Freescale Semiconductor, Inc.
Application Note

Document Number: AN5208
Rev. 0, 11/2015

Software-friendly Method of Using VLLS3 Power Mode with Kinetis L Microcontrollers

Željko Stefanović

1. Introduction

Kinetis microcontrollers include a rich portfolio of power modes enabling an application to balance performance and energy consumption. Application notes listed in the reference section cover these topics extensively.

Very Low–Leakage Stop3 (VLLS3) is the lowest power mode Kinetis L uses to maintain a complete SRAM powered on while consuming a few micro-amps. This helps any application including those requiring a lot of SRAM, such as communication stacks keep their data safe and available at all times.

VLLS3 is one of the Very Low Leakage Stop (VLLSx) low power modes. The device waking up from a VLLSx mode goes through a reset cycle, contrary to waking up from a non-VLLSx low power mode when the device resumes code execution where it left off. Writing software to handle VLLS3 wake-up resets, including bypassing the SRAM initialization, is not always easy within the given Integrated Development Environment (IDE) and can even prevent some users from capitalizing on VLLS3's outstanding low current consumption.

This application note introduces a simple interface between the Kinetis L application and the VLLS3 low power mode. It makes waking up via the VLLS3 reset resemble an ordinary interrupt, looking like as if the application were picking up from where it left off.

Contents

2. VLLS3 Basics

VLLS3 power mode key features are:

- VLLS3 can be entered from any MCG mode.
- Most peripherals are disabled (with clocks stopped). However, OSC, LLWU, LPTMR, RTC, CMP, and TSI can be used.
- NVIC is disabled. LLWU is used to wake up the device (device exits in FEI mode -reset state).
- SRAM_U and SRAM_L remain powered on (content retained and I/O states held).

The MCU enters VLLS3 mode if:

- SMC_PMPROT[AVLLS]=1, SMC_PMCTRL[STOPM]=4, SMC_STOPCTRL[VLLSM]=3; and
- Sleep-now or sleep-on-exit mode is entered with SLEEPDEEP set (controlled by ARM core's System Control Register)

In VLLS3 the on-chip voltage regulator is in stop-regulation state while most digital logic is powered off. Before entering VLLS3 mode, configure the Low-Leakage Wake-up (LLWU) module to enable the desired wakeup source(s). The device's reference manual configuration chapter specifies wake-up sources available in the VLLS3 mode.

After waking up from VLLS3, the device returns to normal RUN mode with RCM_SRS0[WAKEUP]=1 (indicating a wake up reset) and a pending LLWU interrupt. In the LLWU interrupt service routine (ISR) the user can poll LLWU flags to determine the wake-up source.

An asserted RESET pin causes the device to exit the VLLS3 mode. In this instance, the device returns to normal RUN mode with RCM_SRS0[WAKEUP]=1 and RCM_SRS0[PIN]=1.

After VLLS3 mode is invoked, each I/O pin is latched as configured before the microcontroller enters this mode. While in VLLS3, the device's digital logic is turned off and all port and peripheral data is lost. This information must be restored before PMC_REGSC[ACKISO] is cleared following wake up.

If the oscillator module is used to create the application clock(s), the oscillator pins EXTAL0/XTAL0 must be reconfigured first, followed by clearing the ACKISO bit and setting up the oscillator parameters (e.g., RANGE0, HGO0, ERCLKEN, and so on.) for the oscillator to start running. Only with the oscillator running the MCG mode can be successfully updated. Attempting to clear the ACKISO bit at the end of the MCG setup sequence prevents the oscillator from running and causes the application to indefinitely wait for the MCG to update.

3. Power-up and Wake-up Reset Handler Details

The subsequent sections describe power-up and wake-up reset handlers using a Kinetis L application written in C with the Multipurpose Clock Generator (MCG) mode set to FEI, so that application clock(s) can be set before clearing the ACKISO after the system wakes-up. For details about how using VLLS3 affects other MCG modes, see the Appendix.

3.1. Power-up reset handler

A power-up event results in a system reset with the RCM_SRS0[POR]=1. Software can use this flag to test if the power-up reset handler is to be executed or not.

A typical Kinetis L power-up reset handler does the following:

1. Ensures that Watchdog (WDOG) is operating properly. The WDOG is often disabled in software after a power-up to make code development and debugging easier.
2. Calls the SystemInit() function. This routine sets up the MCG and configures essential clocks (i.e., Core/system/Bus/Flash clocks, and so on).
3. Initializes the SRAM. Depending on the application requirements, portions of the SRAM are allocated to the stack(s), heap, and user application data and variables.
4. Jumps to the user-supplied main() function routine.

In most use cases, step 1 provides two possible choices, which involve either disabling the WDOG or feeding it. Step 2 can be as simple as not doing anything if the desired MCG mode and essential clock configuration match the default/reset setup, for example KL26 resets MCG to the FEI mode with Core/system clock of 21 MHz and Bus/Flash clock of 10.5 MHz.

While steps 1 and 2 are directly controlled by the user (and quite often the user is allowed to enter his own code there), steps 3 and 4 are almost exclusively provided by the IDE. After the programmer defines size for the stack(s) and heap, the tool allocates the SRAM accordingly and initializes global variables (if any) and other parameters.

3.2. VLLS3 wake-up reset handler

A wake-up event causes a system reset with the RCM_SRS0[WAKEUP]=1. By testing this flag software can recognize a wake-up reset and proceed with executing the wakeup-up reset handler.

With the on-chip SRAM retained all the way to VLLS3, the application can store CPU parameters, the CONTROL and core's r0-r14 registers, into the SRAM before entering the VLLS3 power mode. After a wake-up reset occurs, the reset handler does the following:

1. Services the WDOG if needed.
2. Restores the MCG configuration (i.e., calls the SystemInit() function).
3. Selects the stack (chooses between the main and process stack by looking at the CONTROL register) and restores core registers (r0-r14). After this step is complete, the application continues running from the point when the VLLS3 was invoked.

Following step 3, the application still has to reconfigure the peripherals and ports to prevent glitches on pins after the isolation is acknowledged (writing 1 to PMC_REGSC[ACKISO]). For details on the mechanism behind the ACKISO bit and how to handle it, see device's reference manual.

Ideally, this handler does not provide any SRAM initialization because the SRAM is retained in VLLS3.

3.3. One handler for power-up and wake-up resets

Quite often the user's application gets a very simple power-up reset handler if crafting of the reset handler is left to the IDE. However, adding support for VLLS3 wake-up resets is not a problem at all.

Thanks to modern assemblers' ability to declare a portion of the code as "weak", the software developer can replace tool-generated (i.e., the default) code with user-provided code.

Exception and interrupt handlers are the prime beneficiaries of the "weak" declaration. Most IDEs generate default handlers used to automatically populate entries into the NVIC Vector Table. However, if the user provides a different routine to manage a particular fault or an interrupt handler, the tool picks-up the user-provided code when generating an application image.

The default code can be replaced with software written in any programming language supported by the IDE. The utility for handling VLLS3 wake-up resets described in this application note is partially written in assembly language and a partially in C.

4. VLLS3 utility implementation details

This solution was tested with IAR 7.0, KDS 3.0.0 and Keil/ARM® µVision 5.15 using a FRDM-KL26Z Freescale Freedom development platform. As a result of unique characteristics of three different assemblers, this application note is accompanied by dedicated *.s files and tool-specific projects. Code examples listed in the latter part of this document are based on a generic assembly language as an illustration of the actual implementation.

Adding support for VLLS3 wake-up resets becomes as simple as including a small group of source files into a user's project and rebuilding it. The default reset handler gets replaced with the code that can process both power-up and wake-up resets. This utility allocates a small portion of the SRAM (16 bytes) to track CPU's operating parameters when the device switches between power modes.

Add-on source files are:

- vlls3_utility.s
- vlls3_utility.c and vlls3_utility.h
- kill_switch.c and kill_switch.h

4.1. File: vlls3_utility.s

NOTE

Modify this file only if adding a new feature to the VLLS3 utility.

This assembly language file provides two routines, Reset_Handler() and VLLS3_UTILITY_WFI(). The best way to explain their interaction is to start with the code that the user writes to invoke the VLLS3 mode:

Example 1. Entering VLLS3 using the new WFI routine

```
//Allow access to VLLSx modes
SMC->PMPROT = SMC_PMPROT_AVLLS_MASK;

//Next low power mode to be entered is VLLSx (select VLLS3)
SMC->PMCTRL = (SMC->PMCTRL & ~SMC_PMCTRL_STOPM_MASK) | SMC_PMCTRL_STOPM(4);
SMC->STOPCTRL = (SMC->STOPCTRL & ~SMC_STOPCTRL_VLLSM_MASK) | SMC_STOPCTRL_VLLSM(3);

//Sleep-now or sleep-on-exit modes entered with SLEEPDEEP set,
//which is controlled in System Control Register in ARM core.
SCB->SCR ~= SCB_SCR_SLEEPONEXIT_Msk;
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;

//Enter VLLS3 mode
VLLS3_UTILITY_WFI();
```

The only difference between this and any other sequence invoking the VLLS3 mode is the call to the VLLS3_UTILITY_WFI() function instead of __WFI() function.

The VLLS3_UTILITY_WFI() function does the following:

1. Pushes r0-r14 onto the stack.
2. Saves CONTROL register with bit 0 set to 0 (forcing post-wake up Thread mode to Privileged).
3. Saves Main Stack pointer (MSP).
4. Saves Process Stack Pointer (PSP).
5. Saves “go-to” address the MCU jumps to once the VLLS3 wake-up reset code finishes.
6. Executes WFI.

Example 2. VLLS3_UTILITY_WFI (generic assembly)

```
                PUSH    {r0-r7}                 ;Save r0-12,14 onto stack
                MOV     r0, r8
                MOV     r1, r9
                MOV     r2, r10
                MOV     r3, r11
                MOV     r4, r12
                MOV     r5, r14
                PUSH    {r0-r5}

                LDR     r0, =VLLS3_parameters   ;Save SPSEL and set future nPRIV=0
                MRS     r1, CONTROL
                LDR     r2, =0x02
                ANDS    r1, r1, r2
                STR     r1,[r0]

                ADDS    r0, r0, #4              ;Save MSP
                MRS     r1, MSP
                STR     r1,[r0]

                ADDS    r0, r0, #4              ;Save PSP
                MRS     r1, PSP
                STR     r1,[r0]

                ADDS    r0, r0, #4              ;Save recovery PC
                LDR     r1, =VLLS3_RECOVERED
                LDR     r2, =0x01
                ORRS    r1, r1, r2
                STR     r1,[r0]

                dsb                             ;As per ARM Application Note 321
                wfi                             ;Enter selected low power mode

VLLS3_RECOVERED…                                ;Jump here once wakeup reset handler finished
```

Core's r0-r14 are saved onto the stack while the CONTROL, MSP and PSP registers and the "go-to" address are saved into a dedicated four 32-bit word array called VLLS3_parameters[]. The latter four words are of crucial importance for the microcontroller's operations after a wake-up and their storage address must be known at all times.

Even though the SRAM is retained in VLLS3, it is because of the wake-up reset that information on the stack selection (Main vs. Process) and the stack pointer values is lost. After any reset, by default, the Kinetis L MCU starts running in the Privileged mode and uses the Main Stack Pointer. The stack pointer is initialized based on the first entry into the core's vector table (typically it is the 32-bit word at the bottom of the flash). This is how any stack pointer setup information that was there before VLLS3 was entered is voided.

4.1.1. New wake-up reset handler details

Example 3. Reset Handler servicing both power-up and wake-up resets (generic assembly)

```
Reset_Handler
             ;check if this is a VLLSx wake-up (RCM_SRS0[WAKEUP]=1)
             LDR    r0, =0x4007F000
             LDR    r0, [r0]
             LDR    r1, =0x01
             ANDS   r0, r0, r1
             BEQ    NONVLLSX_WAKEUP

;this is a VLLSx wake-up reset => set the clocks and jump to where you left
VLLSX_WAKEUP
             LDR    r0, =SystemInit
             BLX    r0
             LDR    r0, =VLLS3_parameters+12
             LDR    r0, [r0]
             BX     r0

;this is a non VLLSx wake-up reset => check the kill switch, set clocks and go to main
NONVLLSX_WAKEUP
             LDR    r0, =_kill_switch
             BLX    r0
             LDR    r0, =SystemInit
             BLX    r0
             LDR    r0, =__main
             BX     r0
             BX     lr
```

Following a reset, the Reset_Handler() starts running by choosing between the power-up and the wake-up branch (based on the content of the RCM_SRS0 register at 0x4007_F000).

In case of a wake-up reset (RCM_SRS0[WAKEUP]=1), the SystemInit() routine is called where the WDOG, MCG, and essential clocks are set and using the “go-to” address the MCU continues executing code past the WFI call. As mentioned earlier, the “go-to“ address is one of the entries in VLLS3_parameters[] array.

The “go-to” location sits just after the WFI call within the VLLS3_UTILITY_WFI() routine. The post-WFI code restores the CONTROL, MSP, and PSP registers using data from VLLS3_parameters[]. At this point, the active stack setup is restored and registers r0-r14 can be pulled from the stack. The only thing left to do is for the code to return from the (VLLS3_UTILITY_WFI) routine.

Example 4. VLLS3_RECOVERED (generic assembly)

```
VLLS3_RECOVERED

             LDR    r0, =VLLS3_parameters      ;Restore stack (stay in the Privileged mode)
             LDR    r1, [r0]
             MSR    CONTROL, r1
             ISB                               ;As per ARM Application Note 321

             ADDS   r0, r0, #4                 ;Restore MSP
             LDR    r1, [r0]
             MSR    MSP, r1

             ADDS   r0, r0, #4                 ;Restore PSP
             LDR    r1, [r0]
             MSR    PSP, r1
```

```
POP     {r0-r5}                     ;Restore r0-12,14
MOV     r8, r0
MOV     r9, r1
MOV     r10, r2
MOV     r11, r3
MOV     r12, r4
MOV     r14, r5
POP     {r0-r7}

BX      lr
```

The CONTROL register is important because it selects the stack to be used in the Thread mode (when the core is not in the Handler mode). The same register sets the Thread mode privilege level.

Many applications run in the Privileged mode using the MSP (IDEs usually default to this combination). Since the ARM Cortex®-M0+ core lets the software developer choose between the MSP and PSP, this VLLS3 utility restores the user's choice, whatever that choice may be.

On the other hand, the MCU is kept in Privileged mode after a VLLS3 wakeup reset, even if a call to VLLS3_UTILITY_WFI() was made from the Unprivileged mode.

Some of the Cortex-M0+ core features, such as the SysTick, can be configured exclusively from the Privileged mode and these features might have to be set following the return from VLLS3_UTILITY_WFI(). Therefore, if the user's application needs to configure such feature after every wake-up reset, it can do it before the application goes back to the Unprivileged mode. If an application only runs in Privileged mode, then no change to the Thread mode privilege level is needed.

CAUTION

Before returning from the VLLS3_UTILITY_WFI(), the utility restores the stack pointer according to the user's selection prior to entering the VLLS3. However, it is the user's responsibility to switch back to the Unprivileged mode if required by the application.

4.1.2. New power-up reset handler

This utility introduces a new routine called "kill_switch()" in front of a conventional SystemInit() to __main sequence of calls.

The kill_switch() routine is a safety mechanism that becomes handy when the device enters a state from which it can't be reliably recovered by the debugging tool (CMSIS-DAP, P&E Micro, and so on).

This demo's kill_switch() subroutine uses two GPIO pins, one configured as an output and one as an input. A sequence of pulses is generated on the output pin and the code loops until the same sequence is detected on the input pin. It works best with a pair of adjacent physical pins equipped with a jumper. If these pins are shorted, the MCU quickly leaves this routine. Otherwise, the routine loops indefinitely.

In a system with scarce GPIOs, a simple way to implement this safety mechanism involves using the NMI pin. Having this pin grounded at power-up forces the microcontroller to enter the NMI interrupt handler and stay there indefinitely, never reaching the rest of the application.

This safety feature is an exclusive part of the power-up handler. If at any point in time control over the device is lost, remove the the kill_switch() jumper, power cycle the setup and the MCU will be waiting for the pattern on the input pin and the debugging tool will be able to restore control.

After the application is fully tested, the kill_switch() can be completely removed from the project.

4.2. Files: vlls3_utility.c and vlls3_utility.h

NOTE

Modify these files only if adding a new feature to the VLLS3 utility.

The four 32-bit word array VLLS3_parameters[] is defined and VLLS3_UTILITY_WFI() is made available for the rest of the application.

4.3. Files: kill_switch.c and kill_switch.h

These files add the kill switch feature to the project.

The header file is where the pair of GPIO pins is specified. It can be modified according to pins available in the setup.

The signal pattern implemented in the C file is a simple pulse integrator. After changing the output pin level, the input pin is probed. If the change is not detected, the integrator count is set to 0. Otherwise, the integrator count is incremented by one. After reaching the predefined count, the routine restores two pins' default configuration and returns to the application.

5. Conclusion

This application note introduces a software approach that simplifies coding of the power-up and wake-up reset handlers in Kinetis L systems that make use of the VLLS3 power mode. IAR, KDS IDE, and Keil demo projects are provided to illustrate basic steps needed by the application to accomplish this.

The SystemInit() routine assumes a Kinetis L setup with clock(s) derived from the FEI mode. The routine also works well with the BLPI and FBI modes.

For details about addressing other MCG modes while making the most of VLLS3, see the Appendix.

Appendix A. VLLS3 wake-up reset and MCG modes

From the VLLS3 point of view, the biggest difference between external oscillator-driven MCG modes (the “xyE” MCG modes: FEE, FBE, BLPE, PBE, PEE) and IRC based configurations (the “xyI” MCG modes: FEI, FBI, BLPI) is the wake-up sequence needed to restore the MCG mode, reconfigure pins, and set up the peripherals in a glitch-free manner.

For “xyE” modes, the oscillator pins are configured before setting up the oscillator and updating the MCG. Based on the utility from this document, an “xyE” VLLS3 wake-up code can do the following:

1. Handles the WDOG.
2. Jumps to the post-WFI code.
3. Restores pins and peripherals.
4. Acknowledges the isolation.
5. Sets-up the MCG.

“xyI” modes allow the MCG to be updated before the ACKISO bit is cleared. These systems can use the following wake-up sequence:

1. Handles the WDOG.
2. Sets-up the MCG.
3. Jumps to the post-WFI code.
4. Restores pins and peripherals.
5. Acknowledges the isolation.

Regardless of the desired MCG mode, port pins and peripherals must be reconfigured before the ACKISO bit is cleared for a guaranteed glitch-free operation.

Setting up the MCG is not a trivial task and a carefully crafted sequence of steps is needed for the MCG to transition to the desired mode. This is why IDEs provide support for the most common setups. In general, support for MCG is embedded within the SystemInit() routine.

As shown in earlier chapters, a SystemInit() routine handling the WDOG and MCG is a perfect fit for Kinetis L VLLS3 applications with the MCG mode set to “xyI”.

One may have to deal with glitches if an IDE provided SystemInit() is used to support “xyE” MCG modes in an embedded system alternating between VLLS3 and RUN modes. This is because for an “xyE” mode to be successfully deployed following a wakeup reset, the oscillator has to be enabled first and ACKISO cleared. However, clearing the ACKISO bit before the GPIOs and peripheral pins are configured can cause glitches on the pins used by the application. Since many IDEs place the call to SystemInit() way ahead of the user code this scenario is quite common.

One way to allow a Kinetis L setup to combine the VLLS3 low power mode with any MCG mode is to implement the following steps after a wake-up reset event:

1. Handle the WDOG while keeping the MCG in the default (and that is the FEI) mode
2. Jump to the post-WFI code

3. Restore pins and peripherals
4. Acknowledge the isolation
5. Set up the MCG to the desired mode
6. Continue with the application

The same setup can do the following at power-up:

1. Handle the WDOG while keeping the MCG in the default (FEI) mode
2. Set up the SRAM
3. Set up the MCG to the desired mode
4. Configure pins and peripherals
5. Run the application

In most use cases, the IDE provided SystemInit() can be easily modified to keep the MCG in its default (FEI) mode. Setting up the MCG to a particular “xyI” or “xyE” mode can be handled in a standalone routine. Examples that show how to do this can be found on Freescale’s homepage and in the Freescale online community.

The rest of the wakeup and reset handler’s steps are basic building blocks of any application. The developer handles configuration/restoration of the port pins and peripherals, the IDE is in charge of the SRAM initialization, and the VLLS3 utility helps with jumping to the post-WFI code.

6. References

Please see freescale.com/kinetis for the detailed documentation and updates.

- Kinetis L – Most energy efficient microcontrollers, up to 48 MHz, 8KB-256KB Flash, up to 32 KB RAM, low power timer and smart peripherals.
- MCU Reference Manuals: The reference manuals contain MCU-specific implementation details in the Chip Configuration chapters and the module-specific sections. Look for a detailed description of the Clocks, Resets and Power Management Features of each MCU.
- MCU Data Sheet Specifications: The data sheet includes all of the MCU specifications, including clock rates, low-power mode power consumption expectations.
- *Power Management for Kinetis MCUs* (document AN4503)
- *Using Low Power modes on Kinetis family* (document AN4470)
- *ARM Application Note 321: ARM Cortex™-M Programming Guide to Memory Barrier Instructions*

7. Revision History

The following table summarizes content changes since the previous release of this document.

Table 1. **Revision history**

Revision number	Date	Substantive changes
0	11/2015	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale 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.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale 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 Freescale 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. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Document Number: AN5208
Rev. 0
11/2015