond delay function
    * @param    Delay；delay value
    * @retval None
    */
static void APP_uDelay(uint32_t Delay)
{
    uint32_t temp;
    SysTick->LOAD=Delay*(SystemCoreClock/1000000);
    SysTick->VAL=0x00;
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk;
    do
    {
        temp=SysTick->CTRL;
    }
    while((temp&0x01)&&!(temp&(1<<16)));
    SysTick->CTRL=0x00;
    SysTick->VAL =0x00;
}

/**
    * @brief    System clock configuration function
    * @param    None
    * @retval None
    */
static void APP_SystemClockConfig(void)
{
    /* Enable HSI */
    LL_RCC_HSI_Enable();

    while(LL_RCC_HSI_IsReady() != 1)
    {
    }
        LL_RCC_HSE_Enable();
        LL_RCC_HSE_SetFreqRegion(LL_RCC_HSE_16_32MHz);
```

```
    while(LL_RCC_HSE_IsReady() != 1)
  {
  }


  /* Set AHB prescaler */
  LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);

  /* Configure HSISYS as system clock source */
  LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSISYS);
  while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_HSISYS)
  {
  }

  /* Set APB1 prescaler */
  LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
  LL_Init1msTick(8000000);

  /* Update system clock global variable SystemCoreClock (can also be updated by calling
SystemCoreClockUpdate function) */
  LL_SetSystemCoreClock(8000000);
}

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  None
  * @retval None
  */
void APP_ErrorHandler(void)
{
  /* Infinite loop */
  while(1)
  {
  }
}
```

4.2 PY32F030/PY32F003/PY3F002A Low Power Mode, LSE as LPTIM Clock Source Timed Wakeup Feeder Dog Routine (HAL Library)

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Systick */
  HAL_Init();

  APP_SystemClockConfig();

  /* Clock configuration */
  APP_RCCOscConfig();

  __HAL_RCC_LSI_ENABLE();

  /* Initialize LED */
  BSP_LED_Init(LED3);

  /* Initialize button */
  BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);

  /* LPTIM initialization */
  APP_LPTIMInit();

  /* Enable PWR */

  /* Turn on LED */
  BSP_LED_On(LED_GREEN);

  /* Wait for button press */
  while (BSP_PB_GetState(BUTTON_USER) != 0)
  {
  }

  /* IWDG initialization */
  APP_IWDGInit();

  /* Turn off LED */
  BSP_LED_Off(LED_GREEN);

  while (1)
  {
    /* Disable LPTIM */
    __HAL_LPTIM_DISABLE(&LPTIMCONF);

    /* Enable LPTIM and interrupt, and start in single count mode */
    APP_LPTIMStart();

     /* Suspend Systick interrupt */
    HAL_SuspendTick();
    /* Enter STOP mode with interrupt wakeup */
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON,PWR_STOPENTRY_WFI);
    /* Resume Systick interrupt */
        HAL_ResumeTick();
    if (HAL_IWDG_Refresh(&IwdgHandle) != HAL_OK)
    {
      Error_Handler();
    }
          /* LED Toggle */
    BSP_LED_Toggle(LED_GREEN);
```

```
    }
}

static void APP_RCCOscConfig(void)
{
    RCC_OscInitTypeDef OSCINIT;
    RCC_PeriphCLKInitTypeDef LPTIM_RCC;

    /* LSI clock configuration */
    OSCINIT.OscillatorType = RCC_OSCILLATORTYPE_LSE;    /* Set the oscillator type to LSE*/
    OSCINIT.LSEState = RCC_LSE_ON;                      /* Enable LSE */
    /* Clock initialization */
    if (HAL_RCC_OscConfig(&OSCINIT) != HAL_OK)
    {
        Error_Handler();
    }

    /* LPTIM clock configuration */
    LPTIM_RCC.PeriphClockSelection = RCC_PERIPHCLK_LPTIM;         /* Select peripheral clock: LPTIM */
    LPTIM_RCC.LptimClockSelection = RCC_LPTIMCLKSOURCE_LSE;      /* Select LPTIM clock source:
LSE */
    /* Peripheral clock initialization */
    if (HAL_RCCEx_PeriphCLKConfig(&LPTIM_RCC) != HAL_OK)
    {
        Error_Handler();
    }
static void APP_SystemClockConfig(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /* Configure clock sources HSE/HSI/LSE/LSI */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE | RCC_OSCILLATORTYPE_HSI |
RCC_OSCILLATORTYPE_LSI | RCC_OSCILLATORTYPE_LSE;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;                                                    /*
Enable HSI */
    RCC_OscInitStruct.HSIDiv = RCC_HSI_DIV1;
/* HSI not divided */
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_24MHz;
/* Configure HSI output clock as 8MHz */
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;                                                     /*
Enable HSE */
    RCC_OscInitStruct.HSEFreq = RCC_HSE_16_32MHz;
/* HSE frequency range 16MHz to 32MHz */
    RCC_OscInitStruct.LSIState = RCC_LSI_OFF;                                                   /*
Disable LSI */
    RCC_OscInitStruct.LSEState = RCC_LSE_OFF;
/* Enable LSE */
    RCC_OscInitStruct.LSEDriver = RCC_ECSCR_LSE_DRIVER_1;
/* LSE with default driving capability */
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_OFF;
/* Disable PLL */
    /* Initialize RCC oscillator */
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        APP_ErrorHandler();
    }

    /* Initialize CPU, AHB, and APB bus clocks */
```

```
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
RCC_CLOCKTYPE_PCLK1; /* RCC system clock types */
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
/*    SYSCLK source selection as LSE */
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
/* AHB clock not divided */
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
/* APB clock not divided */
    /* Initialize RCC system clock (FLASH_LATENCY_0=24M or below; FLASH_LATENCY_1=48M) */
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
    {
       APP_ErrorHandler();
    }
}
    /* Enable LPTIM clock */
    __HAL_RCC_LPTIM_CLK_ENABLE();
}
static void APP_LPTIMInit(void)
{
    /* LPTIM configuration */
    LPTIMCONF.Instance = LPTIM;                                    /* LPTIM */
    LPTIMCONF.Init.Prescaler = LPTIM_PRESCALER_DIV128;    /* Prescaler: 128 */
    LPTIMCONF.Init.UpdateMode = LPTIM_UPDATE_IMMEDIATE; /* Immediate update mode */
    /* Initialize LPTIM */
    if (HAL_LPTIM_Init(&LPTIMCONF) != HAL_OK)
    {
       Error_Handler();
    }
}

static void APP_LPTIMStart(void)
{
    /* Enable autoreload interrupt */
    __HAL_LPTIM_ENABLE_IT(&LPTIMCONF, LPTIM_IT_ARRM);

    __HAL_LPTIM_DISABLE(&LPTIMCONF);
    /* Delay 120us */
    APP_delay_us(120);                    //A delay of more than 120 microseconds must be added here

    /* Enable LPTIM */
    __HAL_LPTIM_ENABLE(&LPTIMCONF);

    /* Load autoreload value */
    __HAL_LPTIM_AUTORELOAD_SET(&LPTIMCONF, 51);

    /* Start single count mode */
    __HAL_LPTIM_START_SINGLE(&LPTIMCONF);
}
    static void APP_IWDGInit(void)
    {
       IwdgHandle.Instance = IWDG;                                /* Select IWDG */
       IwdgHandle.Init.Prescaler = IWDG_PRESCALER_32;    /* Configure prescaler to 32 */
       IwdgHandle.Init.Reload = (1000);                        /* Set IWDG counter reload value to 1000, 1s */
       /* Initialize IWDG */
       if (HAL_IWDG_Init(&IwdgHandle) != HAL_OK)
       {
          APP_ErrorHandler();
       }
    }
```

```
  static void APP_delay_us(int us)
 {
       unsigned t1, t2, count, delta, sysclk; sysclk = 24 ; //Modify this according to the system clock

       t1 = SysTick->VAL;
       while(1)
       {
            t2 = SysTick->VAL;
            delta = t2<t1?(t1-t2):(SysTick->LOAD - t2 + t1);
            if(delta >= us * sysclk)
            break;
       }
 }

 void Error_Handler(void)
{
   /* Infinite loop */
   while (1)
   {
   }
}
```