Application Report
SLAA656C–March 2015–Revised March 2017

Migrating to the SimpleLink™ MSP432™ Family

Dung Dang — *MSP432 Marketing*
Evan Wakefield — *MSP432 Applications*
Priya Thanigai — *MSP432 Applications*

ABSTRACT

The 16-bit MSP430™ and the 32-bit SimpleLink™ MSP432™ microcontroller (MCU) families complement each other in low-power and performance. The goal of this migration guide is to help developers accurately assess the effort to migrate an existing application from the 16-bit MSP430 to the 32-bit SimpleLink MSP432 ARM® platform if they so choose to. Ultimately, the migration guide is built to help derive a migration strategy with complete hardware and software coverage that properly migrates the existing application without introducing bugs due to platform differences while still taking advantage of the unique features or performance improvements that the 32-bit MSP432 devices bring.

Having a good understanding of how each hardware or software component should be migrated or retained, developers can correlate these components back into their specific use in their system, and accurately assess the overall migration effort required for the entire application.

Contents

List of Figures

List of Tables

Trademarks

MSP430, SimpleLink, MSP432, Code Composer Studio are trademarks of Texas Instruments.
ARM, Cortex, Keil, µVision are registered trademarks of ARM Ltd.
Bluetooth is a registered trademark of Bluetooth SIG.
IAR Embedded Workbench is a registered trademark of IAR Systems.
Wi-Fi is a registered trademark of Wi-Fi Alliance.

1 SimpleLink MSP432 Platform Migration Overview

The MSP432 device is part of the SimpleLink microcontroller (MCU) platform which consists of Wi-Fi®, *Bluetooth*® low energy, Sub-1 GHz, and host MCUs. All share a common, easy-to-use development environment with a single core software development kit (SDK) and rich tool set. A one-time integration of the SimpleLink platform lets you add any combination of devices from the portfolio into your design. The ultimate goal of the SimpleLink platform is to achieve 100 percent code reuse when your design requirements change.

The MSP432 devices combine the 32-bit ARM Cortex®-M4F core and the MSP ultra-low-power DNA while showcasing highly integrated peripherals. Careful considerations were taken to enable both Cortex-M and CMSIS compliance along and pain-free migration from 16-bit MSP430 devices to the SimpleLink MSP432 device. For more information on the SimpleLink platform, visit www.ti.com/simplelink.

This migration guide is structured into the following sections:

- The first part of the application report is a device-level comparison of the two MSP platforms and highlights some key hardware migration considerations including the core, system-level considerations, peripheral modifications, and additions.
- The majority of the application report focuses on the software migration, as software often consumes the greatest effort when migrating an existing application to another MCU platform. The software overview lays out a high-level guide to assess the migration scope based on the chosen hardware and software options. The following sections divide the migration procedures based on core system, peripheral code, compiler- and intrinsic-specific code, higher-level libraries, and application code.

2 CPU and Core

The MSP430 CPU is a 16-bit RISC architecture that incorporates features specifically designed for modern and efficient programming such as orthogonal architecture, single-cycle register instruction, unified memory map with no paging, and direct memory-to-memory transfer, just to name a few. The MSP432 CPU is built around the industry standard ARM Cortex-M4F 32-bit core and benefits from the complete ecosystem of development tools and software solutions associated with ARM processors. Although they may be perceived as radically different, these two processors indeed share many architectural similarities as summarized in Table 1.

Table 1. MSP430 and MSP432 Core Comparison

Name	MSP430™ Core	MSP432™ Core
Data size	16 bit	32 bit
Program bus width	16-bit (CPU) or 20-bit (CPUX) address bus	32 bit
Bus type	16-bit MSP430 bus	AHB
Architecture	Von Neumann (Princeton): data op and instruction fetch share same bus	Harvard: separate data and instruction buses
Instruction Set	RISC, MSP430 proprietary	RISC, thumb and thumb2
Instruction size	16-bit (and 16-bit for each operand)	16-bit and 32-bit
Instruction Cycle (typical)	1-4 cycles	1-2 cycles
Pipeline	None	3-stage pipeline
Prefetch buffer	128 bit	128 bit
Power Modes	Active, LPM0-LPM4, LPMx.5	Active, Low Frequency, LPM0, LPM3, LPMx.5
Debug Interface	MSP430 4-wire JTAG and 2-wire SBW	ARM JTAG in 4-wire and 2-wire modes
Math support	Hardware Multiplier (MPY)	Hardware multiplier and divider, DSP extension, and integrated FPU

With the advancement and continuous improvements of compilers, many intricacies and differences at the CPU and core level are managed by the compiler and linker tools. While this alleviates some of the considerations for the C programmer, understanding the differences and advantages of each architecture can prove useful when optimizing the software for performance, size, or real-time behaviors. Section 4 also touches on some of the constraints when migrating software from one platform to another (16-bit to 32-bit and vice versa).

3 Hardware Features and Migration Considerations

3.1 System Features

3.1.1 System-Level Comparison

Table 2. Overall System-Level Comparison of MSP430 and SimpleLink MSP432 MCUs

Parameter	MSP430™ MCUs	MSP432™ MCUs
Supply voltage range	1.8 V to 3.6 V	1.62 V to 3.7 V
Analog supply voltage	1.8 V or 2.2 V to 3.6 V	1.8 V to 3.7 V
Maximum system frequency	8, 16, 20, or 25 MHz	48 MHz
Nonvolatile (flash or FRAM) memory	512 bytes to 512K bytes	Up to 256K bytes
RAM memory	128 bytes to 64K bytes	Up to 64K bytes

3.1.2 Reset

The reset circuit of the MSP432 MCUs is similar to the reset circuit of the MSP430 MCUs. The reset pin can be configured as a reset function (default) or as an NMI function in the Special Function Register (SFR), SFRRPCR.

In reset mode, the RST/NMI pin is active low, and a pulse applied to this pin that meets the reset timing specifications generates a BOR-type device reset. Setting SYSNMI causes the RST/NMI pin to be configured as an external NMI source. The external NMI is edge sensitive, and its edge is selectable by SYSNMIIES. Setting the NMIIE enables the interrupt of the external NMI. When an external NMI event occurs, the NMIIFG is set. The RST/NMI pin can have either a pullup or pulldown that is enabled or not. SYSRSTUP selects either pullup or pulldown, and SYSRSTRE causes the pullup (default) or pulldown to be enabled (default) or not.

If the RST/NMI pin is unused, it is required either to select and enable the internal pullup or to connect an external pullup resistor to the RST/NMI pin as well as with a decoupling capacitor to VSS.

3.1.3 Power

The MSP432 device family introduces a number of new power system features including dual-regulator (LDO and DC-DC), generating two VCORE levels for internal logic and other components.

The core voltage frequency throttling and runtime selectable regulators are some of the features that offer additional flexibility that can further optimize the power system and power consumption of a system. See *Maximizing MSP432P4xx Voltage Regulator Efficiency* for more information.

For more resources on designing the power system and specifying the power profile of the application, see *Designing an Ultra-Low-Power (ULP) Application With SimpleLink™ MSP432™ Microcontrollers*.

3.1.4 Low-Power Modes

Retaining the ultra-low-power architecture of the MSP430 MCUs, the MSP432 MCU family provides a similar power and low-power structure, upon which one can develop a power-efficient application. In addition to active mode, both MSP430 and MSP432 MCU platforms offer various low-power modes where different clocks and peripherals are power-gated providing flexibility to optimize for power consumption in different application states. On the MSP430 platform, these low-power modes are numbered from LPM0 to LPM4, recently extended to LPM3.5 and LPM4.5. Each level offers a different level of clock and peripheral availability. MSP432 MCU platform continues to use a similar structure for the low-power modes, retaining the most useful power modes and extending with two new modes for slow-speed execution. Table 3 summarizes the power modes on MSP platforms and how they correlate to each other.

Table 3. Power Modes of MSP430™ and MSP432™ MCUs

MSP430™ MCUs	MSP432™ MCUs	Industry Description	Comments
Active	Active	Active Mode	CPU and peripherals
	Low-Frequency Active	Low-Power Run	CPU and peripherals <128 kHz
LPM0	LPM0	ARM: Sleep	Peripherals on, CPU off
	Low-Frequency LPM0		Sleep + CLK < 128 kHz
LPM1	N/A	MSP430-specific mode	
LPM2	N/A	MSP430-specific mode	
LPM3	LPM3	ARM: Deep-sleep Standby with RAM and RTC	A/BCLK, <32 kHz, some peripherals available
LPM4	N/A	Standby with RAM	No clocks, some peripherals available
LPM3.5	LPM3.5	ARM: Shutdown	RTC without RAM
LPM4.5	LPM4.5	ARM: Shutdown	Shutdown

3.1.5 Clock

The MSP platform integrates a robust unified highly orthogonal clocking system that is easy to use and consistent across different MSP families and generations of devices. The clock module provides a unified clock tree with up to four clock signals for high-speed and low-speed clocking, offering the flexibility and balance for performance and low power consumption. A number of internal and external clock sources with varying degrees of frequency and accuracy can be used to supply the clock signals. The clock signals in turn can be used to source the CPU (MCLK) and clock the peripherals. While clocking peripherals are clock-gated and clock selection can be configured at the peripheral levels (using peripheral registers), clock signals can be enabled by the clock module or peripheral requests without the need to enable the peripheral clock itself.

MSP432P4xx family retains a clocking system similar to the MSP430F5xx, MSP430FR5xx, and MSP430FR6xx, combining some of the best features that these clocking systems have to offer. Applications that require a high-speed and high-accuracy internal clock source can leverage the improved DCO that offers up to 48-MHz operation, is highly tunable without using modulation, has low jitter, and offers a higher-accuracy option using an external resistor. Not only extending the support for higher speed operations, MSP432 also adds more clock sources and signals in between the frequencies ranges to provide more granularity and clocking options to best meet various application needs. When migrating an application from another platform to the MSP432 platform, these options (see Table 4) can be considered to optimize for the application use case.

3.1.6 Core-System Dependencies and Configuration

Table 4 lists the recommended system configurations based on the selected system frequency.

Table 4. System-Level Configuration

System Frequency	VCORE	Recommended Regulator	Flash Wait States
0 to 12 MHz	0	LDO	0
12 to 16 MHz	0	LDO	1
16 to 24 MHz	0	LDO	1
24 to 32 MHz	1	DC-DC for high active duty cycle application, LDO otherwise	1
32 to 48 MHz	1	DC-DC for high active duty cycle application, LDO otherwise	1

3.2 Peripherals

The initial MSP432 family provides a blend of the ultra-low-power peripherals previously introduced in MSP430 and a number of new peripherals to provide enhanced performance.

The shared peripherals operate in the same manners across MSP430 and MSP432 platforms, with the only exception being their interrupt signals and handling procedures due to the new Nested Vectored Interrupt Controller (NVIC) module on MSP432 MCUs. See Section 4 for more details on migrating MSP430 interrupt to NVIC. Register definitions and their descriptions in the header files are therefore identical across different families, enabling code reuse whether using register-level access or higher abstraction code such as MSP Driver Library.

Some peripherals on MSP432 MCUs receive slight modifications and performance improvements over their MSP430 MCUs counterparts. One example is the ADC14, which is an improved version of the ADC12_B. While their shared features and operations can be leveraged and reused, the new and improved features such as improved resolution (from 12-bit to 14-bit) or increased sampling rate (from 200 ksps to 1 Msps) require new design and software considerations.

The last group of peripherals on MSP432 MCUs that are new to the MSP platform are the peripherals that were derived or inherited from ARM such as the µDMA, Timer32, and SysTick. These modules add new functionalities to the device, and require further modification to the existing system and code to realize the new functions. Particularly with the µDMA module, many new features and capabilities were introduced that can be seen as an upgrade from the MSP430 DMA. Further investigation at the system design level should be paid to leverage these features in an MSP432 MCU application.

4 Software Migration

4.1 Software Migration Overview

When moving existing software from one device to another, one of the first steps is to assess and identify which existing code components can be reused and which components require modifications or developed from new. This up-front evaluation can be helpful to estimate the amount of effort required for the migration to accurately plan on resources and schedule for the software into two categories: code that can be reused, and code that needs to be modified or created new due to platform differences or device's new features. For cross-device or cross-platform migration purposes, the software components can be sorted into the categories: CPU and core-related code, system peripherals (for example, power, clock, and memory) code, peripheral code (whether using register accesses or peripheral driver libraries), interrupt-related code, intrinsic and compiler-specific code, and lastly software libraries or top-level application code of higher abstraction layers. Depending on the degree of differences between the devices being migrated, the number of software components that require updates can vary.

When migrating to the MSP432 device family, it might be worthwhile to look into using an RTOS. Many of the groups in Table 6 can be easily managed using an RTOS such as TI-RTOS. As Table 5 shows, the SimpleLink SDK comes with a number of new features that are standard for the SimpleLink SDK that MSPWare previously did not have including having the TI-RTOS kernel preinstalled and being POSIX compliant. SimpleLink SDKs also support alternative RTOS kernels, such as FreeRTOS, giving developers flexibility. Each kernel provides real-time multitasking services such as timing and scheduling of tasks. The RTOS kernel runs the hardware abstraction layer and a suite of functional drivers for all on-chip peripherals. For instance, when using an RTOS, the RTOS automatically can detect when the device should enter into a lower-power mode for idle times and manage that for the developer.

Table 5. Software Components

	MSPWare	SimpleLink™ SDK
Examples and Demos	Yes	Yes
TI drivers	No	Yes
DriverLib and HAL	Yes	Yes
Plug-ins	No	Yes
TI-RTOS	TI-RTOS link	Yes
FreeRTOS	N/A	Yes
POSIX-compliant API	No	Yes
Platform-specific libraries	Yes	Yes
Documentation	Yes	Yes

Table 6 breaks it down a step further for when migrating to the MSP432 platform of devices. The peripheral set used in the table examples is based on the configuration available on the MSP432P4xx family of devices.

Table 6. Migrating Software Components From MSP430Ware to SimpleLink SDK

Group	Component	Compatibility Level	Compiler Support	Register-Access Code	DriverLib APIs
Core	CPU and core-related	Low		Develop New	New, intelligent APIs
	Data types	Medium		Use explicit types	Use explicit types
	Interrupts	Low	Some	New	Available as APIs
	Intrinsics	Some	High		Some available as APIs
System	System modules: power, clock, memory	Low	N/A	Develop New	New, easy-to-use DriverLib APIs recommended
MSP Peripherals	Shared MSP peripherals: Timer_A, eUSCI, REF_A, COMP_E, WDT_A, RTC_C, GPIO, AES256	High	N/A	Compatible, check for native data types and register width (16-bit)	Compatible, check for native data types and BASE_ADDRESS use
	Extended or new revisions of MSP430 peripherals: ADC14, CRC32	Medium to High	N/A	Partially compatible, check for different register width (16-bit to 32-bit), minor bit relocation, new flag clearing mechanism	Partially compatible, account for new APIs, and arguments for existing APIs. Check for data types and BASE_ADDRESS use
ARM Peripherals	ARM peripherals: DMA, SysTick, Timer32	None	N/A	Develop New	New, DriverLib APIs recommended
Software libraries, top-level application code		High	N/A	Compatible using grlib, iqmathlib, and other libraries	

The migration scoring system in Table 7 allows developers to break down their existing application to similar components and groups. This exercise might yield different results in different application with varying degrees of abstraction, but typically this is possible in well-constructed software with sufficient layers of abstraction to detach low-level system and peripheral code from drivers, libraries, and top-level application code. Having the knowledge on the distribution of each group and component in the

application code, the developers can leverage the migration effort in Table 7 to not only determine the level of migration effort for their application but also to understand at a high level what needs to be accomplished for each component to create a compatible and robust application on the new MSP platform. Table 7 shows the distribution of the software components in a typical application assuming all components and peripherals are being used. Using this chart, after identifying the distribution of a specific application, one can derive from a high level the overall migration effort required for that application.

Table 7. Distribution of Software Components Migrating to MSP432P4xx

Group	Component	Migration Effort	Percentage in Your Application Code
Core	CPU and core-related	Low-Medium	
	Data types	Medium	
	Interrupts	Medium	
	Intrinsics	Medium	
System	System modules: power, clock, memory	Medium	
MSP Peripherals	**Shared MSP Peripherals:** Timer_A, eUSCI, REF_A, COMP_E, WDT_A, RTC_C, GPIO, AES256	Low	
	Extended or new revisions of MSP430 peripherals: ADC14, CRC32	Low-Medium	
ARM Peripherals	ARM Peripherals: DMA, SysTick, Timer32	Low-Medium	
Software libraries, top-level application code		Low	

Notice that the migration effort for each section is given as a general guidance. The exact score can only be determined after these components are carefully assessed in the context of a specific application. The following sections describe the migration steps for each of the components in more detail and highlight some of the important details to watch out for in each component.

4.2 CPU, Intrinsics and Compiler Support

4.2.1 Data Types, 16 Bit and 32 Bit

The first noticeable difference between MSP430 and MSP432 is the 16-bit and 32-bit CPU architectures. This results in different native data types and sizes. In C language, the **int** type takes the architecture's native size; hence int means 16-bit integer for MSP430 MCUs and 32-bit integer for MSP432 MCUs. If a variable relies on its data size and type, especially when considering types and operations that might involve sign and overflow, incorrect data types might result in incorrect calculation results. If the variables are used for indexing purposes, using incorrect types might result in invalid or out of bound memory access. When migrating from MSP430 MCUs to MSP432 MCUs, using **int** might be less potentially dangerous because the new variables have 16 extra bits to buffer the existing 16-bit use.

To remedy incorrect or implicit data type use, developers can use C99 types where explicit data size and type is always specified. Instead of **unsigned int** or **int**, using **uint32_t** or **int16_t** not only prevents variable type misuse but can potentially help developers better identify variable types in the code and understand their use. When the software is to be written for multiple platforms, to ensure the correct types are used, but at the same time the best types are used for their specific purposes (indexing variable, data storage, temporary calculation, etc.), and to leverage the best data types for a particular MSP platform, the C99 fast and least types such as **uint_fast32_t** or **uint_least16_t** can be also be used.

The fast integer type leverages the faster integer type available on the platform with at least the amount of bits required. **uint_fast16_t** would use 16-bit unsigned integer type for MSP430 MCUs, but 32-bit unsigned integer type on MSP432 MCUs, because native 32-bit operation is always the fastest on MSP432 MCUs. On the other hand, the leastN integer type designates an integer type available on a particular architecture with the with a width of at least N bits.

Correct and robust data types should be used not only for the CPU or low-level code, but also across all layers of drivers or software libraries.

4.2.2 MSP430 Core and Cortex-M4 Core Use and Intrinsics

The different cores used by MSP430 MCUs and MSP432 MCUs introduce different sets of intrinsics to be used with their associated devices. Some of these intrinsics share common functionalities, and an intrinsic translation layer is provided to provide compatibility for existing MSP430 MCU code. The file msp_compatibility.h helps translate several existing MSP430 MCU intrinsics to MSP432 MCU equivalents. Example 1 defines some of the frequently used intrinsics.

Example 1. MSP Intrinsics and Definition on MSP432 (msp_compatibility.h)

```
#define __sleep()                       __wfi()
#define __deep_sleep()                  { (*((volatile uint32_t *)(0xE000ED10))) |=
0x00000004; __wfi(); (*((volatile uint32_t *)(0xE000ED10))) and= ~0x00000004; }
#define __low_power_mode_off_on_exit()  { (*((volatile uint32_t *)(0xE000ED10))) &=
~0x00000002; }
#define __get_SP_register()             __get_MSP()
#define __set_SP_register(x)            __set_MSP(x)
#define __get_interrupt_state()         __get_PRIMASK()
#define __set_interrupt_state(x)        __set_PRIMASK(x)
#define __enable_interrupt()            __asm("  cpsie i")
#define __enable_interrupts()           __asm("  cpsie i")
#define __disable_interrupt()           __asm("  cpsid i")
#define __disable_interrupts()          __asm("  cpsid i")
#define __no_operation()                __asm("  nop")
```

CMSIS also provides a collection of instruction and function intrinsics applicable for Cortex-M4 core (see Example 2).

Example 2. MSP432 Intrinsics Defined in cmsis_ccs.h

```
__attribute__( ( always_inline ) ) static inline void __nop(void)
{
     __asm("  nop");
}

// Wait For Interrupt
__attribute__( ( always_inline ) ) static inline void __wfi(void)
{
     __asm("  wfi");
}

// Wait For Event
__attribute__( ( always_inline ) ) static inline void __wfe(void)
{
     __asm("  wfe");
}

// Enable Interrupts
__attribute__( ( always_inline ) ) static inline void __enable_irq(void)
{
     __asm("  cpsie i");
}

// Disable Interrupts
__attribute__( ( always_inline ) ) static inline void __disable_irq(void)
{
     __asm("  cpsid i");
```

Example 2. MSP432 Intrinsics Defined in cmsis_ccs.h (continued)

```
}

// Data Synchronization Barrier
__attribute__( ( always_inline ) ) static inline void __DSB(void)
{
     __asm("  dsb");
}
```

For DriverLib users, DriverLib APIs are also provided to exercise various functionalities achieved by the intrinsics. Additional APIs are also available to execute a sequence of instructions to provide further intelligence. These DriverLib APIs can either be found in CPU module (cpu.h) or the power module (pcm.h).

```
PCM_setPowerState();
PCM_shutdownDevice();
PCM_gotoSleep();
PCM_gotoDeepSleep();
```

For more information on DriverLib APIs, visit www.ti.com/tool/mspdriverlib and see the *MSP432 DriverLib User's Guide* (included with DriverLib).

4.2.3 Interrupt System

MSP430 MCUs and MSP432 MCUs use two significantly different interrupt systems. Any effort to migrate software between these two platforms requires special consideration of interrupt configuration and handling.

The interrupt system on MSP430 MCUs is heavily integrated into the MSP430 core. The interrupt system and peripheral interrupt sources are tied directly back to the main interrupt system, which is controlled by a single bit GIE, part of the SR register.

On the other hand, the MSP432 MCUs employ the Cortex-M4 integrated Nested Vectored Interrupt Controller (NVIC), which takes more control of the interrupt management from the CPU core and at the same time provides more flexibility and features such as configurable priority, efficient tail-chaining, and individual peripheral interrupt control.

4.2.3.1 Enabling Interrupt on MSP432 MCUs

From the perspective of the migration process, the last feature is the main one that requires extra software to register and enable the peripheral interrupt source on the NVIC system. This is in addition to enabling the peripheral's individual interrupt triggers, as shown in Figure 1.

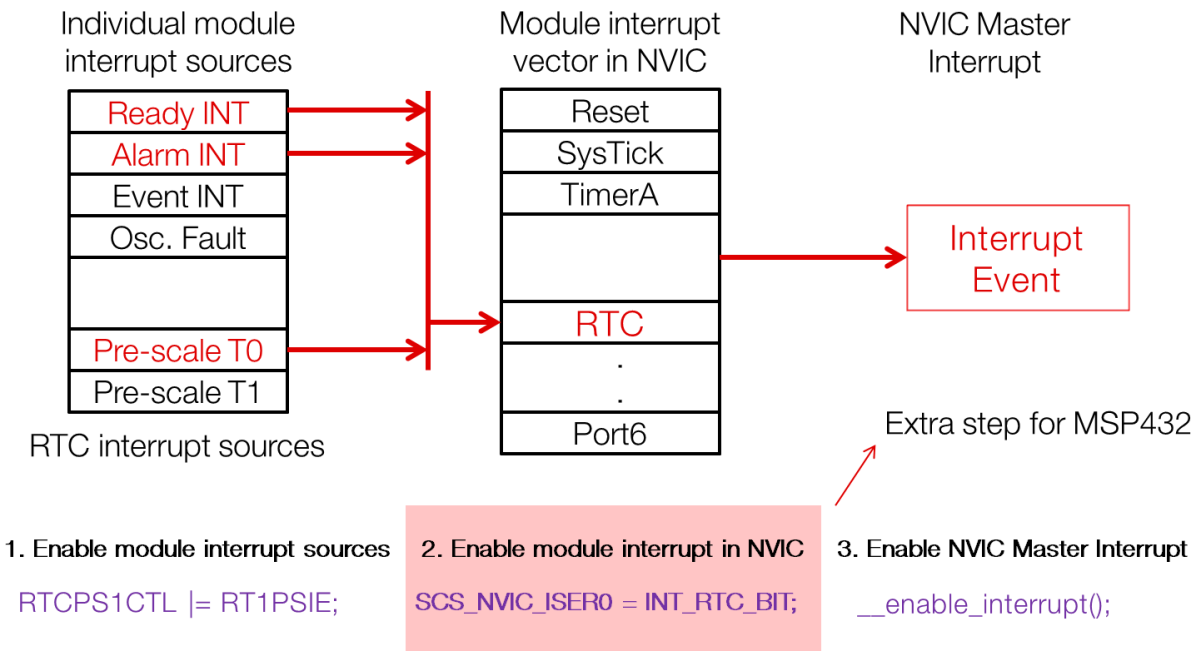


Figure 1. Enabling Interrupt on MSP432™ MCUs

Note that this additional step is done at the NVIC module. That means when using DriverLib, the NVIC interrupt API to be used for this step is `Interrupt_enableInterrupt(PERIPHERAL_INT);`.

4.2.3.2 Registering Interrupt and Interrupt Vector Table

For MSP430 MCUs, the compiler usually manages allocating memory space for the interrupt vector table, reserves default values for unused interrupt vectors, and registers interrupt vectors being used by detecting the special interrupt service routine prefixed by the keywords **#pragma vector.** The ISR function also requires the *__interrupt* keyword.

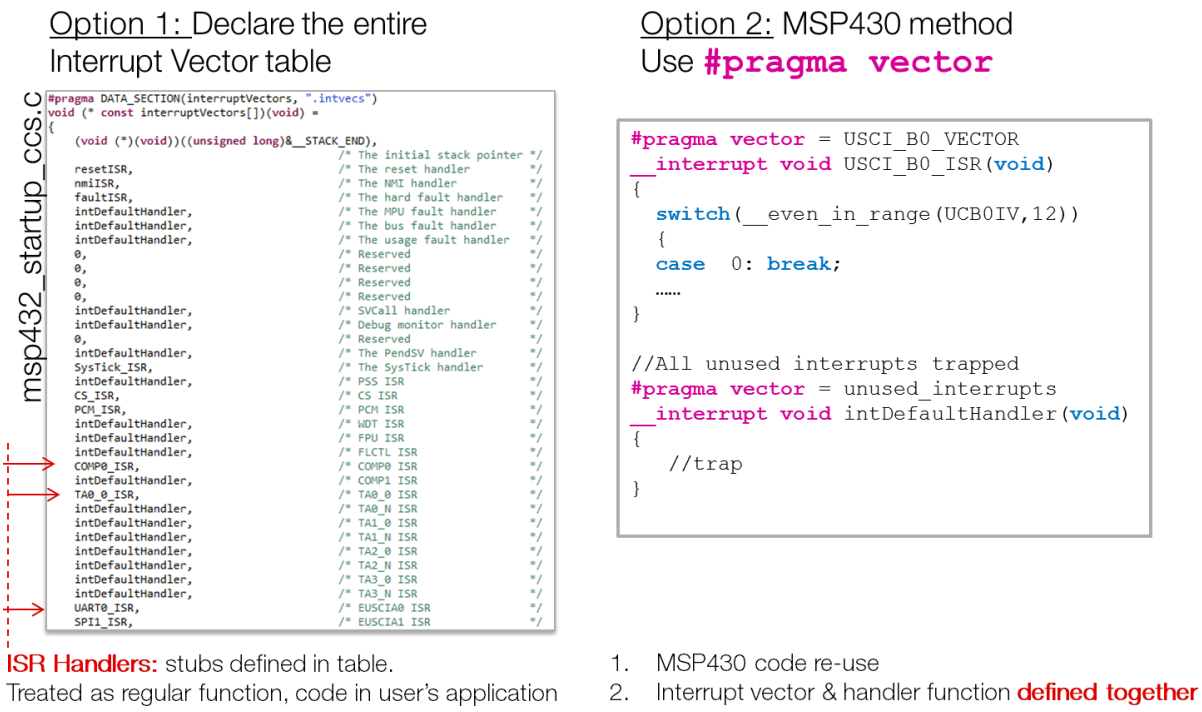
```
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
```

The default method to define the interrupt vector table for MSP432 MCUs is the common standard across different Cortex-M platforms. The entire interrupt vector table is defined as an array with fixed starting location at 0x00000000. The table consists of interrupt vector addresses that point to the addresses of the interrupt service routines, which are defined as regular functions.

NOTE: Feature in work

The CCS compiler is currently under work to enable the MSP430 interrupt #pragma feature on MSP432 that is described below. When this feature is available, this document will be updated with the first CCS and TI compiler versions that support this feature.

To maintain compatibility between MSP platforms, MSP432 MCUs also supports the traditional #pragma vector method to define interrupts. For migration purposes, this method can still be employed to reuse the existing interrupt code from an MSP430 MCU. Figure 2 shows both options to define and allocate for interrupt vector tables on an MSP432 MCU.



NOTE: Option 2 is currently not available. See the "Feature in work" note in Section 4.2.3.2 for additional information.

Figure 2. Defining Interrupt Service Routines (ISR) on MSP432™ MCUs

4.2.4 Interrupt and LPM0 or LPM3 (Sleep) Modes

One of the architectural differences between the MSP430 CPU and the MSP432 CPU is that the MSP430 CPU provides an atomic instruction that allows the user to enable or disable global interrupts and go into low-power modes at the same time. In MSP432 architecture, these two operations are separate; hence, the application typically needs to first enable or disable the master interrupt and then go into LPM0 (sleep) or LPM3 (deep-sleep) mode.

On MSP430 MCUs, the following atomic instruction can be used to enable interrupt and go into low-power mode.

```
__bis_SR_register(LPM0_bits + GIE);
```

On MSP432 MCUs, however, the same procedure requires two separate instructions to complete. The procedure therefore becomes nonatomic.

```
__enable_irq();
__wfi();
```

The above CMSIS-style intrinsics first enable the interrupts by disabling the PRIMASK and then go into LPM0 using the wait for interrupt instruction.

Notice that the MSP432 header file also provides a translation layer to recreate similar intrinsics such as __enable_interrupts() or __sleep() mimicking MSP430 MCU's behavior yet stay functionally the same as the CMSIS intrinsics.

The biggest difference due to the nonatomic operation on MSP432 MCUs is that in the event of a pending interrupt or an asynchronous interrupt occurs between the enable_irq() and wfi() instructions, the interrupt might get serviced prior to the device entering LPM0 or LPM3 mode. If that interrupt is the only expected interrupt and the only method for the device to return from low-power mode, the application needs to ensure that device does not permanently stay in low-power mode. In this unique scenario, the following code sequence can be used instead:

```
__disable_irq();
__wfi();
__enable_irq();
__ISB();
__disable_irq();
```

In this scenario, if the PRIMASK is enabled resulting in the master interrupt being disabled, in the event of a pending interrupt, the __wfi() command will either execute as a NOP instruction if in active. If the device has already executed the __wfi() command, an interrupt triggers with the master interrupt disabled would result in device wake up from LPM0 or LPM3 mode. The last three instructions essentially immediately enable the interrupt module to ensure the interrupt is serviced. The drawback of this sequence is that every interrupt triggered will result in device wake-up. It is up to the developers to determine the optimal yet safe interrupt and low-power mode sequence based on their application requirements and interrupt sources and low-power mode profiles.

Interrupt wake-up operation is also slightly different between MSP430 and MSP432 MCUs. On MSP430 MCUs, the application can typically instruct the CPU to wake up at the end of an Interrupt Service Routine (ISR) by clearing the appropriate bits in the SR register through the use of the intrinsic function __bic_SR_Register_on_exit(). On MSP432 Cortex-M CPU, the application can configure whether the CPU wakes up or goes back to low-power mode at the end of any interrupt service routine by using the SLEEPONEXIT bit. Specifically, if the SLEEPONEXIT bit is set, the device automatically reenters low-power state at the end of the ISR of the last pending interrupt. If the SLEEPONEXIT is clear, the device wakes up and resumes CPU execution after servicing the last pending interrupt.

The MSP432 DriverLib incorporates several APIs to accomplish different interrupt and LPM operations, including APIs to execute both of the above interrupt and LPM sequences.

4.2.5 Memory

While both MSP430 and MSP432 MCUs use a unified memory map without paging, their memory map layouts significantly differ. Applications that have some memory components or dependencies such as data logging or field firmware updates need to restructure the overall memory layout.

Specific component placements are detailed in Figure 3 and Figure 4.

Can generate NMI on read/write/fetch								
Generates PUC on fetch access								
Protectable for read/write accesses								
Always able to access PMM registers from[1]; Mass erase by user possible								
Mass erase by user possible								
Bank erase by user possible								
Segment erase by user possible								
Address Range	**Name and Usage**	**Properties**						
00000h-00FFFh	Peripherals with gaps							
00000h-000FFh	Reserved for system extension							
00100h-00FEFh	Peripherals						x	
00FF0h-00FF3h	Descriptor type[2]						x	
00FF4h-00FF7h	Start address of descriptor structure						x	
01000h-011FFh	BSL 0	x				x		
01200h-013FFh	BSL 1	x				x		
01400h-015FFh	BSL 2	x				x		
01600h-017FFh	BSL 3	x			x	x		
017FCh-017FFh	BSL Signature Location							
01800h-0187Fh	Info D	x						
01880h-018FFh	Info C	x						
01900h-0197Fh	Info B	x						
01980h-019FFh	Info A	x						
01A00h-01A7Fh	Device Descriptor Table						x	
01C00h-05BFFh	RAM 16 KB							
05B80-05BFFh	Alternate Interrupt Vectors							
05C00h-0FFFFh	Program	x	x[1]	x				
0FF80h-0FFFFh	Interrupt Vectors							
10000h-45BFFh	Program	x	x	x				
45C00h-FFFFFh	Vacant							x[3]

Figure 3. MSP430™ MCU Memory Map (MSP430F548A)

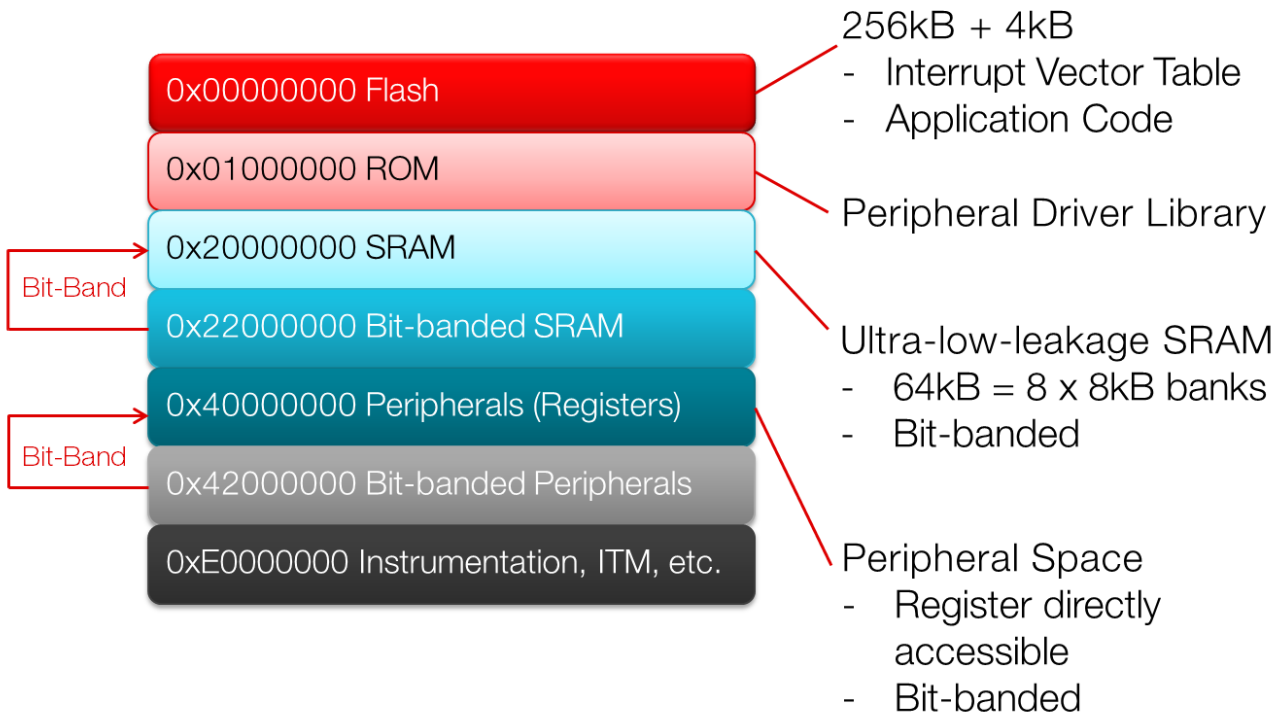


Figure 4. MSP432™ MCU Memory Map (MSP432P401R)

A few key highlights include:

- Interrupt vector table
  - MSP430 MCUs start at 0xFFFF with Reset Vector and grow downward.
  - MSP432 MCUs start at 0x0000 with Reset Vector and grow upward.
- Flash, RAM, and other memory placements:
  - MSP430 memory components share 64KB space (for 16-bit CPU devices) or 1MB space (for 20-bit-supported CPUX), and addresses for each component potentially vary per device.
  - MSP432 MCUs use the abundant 32-bit address space to reserve separate memory locations for different components.
- MSP432 MCUs introduce the bit-band feature to compensate for the Read-Modify-Write nature of the Cortex-M instruction set.
- Individual bit manipulation in SRAM or in peripheral space can take advantage of bit-banding to reduce the Read-Modify-Write operation down to a single instruction. DriverLib APIs take advantage of bit-banding when manipulating peripheral registers. When writing your own register-access code, use the following defines to leverage bit-banding when possible.

Example 3. MSP432 Bit-Banding Defines

```
#define HWREGBIT8(x, b)
#define HWREGBIT16(x, b)
#define HWREGBIT32(x, b)
#define BITBAND_SRAM(x, b)
#define BITBAND_PERI(x, b)
```

4.3 Core System Software

The core system peripherals vary between device families to provide the best fit for objectives of a particular device family. This can be power management (Vcore, no Vcore), clocking scheme (DCO or FLL), or nonvolatile memory type (flash, FRAM). For such reasons, migration code from one device to another family or platform always requires developing new code for the core system peripherals.

Table 8. New Core System Peripherals for MSP432 MCUs

System Modules	Compatibility Level	Register-Access Code	DriverLib APIs
Power, Clock, Memory	Low	Develop New	New easy-to-use DriverLib APIs recommended

DriverLib APIs can be leveraged to most properly and efficiently configure the core system. While this does not take away the requirement for the developers to comprehend the device operations, it does alleviate the time investment to recreate and develop the code for the core modules. Having a good understanding of the inter-peripheral requirements to configure the core such as from Section 3.1.6, one can quickly devise a configuration routine that uses DriverLib APIs for power, clock, and memory modules to achieve the core configuration requirement.

```
MAP_FlashCtl_setWaitState(FLASH_BANK0, 2);
MAP_FlashCtl_setWaitState(FLASH_BANK1, 2);
PCM_setPowerState(PCM_AM_DCDC_VCORE1);
MAP_CS_setDCOCenteredFrequency(CS_DCO_FREQUENCY_48);
```

Additionally, because the core configuration code is usually done once in the initialization code at the beginning of the application, the total amount of software needed is relatively small in comparison to the entire program. Hence while having to develop new, with the availability of the DriverLib APIs and helpful code examples, the actual code development required to migrate to the new core peripheral system is manageable. The majority of the effort is spent learning how the new system works and figuring out the most optimal core configuration to fit the application goals. For more information on how to leverage new features of the core system, especially of the unique power system new to MSP432 MCUs, see *Designing an Ultra-Low-Power (ULP) Application With SimpleLink™ MSP432™ Microcontrollers*.

4.4 Peripheral Software

4.4.1 MSP-Shared Peripherals

The MSP devices are designed with cross-platform-compatible peripherals in mind. Many of the highly integrated and ultra-low-power peripherals from the MSP430 family continue to be useful in the higher performance MSP432 family. When moving from one platform to another, these peripherals retain their core design, functionalities, and the interfaces by which they are accessed and controlled. From the developer's perspective, the peripheral use, register and bit definitions, and header file supports are identical moving across device families and platforms. From the software standpoint, this means that any existing code written for these peripherals is functional and compatible on a new platform. However, keep in mind the additional interrupt enabling step for each of the peripherals as explained in Section 4.2.3.

Table 9. Shared Peripherals Across MSP Platforms

Shared MSP Peripherals	Compatibility Level	Register-Access Code	DriverLib APIs
Timer_A, eUSCI, REF_A, COMP_E, WDT_A, RTC_C, GPIO, AES256	High	Compatible, check for native data types and register width (16 bit)	Compatible, check for native data types and BASE_ADDRESS use

Example 4 and Example 5 are examples of the highly compatible peripheral register-access code for Timer_A for MSP430 and MSP432 MCUs, respectively.

Example 4. MSP430 Timer_A Register-Access Code

```
P1DIR |= 0x01;                // P1.0 output
TA0CCTL0 = CCIE;              // CCR0 interrupt enabled
TA0CCR0 = 50000;
TA0CTL = TASSEL_2 | MC_1 | TACLR;      // SMCLK, upmode, clear TAR
```

Example 5. MSP432 Timer_A Register-Access Code

```
P1DIR |= 0x01;                // P1.0 output
TA0CCTL0 = CCIE;              // CCR0 interrupt enabled
TA0CCR0 = 50000;
TA0CTL = TASSEL_2 | MC_1 | TACLR;      // SMCLK, upmode, clear TAR

Add the line to enable nvic for timer  // Enable interrupts
```

Example 6 and Example 7 demonstrate the highly compatible Driver Library APIs exercised to configure the serial communication peripheral USCI on MSP430 and MSP432 MCUs, respectively. Notice that the only difference is due to the clock module API. The MSP432 Driver Library also introduces new argument definitions for the base addresses in addition to the existing MSP430 base address definitions (USCI_B0_BASE and USCI_B0). Either definition can be used to select the specified peripheral instance.

Example 6. MSP430 USCI DriverLib code

```
// Initialize Master
USCI_B_I2C_masterInit(USCI_B0_BASE, USCI_B_I2C_CLOCKSOURCE_SMCLK,
           UCS_getSMCLK(UCS_BASE), USCI_B_I2C_SET_DATA_RATE_400KBPS);

// Specify slave address
USCI_B_I2C_setSlaveAddress(USCI_B0_BASE, SLAVE_ADDRESS);

// Set Master in receive mode
USCI_B_I2C__setMode(USCI_B0_BASE, USCI_B_I2C_RECEIVE_MODE);

// Enable I2C Module to start operations
USCI_B_I2C_enable(USCI_B0_BASE);
```

Example 7. MSP432 eUSCI DriverLib Code

```
// Initialize Master
USCI_B_I2C_masterInit(USCI_B0_BASE, USCI_B_I2C_CLOCKSOURCE_SMCLK,
           CS_getSMCLK(), USCI_B_I2C_SET_DATA_RATE_400KBPS);

// Specify slave address
USCI_B_I2C_setSlaveAddress(USCI_B0_BASE, SLAVE_ADDRESS);

// Set Master in receive mode
USCI_B_I2C__setMode(USCI_B0_BASE, USCI_B_I2C_RECEIVE_MODE);

// Enable I2C Module to start operations
USCI_B_I2C_enable(USCI_B0_BASE);
```

In summary, migrating shared peripheral codes for MSP430 and MSP432 requires a low level of effort. Developers should, however, make sure that additional interrupt code is accounted for based on the platform's interrupt system, and that correct data types, preferably explicit types, are used. One additional migration step that is always required even when moving between two devices in the same family is to ensure the port pins being used by the peripherals are updated on the new device.

4.4.2 Extended Peripherals

Enhanced peripherals can be identified by their use of revision letters such as REF, REF_A, REF_B. Sometimes in the case of analog modules, the resolution can be used as the identifier such as in the case of the ADC module: ADC12, ADC12_A, ADC12_B, ADC14. The MSP432P4xx family introduces the 14-bit ADC14 module as an expanded enhanced version of the ADC12_B, featuring 14-bit resolution and 1-Msps sampling rate among other digital enhancements.

Table 10. Level of Compatibility for Extended Peripherals

Extended and New Revision Peripherals	Compatibility Level	Register-Access Code	DriverLib APIs
ADC14	Medium to High	Partially compatible, check for different register width (16-bit to 32-bit), minor bit relocation, new flag clearing mechanism.	Partially compatible, account for new APIs and arguments for existing APIs. Check for data types and BASE_ADDRESS use.

For these types of peripherals with incremental changes, shared functionalities with prior revisions are often retained; therefore include the same register and bit definitions in the header file. In some cases, minor register bits might be shifted or added to enable the enhanced features. As a result, some of the existing register-access code can be reused in similar fashion as described for shared peripherals (see Section 4.4.1). On the other hand, new features introduce new registers and bit definitions. Developers are offered two options: using register-access code or new MSP432 DriverLib APIs. For both methods, in addition to leveraging the specification and the description of the peripheral from the *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual*, developers can also take advantage of various device code examples written in both register-access and DriverLib formats.

Example 8. MSP432 ADC14 DriverLib APIs

```
// ADC14 DriverLib
// APIs similar to MSP430 ADC12_A/B APIs but with new namespace or updated parameters
extern bool ADC14_initModule(uint32_t clockSource, uint32_t clockPredivider,
        uint32_t clockDivider, uint32_t internalChannelMask);
extern void ADC14_setResolution(uint32_t resolution);
extern bool ADC14_configureConversionMemory(uint32_t memorySelect,
        uint32_t refSelect, uint32_t channelSelect, bool differntialMode);
extern bool ADC14_enableComparatorWindow(uint32_t memorySelect,
        uint32_t windowSelect);
```

In summary, extended peripherals usually retain some level of compatibility both in functions and in software. TI recommends checking the existing code to ensure functionality is retained. Some additional new code is likely required to fully exercise the features of the new peripheral. This migration and new development effort can be aided by looking through device code examples available in both register access and DriverLib APIs.

4.4.3 New and Cortex-M Peripherals

The last category of peripherals first introduced to MSP portfolio on the MSP432 platform includes the new Cortex-M peripherals that are either part of the Cortex-M4F core or from Cortex-M peripheral set due to integration and performance advantages.

Table 11. Cortex-M Peripherals

Cortex-M Peripherals	Compatibility Level	Register-Access Code	DriverLib APIs
SysTick, Timer32	None	Develop New, simple	New, simple APIs
DMA	None	Possible but very complex	APIs highly recommended
FPU	None	Simple, built-in compiler support	Simple, built-in compiler support

Because these peripherals are available from ARM, software written using CMSIS definitions (in addition to MSP register access and DriverLib APIs) may also be available.

The FPU is also another module that has low software requirement as most modern ARM compilers have built-in support to control the FPU for all floating point calculation in the code.

On the other hand, modules such as the µDMA can be relatively demanding when new code creation is required. When migrating existing DMA code on MSP430 platform to the new µDMA module on MSP432, TI highly recommends considering using DriverLib APIs and various code examples written for the module to quickly ramp up on learning how to properly configure and use this module. A significant level of migration is to be expected as system-level considerations are required to restructure the DMA configuration and use. See the *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual* and the code examples for more information on the DMA module.

4.4.4 Traditional C Code and Register Access

Take advantage of bit-banding for single register bit manipulation. Check for 8-bit, 16-bit, and 32-bit register accesses in the application.

4.4.5 Peripheral Driver Library

MSP430 and MSP432 DriverLib share some common APIs for the peripherals available on both platforms. MSP432 DriverLib does however incorporate small changes in function calling, argument passing, and how to handle peripherals with multiple instances.

4.4.6 Other Peripheral Manipulation Software

In addition to leveraging register-access code and MSP DriverLib APIs, developers can also take advantage of other peripheral driver software available by third-party developers, community-driven projects, or from ARM ecosystem. The strategy for migrating this software remains similar to when using traditional register access code or TI-provided MSP Driver Library.

CMSIS is the coding standard provided by ARM. Out of CMSIS, several software components have been defined and developed by various silicon vendors. In addition to regular header and support files, MSP432 also provides CMSIS-style device header file to enable CMSIS users. While this approach can leverage the CMSIS standard for code development, migration effort from MSP430 to MSP432 MCUs using CMSIS is slightly more demanding.

4.4.7 Application Code and Software Libraries

Check for data types.

Check for real-time behavior.

Abstract the code to isolate core-dependent code from higher abstraction layers.

5 Tools and Ecosystem

The MSP432 device is part of the SimpleLink MCU platform which consists of Wi-Fi, Bluetooth low energy, Sub-1 GHz, and host MCUs. All share a common, easy-to-use development environment with a single core SDK and rich tool set. A one-time integration of the SimpleLink platform lets you add any combination of devices from the portfolio into your design. The ultimate goal of the SimpleLink platform is to achieve 100 percent code reuse when your design requirements change. For more information, visit www.ti.com/simplelink.

MSP432 MCUs are supported by Cortex-M-capable debuggers including TI's XDS200, XDS100v2/3, XDS110, IAR I-jet, and SEGGER J-LINK. For more information on supported debuggers and how to take advantage of these tools when debugging MSP432 MCUs, see the *MSP432™ SimpleLink™ Microcontrollers Hardware Tools User's Guide*.

MSP432 MCU code development is available on the same integrated development environments (IDEs) as MSP430 MCUs, including Code Composer Studio™ IDE, IAR Embedded Workbench® IDE, and gcc. The compatible tool chains also provide compatible header files, support packages, code examples, and software libraries to maximize the similar experience on both MSP430 MCU and MSP432 MCU platforms.

In addition to the common IDEs, MSP432 MCUs are also supported in Keil® µVision® MDK. While the migration effort from MSP430 MCUs to MSP432 MCUs using Keil µVision can be more involved, developers can leverage the ARM software ecosystem including various layers of CMSIS software in their MSP432 MCU code development.

6 Migration Example and Analysis

The following steps summarize the process of migrating an existing application on the MSP430F5529 MCU to the MSP432P401R MCU.

1. **Core module configuration**
   (a) Set up power and VCore
   (b) Set up flash wait-state based on the system frequency
   (c) Set up DCO frequency
2. **Peripheral code: three types of MSP432 code**
   (a) Migrate register access code for Timer_A modules
   (b) Migrate DriverLib APIs for ADC12 module to ADC14 module
   (c) Migrate RTC module mode using CMSIS
3. **Interrupt-related configuration code**
   (a) Module interrupt enable instructions to be done using both register-access and DriverLib APIs
   (b) Enable module interrupt on NVIC to be done
   - Using DriverLib API: Interrupt_enableInterrupt for ADC14
   - Using register access: SCS_NVIC_ISERx register for RTC

   (c) Enable NVIC master interrupt (equivalent to the GIE bit on MSP430 MCUs)
   - Using DriverLib API
   - Using MSP432 intrinsic function
4. **Add ISR stubs to interrupt vector table**
5. **Core, CPU, and system-related functions**
   (a) How to go to LPM0 (sleep)
   (b) How to go to LPM3 (deep sleep)
   (c) Control when and how to wake up to active from ISR
   (d) Control when and how to go back to sleep from ISR
6. **Tie all pieces together and debug**

7 References

1. *Designing an Ultra-Low-Power (ULP) Application With SimpleLink™ MSP432™ Microcontrollers*
2. *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual*
3. *MSP432™ SimpleLink™ Microcontrollers Hardware Tools User's Guide*
4. *MSP432P401R, MSP432P401M SimpleLink™ Mixed-Signal Microcontrollers*

Revision History

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

Changes from March 16, 2016 to March 8, 2017 **Page**

- Changed the title of this document ... 1
- Added to the authors of this document ... 1
- Updated the abstract ... 1
- Changed the title and updated the content of Section 1, *SimpleLink MSP432 Platform Migration Overview* ... 2
- Changed the Flash Wait States column for "32 to 48 MHz" in Section 3.1.6, *Core-System Dependencies and Configuration* ... 5
- Removed former Table 5, *Software Components for Different Levels of Porting* ... 6
- Added the second paragraph and Table 5, *Software Components*, in Section 4.1, *Software Migration Overview* ... 6
- Changed the title of Table 6, *Migrating Software Components From MSP430Ware to SimpleLink SDK*, and changed the bottom right table cell ... 6
- Changed msp430dna.h to msp_compatibility.h in Section 4.2.2, *MSP430 Core and Cortex-M4 Core Use and Intrinsics* . 8
- Changed the first paragraph and added XDS110 in Section 5, *Tools and Ecosystem* ... 18
- Updated titles of referenced documents as necessary ... 19

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