Application Report
SLAA704–July 2016

Flash Operations on MSP432™ MCUs

Timothy Logan *MSP Applications*

ABSTRACT

The flash controller of the MSP432™ MCUs is a very powerful and intricate peripheral that has a robust programming model and feature-rich implementation. With a proper software layer, the programmer is able to abstract out the complexities of the flash controller's programming model and keep application code free of any flash controller overhead. Additionally, executing DriverLib from ROM enables the user to not have to worry about any significant boundary requirements when verifying flash operations and changing flash banks into special read modes. Project collateral discussed in this application report can be downloaded from the following URL: www.ti.com/lit/zip/SLAA704.

Contents

(1) MSP432, Code Composer Studio are trademarks of Texas Instruments.
(2) Keil is a registered trademark of ARM Limited.
(3) IAR Embedded Workbench is a registered trademark of IAR Systems AB.
(4) Wi-Fi is a registered trademark of Wi-Fi Alliance.

1 Introduction

The flash controller of the MSP432 is a very powerful and intricate peripheral that has a robust programming model and feature-rich implementation. With a proper software layer, the programmer is able to abstract out the complexities of the flash controller's programming model and keep application code free of any flash controller overhead. Additionally, executing DriverLib from ROM lets the user not worry about any significant boundary requirements when verifying flash operations and changing flash banks into special read modes. While it is certainly possible, TI strongly discourages all but the most advanced power users from writing their own flash driver. The strict verification and very sequential programming model makes development of a custom flash driver a somewhat daunting task. Instead, by using the robust and comprehensive peripheral Driver Library, the programmer can achieve optimal performance with little overhead introduced into the application code.

2 Basic Flash Operations

Due to the complexity of the flash controller on the MSP432, simple operations such as erases and programs require a high degree of software overhead and special configuration. Driver Library for MSP430 (referred to as *DriverLib* in this document) provides a versatile array of functions to fully use all of the advanced features of the flash controller. The DriverLib APIs provide a robust front-end that greatly simplifies the user experience and removes the need for the programmer to worry about mundane requirements such as multipulse programming, bit masking, and dynamically changing flash bank read modes. All of these requirements are handled automatically in the software layer and provided to users in a packaged and optimized library.

With the MSP432 flash controller, a program operation may take several attempts before the intended data is actually persisted to memory. Likewise, with an erase operation, several erase attempts might be needed before the erase operation actually goes through and all bits in memory show a high erased state of binary 1. In between these attempts, the specification for the flash controller lays out a set of very strict and stringent requirements for verifying the success or failure of the operation. A full set of these requirements can be found in detail in the comprehensive flowcharts and programming model section of the technical reference manual. TI highly recommends that users utilize the DriverLib APIs for all flash operations and do not attempt to develop their own flash driver. This document does not provide a description of the inner workings of the DriverLib flash APIs. Instead, it provides users with practical and easy-to-follow use cases for the fundamental flash operations APIs and provides examples of advanced programming techniques. Users who are interested in the inner workings of the flash APIs should refer to the flow charts and programming models that are provided by the flash controller chapter of the MSP432P4xx Family Technical Reference Manual.

3 TLV Parsing

As stated in the previous section, one of the required stages of programming or erasing flash memory on the MSP432 is verifying that the operation succeeded. It is not uncommon for the flash controller on the MSP432 to require multiple pulses for an operation to complete successfully. The maximum number of pulses varies per chip and is stored in the device's TLV structure. The TLV structure is a special area of memory on MSP devices that is used to store information such as calibration and configuration information for the device. For the flash controller, two different values are stored: the maximum number of retries to attempt for an erase operation and the maximum number of retries to attempt for a programming operation. If a flash operation takes more tries than these maximum values, the operation is considered an error and should be aborted immediately. While DriverLib's blocking APIs take care of managing the number of retries, these numbers are important to know for those who want to use dual bank erasing or are interested in real-time performance benchmarking and latency. DriverLib provides a specialized API for retrieving information from the TLV structure.

In this API, the user specifies the TLV_TAG_FLASHCTL tag is passed into the API. This is a unique tag that tells DriverLib to parse the TLV structure in memory and find the unique section that is allocated for flash memory information. The second parameter specifies the module instance to search for. With all MSP432 devices, there is only one flash controller, so a zero is passed to this value. The third and fourth parameters are values that are returned from the API to be used in the actual application. The tlvLength parameter is used by the application to verify that the provided TLV tag was found. If the user provided an invalid TLV tag, this value would be returned as zero. The last parameter is a pointer to a structure that contains the TLV information for the flash controller. This structure is defined in sysctl.h as:

```
uint_fast8_t mTries, tlvLength;
SysCtl_FlashTLV_Info *flInfo;

/* Parsing the TLV and getting the maximum erase pulses */
SysCtl_getTLVInfo(TLV_TAG_FLASHCTL, 0, &tlvLength, (uint32_t**) &flInfo);

if (tlvLength == 0 || flInfo->maxErasePulses == 0)
{
    mTries = MAX_ERASE_NO_TLV;
} else
{
    mTries = flInfo->maxErasePulses;
}
```

The DriverLib API automatically searches for the flash information and copies or casts that information into the provided memory structure. The user can then access the information from that structure and use it in the application. In this case, the user would be able to know what the maximum number of retries that are needed for flash program or erase operations to qualify them as errors. If this information is not found (on some pre-release versions of the device, this information has not been populated), the number of tries is set to a constant value.

4 Executing DriverLib APIs From ROM

With the requirement of multiple pulses for programming or erasing, an additional requirement of changing the read mode of the flash bank is introduced. By changing the read mode of the flash bank, the device is put in a "stressed" mode that raises the minimum charge on the flash lane. This allows the user to expose "marginal" bits that are not fully programmed and mask away bits that have already been adequately programmed (and prevent over programming). When the read mode of a flash bank is changed, however, any reads or instruction fetches from that bank become unreliable. When executing code from flash memory, this puts a burden on the programmer to be conscious of which bank they are executing the code from and requires that they do not program out of the same bank they are executing. A simple solution is to use the DriverLib flash APIs that are located in ROM. Every MSP432 device contains a full copy of DriverLib preprogrammed in ROM. When executing from ROM, the user does not have to worry about which bank they are executing from or accidently changing the read mode of the wrong flash bank.

To execute DriverLib APIs from ROM, simply place the MAP_ prefix in front of the desired API call. For the *FlashCtl_programMemory* API for example, using the *MAP_FlashCtl_programMemory* API call instead would map the API call to execute from ROM instead of user memory (either in SRAM or flash depending on the linker file). It is important to note that not all DriverLib APIs have a ROM equivalent (due to technical restrictions or patching). If the *rom_map.h* header file is opened, the user can see a set of preprocessor defines that adds intelligence to using the ROM APIs:

```
#ifdef ROM_FlashCtl_programMemory
#define MAP_FlashCtl_programMemory
\
        ROM_FlashCtl_programMemory
#else
#define MAP_FlashCtl_programMemory
\
        FlashCtl_programMemory
#endif
```

As can be seen, if the ROM API is available, the *MAP_FlashCtl_programMemory* macro maps to the *ROM_FlashCtl_programMemory* version of the API. If the ROM API is not available, the macro maps to the *FlashCtl_programMemory* API that runs in user memory. Due to the strict verification requirements that were discussed earlier, all of the fundamental flash APIs have ROM equivalents. The MAP system is designed as a way to "patch out" ROM function if issues arise later. By using the latest DriverLib release with the latest header files, the user can ensure that the correct DriverLib APIs are being called. For more details on working with DriverLib APIs in ROM, refer to the *DriverLib's User Guide*, which is provided with the DriverLib release.

5 Protecting and Unprotecting Sectors

The first step to any flash operation is to configure the protection settings to allow flash erases or programs to the desired address range in memory. For MSP432, protection is organized by flash banks and flash sectors. Each MSP432 has two banks of memory and a varying number of sectors depending on the memory configuration. The default sector size for MSP432 is static and set to 4096 bytes. This means that for the MSP432P401R (with 256KB of flash memory) there are a total of 64 sectors divided evenly over two banks. For the 128KB MSP432P401M variety, there are a total of 32 sectors divided evenly over two banks.

Configuring which sectors are protected and unprotected can be accomplished by using the *FlashCtl_protectSector* and *FlashCtl_unprotectSector* APIs. These APIs accept two parameters: one for the memory space to protect (information or main memory and bank number), and one for the sector within that memory space to protect or unprotect. The first parameter can consist of the following four values:

- FLASH_MAIN_MEMORY_SPACE_BANK0
- FLASH_MAIN_MEMORY_SPACE_BANK1
- FLASH_INFO_MEMORY_SPACE_BANK0
- FLASH_INFO_MEMORY_SPACE_BANK1

These parameters correspond to the different memory spaces and bank numbers of the MSP432 memory configurations. The main memory space corresponds to the main user memory where the majority of the constant data and code space. The information memory is a special area of memory that contains special information such as the bootloader and user calibration data. For more information about the differences between main and information memory, refer to the MSP432P4xx Family Technical Reference Manual. The bank numbers of each parameter depend on the memory configuration of the device. For the 128KB MSP432P401M variant, bank one of main memory starts at 0x10000 (half of the memory), while for the 256KB variant, bank one starts at 0x20000. The second parameter is the sector to protect or unprotect.

The second parameter that is passed into the protection functions is the sector to protect or unprotect. To calculate the sector from the physical address, use the formula in Equation 1.

(Address – Base Address) / 4096 (1)

Take the address of the memory to protect or unprotect, subtract the base address of the memory section specified in the first parameter (see above), and divide this by the flash block size (for MSP432 MCUs, 4096). For example, the following API call unprotects address 0x23000:

```
FlashCtl_unprotectSector(FLASH_MAIN_MEMORY_SPACE_BANK1, FLASH_SECTOR3);
```

6 Sector Erase (Blocking)

Before any flash programming operation occurs, the user must take steps to ensure that the destination memory to write has been properly erased. The DriverLib APIs provide several erase functions such as single sector erase, mass erase, and a nonblocking dual bank erase. The nonblocking erase is described in Section 10, however this section describes the use of the blocking single sector and mass erase APIs. The term "blocking" means that when the API is called, control is only given back to the user application when the operation has completed or the operation has failed due to an error. The majority of the flash APIs for DriverLib return a Boolean value (true if the operation succeeded, false otherwise).

The following is the prototype of the API used to erase one sector:

```
bool FlashCtl_eraseSector(uint32_t addr)
```

This API accepts one parameter; the address of the sector to erase. Because the minimum granularity for sectors is defined by the sector size (4096 bytes), any address that is given to this function is automatically rounded down and truncated to corresponding aligned sector address. Note that all of the verifications requirements are handled inclusive of the API. While there is a *FlashCtl_verifyMemory* function available, this function is used for the nonblocking dual bank erase.

The mass erase API requires no parameters. The user should simply unprotect the sectors to erase and call the *FlashCtl_performMassErase* function. As with the sector erase function, all verification requirements are handled within the API itself. The API automatically detects which sectors are unprotected and performs verification. If at any point the maximum number of verification failures occurs, a Boolean value of false is returned. For full code examples of how to use both the sector and mass erase functions, refer to the DriverLib release folder.

7 Programming Flash (Blocking)

One of the most fundamental flash operations a user can execute is programming the flash memory with data from an external source. This external source can be anything from a serial protocol such as I^2C to an attached radio such as BLE or Wi-Fi®. DriverLib provides a minimalistic API that handles the entire complex requirement of the MSP432 flash controller. Due to the amount of software overhead that a nonblocking flash program would introduce to the user, DriverLib does not support a nonblocking flash program. TI highly recommendeds that the user exclusively use the DriverLib APIs for not only program, but any flash operation that is needed in the application.

The following is the prototype of the DriverLib API for programming flash:

```
bool FlashCtl_programMemory(void* src, void* dest, uint32_t length)
```

As can be seen, the API simplifies all of the complexities that are associated with programming through the MSP432 flash controller to three simple parameters: a source pointer, a destination pointer, and a length. The DriverLib API dynamically examines the parameters provided and uses the optimal hardware features based on the given situation. The following is a list of guidelines that users can follow to ensure optimal performance from the DriverLib APIs:

- Provide 4096 aligned addresses for both source and destination
- When possible, use larger transfer size (4096 byte plus)
- Make sure transfer size is power of two

As with the erase APIs, refer to the DriverLib release folder for a full set of code examples that show the practical use of these APIs.

8 Interrupts and Blocking APIs

One important aspect to note about using the flash APIs is that each blocking program or erase API described in Section 7 disables master interrupts during its execution. The interrupt state is retained (if interrupts are enabled before the API call, they are re-enabled after the API exits). Because the flash APIs are meant to be run from ROM, any interrupt that occurs could cause the interrupt to access an area of flash that is considered to be in an invalid read mode for reads or instruction fetches. Because of this, the APIs disable global interrupts during the execution window when this corner condition could happen. Any interrupt that pends automatically fires when the interrupts are re-enabled within the API (if interrupts were enabled before calling the API).

9 Performance of DriverLib APIs

The speed of flash program can largely vary on how many attempts each operation takes. Under normal working conditions, operations typically pass on one attempt. At higher and lower extremes of temperature, this can largely vary; however, for most use cases the number of attempts is quite minimal. Table 1 lists the performance of several typical use cases.

Table 1. Typical Flash API Latencies

Operation	Size	Typical Duration (Speed)
FlashCtl_programMemory	4 bytes	39 µs (103 KB/s)
	32 bytes	52.5 µs (609 KB/s)
	4096 bytes (sector)	4.378 ms (936 KB/s)
FlashCtl_eraseSector	One sector	9.085 ms
FlashCtl_performMassErase	64 sectors (all)	11.53 ms

10 Dual Bank Erase (Nonblocking)

Dual bank erase is an advanced flashing technique that lets power users queue off a program operation on one bank and then return to executing user application code. The main benefit to this approach is enabling the user to go into low-power mode and optimize power use while the erase operation occurs. The disadvantage to this approach is that it introduces a level of complexity to the user application as all post-operation verification (and multiple erase attempts) need to be handled in the user application. This application report provides a simple reference example (available for use in GCC, Keil® MDK, IAR Embedded Workbench® IDE, and Code Composer Studio™ IDE).

As the flash controller has strict verification requirements, all of the verification logic that is handled within the blocking API must now be handled within the user application. For simplicity, the following example describes only one specific sector to do the nonblocking erase. Figure 1 shows a flow chart of the complete sample application for nonblocking erase.

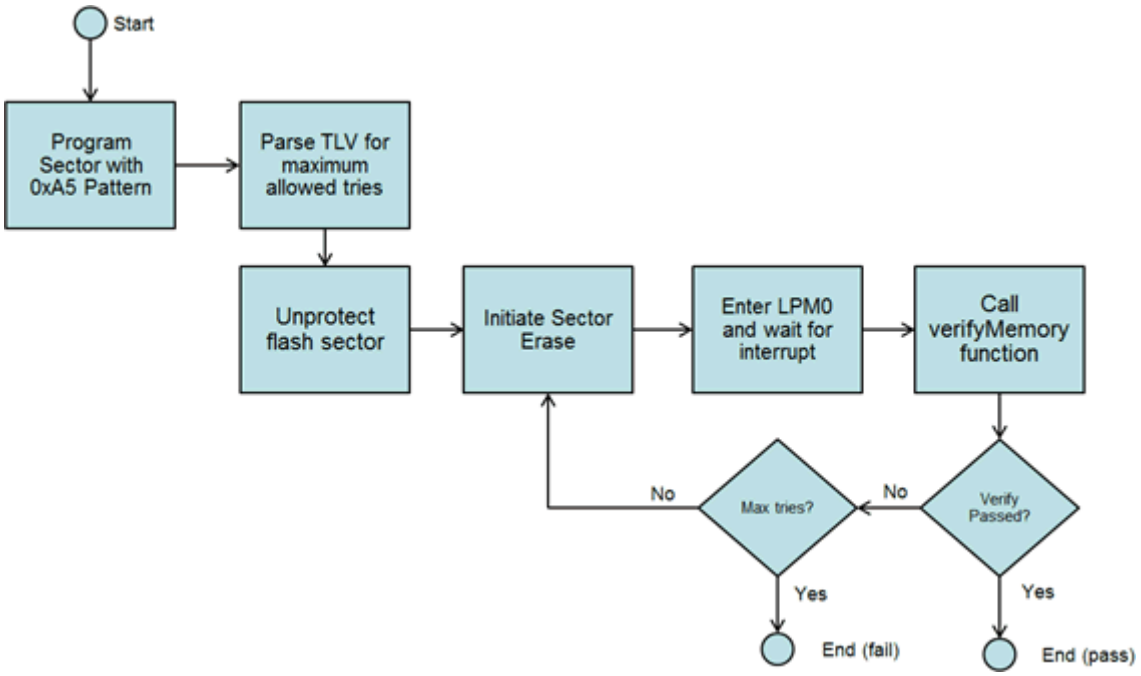

Figure 1. Flow Chart of Dual Bank Erase

The first part of the code must set up the device before initiating the erase. As stated in Section 3, each device has a specific maximum number of retries that can be attempted before a flash operation is configured erroneous. The user is able to get this number with the *SysCtl_getTLVInfo* function as described in Section 3. Additionally to verify functionality of the erase, use the *MAP_FlashCtl_programMemory* API in conjunction with memset to program the sector of memory to a pattern of 0xA5. To perform a persistent operation on memory, the corresponding flash sector must be unprotected:

```
/* Filling our buffer and writing it to memory using the blocking program
 * API
 */
memset(writeBuffer, 0xA5, 4096);
MAP_FlashCtl_unprotectSector(FLASH_MAIN_MEMORY_SPACE_BANK1, FLASH_SECTOR0);
MAP_FlashCtl_programMemory(writeBuffer, (void*) FLASH_ERASE_LOCATION, 4096);
```

After the destination flash sector has been programmed and information has been parsed from the TLV structure, the application can begin to initiate the erase operation. This is done by calling the *MAP_FlashCtl_initiateSectorErase* API with the address of the sector to erase. In this example, program the address 0x20000, which corresponds to the start of bank one. It is important to make sure that interrupts are enabled at this point. When the erase operation completes, an interrupt is fired to the *FLCTL_IRQHandler* interrupt. A breakpoint can be set in this interrupt to observe the flash controller operation.

```
void FLCTL_IRQHandler(void)
{
    uint32_t status;

    status = MAP_FlashCtl_getEnabledInterruptStatus();
    MAP_FlashCtl_clearInterruptFlag(status);

    if (status & FLASH_ERASE_COMPLETE)
    {
        __no_operation();
    }
}
```

After the interrupt is fired, verify that the erase operation completed successfully and that all of the bits that were intended to be erased are at a logical state of 1 binary. To do this, the *MAP_FlashCtl_verifyMemory* function is used in the main loop. This function takes advantage of the "burst read" hardware feature of the flash controller to effectively and efficiently verify all bits of the flash erase operation. If passed, the program flow goes into a LED blink operation to signify successful erasure of the flash sector. If the verify feature fails, the process is repeated the maximum number of times that was specified in the TLV structure. If the maximum number of attempts fails, the program flow turns the LED on P1.0 a solid out to signify failure.

11 References

1. MSP432P401R, MSP432P401M Mixed-Signal Microcontrollers
2. MSP432P4xx Family Technical Reference Manual
3. MSP Peripheral Driver Library

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

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