Flash Tools & Memory Management for ATSAM3X8E Microcontroller

Dave Dorzback
Flash & Mem Mgmt for ATSAM3X8E MCU

Abstract—This document will describe the implementation of the 'FlashTools' API, an API for the ATSAM3x8E Microcontroller that allows internal read/write flash operations and memory management via the MCU's integrated Memory Protection Unit (MPU).

I. Introduction

The ATSAM3x8E is an ARM based microcontroller that runs on a 32-bit ARM Cortex M3 Microprocessor. The MCU has 512kb of internal flash in the form of dual 256kb flash banks, 1024 pages each. The aim of this project was to write a low level abstraction layer in the form of an API, which will implement and handle some of the operations and features that, in an OS environment, would typically be part of the kernel. The ATSAM3x8E has no NVRAM or EEPROM, but by enabling internal flash writes, the internal flash can be used as a source of non-volatile data storage for program data. Moreover, interfacing with Cortex-M3's Memory Protection Unit provides the API with control over all memory in the system's memory map, making it possible for memory protection, code separation, and memory management to be implemented.

A. Enhanced Embedded Flash Controller

The Enhanced Embedded Flash Controller (EEFC) manages access performed by masters of the system and enables reading the flash and writing to the write buffer. It ensures the interface of the flash block with the 32-bit internal bus, and manages programming, erasing, locking, and unlocking of the flash. The user interface of the EEFC is integrated within the system controller at the base addresses 0x400E00A00 and 0x400E0C00. There is one EEFC mapped for each internal flash bank.

Each EEFC contains four 32-bit wide registers, as shown in Figure 1, with different bits representing different values. In the EEFC's Flash Command Register (EEFC_FCR), bits 0-7 are for the Flash Command (FCMD), bits 8-23 for the Flash Argument (FARG), and bits 24-31 the Flash Write Protection Key (FKEY). The Flash Command refers to a set of predefined values listed in Figure 2. The Flash Argument is required for several commands and depends on the specific command, for example in flash writes this value should be the number of the flash page to be programmed. The Flash Write Protection Key should be written with the value 0x5A (provided in datasheet), and if a different value is written the command is not performed.

Table 20-4. Register Mapping

Offset	Register	Name	Access	Reset State
0x00	EEFC Flash Mode Register	EEFC_FMR	Read-write	0x0
0x04	EEFC Flash Command Register	EEFC_FCR	Write-only	–
0x08	EEFC Flash Status Register	EEFC_FSR	Read-only	0x00000001
0x0C	EEFC Flash Result Register	EEFC_FRR	Read-only	0x0
0x10	Reserved	–	–	–

Fig. 1. Enhanced Embedded Flash Controller UI [1]

Table 20-2. Set of Commands

Command	Value	Mnemonic
Get Flash Descriptor	0x00	GETD
Write page	0x01	WP
Write page and lock	0x02	WPL
Erase page and write page	0x03	EWP
Erase page and write page then lock	0x04	EWPL
Erase all	0x05	EA
Set Lock Bit	0x08	SLB
Clear Lock Bit	0x09	CLB
Get Lock Bit	0x0A	GLB
Set GPNVM Bit	0x0B	SGPB
Clear GPNVM Bit	0x0C	CGPB
Get GPNVM Bit	0x0D	GGPB
Start Read Unique Identifier	0x0E	STUI
Stop Read Unique Identifier	0x0F	SPUI
Get CALIB Bit	0x10	GCALB

Fig. 2. Enhanced Embedded Flash Controller Commands [1]

To perform commands, the Flash Command Register must be written with the appropriate FCMD, FARG, and FKEY values. Once the command is written, the Flash Ready (FRDY) bit and the Flash Value (FVALUE) field in the Flash Result Register (EEFC_FRR) are cleared, and once the command is completed the FRDY bit is automatically set. If the command was not successful or errors were encountered, error flags are set in the Flash Status Register (EEFC_FSR).

Located in ROM is the In Application Programming (IAP) routine, which sends the desired flash command to the EEFC and waits for the FRDY bit to be set. The IAP function can be retrieved by reading the NMI vector in ROM (0x00800008).

B. Memory Protection Unit

The Coretex-M3 processor has an integrated Memory Protection Unit (MPU) that provides memory control and enables memory protection. Programming of the

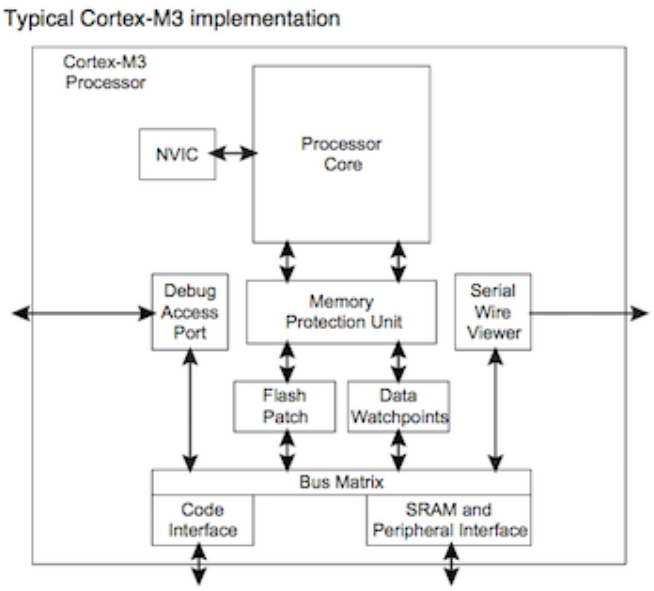

Fig. 3. Typical Cortex-M3 Implementation [1]

Table 12-35. MPU registers summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000ED90	TYPE	RO	Privileged	0x00000800	"MPU Type Register" on page 201
0xE000ED94	CTRL	RW	Privileged	0x00000000	"MPU Control Register" on page 202
0xE000ED98	RNR	RW	Privileged	0x00000000	"MPU Region Number Register" on page 204
0xE000ED9C	RBAR	RW	Privileged	0x00000000	"MPU Region Base Address Register" on page 205
0xE000EDA0	RASR	RW	Privileged	0x00000000	"MPU Region Attribute and Size Register" on page 206
0xE000EDA4	RBAR_A1	RW	Privileged	0x00000000	Alias of RBAR, see "MPU Region Base Address Register" on page 205
0xE000EDA8	RASR_A1	RW	Privileged	0x00000000	Alias of RASR, see "MPU Region Attribute and Size Register" on page 206
0xE000EDAC	RBAR_A2	RW	Privileged	0x00000000	Alias of RBAR, see "MPU Region Base Address Register" on page 205
0xE000EDB0	RASR_A2	RW	Privileged	0x00000000	Alias of RASR, see "MPU Region Attribute and Size Register" on page 206
0xE000EDB4	RBAR_A3	RW	Privileged	0x00000000	Alias of RBAR, see "MPU Region Base Address Register" on page 205
0xE000EDB8	RASR_A3	RW	Privileged	0x00000000	Alias of RASR, see "MPU Region Attribute and Size Register" on page 206

Fig. 4. Summary of MPU Registers [1]

Table 12-37. TEX, C, B, and S encoding

TEX	C	B	S	Memory type	Shareability	Other attributes
b000	0	0	x(1)	Strongly-ordered	Shareable	-
		1	x(1)	Device	Shareable	-
	1	0	0	Normal	Not shareable	Outer and inner write-through. No write allocate.
			1		Shareable	
		1	0	Normal	Not shareable	Outer and inner write-back. No write allocate.
			1		Shareable	
b001	0	0	0	Normal	Not shareable	
			1		Shareable	
		1	x(1)	Reserved encoding		-
	1	0	x(1)	Implementation defined attributes.		-
		1	0	Normal	Not shareable	Outer and inner write-back. Write and read allocate.
			1		Shareable	
b010	0	0	x(1)	Device	Not shareable	Nonshared Device.
		1	x(1)	Reserved encoding		-
	1	x(1)	x(1)	Reserved encoding		-
b1BB	A	A	0	Normal	Not shareable	
			1		Shareable	

Fig. 5. Encodings for TEX, C, B, and S [1]

Table 12-39. AP encoding

AP[2:0]	Privileged permissions	Unprivileged permissions	Description
000	No access	No access	All accesses generate a permission fault
001	RW	No access	Access from privileged software only
010	RW	RO	Writes by unprivileged software generate a permission fault
011	RW	RW	Full access
100	Unpredictable	Unpredictable	Reserved
110	RO	RO	Read only, by privileged or unprivileged software
111	RO	RO	Read only, by privileged or unprivileged software

Fig. 6. Encodings for AP [1]

MPU splits the memory map into regions, each having a defined memory type and attributes. Memory types include Normal, Device, Strongly-ordered, Shareable, and Execute Never. The memory type and attributes of a region determine the behavior of accesses to the region.

An embedded OS typically uses the MPU for memory protection. Attempting to access a memory location prohibited by the MPU or attempting to fetch an instruction from an Execute Never region will cause the processor to generate a Memory Management Fault. This will cause a fault exception, and in an OS environment might lead to termination of the offending process. Moreover, in an OS environment, the MPU can update the MPU settings dynamically based on the process to be executed.

The MPU is located within the System Control Block at base address 0xE000E000 + 0x0D90. The MPU uses registers to define the MPU regions and their attributes, and a summary of them is listed in Figure 4. The MPU Region Attribute and Size Register (RSAR) defines the region size and attributes of the MPU region specified by the MPU Region Number Register (RNR). The access permission bits TEX, C, B, S, AP, and XN of the Region Attribute and Size Register control access to the corresponding memory region. Encodings for these bits are listed in Figures 5 and 6, and suggested memory region attributes are listed in Figure 7.

Table 12-40. Memory region attributes for a microcontroller

Memory region	TEX	C	B	S	Memory type and attributes
Flash memory	b000	1	0	0	Normal memory, Non-shareable, write-through
Internal SRAM	b000	1	0	1	Normal memory, Shareable, write-through
External SRAM	b000	1	1	1	Normal memory, Shareable, write-back, write-allocate
Peripherals	b000	0	1	1	Device memory, Shareable

Fig. 7. Memory Region Attributes [1]

II. Methodology

The API provides an interface in the form of a C++ class (FlashTools) with the MCU's Embedded Flash Controllers and Memory Protection Unit. Moreover, it was developed using the Arduino Due, an ATSAM3x8E-based development board. In the class constructor, the In Application Programming Routine is retrieved from ROM, the flash controllers are initialized (flash wait states and flash access mode set), and Memory Management Fault Exceptions are enabled via the System Control Block.

The API provides the public member functions write(), read(), lock(), unlock() and islocked() for performing internal flash operations. The public member function getPageAddress() can be used to convert a flash page number 0-2047 to a corresponding flash page base address. Moreover, the API provides the public member function MPUConfigureRegion() for programming the MPU.

A. Definitions

In FlashTools.h are struct definitions for representing the EFCs and MPU. Each member represents a Read-Write or Read-Only register. Refer to Figure 1 and Figure 4 for summaries of the EEFC and MPU registers.

```
/* Register Definitions */
typedef volatile       uint32_t RWREG;
/* Read-Write Register */
typedef const volatile uint32_t ROREG;
/* Read-Only Register */

/* Embedded Flash Controller Definition */
typedef struct {
    RWREG EEFC_FMR;
    RWREG EEFC_FCR;
    ROREG EEFC_FSR;
    ROREG EEFC_FRR;
} EfcInstance;

/* MPU Definition */
typedef struct {
    ROREG TYPE;
    RWREG CTRL;
    RWREG RNR;
    RWREG RBAR;
    RWREG RASR;
    RWREG RBAR_A1;
    RWREG RASR_A1;
    RWREG RBAR_A2;
    RWREG RASR_A2;
    RWREG RBAR_A3;
    RWREG RASR_A3;
} MpuInstance;
```

These type definitions are then used to cast the EFC base addresses and MPU base address to pointers of EfcInstance and MpuInstance types. The following private members of the FlashTools class define the mapped EFC and MPU:

```
EfcInstance *efc = ((EfcInstance*)EFC0_ADDR);
MpuInstance *mpu = ((MpuInstance*)MPU_ADDR)
```

Base addresses for the internal flash banks, ROM, EFCs, MPU, and IAP are also defined here:

```
#define IFLASH0_ADDR          (0x00080000u)
#define IFLASH1_ADDR          (0x000C0000u)
#define IROM_ADDR             (0x00100000u)
#define EFC0_ADDR             (0x400E0A00u)
#define EFC1_ADDR             (0x400E0C00u)
#define MPU_ADDR              (0xE000E000ul + 0x0D90ul)
#define SCB_ADDR              (0xE000E000ul + 0x0D00ul)
#define IAP_ENTRY_ADDRESS     (IROM_ADDR + 8)
```

Flash Commands from Figure 2 that are used by the API, and the Flash Write Protection Key, are also defined as macros:

```
#define EFC_FCMD_EWP 0x03
/* Erase page and write page */
#define EFC_FCMD_EA  0x05  /* Erase all */
#define EFC_FCMD_CLB 0x09  /* Clear lock bit */
#define EFC_FCMD_SLB 0x08  /* Set lock bit */
#define EFC_FCMD_GLB 0x0A  /* Get lock bit */

#define FWP_KEY      0x5Au
```

B. Flash Layout & Sending Flash Commands

The ATSAM3x8E uses a paged storage system for internal flash in the form of dual 256kb flash banks. Each bank contains 1024 pages with 256b per page. Some of these properties are defined in FlashTools.h:

```
#define IFLASH0_SIZE                (IFLASH1_ADDR - IFLASH0_ADDR)
#define IFLASH1_SIZE                (IFLASH0_SIZE)
#define IFLASH_ADDR                 (IFLASH0_ADDR)
#define IFLASH_PAGE_SIZE            (256u)
#define IFLASH_NB_OF_PAGES          (1024u)
#define IFLASH_LOCK_REGION_PAGES    (64u)
#define IFLASH_WORD_SIZE            (sizeof(uint32_t))
#define IFLASH_LOCK_REGION_SIZE     (IFLASH_PAGE_SIZE * IFLASH_LOCK_REGION_PAGES)
#define IFLASH_WORDS_PER_PAGE       (IFLASH_PAGE_SIZE / IFLASH_WORD_SIZE)
#define IFLASH_LAST_PAGE_ADDRESS    (IFLASH1_ADDR + IFLASH1_SIZE - IFLASH_PAGE_SIZE)
#define IFLASH_TOTAL_PAGES          (IFLASH0_NB_OF_PAGES + IFLASH1_NB_OF_PAGES)
```

The public member function getPageAddress() accepts a page number 0-2047 and an optional offset value and will return the corresponding address at that page and offset. Page number arguments between 0 and 1023 correspond to pages in internal flash bank 0, while pages 1024-2047 correspond to internal flash bank 1. Addresses are calculated with the formula flash bank start address + (IFLASH_PAGE_SIZE * page number) + offset:

```
template <typename Type>
Type *FlashTools::getPageAddress(uint32_t page_num, uint32_t offset = 0) {
    if (page_num < 0 || page_num >= IFLASH_TOTAL_PAGES) {
        return NULL;
    }
    if (page_num <= IFLASH_NB_OF_PAGES) {
        return reinterpret_cast<Type *>(IFLASH0_ADDR + (IFLASH_PAGE_SIZE * page_num)) + offset;
    } else {
        page_num \%= IFLASH_NB_OF_PAGES;
        return reinterpret_cast<Type *>(IFLASH0_ADDR + (IFLASH_PAGE_SIZE * page_num)) + offset;
    }
}
```

C. In Application Programming Routine

The In Application Programming Routine is represented in the code as a private member of the FlashTools class having a function pointer type. The type definition and static member declaration are as follows:

```
typedef uint32_t (*IAP_FPTR)(uint32_t EFCidx, uint32_t cmd);
static IAP_FPTR IAP;
```

The IAP_FPTR member is then assigned a value in the class constructor by reading the NMI vector in ROM:

```
IAP = (uint32_t(*)(uint32_t EFCidx, uint32_t cmd)) *((uint32_t *)IAP_ENTRY_ADDRESS);
```

The IAP routine has two parameters; the first being the EFC number to write the command to (0 for EFC0, i.e. doing an operation on internal flash 0, or 1 for EFC1, i.e. doing an operation on internal flash 1), and the second

20.5.2 EEFC Flash Command Register

Name: EEFC_FCR

Address: 0x400E0A04 (0), 0x400E0C04 (1)

Access: Write-only

Offset: 0x04

31	30	29	28	27	26	25	24
FKEY							
23	22	21	20	19	18	17	16
FARG							
15	14	13	12	11	10	9	8
FARG							
7	6	5	4	3	2	1	0
FCMD							

Fig. 8. Flash Command Register [1]

being the Flash Command value. The private member function cmd() acts as a wrapper around the IAP function, and has a parameter for the Flash Command value and Flash Argument value:

```
uint32_t FlashTools::cmd(uint32_t cmd, uint32_t arg) {

    /* EFC Flash Command Register definition */
    union {
        uint32_t FULL;
        struct {
            uint32_t FCMD:8;  // Flash Command (8) - bits 0-7
            uint32_t FARG:16; // Flash Argument (16) - bits 8-23
            uint32_t FKEY:8;  // Flash Write Protection Key (8) - bits 23-31
        } SECTION;
    } EFC_FCR_REGISTER;

    EFC_FCR_REGISTER.FULL = 0;                  // Init. all bits to 0
    EFC_FCR_REGISTER.SECTION.FCMD = cmd;        // Set bits 0-7 with flash command
    EFC_FCR_REGISTER.SECTION.FKEY = FWP_KEY;    // Set bits 8-23 with flash argument
    EFC_FCR_REGISTER.SECTION.FARG = arg;
// Set bits 23-31 with flash write protection key

    /* Send the corresponding EFC index and command */
    IAP((efc == EFC0 ? 0 : 1), EFC_FCR_REGISTER.FULL);

    /* Return Flash Status Register value -- 0 if successful or error flags */
    return (efc->EEFC_FSR & EEFC_ERROR_FLAGS);
}
```

In the cmd() function a union type is used to represent the EEFC's Flash Command Register. As mentioned previously and shown in Figure 8, the Flash Command Register has three parts: FCMD, FARG, and FKEY. The union has two members, an unsigned 32-bit integer (FULL) representing the full register value, and a struct with three bit fields (SECTION) representing the three separate register values. Since both members of the union share the same memory, this allows the full 32-bit value to be accessed in its entirety through the FULL member, or for the individual register values to be accessed/set through the SECTION member. In the function the FCMD, FKEY, and FARG values are set via the SECTION member, and in the next line the EFC number and register value are passed to the IAP routine to perform the command. The function then returns the value of the Flash Status Register, which will be 0 upon success or an error flag.

D. Internal Flash Writes

The API allows internal flash writes to be performed via the write() function. This function has parameters for the flash address to write to, a buffer containing the data to write, and the size of the data to write. The function will validate that the address is valid then check if the flash is locked via the islocked() function. If the flash is locked, the unlock() function is used to unlock the region of flash where the data is to be written.

```
template<typename Type>
uint32_t FlashTools::write(uint32_t addr, Type *data, uint32_t data_size) \{

    /* Validate flash address then unlock flash region */
    if (addr >= IFLASH_LAST_PAGE_ADDRESS + IFLASH_PAGE_SIZE || addr < IFLASH_ADDR || addr & 3) {
        return INVALID;
    } else if (islocked(addr, addr + data_size - 1)
                && unlock(addr, addr + data_size - 1) != SUCCESS) {
        return ERROR;
    }
```

Afterwards, the address argument is compared to internal flash bank 1's base address to determine which flash bank the data is being written to. The write command must be sent to the appropriate flash controller, which depends on which bank the data is being written in. Either EFC0 or EFC1 is used, depending on which internal flash bank the address is located in. The page number the address is located on is then calculated with (address - flash bank start address) / flash page size, and offset from the start of the page with (address - flash bank start address) % flash page size.

```
    /* Determine whether addr is in flash bank 0 or 1 and set appropriate flash bank start
       address and EFC instance (EFC0 for flash bank 0, EFC1 for flash bank 1)
*/
    const uint32_t FLASH_START_ADDR {addr >= IFLASH1_ADDR ? IFLASH1_ADDR : IFLASH0_ADDR};
    efc = addr >= IFLASH1_ADDR ? EFC1 : EFC0;

    /* Calcuate page number of addr and offset of addr from start of page */
    uint16_t page_num {(addr - FLASH_START_ADDR) / IFLASH_PAGE_SIZE};
    uint16_t offset   {(addr - FLASH_START_ADDR) \% IFLASH_PAGE_SIZE};
```

Flash pages must be programmed one page at a time, so a loop is used to write the data to flash one page at a time. One full page is written each iteration before sending the write page command. On each iteration, write_size represents the size of new data being written to the page. This value is calculated at the beginning of each iteration by taking the minimum of IFLASH_PAGE_SIZE - offset (the size of a flash page minus the address' offset from the page start) and data_size (size of remaining data to be written). Note that offset will be non-zero on the first iteration if the address argument is offset from the beginning of the flash page it's located on, and for all other iterations it will be zero. The page address is then calculated each iteration by flash bank start address + page number * flash page size, then padding size is calculated by flash page size - offset - write_size. Also note that the padding size will be non-zero on the last iteration if the remaining data to write does not take up a full page, and for all other iterations it will be zero.

```
    /* Write all data one flash page at a time until all data has been written */
    for (uint32_t write_size; data_size > 0; data_size -= write_size) {
        write_size = IFLASH_PAGE_SIZE - offset < data_size ? IFLASH_PAGE_SIZE - offset : data_size;

        // Calculate page address and padding size
        uint32_t page_address {FLASH_START_ADDR + page_num * IFLASH_PAGE_SIZE};
        uint16_t padding_size {IFLASH_PAGE_SIZE - offset - write_size};

        // Copy 1 page of data to flash in 3 parts: offset, data, padding
        flashcpy(page_address, data, offset, write_size, padding_size);

        // Send EFC command. Return error flag on failure
        if (cmd(EFC_FCMD_EWP, page_num) != SUCCESS) {
            return efc->EEFC_FSR & EEFC_ERROR_FLAGS;
        }

        data += (write_size/sizeof(Type));
        ++page_num;
        offset = 0;
    }
    setfws(fws);
    return SUCCESS;
\}
```

The private flashcpy() function is then passed the page address, data to be written, offset size, write size, and

padding size. This function copies data into a buffer in three parts: offset data, write data, and padding data, then writes the contents of this buffer to flash the flash page. Data of size offset is copied from the page address into the buffer, then data of size write size is copied from the data buffer argument into the buffer, and finally data of size padding size is copied from the flash page to the buffer. The contents of the buffer are then written to the flash page in 32-bit words. The page data is copied to this intermediary buffer first and then written to flash to preserve data on the page that will not be overwritten and to avoid issues with alignment.

Once data is done being copied to the flash page, the cmd() function is used to send the write page command and program the flash page. The write() function returns 0 if all data was written to flash successfully.

E. MPU Programming

The API's public member function MPUConfigureRegion() provides control over memory via the MPU. The function has three parameters related to details about the memory to be configured: addr, address of memory to configure; size, size of memory to configure; and region, the region to configure. Additionally, there are six access permission attribute parameters tex, c, b, s, ap, and xn (refer to Figures 5 & 6).

The function calls the Data Synchronization Barrier instruction, which ensures the effect of the MPU takes place immediately at the end of context switching, and the Instruction Synchronization Barrier instruction, which ensures the new MPU setting takes place immediately after programming the MPU regions.

```
uint32_t FlashTools::MPUConfigureRegion(uint32_t *addr, uint32_t size, uint32_t region,
                                        uint32_t tex, uint32_t c, uint32_t b, uint32_t s,
                                        uint32_t ap, uint32_t xn) \{

    /* Data Synchronization Barrier -- see datasheet pg. 75, 149 */
    /* Instruction ensures effect of MPU takes place i
        immediately at the end of context switching */
    __DSB();

    /* Instruction Synchronization Barrier -- see datasheet pg. 75, 150 */
    /* Instruction ensures new MPU setting takes effect
        immediately after programming MPU regions */
    __ISB();
```

It then defines union types for the three MPU registers values will be set in, the Region Control Register (CTRL), the Region Base Address Register (RBAR), and the Region Size and Attribute Register (RASR). These MPU registers are represented using union types in a way analogous to how the EEFC's Flash Command Register is represented in the cmd() function, and this allows for the setting of individual register values.

```
    /* MPU CTRL Register -- see datasheet page 202 */
    union {
        uint32_t FULL;
        struct {
            uint32_t ENABLE:1;
            uint32_t HFNMIENA:1;
            uint32_t PRIVDEFENA:1;
            uint32_t RESERVED0:29;
        } SECTION;
    } MPU_CTRL_REGISTER;

    /* MPU RBAR Register -- see datasheet page 205 */
    union {
        uint32_t FULL;
        struct {
            uint32_t REGION:4;
            uint32_t VALID:1;
            uint32_t ADDRESS:27;
        } SECTION;
    } MPU_RBAR_REGISTER;

    /* MPU RASR Register -- see datasheet page 206 */
    union {
        uint32_t FULL;
        struct {
            uint32_t ENABLE:1;
            uint32_t SIZE:5;
            uint32_t RESERVED0:2;
            uint32_t SRD:8;
            uint32_t B:1;
            uint32_t C:1;
            uint32_t S:1;
            uint32_t TEX:3;
            uint32_t RESERVED1:2;
            uint32_t AP:3;
            uint32_t RESERVED2:1;
            uint32_t XN:1;
            uint32_t RESERVED3:3;
        } SECTION;
    } MPU_RASR_REGISTER;
```

Values for the MPU's RBAR and RASR register are then built by setting the values in the register:

```
    /* Set MPU Register Base Address Register -- see datasheet pg. 205 */
    MPU_RBAR_REGISTER.FULL              = 0;
    MPU_RBAR_REGISTER.SECTION.REGION  = region;
    MPU_RBAR_REGISTER.SECTION.VALID   = 1;
    // Region size in bytes = 2^(size+1) -- datasheet pg. 207
    MPU_RBAR_REGISTER.SECTION.ADDRESS = ((uint32_t)addr >> 5) & (0xffffffff << (size - 4));

    /* MPU Register Attribute and Size Register -- see datasheet pg. 206 */
    MPU_RASR_REGISTER.FULL              = 0;
    MPU_RASR_REGISTER.SECTION.SIZE    = size;
    MPU_RASR_REGISTER.SECTION.ENABLE = 1;
    MPU_RASR_REGISTER.SECTION.SRD     = 0;
    /* See datasheet pg. 207-209 for attribute tables */
    MPU_RASR_REGISTER.SECTION.TEX = tex;
    MPU_RASR_REGISTER.SECTION.C   = c;
    MPU_RASR_REGISTER.SECTION.B   = b;
    MPU_RASR_REGISTER.SECTION.S   = s;
    MPU_RASR_REGISTER.SECTION.AP  = ap;
    MPU_RASR_REGISTER.SECTION.XN  = xn;

    /* MPU Control Register -- see datasheet pg. 202 */
    MPU_CTRL_REGISTER.SECTION.PRIVDEFENA = 1;
    MPU_CTRL_REGISTER.SECTION.HFNMIENA   = 0;
    MPU_CTRL_REGISTER.SECTION.ENABLE     = 1;
```

Values for the CTRL register are then set, which will enable the MPU:

```
    /* MPU Control Register -- see datasheet pg. 202 */
    MPU_CTRL_REGISTER.SECTION.PRIVDEFENA = 1;
    MPU_CTRL_REGISTER.SECTION.HFNMIENA   = 0;
    MPU_CTRL_REGISTER.SECTION.ENABLE     = 1;
```

Finally, the values are assigned to the actual mapped MPU registers and the function returns:

```
    /* Set MPU Registers */
    mpu->RBAR = MPU_RBAR_REGISTER.FULL;
    mpu->RASR = MPU_RASR_REGISTER.FULL;
    mpu->CTRL = MPU_CTRL_REGISTER.FULL;

    return SUCCESS;
}
```

Setting the RBAR, RASR, and CTRL register values enables the MPU and configures the memory at the address argument with the access permission attribute arguments.

III. Results

In order to test the API, an Arduino sketch using the API was made and uploaded to the Arduino Due. A through-hole LED was connected between pin 13 on the Arduino Due and GND.

```
#include "FlashTools.h"
#include <Arduino.h>

int ledPin = 13;        // LED connected to this pin
```

In addition, the variables uint32_t blinks and uint32_t *flash_addr and a FlashTools object are defined in the sketch:

```
int ledPin = 13;        // LED connected to this pin
uint32_t blinks[1];     // How many times the LED blinks
uint32_t *flash1_addr; // Pointer to hold flash address
FlashTools flash1;      // FlashTools object
```

An Interrupt Service Routine is also defined in the sketch to handle Memory Mangement Faults. If a Memory Management Fault occurs, an error message is printed to the serial monitor:

```
/* Interrupt Service Routine for Memory Management Faults */
ISR(MemManage_Handler) {
    while (1) {
      SerialUSB.println("Error: Memory Management Fault");
    }
}
```

The sketch has a setup() function, which runs once when the microcontroller is powered on, and a loop() function, which loops continuously after setup(). In the setup() function, an address is obtained for flash page 1030, then a single uint32_t value is read in from this address to the blinks variable. If no value has been written to flash here, the blinks variable is set to 1, otherwise, the value is incremented. If the value of blinks is greater than 3, the MPU is enabled and region for the flash address is set to read-only. Finally, the value for blinks is written back to flash at the same address.

```
/* Set up - Runs once on power up*/
void setup() {
  SerialUSB.begin(9600);

  // Get start address for flash page 1030
  flash1_addr = flash1.getPageAddress<uint32_t>(1030);

  // Read in value 'blinks'
  uint32_t val = flash1.read<uint32_t>(flash1_addr);
  blinks[0] = val == 0xffffffff ? 1 : val + 1;

  if (blinks[0] > 3) {
    flash1.MPUConfigureRegion(flash1_addr, 4, 0,
                              0b000, 1, 0, 0, 0b101, 1);
  }

  // Write blinks back to flash at the same address
  flash1.write<uint32_t>(flash1_addr, blinks, sizeof(uint32_t));
}
```

In the main program loop, a for loop will briefly write the LED pin high before writing it low again for a number of times based on the value of 'blinks'. The program then sleeps for 5 seconds before blinking the LED again.

```
void loop() {
  // Blink LED 'blinks' times
  for (int i = 0; i < blinks[0]; ++i) {
      digitalWrite(ledPin, HIGH);
      delay(500);
      digitalWrite(ledPin, LOW);
      delay(200);
  }
  // Sleep for 5 seconds
  delay(5000);
}
```

Based on this sketch, it's expected for the LED to blink a single time every five seconds after the MCU is initially programmed. Then, after resetting the MCU or powering it off/on, the LED should blink twice every five seconds if the data is being written to flash successfully. Moreover, after resetting the MCU or connecting/disconnecting power 3 times, the MPU should be enabled and the region of flash where the value for blinks is being written protected as read-only. After protecting the region of flash as read-only, the subsequent write attempt should trigger a Memory Management Fault, and the ISR should print an error message to the serial monitor.

When the Arduino Due is programmed with this sketch the expected results do occur. The LED blinks one time every 5 seconds initially, then an additional time every 5 seconds when powering the MCU off/on. This indicates the value is being successfully written to and read from flash, and the value persists in flash even when the MCU is powered off. Moreover, after powering the MCU off/on 3 times the value for blinks reaches 4, and the address where it's being written is successfully protected as read-only via the MCU. This is indicated because the LED will no longer blink at this point, and instead a Memory Management Fault occurs and an error is printed to the serial monitor by the ISR.

IV. Conclusion

This API provides an interface with the ATSAM3X8E's embedded flash controllers and memory protection unit hardware registers, and as a result gives the user significant control over system storage and memory.

While the microcontroller does not include any built in EEPROM or NVRAM, non-volatile data stroage of program data is still possible without adding any external peripherals. Using the in application programming routine stored in the NMI vector of ROM, internal flash writes are possible and flash memory can be used as a source of non-volatile data storage.

Moreover, while the ATSAM3X8E's microprocessing unit does not have a memory management unit or virtual address space, memory protection and management can be achieved by interfacing with the Cortex-M3's integrated memory protection unit. This provides the user with control over all memory in the system memory map, flash, RAM, device, etc., and allows for security privilege levels, code separation, and more.

References

[1] Atmel. "AT91SAM ARM-based Flash MCU, SAM3X/SAM3A Series," ATSAM3X8E datasheet, July 2012.