Application Report
SLAA294B–March 2006–Revised March 2018

Software Coding Techniques for MSP430™ MCUs

Keith Quiring *MSP430 Applications*

ABSTRACT

This application report describes software techniques and related topics of interest to programmers of MSP430™ MCUs. Code examples are available from http://www.ti.com/lit/zip/slaa294.

Contents

Trademarks

1 Introduction

This application report covers software techniques that are widely applicable in applications based on MSP430 MCUs. Some of these techniques should be used in every program, while some are situation dependent. All are designed to save time for the developer or increase system robustness.

The first part of the document discusses the standard interrupt-based code flow model for MSP430 MCUs, recommended for the vast majority of applications. The next part discusses techniques that should be considered when developing any application based on an MSP430 MCU. Using these methods can greatly reduce debug time or provide additional robustness in the field. They include initialization procedures, validation of supply rails before performing voltage-sensitive operations, and use of special functions.

As with all application reports for MSP430 MCUs, this document is designed to support the user's guides, so also see the relevant family user's guide while reading this report.

2 Top-Level Code Flow for MSP430 MCUs

Most software applications are best served by adhering to a flow similar to the one in Figure 1. This flow is designed to maximize power efficiency.

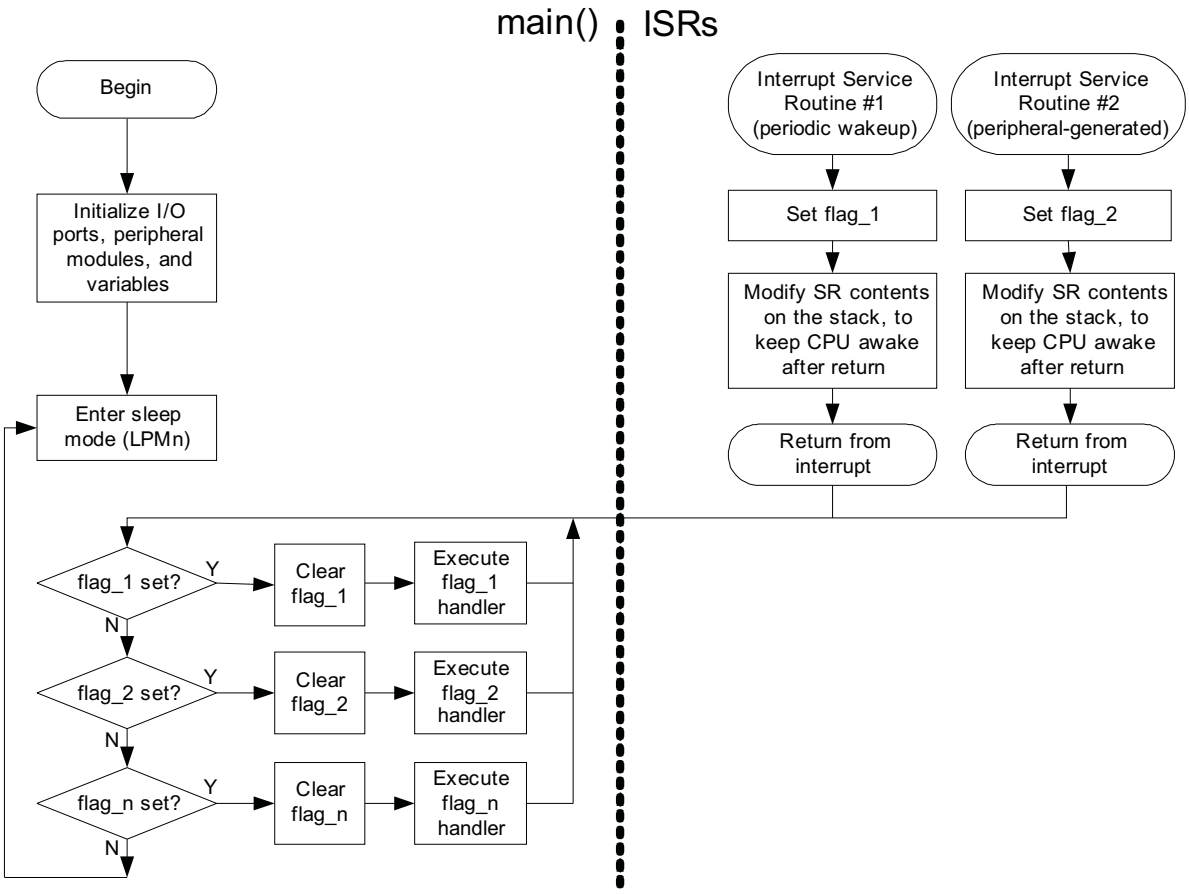

Figure 1. Top-Level Code Flow

The code architecture is interrupt driven, because doing so provides the most opportunities to power down the device. The device sleeps until an interrupt is received, thereby maximizing power efficiency.

To understand the way the interrupt service routines (ISRs) in Figure 1 are implemented, it is beneficial to review the way in which the MSP430 MCU manages low-power modes. The power modes are controlled by bits within the status register (SR). The advantage of this is that the power mode in place before ISR execution is saved onto the stack. When the ISR reloads that value upon completing execution, program flow returns to that saved power mode. However, by manipulating the saved SR value on the stack from within the ISR, program flow after the ISR can be diverted to a different power mode.

This mechanism is an integral part of low-power operation, because it allows the device to quickly wake up in response to an interrupt. As an example, suppose a device is in the LPM0 low-power mode when an interrupt occurs. The device prepares for ISR execution, including the saving of the SR to the stack and clearing the SR. Clearing the SR causes an exit from LPM0 into active mode. Within the ISR, the code developer places a statement that modifies the saved SR value by clearing the low-power bits. When the ISR completes, it reloads the values from the stack to their respective registers. Without having modified the bits, this action would put the device back into LPM0. Because the SR value has been modified to reflect a fully active device, the device stays active, and execution resumes at the PC value that had been saved to the stack prior to ISR execution.

Given this ability to change the power mode from the ISR, the developer can implement the full functionality of the ISR within the routine itself, or use the ISR to wake up the processor and let the main loop handle all or part of the resulting functionality. Handling within the ISR ensures that the response to the interrupt event is immediate, provided that interrupts are enabled at the time of the event. However, while handling an ISR, interrupts are not enabled and will not be enabled until the ISR is completed. As a result, long ISRs may decrease system responsiveness. The developer must choose which of these options best fits the application.

Applying this understanding to the flow in Figure 1, the figure shows two interrupts that allow their functionality to be handled within the main loop. These ISRs execute two tasks. First, they change the saved SR value on the stack to reflect a device in active mode. This allows one run through the main loop, before returning to sleep again. The interrupt can be any applicable event, such as a timer, pushbutton, or completion of an analog-to-digital conversion. The second task of the ISR is to set a flag that communicates to the main loop what action needs to be taken.

If the action to be taken in response to an interrupt is brief enough to be placed within the ISR itself, there may be no need to handle it in the main loop. In this case, there is no need for the ISR to set a flag or alter the SR power mode bits. The CPU would return to sleep upon exiting the ISR.

This flow can be adapted according to the complexity of the application. For example, if only one interrupt has the ability to wake the main loop, a flag system is unnecessary. In this case, the interrupt wakes the main loop, and the main loop performs its one function and puts the CPU back to sleep. An application that requires the CPU never sleep may have no need for interrupts at all.

The generalized sleep mode "LPMn" is shown in Figure 1. The actual mode to be used depends on the modules that must stay awake during the sleep mode. If a timer is responsible for waking up the device, and the timer is driven by ACLK, then ACLK must be kept active; LPM3 can be used. However, if the timer is driven from the DCO, then LPM0 must be used.

All the techniques in this application report assume that some version of this code structure is in place. Nearly every piece of code provided by TI, whether in the code examples or application reports, reflects this architecture and can be referenced for further study.

It is beneficial to gain a solid understanding of this interrupt-driven code flow as it pertains to a particular program. Consider where program execution might be when a given interrupt occurs, and what effect the interrupt will have on the code that was originally being executed.

3 Techniques

3.1 First Things First: Configure the Watchdog and Oscillator

Configuring the watchdog should be among the first actions taken by any MSP430 program. If it is not handled quickly, the watchdog expires, and a PUC system reset occurs. This then occurs iteratively to form an endless loop. To prevent this, the watchdog should be configured at the beginning of the program by resetting the timer value, setting the hold bit, or disabling watchdog mode

Similarly, when using a low-frequency crystal on LFXT1 with a device from the 4xx or 2xx families, the code should configure the internal load capacitance using the FLL_CTL0 register very early in the code. Without this, the oscillator may not run properly. Note that it could still be a relatively long time after this before the clock is stable see Section 3.4.

These techniques can be found in most code examples for MSP430 MCUs available from TI.

3.2 Always Use Standard Definitions From TI Header Files

TI provides header files for every MSP430 MCU in production. These header files have constants for all the registers and bits in a given device, which match the names provided in the user's guides. Using these constants within code greatly enhances its readability. It also gives any code that uses them a similar look-and-feel that enables another engineer familiar with MSP430 MCUs, including TI's support team, to more quickly grasp the code. Every code example and application report from TI uses these headers where applicable.

3.3 Using Intrinsic Functions to Handle Low-Power Modes and Other Functions

Several intrinsic functions are available in development environments when writing in C. Sometimes, the only way to accomplish a critical task is to use an intrinsic function. Other intrinsics provide an opportunity to do things more efficiently.

The most common example of a critical task that can only be done using intrinsic functions is entering or exiting low-power modes. Doing so requires manipulation of bit values not otherwise accessible at the level of C, because they reside within the status register of the CPU. If entering LPM3 within the IAR development environment, an intrinsic function is used:

```
_BIS_SR(LPM3_bits + GIE);
```

Other intrinsic functions provide opportunity for optimization, such as those that provide access to the BCD math assembly instructions. Doing BCD math without these instructions requires a considerable amount of C code, and the compiler does not automatically translate this code to the BCD math instructions. Using the intrinsic functions allows the C programmer to take advantage of these instructions, maximizing memory and power efficiency.

The documentation for the development environment includes a list of these functions. See this list during development, and check it whenever new versions of the environment are released.

3.4 Write Handlers for Oscillator Faults

The MSP430 MCUs have circuitry that checks the integrity of the clocks. All the families provide this function for the DCO and high-frequency crystal sources. The 4xx and 2xx families also provide this function for a low-frequency (32.768-kHz) source. The specifics are covered in the user's guides.

Two kinds of oscillator faults should be considered, and a decision made regarding whether or not to handle them:

1. Stabilization of a crystal oscillator as it powers up. This happens every time the device runs. The time to stabilization is particularly long for low-frequency crystals, often in the hundreds of milliseconds.
2. Failure during operation. This can occur if a conductive substance is allowed to short the leads of the crystal. Some applications may be particularly susceptible to failure or intolerant of it and, therefore, need to handle it in a particular way. If a crystal oscillator fails, the DCO can drive the CPU while it handles the failure.

If an oscillator sourcing ACLK or SMCLK fails or has not yet stabilized, any peripherals supplied by those clocks are affected, and the only way to prevent this is to catch and handle it in software. A common problem is for a timing-sensitive peripheral (for example, a timer), sourced by a low-frequency crystal, to produce poor initial results because the crystal has not yet stabilized. If the code does not wait for the crystal to stabilize, the output of the peripheral may be corrupted.

If LFXT1 or XT2 sourcing MCLK fails, supply of MCLK reverts to the DCO. While this is an intelligent and robust fail-safe, it may have a negative effect on operation of the circuit and, therefore, needs to be caught and handled by software, rather than continuing as if nothing happened.

An easy way to handle initial stabilization is to repeatedly clear, wait, and check the fault flags until they stay cleared, as shown in the user's guides. (For the 1xx family, which cannot detect low-frequency oscillator faults on LFXT1, a fixed delay can be used, with a period sufficient for the worst-case stabilization length.) This method does not catch a fault during normal operation. A method that can catch both scenarios is to set the OFIE bit and implement a handler in the NMI interrupt service routine.

Fault-init.c in the accompanying zip file shows a check performed at startup to ensure the clock has stabilized. An example of using the OFIE bit and NMI service routine to trap and handle oscillator failures during operation can be found in code example FET410_LFxtal_nmi.c, in the MSP430x41x, MSP430F42x C Examples.

3.5 Increasing the MCLK Frequency

MCLK can be configured up to 8 MHz (16 MHz on the 2xx family devices). However, the V_{CC} requirement increases with frequency. If MCLK is set for a frequency that requires a V_{CC} level higher than what is applied to the device, unpredictable behavior can occur. The device-specific data sheet indicates the V_{CC} required for a particular MCLK frequency.

Even if the stabilized V_{CC} value is high enough for a given frequency, a slow V_{CC} ramp can prevent that level from being reached before the program increases the MCLK frequency. This is one reason it is good for the programmer to have knowledge of the power-up characteristics of the supply rail, not just on a single prototype, but characterized thoroughly for the device being produced.

If the device in question possesses an SVS module, it can be used to alert the system when V_{CC} has reached the necessary level. If the device does not contain SVS, but does contain an available analog-to-digital conversion (ADC) module, the ADC module can be used to sample the V_{CC} level and determine if its high enough before proceeding with the change.

If the device has neither SVS nor an available ADC, a fixed delay period can be used to wait until V_{CC} has reached the necessary level. Note that the delay period must be sufficiently long to handle the worst-case ramp scenario, taking into account variability over production windows and temperature.

A code example showing the use of the SVS module for this purpose is given as mclk.c in the accompanying zip file.

3.6 Using a Low-Level Initialization Function

By default, when a C compiler generates assembly code, it creates code that initializes all declared memory and inserts it before the first instruction of the main() function. If the amount of declared memory is large (either a large number of variables or one or more large memory spaces) this could pose a problem with the watchdog. The time required to initialize the long list of variables may be so long that the watchdog expires before the first line of main() can be executed. This means the watchdog configuration code is never executed, and an endless loop results. Generally, this can happen only on devices containing more than 2KB of RAM.

The easiest way to prevent this is to use a compiler directive that disables the initialization of memory elements that do not need preinitialization. For example, if using IAR, and a single large array is contributing to a watchdog-expiration problem, it could be handled as such:

```
__no_init int x_array[2500];
```

If this directive is not available in a given development environment, another possibility is to use a compiler-defined low-level initialization function to handle the watchdog before memory is initialized. The memory would be initialized as usual, but the watchdog configuration would happen first. In the IAR environment, this is accomplished by adding a function by the name of *__low_level_init()* and inserting the watchdog configuration code. For example:

```
void __low_level_init(void) { WDTCTL = WDTPW+WDTHOLD; }
```

A code example portraying the low-level init function is given as init.c in the accompanying zip file.

If neither of these functions is available in the compiler environment being used, one more option is to edit the startup file the compiler inserts at the beginning of every C program. See the compiler documentation for more information on these options.

If large amounts of memory are being defined on a device from the 4xx family, it is also a good idea to configure the LFXT1 oscillator capacitance within the low-level init function (or the startup file) (see Section 3.1). This gives extra time for the oscillator to stabilize before main execution begins.

3.7 In-System Programming (ISP)

If using the ISP functionality to write to flash memory, a few actions must be taken to ensure proper results:

1. Set the correct f_{FTG} value, as specified in the data sheet. Without this, the results are unpredictable. If the clock is too slow, there is the potential for overstressed flash cells. If the clock is too fast, there is the potential for incomplete write or erase operations.
2. Set the flash lock bit after the ISP operation is complete. This prevents accidental writes.
3. Ensure that the cumulative programming time for a flash block is not exceeded.
4. Provide sufficient V_{CC}. The level required for flash write and erase is higher than what is required for CPU operation.

The user's guides and data sheets contain more information on these points.

V_{CC} must be above the minimum specified by the device data sheet for flash erase/programming. Common ways in which this could be violated include a power-up ramp that has not yet completed, or a battery that has drifted too low for flash programming but is still high enough for CPU operation. Even if the level is high enough initially, the current draw associated with flash erase/programming could potentially stress smaller power supplies, pulling the voltage below the minimum threshold.

V_{CC} can be checked using the SVS module, if available, or with analog-to-digital conversion. SVS makes it advantageous in that it provides continuous checking of the rail during the operation. The code example given in Section 3.5 (mclk.c) can be used as a reference for how to configure the SVS prior to performing a V_{CC}-sensitive operation. The code example sets available from the TI website show how to use the ADC modules.

3.8 Using Checksums to Verify Flash Integrity

To ensure integrity of the data in flash memory during critical applications, a checksum function can be implemented that periodically verifies the data. The checksum value can be stored in one or more locations, depending on the redundancy needed. The value provided by a checksum scheme is that it provides the device an opportunity to handle an error if one should occur.

A code example showing the use of flash checksum verify is given as checksum.c in the accompanying zip file.

4 References

1. MSP430x1xx Family User's Guide
2. MSP430x2xx Family User's Guide
3. MSP430x4xx Family User's Guide
4. MSP430 MCU Code Examples

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from July 18, 2006 to March 14, 2018	Page
• Editorial updates throughout, including change to document title	1
• Added link to MSP430x41x, MSP430F42x C Examples in the last paragraph of Section 3.4, *Write Handlers for Oscillator Faults*	4

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265