Low-Power Modes on the STM32L0 Series

由 Matt Mielke创建, 最后修改于四月 11, 2019

Introduction

The STMicroelectronics family of ultra-low-power MCUs utilize a low-leakage technology and an optimized design in order to achieve outstandingly low current consumption, making them ideal for battery powered and energy harvesting applications. In order to take full advantage of the low-power capabilities of these devices, it is necessary to know what low-power modes are available, how they can be configured, and for what tasks they are best suited. This article provides an overview of the low-power modes on the STM32L053C8 MCU. However, any STM32L0 device could have been used as the low-power modes are the same across the series. Also included in the ultra-low-power family are the STM32L1 series and the STM32L4 series. These devices are higher performance products featuring more advanced cores, additional memory, and a larger set of peripherals. They feature the same low-power modes as the L0 series (plus some extras in the case of the L4), so this article is a great place to start for them as well. Figure 1, taken from one of ST's brochures, concisely summarizes the features and benefits of the L0, L1, and L4 series.

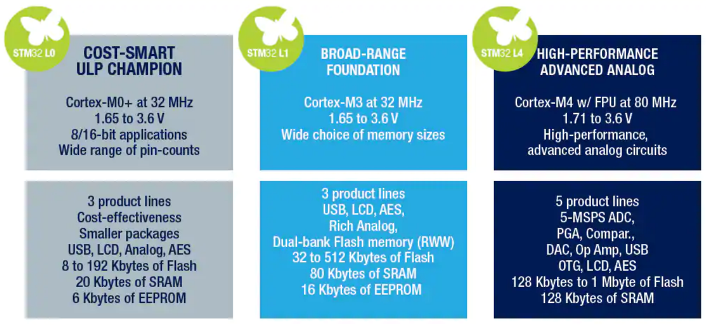
STM32 L0
COST-SMART ULP CHAMPION
Cortex-M0+ at 32 MHz
1.65 to 3.6 V
8/16-bit applications
Wide range of pin-counts
3 product lines
Cost-effectiveness
Smaller packages
USB, LCD, Analog, AES
8 to 192 Kbytes of Flash
20 Kbytes of SRAM
6 Kbytes of EEPROM
STM32 L1
BROAD-RANGE FOUNDATION
Cortex-M3 at 32 MHz
1.65 to 3.6 V
Wide choice of memory sizes
3 product lines
USB, LCD, AES,
Rich Analog,
Dual-bank Flash memory (RWW)
32 to 512 Kbytes of Flash
80 Kbytes of SRAM
16 Kbytes of EEPROM
STM32 L4
HIGH-PERFORMANCE ADVANCED ANALOG
Cortex-M4 w/ FPU at 80 MHz
1.71 to 3.6 V
High-performance,
advanced analog circuits
5 product lines
5-MSPS ADC,
PGA, Compar.,
DAC, Op Amp, USB
OTG, LCD, AES
128 Kbytes to 1 Mbyte of Flash
128 Kbytes of SRAM

Figure 1: Comparison of the STM32 ultra-low-power product series

When doing any work with an MCU from ST, there are two documents that should be readily available. The first is the reference manual, which in the case of the STM32L053C8 is the STM32L0x3 reference manual. This document contains detailed information about the STM32L0x3 line, i.e. how to use the memory and peripheral set. For more detailed information about a particular device in the product line, such as pin mappings, electrical characteristics, and package information, the datasheet (which in this context is the STM32L053xx datasheet) should be used as a reference. As far as low-power modes are concerned, the reference manual will explicitly detail how to enter and exit them, while the datasheet will specifically define the peripheral availability, possible wake-up sources, and current consumption estimates.

Background

The STM32L0 is built on a Cortex-M0+ core, which means its low-power capabilities are dependent on this core's power management features. These features can be configured using the System Control Register (SCR), which is found in the System Control Block. Unfortunately, the core registers are not documented in either the reference manual or the datasheet. ST instead provides the STM32L0 Series Cortex-M0+ Programming Manual for those seeking concise documentation on the Cortex-M0+. The complete documentation on the Cortex-M0, M0+, and M1 cores may be found in the ARMv6-M Architecture Reference Manual. Both documents have a section on *Power Management*, which is a great place to start for this topic.

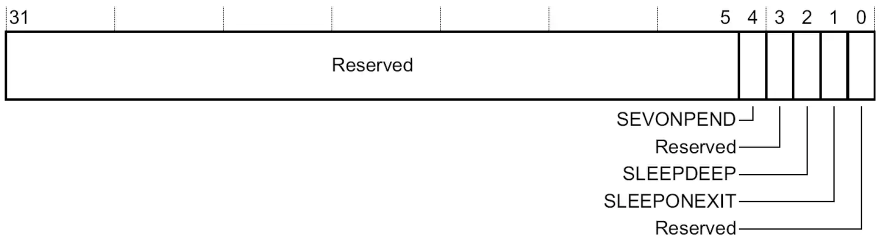

Figure 2: SCR register bits

As Figure 2 shows, the SCR consists of three bits: SEVONPEND, SLEEPONEXIT, and SLEEPDEEP. The SEVONPEND (Send EVent ON PENDing) bit allows interrupts entering the pending state to trigger wake-up events. Note that if these interrupts are not enabled in the NVIC, a wake-up event will still be generated, but the ISR will not be entered. For more information on pending interrupts, enabling interrupts, or the NVIC in general, see the *Nested Vectored Interrupt Controller* section in either of the previously mentioned Cortex-M0+ manuals. The SLEEPONEXIT bit provides the option to put the processor into low-power mode after an exception return, before the program resumes execution. This is ideal for applications that only need to be awake to service interrupts. Finally, the SLEEPDEEP bit allows entry into a deep sleep state, as opposed to the regular sleep state. The manufacturers of the chips that utilize the Cortex-M0+ core determine the exact behavior of the device in these states. In the case of ST, the sleep state is used as a base for both Sleep mode and Low-power sleep mode, while the deep sleep state is used as a base for Stop mode and Standby mode.

There are three ways to enter low-power mode on the Cortex-M0+. The first is to use the WFI (Wait For Interrupt) instruction. Just as the name suggests, if the device enters low-power mode as a result of this instruction, an interrupt (enabled in the NVIC) is capable of waking the device. The second way to enter low-power mode is by executing the WFE (Wait For Event) instruction. This is very similar to the WFI instruction, but offers more flexibility. Not only can the device be woken up by events configured in the extended interrupt and events controller (EXTI), but also by interrupts that are disabled in the NVIC (so long as they are enabled in the corresponding peripheral control register). The third way to enter low-power mode has already been mentioned. By setting the SLEEPONEXIT bit in the SCR, an exception return will cause the device to enter low-power mode as if the WFI instruction was executed. Note that in all of these cases, low-power mode will only be entered if no interrupt or event is pending. Because WFI and WFE are not guaranteed to halt program execution, they are often referred to as "hint instructions".

The final core register worth mentioning is the PRIMASK register. It contains only one configurable bit, PM (Prioritizable interrupt Mask), which will disable all interrupts with configurable priority if set to '1'. Not only can this be used to perform atomic operations, but also to delay execution of an ISR if the system first needs to be restored to a working state. An example of this will be provided in the section detailing Stop mode.

In order to allow the programmer easy access the WFI and WFE instructions when developing a C application, the CMSIS-CORE standard provides the `__WFI()` and `__WFE()`

functions. All of the example functions in the following sections use `__WFI()` to execute the WFI instruction and enter low-power mode. Also, rather than providing access to the PRIMASK register directly, CMSIS implements the `__disable_irq()` and `__enable_irq()` functions in order to set and clear the PM bit, respectively. In order to check the status of the PM bit, the `__get_PRIMASK()` function will return its current state. Most IDEs make it very simple to add CMSIS drivers to projects. For example, in Keil, be sure that ARM::CMSIS is installed with the Pack Installer and simply check the "CORE" pack (under the CMSIS component) in the Run-Time Environment Manager when creating a new project.

Low-Power Modes

The STM32L0 devices implement five low power modes: Low-power run mode, Sleep mode, Low-power sleep mode, Stop mode, and Standby mode. The differences between these modes can be described in terms of power consumption, performance, wake-up time, and wake-up sources. If for each of these parameters, the modes are put in order from best (1) to worst (5), it becomes clear what the trade-offs are. Generally speaking, as power consumption goes down; the performance decreases, the wake-up time increases, and the number of wake-up sources decreases. Table 1 summarizes the ranking of the low-power modes. As an example of interpretation, consider Low-power run mode. It has the best performance, the most wake-up sources, the second fastest wake-up time, and the fourth lowest current consumption.

Table 1: Ordinal ranking of the STM32L0 low-power modes based on various operating parameters

	LPRun	Sleep	LPSleep	Stop	Standby
Performance	1	2	3	4	5
Power Consumption	4	5	3	2	1
Wake-up Time	2	1	4	3	5
Wake-up Sources	1	2	3	4	5

Throughout this section, it will become clear how these rankings were derived. However, it is important to realize early on that they are only true in the general sense. For example, it is entirely possible for Stop mode to consume *more* current than Low-power sleep mode, depending on their configurations and which peripherals are enabled/disabled. But generally this will not be the case because Stop mode restricts the capabilities of the device far more than Low-power sleep mode does in order to conserve more power.

While this article does not give definitive values for the current that will be consumed in each mode, the Using the IDD Current Measurement Feature on the STM32L053 Discovery Board page does provide some example measurements for each low-power mode.

Low-Power Run Mode

It's rather deceitful to market this as a low-power mode because the main way it conserves energy is by requiring a low system clock frequency. Reducing the clock speed of *any* microcontroller to the kilohertz range will dramatically reduce current consumption to the point of being competitive with the average sleep mode. The reason this is not normally done, though, is because the reduction in performance along with the static current consumption (which is not dependent on the clock frequency) can use more energy in the long run. Depending on the application, i.e. which sleep mode is being used or how often the device wakes up, it may be more efficient to consume more current for a shorter period of time rather than consume less current for a longer period of time. The reason ST can get away with classifying this as a low-power mode is because they provide the ability to put the internal voltage regulator into a low-power state. This will reduce the static current consumed by the device, minimizing its impact on the trade-off between performance and total current consumption.

In order to switch the regulator into low-power mode, two conditions must be met. First, the regulator voltage (V_{CORE}) must be in range 2. Luckily, according the PWR_CR register documentation, this is the default configuration of the regulator. So, unless the device's dynamic voltage scaling features are being taken advantage of, there is no need to worry about this prerequisite. The second condition is that the system frequency does not exceed f_{MSI} range 1. According the description of the MSIRANGE bit (in the RCC_ICSCR register), this corresponds to a frequency of around 131.072 kHz. At this speed and power level, the USB, ADC, and TSC (Touch Sensing Controller) peripherals are not available. Any frequency dependent peripherals (USART, Timers, etc.) that were previously initialized in Run mode will have to be reinitialized after the system frequency is changed in order to continue operating properly.

Unlike the other low-power modes, the CPU is not stopped in Low-power run mode. This means it is not entered via the WFI/WFE instruction as previously discussed, but rather by setting the LPSDSR (Low-Power Sleep-Deep/Sleep/low-power Run) and LPRUN (Low-Power RUN) bits in the PWR_CR register. Note that LPSDSR must be set before setting LPRUN, LPRUN must be cleared before clearing LPSDSR, and LPRUN should be cleared before entering any other low-power mode. Since program execution continues in Low-power run mode, the device is "woken" by software rather than being limited to a finite set of interrupts or events. Simply clearing the LPRUN bit and bringing the system frequency back up to full speed will get the system back into Run mode. Listing 1 shows the entire procedure for entering Low-power run mode using the steps outlined in the reference manual. Listing 2 demonstrates how Run mode can be re-entered when the device no longer needs to be in Low-power run mode.

Listing 1: Example of entering Low-power run mode

```c
void enter_LPRun( void )
{
    /* 1. Each digital IP clock must be enabled or disabled by using the
                   RCC_APBxENR and RCC_AHBENR registers */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    /* 2. The frequency of the system clock must be decreased to not exceed the
          frequency of f_MSI range1. */
    Config_SysClk_MSI_131();
    // Reinitialize peripherals dependent on clock speed
    USART1_Init();
    SysTick_Init( 0.001 );
    I2C1_Init();
    /* 3. The regulator is forced in low-power mode by software
          (LPRUN and LPSDSR bits set ) */
    PWR->CR &= ~PWR_CR_LPRUN; // Be sure LPRUN is cleared!

    PWR->CR |= PWR_CR_LPSDSR; // must be set before LPRUN
    PWR->CR |= PWR_CR_LPRUN; // enter low power run mode
}
```

Listing 2: Example of entering Run mode

```c
void enter_Run( void )
```

```
{
    /* Enable Clocks */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;

    /* Force the regulator into main mode */
    // Reset LPRUN bit
    PWR->CR &= ~( PWR_CR_LPRUN );
    // LPSDSR can be reset only when LPRUN bit = 0;
    PWR->CR &= ~( PWR_CR_LPSDSR );
    /* Set HSI16 oscillator as system clock */
    Config_SysClk_HSI16();
    // Reinitialize peripherals dependent on clock speed
    USART1_Init();
    SysTick_Init( 0.001 );
    I2C1_Init();
}
```

Sleep Mode

The most straight forward of the low-power modes, Sleep mode offers the shortest wake-up time at the expense of using the most power. The datasheet states that with all peripherals disabled and a system frequency of 16 MHz, about 1 mA of current will be consumed. This is much higher than the other low-power modes, which can achieve values on the order of microamps or even nanoamps. The wake-up time, however, is almost ten times faster than the most competitive low-power mode. Table 2 shows the amount of time it takes for the device to wake up from each low-power mode and enter Run mode. The values for wake-up time are taken from Table 4 of the datasheet.

Table 2: Wake-up to Run mode times for each low-power mode

Low-power Mode	Wake-up Time
Low-power run	3 µs
Sleep	0.36 µs
Low-power sleep	32 µs
Stop	3.5 µs
Standby	50 µs

In Sleep mode, only the core is stopped while all peripherals continue to run. This makes it practically effortless to enter Sleep mode because the system frequency does not have to be decreased and all of the device's peripherals are available for use. Also, it is very easy to exit Sleep mode since any interrupt or event available in Run mode can wake the device and be serviced with very low latency. Thus, Sleep mode can be used in almost any situation where the CPU is in a spinlock waiting for an event to occur. Rather than enter a busy-wait loop, the user could simply execute WFI or WFE (depending on the wake-up method) to suspend execution and save power until the core is needed again. This works because the SCR is configured for Sleep mode by default, i.e. the SLEEPDEEP bit is cleared. For applications where the CPU is only needed for servicing interrupts, it makes more sense to set the SLEEPONEXIT bit and always enter Sleep mode after the interrupt(s) have been serviced, as opposed to resuming program execution.

Listing 3 is an example of a function that can be used to enter Sleep mode. Because this function is taken from a program that uses more than one low-power mode, the first statement ensures that the SLEEPDEEP bit is cleared to avoid unexpected behavior. Also, in order to avoid wake-up latency, the Flash access control register is configured to keep the non-volatile memory idle while the device is in Sleep mode. Stopping the Flash memory interface clock is discussed further in the Low-power sleep mode section.

Listing 3: Example of entering Sleep mode

```
void enter_Sleep( void )
{
    /* Configure low-power mode */
    SCB->SCR &= ~( SCB_SCR_SLEEPDEEP_Msk );  // low-power mode = sleep mode
    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;     // reenter low-power mode after ISR

    /* Ensure Flash memory stays on */
    FLASH->ACR &= ~FLASH_ACR_SLEEP_PD;
    __WFI();  // enter low-power mode
}
```

Low-Power Sleep Mode

Low-power sleep mode is essentially a combination of Low-power run mode and Sleep mode. Not only is the Cortex-M0+ core stopped, but the regulator is placed into low power mode, which means the same conditions as those from Low-power run mode must be met. Recall that V_{CORE} must be in range 2 (default configuration) and the system clock frequency must be decreased to no more than f_{MSI} range 1 (131.072 kHz). Because of this, the USB, ADC, and TSC peripherals are not available in this mode. Also, any frequency dependent peripherals that continue to run in Low-power sleep mode will have to be reinitialized so they continue to operate properly.

Unlike Low-power run mode, the LPRUN bit is not used to put the regulator into low-power mode. Once the system frequency is decreased, the LPSDSR bit should be set and the same procedure for entering sleep mode should be followed. That is, ensure that the SLEEPDEEP bit is cleared and execute the WFI instruction, the WFE instruction, or set the SLEEPONEXIT bit and wait for an exception return. The LPSDSR bit will automatically place the regulator into a low-power state when the device enters low-power mode. When the device exits low-power mode following a wake-up event, Run mode will be entered with the regulator on at full power.

The reference manual mentions the option of switching off the Flash memory in the Low-power sleep mode section. Setting the SLEEP_PD (SLEEP Power-Down) bit in the FLASH_ACR register will place the non-volatile memory in power-down mode when the device enters either Sleep mode or Low-power sleep mode. While this does increase the wake-up latency, the power consumption is reduced by about 12 µA (Table 34 in the datasheet), which may be significant depending on the application. The increased wake-up time is presumably the reason that no mention of this option is made in the Sleep mode section of the reference manual (even though it does work in Sleep mode). If the application utilizing Sleep mode doesn't need the incredibly fast wake-up time it offers, then Low-power sleep mode should probably be used instead. Listing 4 shows an example function used to enter Low-power sleep mode following the procedure laid out in the reference manual.

Listing 4: Example of entering Low-power sleep mode

```
void enter_LPSleep( void )
{
    /* 1. The Flash memory can be switched off by using the control bits
             (SLEEP_PD in the FLASH_ACR register). This reduces power consumption
             but increases the wake-up time. */
    FLASH->ACR |= FLASH_ACR_SLEEP_PD;
    /* 2. Each digital IP clock must be enabled or disabled by using the
               RCC_APBxENR and RCC_AHBENR registers */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    /* 3. The frequency of the system clock must be decreased to not exceed the
               frequency of f_MSI range1. */
    // Set MSI 131.072 kHz as system clock
    Config_SysClk_MSI_131();
    // Reinitialize peripherals dependent on clock speed
    USART1_Init();
    SysTick_Init( 0.001 );
    I2C1_Init();
    /* 4. The regulator is forced in low-power mode by software
               (LPSDSR bits set ) */
    PWR->CR |= PWR_CR_LPSDSR; // voltage regulator in low-power mode during sleep
    /* 5. Follow the steps described in Section 6.3.5: Entering low-power mode */
    SCB->SCR &= ~( SCB_SCR_SLEEPDEEP_Msk ); // low-power mode = sleep mode
    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk; // reenter low-power mode after ISR
    __WFI(); // enter low-power mode
}
```

Stop Mode

Arguably the most complicated low-power mode available in the STM32L0 series, Stop mode has the potential to achieve current consumption on the order of nanoamps while still retaining the SRAM and register contents. However, if wake-up time is more critical, many of the power saving options may be ignored in order to achieve latency on par with that of Low-power run mode. Further complicating matters are the limited number of available wake-up sources, the multiple mentions of Stop mode in the errata sheet, and the added debugging complexity. Nevertheless, Stop mode is likely the best option for those wanting to use the least amount of power while not having to reinitialize the system upon waking.

In Stop mode, the core is stopped and the only oscillators that are able to run are the LSE, LSI, and the HSI in a limited capacity. The low-speed clocks allow the RTC and IWDG to continue running and wake-up the device. The HSI can provide limited functionality to peripherals capable of running in Stop mode. For example, the USART and I^{2}C are still able to receive data in Stop mode by waking up the HSI when it is needed. The HSI will only feed the peripheral that requested it and will automatically be disabled when it is no longer needed. For a complete list of all peripherals that are functional in Stop mode as well as which wake-up sources are available, see Table 4 in the datasheet. Note that, because the core clock is stopped, a debugging connection will not be able to be sustained once Stop mode is entered. However, according the reference manual, setting the DBG_STOP bit in the DBGMCU_CR register will allow debugging in Stop mode.

In order to enter Stop mode, the SLEEPDEEP bit must be set since both Stop mode and Standby mode are implementations of the deep sleep state provided by the Cortex-M0+ core. Clearing the PDDS (Power Down DeepSleep) bit in the PWR_CR register is how Stop mode is chosen over Standby mode. Also, it is necessary to ensure that the WUF (Wake-Up Flag) bit in the PWR_CSR register is cleared. Unfortunately, this bit cannot be modified by software and must be cleared by writing a '1' to the CWUF (Clear Wake-UP Flag) bit in the PWR_CR register. This will clear WUF after 2 system clock cycles. Once these conditions are met, the user need only execute the WFI instruction, the WFE instruction, or set the SLEEPONEXIT bit and wait for an exception return. Note that by default, the device will select the MSI oscillator as the system clock when it wakes from Stop mode. By setting the STOPWUCK (STOP Wake-Up ClocK) bit in the RCC_CFGR register before entering Stop mode, the HSI16 oscillator will be selected as the system clock instead. Listing 5 shows an example function that will enter Stop mode using this bare minimum configuration. It also shows how to enable an external interrupt capable of waking the device.

Listing 5: Simple example of entering Stop mode

```
void enter_Stop( void )
{
    /* Enable Clocks */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;

    /* Configure PA0 as External Interrupt */
    GPIOA->MODER &= ~( GPIO_MODER_MODE0 ); // PA0 is in Input mode
    EXTI->IMR |= EXTI_IMR_IM0; // interrupt request from line 0 not masked
    EXTI->RTSR |= EXTI_RTSR_TR0; // rising trigger enabled for input line 0

    // Enable interrupt in the NVIC
    NVIC_EnableIRQ( EXTI0_1_IRQn );
    NVIC_SetPriority( EXTI0_1_IRQn, BTN_INT_PRIO );

    /* Prepare to enter stop mode */
    PWR->CR |= PWR_CR_CWUF; // clear the WUF flag after 2 clock cycles
    PWR->CR &= ~( PWR_CR_PDDS ); // Enter stop mode when the CPU enters deepsleep
    RCC->CFGR |= RCC_CFGR_STOPWUCK; // HSI16 oscillator is wake-up from stop clock
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // low-power mode = stop mode
    __WFI(); // enter low-power mode
}
```

If more power savings are desired and wake-up time is less of a concern, there are additional configurations that can be made before Stop mode is entered. The easiest one is to put the regulator into low-power mode by setting the LPSDSR bit in the PWR_CR register. In the same register, the ULP (Ultra-Low Power mode) bit can also be set in order to disable V_{REFINT} (internal voltage reference) if it is not being used by any of the analog peripherals. Furthermore, if V_{REFINT} is not needed immediately after waking, setting the FWU (Fast Wake-Up) bit will not add additional wake-up latency that would otherwise be caused by setting ULP.

The aforementioned changes will significantly reduce current consumption, but the device may still be drawing on the order on microamps. In order to get into the nanoamps range, it is necessary to place all of the GPIO pins into analog mode. According to section 9.3.12 of the reference manual, when an I/O pin is configured as analog, the Schmitt trigger input is deactivated, providing zero consumption for each pin. Doing this, however, means that the GPIOx_MODER register for each port will have to be saved before switching every pin to analog mode. This way, every pin can be restored to its previous mode as soon as the device wakes up. Also, in order to avoid unexpected errors, interrupts should be disabled during both the save and restore process.

The function in Listing 6 is built on the basic function in Listing 5. Not only is the regulator placed in low-power mode and V_{REFINT} switched off, but the I/O context is saved prior to entering Stop mode. Since interrupts were disabled when WFI was executed, i.e. the PM bit was set, the external interrupt will wake the device but its ISR will not be entered. Program execution instead continues from the WFI instruction allowing the context to be restored immediately. Once interrupts are re-enabled, the external interrupt's ISR will be entered, since the interrupt is still in the pending state.

Listing 6: Advanced example of entering Stop mode

```
void enter_Stop( void )
{
    /* Enable Clocks */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;

    /* Configure PA0 as External Interrupt */
    GPIOA->MODER &= ~( GPIO_MODER_MODE0 ); // PA0 is in Input mode
    EXTI->IMR |= EXTI_IMR_IM0;   // interrupt request from line 0 not masked
    EXTI->RTSR |= EXTI_RTSR_TR0; // rising trigger enabled for input line 0

    // Enable interrupt in the NVIC
    NVIC_EnableIRQ( EXTI0_1_IRQn );
    NVIC_SetPriority( EXTI0_1_IRQn, BTN_INT_PRIO );

    /* Prepare to enter stop mode */
    PWR->CR |= PWR_CR_CWUF;      // clear the WUF flag after 2 clock cycles
    PWR->CR &= ~( PWR_CR_PDDS ); // Enter stop mode when the CPU enters deepsleep
    // V_REFINT startup time ignored | V_REFINT off in LP mode | regulator in LP mode
    PWR->CR |= PWR_CR_FWU | PWR_CR_ULP | PWR_CR_LPSDSR;
    RCC->CFGR |= RCC_CFGR_STOPWUCK; // HSI16 oscillator is wake-up from stop clock
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // low-power mode = stop mode

    __disable_irq();

    Idd_SaveContext();
    I2C1->CR1 &= ~I2C_CR1_PE;  // Address issue 2.5.1 in Errata
    __WFI(); // enter low-power mode

    I2C1->CR1 |= I2C_CR1_PE;
    Idd_RestoreContext();

    __enable_irq(); // <-- go to isr
}
```

Both Listings 5 and 6 use an external interrupt from pin PA0 to wake the device. This is because external interrupts are controlled by the EXTI and, other than a system reset, only it can cause the device to exit Stop mode. In short, the EXTI manages 30 internal and external lines capable of generating interrupts and events. Table 52 in the reference manual specifies the line source for each line and whether they are configurable or direct lines. In the above examples, EXTI line 0 is by default mapped to PA0 so the line itself did not have to be configured. All that had to be done was enable the line as an interrupt source, choose how the interrupt would be triggered, and enable the corresponding interrupt in the NVIC. Had Stop mode been entered using the WFE instruction, the line would have been enabled in event mode.

Table 3 summarizes the limitations of Stop mode documented in the STM32L053x6/8 Errata sheet. Clearly, no matter what chip revision is being used, there are issues that should be addressed by the programmer in order to avoid unexpected errors. The function in Listing 6 implements the workaround suggested for the issue caused by not configuring the I^{2}C peripheral to wake the device from Stop mode. Section 2.5.1 in the Errata explains that disabling the I^{2}C peripheral before entering Stop mode and re-enabling it immediately after wake-up avoids any errors. Luckily, that workaround can be done in conjunction with saving and restoring the I/O context.

Table 3: Documented limitations of Stop mode in the STM32L053x6/8 Errata

Section	Issue	Revision A	Revision Z	Revision Y	Revision X
Section 2.1.2	Exiting Stop mode on a reset event is not possible when HSI16 is the clock system and it is selected as wake-up clock	Workaround Available	Fixed	Fixed	Fixed
Section 2.1.10	Flash memory wake-up issue when waking up from Stop or Sleep with Flash in power-down mode	Workaround Available	Workaround Available	Fixed	Fixed
Section 2.1.11	Unexpected system reset when waking up from Stop mode with regulator in low-power mode	Workaround Available	Workaround Available	Fixed	Fixed
Section 2.1.12	I^{2}C and USART cannot wake up the device from Stop mode	No Workaround Available	No Workaround Available	No Workaround Available	Fixed
Section 2.5.1	Wrong behaviors in Stop mode when waking up from Stop mode is disabled in I^{2}C peripheral	Workaround Available	Workaround Available	Workaround Available	Workaround Available

Standby Mode

Unlike Stop mode, entering Standby mode is quite simple because the user has fewer options. The only oscillators available are the LSI and LSE, the only peripherals that can function are the RTC and IWDG, the voltage regulator is completely disabled, and all I/O pins are set as high impedance (so saving the context is pointless). All the user has to do to enter Standby mode is set the SLEEPDEEP bit, set the PDDS bit, and be sure that the WFU bit is cleared by writing a '1' to the CWUF bit. Then, executing the WFI instruction, the WFE instruction, or setting SLEEPONEXIT before an exception return will cause the device to enter this low-power mode. There are also fewer options for exiting Standby mode. Only a rising edge on a wake-up pin (PA0 or PC13), one of the RTC wake-up events, or a IWDG reset will wake the device. The method of choice must be configured and the corresponding wake-up flags cleared before Standby mode is entered. Note that the STM32L053C8 chip only has two wake-up pins whereas the reference manual and datasheet will often mention a third wake-up pin only available on the packages with higher pin counts.

The biggest problem with Standby mode is that it does not preserve the contents of SRAM or the registers (except the RTC registers, RTC backup registers, and the Standby circuitry). After waking from Standby mode, the program execution restarts in the same way as if a reset had occurred. In order to determine whether or not the system was woken from Standby mode, the SBF (StandBy Flag) bit in the PWR_CSR register can be checked as soon as the `main()` function is entered. If this bit is set, then the device was previously in Standby mode and the CSBF (Clear StandBy Flag) should be set in order to clear SBF. The user then knows if the system should be initialized for a first time run or the if the system should be restored to its previous state.

Notice in Table 2 that Standby mode has the longest wake-up time. This estimate does not include the amount of time required to reinitialize the system and resume program execution. Not only does this added latency make immediate responses to wake-up events next to impossible, it can severely limit the power saving capabilities of Standby mode. In order to keep the average current consumed below that of Stop mode, the device will have to be in Standby mode for a very long period of time. Just how long depends on how much current is consumed during the wake-up/reinitialization process. Listing 7 shows an example function that may be used to enter Standby mode where either of the wake-up pins will wake the device.

Listing 7: Example of entering Standby mode

```
void enter_Standby( void )
{
    /* Enable Clocks */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;

    /* Prepare for Standby */
    // if WKUP pins are already high, the WUF bit will be set
    PWR->CSR |= PWR_CSR_EWUP1 | PWR_CSR_EWUP2;

    PWR->CR |= PWR_CR_CWUF; // clear the WUF flag after 2 clock cycles
    PWR->CR |= PWR_CR_ULP;   // V_{REFINT} is off in low-power mode
    PWR->CR |= PWR_CR_PDDS; // Enter Standby mode when the CPU enters deepsleep

    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // low-power mode = stop mode
    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk; // reenter low-power mode after ISR
    __WFI(); // enter low-power mode
}
```

Conclusion

For ultra-low-power applications like gas/water meters, wearables, or IOT sensors; the STM32L family of microcontrollers feature many low-power modes that allow users to carefully balance performance with power consumption. In particular, the STM32L0 series utilizes the most energy efficient ARM processor (the Cortex-M0+) and provides five low-power modes to suit most entry-level applications.

Low-power run mode yields the best performance while still consuming less current than sleep mode. Because the CPU is kept running, the decision to wake the device is made by software, providing the most flexibility in that respect. Unfortunately, having the regulator in low-power mode introduces a wake-up latency that is too great for many real-time applications. This is where Sleep mode really excels, boasting a wake-up time of 0.36 µs as a result of allowing all peripherals to run at full speed while the core is disabled. However, this also makes it the most power hungry low-power mode, which the peripheral performance and full set of hardware wake-up sources do little to compensate for. Low-power sleep mode uses the same approach as Low-power run mode to reduce the current consumption to levels comparable to Stop mode without limiting the number of wake-up sources. If it is sufficient to have the EXIT wake the device, however, Stop mode provides the lowest current consumption while still preserving the SRAM and register contents. It also has a wake-up time comparable to Low-power run mode. Finally, for applications where the MCU is not needed for very long periods of time, Standby mode may be appropriate. It uses the least amount of power at the expense of having very few wake-up sources and the longest wake-up time.

In order to see the example functions presented above used in a functional application, please visit the following page: Using the IDD Current Measurement Feature on the STM32L053 Discovery Board. This application will allow the user to enter the desired low-power mode in order to view the current consumed when in that state. The complete code is available, including the functions used within the above examples, e.g. `Config_SysClk_HSI16()`, `Idd_SaveContext()`, etc. Also, Table 1 on that page summarizes the average current consumption in each mode when these functions are used to enter them.

Contact the Author

The STM32L0 series offers more low-power modes than the average MCU as well as more flexibility in configuring them. Using these in combination with the low-power peripherals can considerably extend battery life and reduce energy harvesting requirements. Hopefully, enough information has been provided to help you get started with developing low-power applications using any MCU in ST's ultra-low-power family. If you have any questions or would like to provide feedback, feel free to send an email to eewiki@digikey.com or visit Digi-Key's TechForum.

无标签

ATLASSIAN