Programming the On-Chip Flash Memory in a Stellaris® Microcontroller

Application Note

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction

The flash memory of any system is routinely updated and re-programmed. This application note provides three methods for updating (erasing and programming) flash memory: the Stellaris® driver library provided by Luminary Micro, software polling for completed updates, and interrupt-driven updates.

This application note does not cover other issues related to flash memory management, such as the protection of flash memory pages.

For more details on programming the on-chip flash memory, review the flash sections in the appropriate Stellaris family data sheet and the *Stellaris® Peripheral Driver Library User's Guide*. In particular, it is important to understand the flash memory controller's interface registers' functions.

Programming Flash Memory with the Stellaris® Peripheral Driver Library

The Luminary Micro StellarisWare™ peripheral driver library provides an efficient and convenient method to program the integrated flash memory. The driver library is documented in the *Stellaris® Peripheral Driver Library User's Guide*. Three API calls are required to program a page of flash memory: `FlashUsecSet`, `FlashErase`, and `FlashProgram`.

FlashUsecSet Function

The `FlashUsecSet` function sets internal timing parameters in the system's flash memory controller. The memory controller uses the supplied parameter as a prescale timer to provide a constant frequency clock to the flash controller. This function must be called before other functions that erase or program memory to assure proper timing.

The prototype for `FlashUsecSet` is:

```
void FlashUsecSet(unsigned long ulClocks);
```

where *ulClocks* is the internal operating frequency of the processor (in MHz). For example, if the processor is executing at 20 MHz, the correct parameter value is 20.

FlashErase Function

The `FlashErase` function erases a 1 KB page of on-chip flash memory. After erasing, the memory is left in a state where all bits in the page are set. Protected pages (those marked as unreadable and unprogrammable) cannot be erased.

The prototype for `FlashErase` is:

```
long FlashErase(unsigned long ulAddress);
```

where *ulAddress* is the start address of the flash block to be erased and must be a 1 KB-aligned address. The function returns 0 on success or -1 on failure (if an invalid block address was specified or the block is write-protected).

FlashProgram Function

The `FlashProgram` function programs a sequence of words into the on-chip flash memory. Programming consists of ANDing the existing flash memory contents with the new data; the resulting AND of the data being stored in the on-chip memory. If a bit in flash memory is a zero, it cannot be set to a one using the `FlashProgram` function. Restoring a bit to a one is accomplished only using the `FlashErase` function. Moreover, a word may be subject to one or two calls to `FlashProgram`. Additional calls must follow a `FlashErase` function call. Excessive calls to `FlashProgram` degrade a flash memory cell's ability to retain data, and so to guarantee maximal data retention, it is best to completely erase and reprogram a flash page following two calls to `FlashProgram` that affect the same word locations.

The prototype for `FlashProgram` is:

```
long FlashProgram(unsigned long *pulData,
   unsigned long ulAddress,
   unsigned long ulCount);
```

where *pulData* is a pointer to the data to be programmed, *ulAddress* is the starting address in flash to be programmed, and *ulCount* is the number of bytes to be programmed. Since the flash is programmed a word at a time, the starting address and byte count must both be multiples of four.

Only one call to each function is required per flash memory page since the `FlashErase` function call erases a page of flash memory, and the `FlashProgram` function call is capable of programming an arbitrary size of memory.

Examples

Example 1 shows the setup of the **USECRL** register. Example 2 shows a single erase call. Example 3 shows a single programming call. Note that in these and the examples that follow, the path for DriverLib needs to be adjusted relative to the installed location.

Example 1. USECRLR Setup

```
//
// The processor operates at 20 MHz. Set USECRL accordingly.
//
#include "driverlib/flash.h"
unsigned long ulOperatingFrequency = 20;// in MHz

FlashUsecSet(ulOperatingFrequency);
```

Example 2. Single Erase Sequence

```
//
// Erase the first page in the flash
//
#include "driverlib/flash.h"
unsigned long ulAddress = 0x0;
long ret_val;
```

```
ret_val = FlashErase(ulAddress);
if (ret_val == -1)
{
    // Handle an error (for example, protection)
    ...
}

// No error (continue)
...
```

Example 3. Single Program Sequence

```
//
// Program the first word (4 bytes) in the flash
//
#include "DriverLib/src/flash.h"
long ret_val;
unsigned long ulData = 0xA5A5A5A5;
unsigned long ulFlashOffset = 0x0;
unsigned long ulLength = 4;

ret_val = FlashProgram(&ulData, ulFlashOffset, ulLength);
if (ret_val == -1)
{
    // Handle an error (for example, protection)
    ...
}

// No error (continue)
...
```

Programming Flash Memory with Direct Register Writes (Polled)

Although efficient and recommended, the Stellaris driver library is not required to program the flash memory. The flash memory interface registers are available for direct control. This section explains how to program the flash memory using these registers.

USECRL, FMD, FMA and FMC Registers

There are four registers used in erase and programming operations. These registers are written in the following order and provide the following functions.

Note: Flash protection is provided through the **FMPRE** and **FMPPE** registers. Although the page erase size is 1 KB, a protection block is 2 KB. See the data sheet for details.

1. **USECRL: U Second Reload register.** The flash memory control logic contains a divider that takes the processor clock and divides it down to a 1-MHz reference clock to synchronize the timing of flash memory operations; in particular erase operations. The **USECRL** register contains an 8-bit field that is used for reloading this counter. The value programmed into this register is of

the formula ‘frequency (in MHz) minus 1’. This register must be programmed before performing any memory operations.

The symbolic define for this register is `FLASH_USECRL_R`.

2. **FMD: Flash Memory Data register.** This 4-byte register contains the data to write when the programming cycle is initiated. This register is not used during erase cycles.

   The symbolic define for this register is `FLASH_FMD_R`.

3. **FMA: Flash Memory Address register.** During a write operation, this register contains a 4-byte-aligned address and specifies where the data is written. During erase operations, this register contains a 1 KB-aligned address and specifies which page is erased. Note that the alignment requirements must be met by software or the results of the operation are unpredictable.

   The symbolic define for this register is `FLASH_FMA_R`.

4. **FMC: Flash Memory Control register.** This is the final register written and initiates the memory operation. There are four control bits in the lower byte of this register that, when set, initiate the memory operation. The most used of these register bits are the `ERASE` and `WRITE` bits. It is a programming error to write multiple control bits and the result of such an operation is unpredictable.

   The symbolic define for this register is `FLASH_FMC_R`. The symbolic defines for the control bits are: `FLASH_FMC_WRITE`, `FLASH_FMC_ERASE` (page erase), and `FLASH_FMC_MERASE` (mass erase entire flash).

   To protect against accidental access, all writes to FMC must contain the fixed value 0xA442 in bits 31:16. If these bits are not set correctly, the register write will be ignored and no programming operation will take place. These bits can be set by ORing `FLASH_FMC_WRKE`Y into the value written to the register.

After the operation is initiated, software can check for the completion of the operation by examining the **FMC** register. The **FMC** register keeps the control bit set (step 4) until the operation completes. After the operation completes, the register is cleared. This allows software to spin and wait until the operation completes.

Examples

Examples 4, 5, and 6 show direct register access implementations equivalent to the peripheral driver library `FlashUsecSet`, `FlashErase`, and `FlashProgram` functions respectively. Each example requires the appropriate IC header file, lm3sxxx.h, which can be found in the StellarisWare/inc subdirectory. The header file for the LM3S811 part is shown here but this should be replaced with the appropriate header for your target IC.

Example 4. FlashUsecSet-Equivalent Call

```
#include "inc/lm3s811.h"
void
DirectFlashUsecSet(unsigned long ulClocks)
{
    FLASH_USECRL_R = ulClocks - 1;
}
```

Example 5. FlashErase-Equivalent Call

```
#include "inc/lm3s811.h"
long
DirectFlashErase(unsigned long ulAddress)
{
    //
    // The address must be block aligned.
    //
    if(ulAddress & (FLASH_ERASE_SIZE - 1))
    {
        return(-1);
    }

    //
    // Clear the flash access interrupt.
    //
    FLASH_FCMISC_R = FLASH_FCMISC_AMISC;

    //
    // Erase the block.
    //
    FLASH_FMA_R = ulAddress;
    FLASH_FMC_R = FLASH_FMC_WRKEY | FLASH_FMC_ERASE;

    //
    // Wait until the block has been erased.
    //
    while(FLASH_FMC_R & FLASH_FMC_ERASE)
    {
    }

    //
    // Return an error if an access violation occurred.
    //
    if(FLASH_FCRIS_R & FLASH_FCRIS_ARIS)
    {
        return(-1);
    }

    //
    // Success.
    //
    return(0);
}
```

Example 6. FlashProgram-Equivalent Call

```
long
DirectFlashProgram(unsigned long *pulData, unsigned long ulAddress,
             unsigned long ulCount)
{
    //
    // The address and count must be word aligned.
    //
    if((ulAddress & 3) || (ulCount & 3))
    {
        return(-1);
    }

    //
    // Clear the flash access interrupt.
    //
    FLASH_FCMISC_R = FLASH_FCMISC_AMISC;

    //
    // Loop over the words to be programmed.
    //
    while(ulCount)
    {
        //
        // Program the next word.
        //
        FLASH_FMA_R = ulAddress;
        FLASH_FMD_R = *pulData;
        FLASH_FMC_R = FLASH_FMC_WRKEY | FLASH_FMC_WRITE;

        //
        // Wait until the word has been programmed.
        //
        while(FLASH_FMC_R & FLASH_FMC_WRITE)
        {
        }

        //
        // Increment to the next word.
        //
        pulData++;
        ulAddress += 4;
        ulCount -= 4;
    }

    //
    // Return an error if an access violation occurred.
    //
```

```
    if(FLASH_FCRIS_R & FLASH_FCRIS_ARIS)
    {
        return(-1);
    }

    //
    // Success.
    //
    return(0);
}
```

Programming Flash Memory with Direct Register Writes (Interrupt-Driven)

The flash memory controller is capable of generating an interrupt at the completion of a memory operation. Using interrupts potentially allows an application to utilize the time spent waiting for the operation to complete (especially given the low interrupt latencies of the ARM® Cortex™-M3 core).

The software polling loop is removed and replaced with an update to local programming registers.

FCRIS, FCIM, and FCISC Registers

There are three registers used in managing the Flash controller while using interrupts:

1. **FCRIS: Flash Controller Raw Interrupt Status register.** This register contains the raw interrupt status. It is set whenever a flash operation is complete. There are two bits defined in the **FCRIS** register. Bit 0 is set if the logic detects that a programming operation is attempted on a protected page. Bit 1 is set when the logic completes a flash operation.

   The symbolic define for this register is `FLASH_FCRIS_R`. The symbolic define of the two bits are `FLASH_FCRIS_PRIS` and `FLASH_FCRIS_ARIS`, for bits 1 and 0 respectively.

2. **FCIM: Flash Controller Interrupt Mask register.** This register contains bits that control whether a raw interrupt condition is promoted to an interrupt sent to the processor. If a mask bit is set, the raw interrupt is promoted; otherwise, if a mask bit is clear, the raw interrupt is suppressed.

   The symbolic define for this register is `FLASH_FCIM_R`. The symbolic define for the two bits are `FLASH_FCIM_PMASK` and `FLASH_FCIM_AMASK`, for bits 1 and 0 respectively.

3. **FCISC: Flash Controller Interrupt Status and Clear register.** This register contains bits that have a dual purpose. If the register is read, the bits indicate that an interrupt has been generated and sent to the processor. If written, an interrupt can be cleared. Writing a 1 to the register clears the bit. Writing a 0 does not affect the state of the bit.

   The symbolic define for this register is `FLASH_FCISC_R`. The symbolic define for the two bits are FLASH_FCMISC_PMISC and FLASH_FCMISC_AMISC, for bits 1 and 0 respectively.

   The conventional use of this register is as follows. In the interrupt handler, software reads the **FCISC** register and handles the appropriate condition. When complete, the content of the **FCISC** register is written back to itself, which clears the handled condition.

Note that in order to implement an interrupt-driven programming method, all code that executes during the flash program operation must execute out of SRAM memory. Moreover, potential interrupt sources must be masked off. These requirements are to ensure that the flash memory is not accessed while processing the write/erase operation.

Issues to Consider

One issue that this application note does not address is the need for exclusive access of the flash memory during an update operation. If more than one process can update the flash memory, the updates must be safely sequenced.

Also keep in mind the limitations documented in the data sheet. The data sheet specifies the values of each of these limits.

- The flash memory is limited to the number of erase and program cycles.
- Each word may not be subject to more than a specific number of programming cycles before an erase cycle is required. In other words, for any given word, `FlashProgram` can only be called twice before `FlashErase` is called.

Conclusion

There are a number of methods to erase and program the flash memory on the Stellaris family of microcontrollers; from easy to less-than-easy-but-not-complex.

References

The following are available for download at www.luminarymicro.com:

- Stellaris microcontroller data sheet, Publication Number DS-LM3S*nnn* (where *nnn* is the part number for that specific Stellaris family device)
- *Stellaris® Peripheral Driver Library User's Guide*, Document order number SW-DRL-UG
- Stellaris Peripheral Driver Library, Order number SW-DRL

Important Notice

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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