Freescale Semiconductor
Application Note

Document Number:AN4695
Rev. 0, 04/2013

Avoiding Read While Write Errors When Developing In-Software Flash Programming Applications for Kinetis and ColdFire+ MCUs

by: Chris Brown, Maclain Lobdell, and Melissa Hunter

Contents

1 Overview

This application note discusses important considerations and guidelines for implementing in-software flash programming on Kinetis and ColdFire+ MCUs. The methods described provide a means to perform in-software flash programming on devices with a single flash block or devices that do not allow memory reads while writes are occurring within the same block.

The techniques described are utilized in the Kinetis 100 MHz sample code. Also, the examples provided in this application note use the TFS Flash Driver Software for Kinetis and ColdFire+ MCUs. See References section for details.

The examples provided in the application note support:

- IAR Embedded Workbench 6.40
- Freescale CodeWarrior 10.2/10.3 with Freescale compiler
- Freescale CodeWarrior 10.3 with GCC compiler

Any other development environment must be able to support the methods described.

Contact IAR (**iar.com**) or Freescale CodeWarrior (**freescale.com/CodeWarrior**) for additional help and support.

2 Introduction: In-software flash programming

Many applications require the storage of data into non-volatile memory while in operation. The ability to erase/reprogram embedded flash memory in-software enables the use of a bootloader to upgrade the application software or the capability to securely store valuable data.

Embedded flash memory is grouped into blocks. Each block contains the circuitry required to read, erase, and program within that block. Most of the flash memory technologies have a limitation of not allowing read operations at the same time as an erase or program operation is occurring within the same block. Thus, erasing/programming a sector is not allowed if code execution (fetching instructions = reading) is taking place within the same block even if the read is in a different flash sector than the erase/program. This is called a Read While Write (RWW) violation and will result in a Read Collision error on Kinetis and other microcontrollers.

To determine whether or not there is a potential for a Read While Write violation, consult the Flash Memory Configuration section of the Chip Configuration Chapter for a specific part and/or the *.map file of the program. The Flash Memory Configuration section contains a table that correlates the specific part numbers to the amount of flash memory, flash blocks, and locations of the flash blocks. An example of such a table is given below.

Device	Program flash (KB)	Block 0 (P-Flash) address range	FlexNVM (KB)	Block 1 (FlexNVM/P-Flash address range	FlexRAM (KB)	FlexRAM address range
MK60DN256ZVLQ10	256	0x0000_0000 – 0x0001_FFFF	–	0x0002_0000 – 0x0003_FFFF	–	N/A
MK60DX256ZVLQ10	256	0x0000_0000 – 0x0003_FFFF	256	0x1000_0000 – 0x1003_FFFF	4	0x1400_0000 – 0x1400_0FFF
MK60DN512ZVLQ10	512	0x0000_0000 – 0x0003_FFFF	–	0x0004_0000 – 0x0007_FFFF	–	N/A
MK60DN256ZVMD10	256	0x0000_0000 – 0x0001_FFFF	–	0x0002_0000 – 0x0003_FFFF	–	N/A
MK60DX256ZVMD10	256	0x0000_0000 – 0x0003_FFF	256	0x1000_0000 – 0x1003_FFFF	4	0x1400_0000 – 0x1400_0FFF
MK60DN512ZVMD10	512	0x0000_0000 – 0x0003_FFF	–	0x0004_0000 – 0x0007_FFFF	–	N/A

The *.map file will list the locations of all of the functions of the program being used. Through this file, the user can know the location where flash programming function is stored as well as any other functions that may be pertinent to avoiding Read While Write errors, (that is, interrupt service routines). For help on how to enable a specific tool to produce a *.map file, consult the help documentation of the specific toolset.

2.1 Avoiding Read While Write violations: Flash command code

The following subsections describe two simple methods to workaround this restriction.

2.1.1 Method 1: Execute flash commands from a separate flash block

The first method is to execute the flash erase/program software subroutines from a different flash block than the one with the sectors being erased/programmed. This works well if the microcontroller has multiple flash blocks. With multiple flash blocks, the application can place the flash subroutines in one block and designate all or part of the other block for data storage.

2.1.2 Method 2: Execute flash commands from SRAM

The second method is recommended for situations when the MCU has only flash block, or the user needs to erase/reprogram within each available block. In these scenarios, the critical parts of the flash subroutines are moved into SRAM for execution. Thus, the Read While Write condition on the flash is avoided.

2.2 Avoiding Read While Write violations: ISR code

If an interrupt occurs during a flash erase/program operation, the Read While Write restriction will be violated if the interrupt service routine (ISR) code is located in the same flash block as the erase/program operation is occurring on.

This situation can be avoided with three possible options:

- The first option is to disable/enable interrupts before/after the flash command operation. If interrupts are disabled, any interrupt that occurs during the flash operation will be pending when interrupts are re-enabled.
- The second option is to ensure that the ISR code is never in the same block as in-software flash erase/programming is performed.
- The third option is to relocate the expected ISR into RAM for the duration of the erase or program time. See Instructions for creating a RAM function for more details on how to create a RAM function.

2.3 What this application note demonstrates

This application note shows how to create RAM functions for executing flash commands in RAM using two Integrated Development Environments, Freescale CodeWarrior and IAR Embedded Workbench. The examples are based on C90TFS Standard Flash Software Drivers for Kinetis and ColdFire+ Microcontrollers, but the concepts apply to other Freescale microcontrollers and software as well. Examples of the RAM function methods can be found in the Kinetis 100 MHz sample code projects. See References section.

2.4 Example application

Consider an example application in which the user wants to erase/reprogram sectors defined by the application as the “data log region” while executing from the “code execution region”. Both of these regions are within the same flash block (P-Flash Block 0). To avoid the Read While Write violation of erasing/programming within the same block as code is executing, use a RAM function to launch the flash commands. This function is copied to SRAM from flash during system initialization.

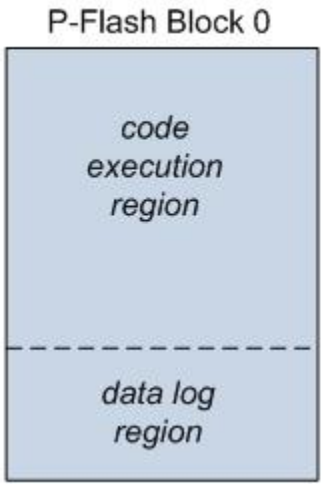

Figure 1. Use-case example: The application code and data log regions are within the same flash block (P-flash block 0).

3 Procedure

The application executes from the “code execution region” in P-Flash Block 0. When the application wants to erase/reprogram a sector in the “data log region”, it first prepares to execute the flash commands by loading the flash registers.

1. Before jumping to SRAM, set the Flash CCOB registers to prepare to launch a flash command.
2. Call an SRAM function which launches the flash command by clearing the CCIF flag, then waits for the CCIF flag to set, indicating the command is complete.
3. After the command is complete, return from the SRAM function back to flash and continue code execution.

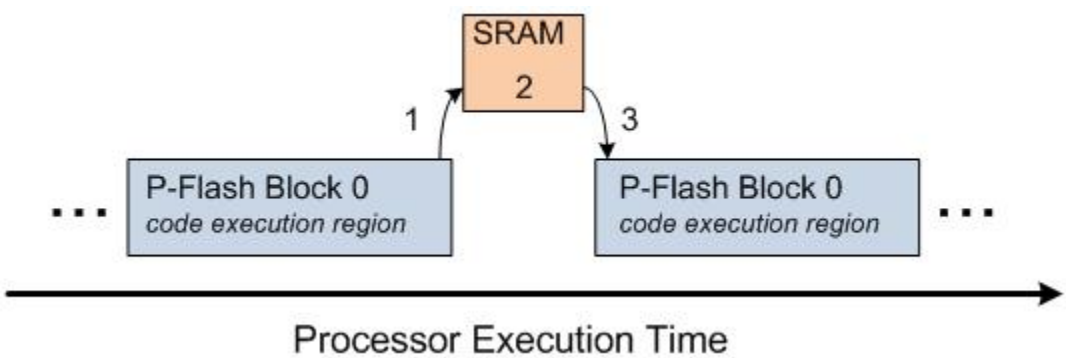

Figure 2. Code Execution: Code execution jumps from flash, to SRAM, and then back to flash.

4 Size of RAM function

The function presented below to execute flash commands in SRAM (FlashLaunchCommand) takes approximately 16 bytes of SRAM memory.

5 Using the C90TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers

The examples in this application note utilize the C90TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers from Freescale.

Following are the instructions on how to add C90TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers into the project.

1. Add C90TFS driver source/header files into the project.
2. Add C90TFS driver paths to the project settings.
3. Include C90TFS driver header files needed in the user source files that will be accessing the flash commands.
4. In SSD_FTFx.h, be sure the FLASH_DERIVATIVE is set to the right configuration for the specific device being used.

```
For MD50DN512CMB10 - "N512":
    #define FLASH_DERIVATIVE     FTFx_KX_512K_0K_0K_2K_0K
For MK50DX256CMD10 - "X256":
    #define FLASH_DERIVATIVE     FTFx_KX_256K_256K_4K_2K_2K
```

5. Be sure to declare a global structure for the flash information, as given below.

```
/* Flash driver structure */
FLASH_SSD_CONFIG ftfl_cfg =
    {
        FTFx_REG_BASE,          /* FTFx control register base */
        PFLASH_BLOCK_BASE,      /* base address of PFlash block */
        PBLOCK_SIZE,            /* size of PFlash block */
        DEFLASH_BLOCK_BASE,     /* base address of DFlash block */
        DBLOCK_SIZE,            /* size of DFlash block */
        EERAM_BLOCK_BASE,       /* base address of EERAM block */
        EERAM_BLOCK_SIZE,       /* size of EERAM block */
        0,                      /* size of EEE block */
        DEBUGENABLE,            /* background debug mode enable bit */
        NULL_CALLBACK           /* pointer to callback function */
    };
```

6. Then initialize the flash driver by calling the `init` function.

```
printf("Initializing Flash Driver: ");
returnCode = pFlashInit(&ftfl_cfg);
```

7. Finally, the C90TFS Flash Driver Software functions are ready to be used in the application. Here is an example of the `swap` function being called:

```
returnCode = pPFlashSwap(&ftfl_cfg, FLASH_SWAP_INDICATOR_ADDR, SwapCallback,
FlashCommandSequence);
```

6 Instructions for creating a RAM function

6.1 IAR Embedded Workbench 6.40

1. In the linker file, initialize the .textrw_init and .textrw sections manually, using the code given below.

```
initialize manually { section .textrw_init };
initialize mnaually { section .textrw };
```

2. Define and place sections in the linker file.

CodeRelocate is a .textrw_init section placed in ROM. This is a flash section which will store RAM functions to be copied to RAM when the system initializes.

CodeRelocateRam is a .textrw section placed in RAM. This is a RAM section, which will be the home of RAM functions after they are copied from flash.

```
define block CodeRelocate { section .textrw_init };
define block CodeRelocateRam { section .textrw };

place in ROM_region   { readonly, block CodeRelocate};

place in RAM_region   { readwrite, block CodeRelocateRam,
                        block CSTACK, block HEAP };
```

3. In the software source files, prefix the function prototype with "`__ramfunc`". IAR will place this function in the CodeRelocate section to be copied to CodeRelocateRam.

   The C90TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers utilize a `FlashCommandSequence` function for executing all flash commands. This entire function can be placed into SRAM. This is the simplest method to implement, but it is the least SRAM-efficient. To reduce SRAM usage, just a critical subset can be placed in SRAM.

   The following function, just performs the critical steps that need to run from SRAM when launching a flash command.

```
/* Flash launch command */
__ramfunc UINT32 FlashLaunchCommand (PFLASH_SSD_CONFIG PSSDConfig)
{
    /* clear CCIF bit */
    REG_WRITE(PSSDConfig->ftfxRegBase + FTFx_SSD_FSTAT_OFFSET,
              FTFx_SSD_FSTAT_CCIF);

    /* check CCIF bit */
    while(FALSE == (REG_BIT_TEST(PSSDConfig->ftfxRegBase +
          FTFx_SSD_FSTAT_OFFSET, FTFx_SSD_FSTAT_CCIF)))
    {
        /* wait till CCIF bit is set */
    }
}
```

4. In the startup initialization source file, reference sections using #pragma directives are shown below.

```
#pragma section = "CodeRelocate"
#pragma section = "CodeRelocateRam"
```

5. In the software initialization routine, declare and initialize pointers for various data sections. These pointers are initialized using values pulled from the linker file.

```
/* Get addresses for any code sections that need to be copied from ROM to RAM
* The IAR tools have a predefined keyword that can be used to mark individual
* functions for execution from RAM. Add "__ramfunc" before the return type in
* the function prototype for any routines you need to execute from RAM
* instead of ROM. ex: __ramfunc void foo(void);
*/

uint8* code_relocate_ram = __section_begin("CodeRelocateRam");
uint8* code_relocate = __section_begin("CodeRelocate");
uint8* code_relocate_end = __section_end("CodeRelocate");
```

6. In the startup initialization routine, copy the RAM functions from ROM to RAM.

```
/* Calculate the number of bytes to copy */
     uint32 n;
     n = code_relocate_end - code_relocate;


    /* Copy functions from ROM to RAM */

            while (n--)
                  *code_relocate_ram++ = *code_relocate++;
```

7. Call the function to launch the flash command from within the `FlashCommandSequence` function of the C90TFS flash software drivers. Call the function just after the CCOB registers are loaded and before the error flags are checked.

```
// CallRAM function to launch command
FlashLaunchCommand (PSSDConfig);
```

6.2 Freescale CodeWarrior for Microcontrollers v.10.2/10.3 with Freescale compiler

1. In a header file, define a custom section for relocating code. Use the pragma compiler directive `#pragma define_section` to define a section called "`.relocate_code`".

   ```
   #pragma define_section relocate_code ".relocate_code"  far_abs  RX
   ```

   NOTE

   In the Kinetis sample code, this is defined in the file CW.h.

2. The attribute `__declspec()` is used to specify a storage location for objects. So, prepending functions with `__declspec(relocate_code)` will place them into the `relocate_code` section that was just defined by the pragma statement.

   The user can define a keyword in the header file such as "`__relocate_code__`" to make an easier-to-read substitution of this attribute.

   ```
   #define __relocate_code__ __declspec(relocate_code)
   ```

3. In the linker file, insert `.relocate_code` section in the `.data_bss` section. The `.data_bss` section is saved space in the SRAM for initialized data, but the RAM function can be placed there as well. This section is linked to the SRAM, but the data is loaded to the flash via a startup routine.

   ```
   AT(__DATA_ROM) designator.

       .data_bss : AT(__DATA_ROM)
       {
            __DATA_RAM      = .;
           *(.data)
           *(.sdata)
      *(.relocate_code)
      *(.relocate_const)
      *(.relocate_data)
      *(.test)
           __DATA_END      = .;
           .               = ALIGN(0x10);
           __START_BSS     = .;
           *(.sbss)
           *(SCOMMON)
           *(.bss)
           *(COMMON)
           __END_BSS       = .;
           .               = ALIGN(0x10);
           __HEAP_START    = .;
   __heap_addr= .;
   __heap_size = (4 * 1024);
   .= . + __heap_size;
           __HEAP_END      = .;
           __SP_END        = .;
   .= . + (1 * 1024);
   __BOOT_STACK_ADDRESS = .;
       } > ram
   ```

4. In the software source files, prefix the function with the keyword "__relocate code" defined in step 2 or use `__declspec (relocate_code)`. CodeWarrior will place this function in the `relocate_code` section.

   The C90TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers utilize a `FlashCommandSequence` function for executing all flash commands. This entire function can be placed into SRAM. This is the simplest method to implement, but it is the least SRAM-efficient. To reduce SRAM usage, just a critical subset can be placed in SRAM.

The following function just performs the critical steps that need to run from SRAM when launching a flash command.

```
/* Flash launch command function prototype */
extern void FlashLaunchCommand(PFLASH_SSD_CONFIG PSSDConfig);


__relocate_code__
void FlashLaunchCommand (PFLASH_SSD_CONFIG PSSDConfig)
{
/* Flash launch command */

    /* clear CCIF bit */
    REG_WRITE(PSSDConfig->ftfxRegBase + FTFx_SSD_FSTAT_OFFSET,
              FTFx_SSD_FSTAT_CCIF);

    /* check CCIF bit */
    while(FALSE == (REG_BIT_TEST(PSSDConfig->ftfxRegBase +
          FTFx_SSD_FSTAT_OFFSET, FTFx_SSD_FSTAT_CCIF)))
    {
        /* wait till CCIF bit is set */
    }
}
```

5. In the software initialization source file, bring in symbols from the header file as externs. These symbols define the sections:

```
extern uint32 __DATA_ROM[];
extern uint32 __DATA_RAM[];
extern char __DATA_END[];
```

6. In the software initialization routine, declare and initialize pointers for various data sections. These pointers are initialized using values pulled from the linker file:

```
uint8 * data_ram, * data_rom, * data_rom_end;

/* Note data_rom_end is actually a RAM address in CodeWarrior */
data_ram = (uint8 *)__DATA_RAM;
data_rom = (uint8 *)__DATA_ROM;
data_rom_end  = (uint8 *)__DATA_END;
```

7. In a startup initialization routine, copy the contents from DATA_ROM to DATA_RAM.

NOTE

`data_rom_end` is actually a RAM address in CodeWarrior, so when calculating the number of bytes to copy, use `data_ram` instead of `data_rom` as the starting point.

```
/* Calculate the number of bytes to copy */
 uint32 n;
 n = data_rom_end - data_ram;

/* Copy initialized data from ROM to RAM */
 while (n--)
       *data_ram++ = *data_rom++;
```

8. Call the function to launch the flash command from within the `FlashCommandSequence` function of the C90TFS flash software drivers. Call the function just after the CCOB registers are loaded and before the error flags are checked.

```
// Call RAM function to launch command
FlashLaunchCommand (PSSDConfig);
```

6.3 CodeWarrior for Microcontroller 10.3 with GCC compiler

1. In a header file, define a custom section for relocating code. Use the `__attribute__` declaration specifier to specify where a function should be placed in memory. The user can define a macro for this attribute as given below.

```
#define __relocate_code__ __attribute__((section(".relocate_code"), long_call))
```

2. In the link file, insert `.relocate_code` section in the `.data section`:

```
.data : AT(___ROM_AT)
{
  . = ALIGN(4);
  _sdata = .;
  *(.data)
  *(.data*)
  *(.relocate_code)
  . = ALIGN(4);
  _edata = .;
} > m_data
```

3. For program execution to copy the section from ROM to RAM, a link copy table must be defined in the linker file. The following is an example of such a table:

```
_romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.user_data2);
.romp : AT(_romp_at)  {
  __S_romp = _romp_at;
  LONG(___ROM_AT);
  LONG(_sdata);
  LONG(___data_size);
  LONG(___m_data2_ROMStart);
  LONG(___m_data2_RAMStart);
  LONG(___m_data2_ROMSize);
  LONG(0);
  LONG(0);
  LONG(0);
} > m_data2
```

NOTE

The startup initialization routine will call the function `__copy_rom_sections_to_ram`, that copies the contents from ROM to RAM.

4. In the flash driver source files, declare and define the `FlashLaunchCommand` function with the keyword "__relocate_code__". For example:

```
/* Flash launch command function prototype */
void FlashLaunchCommand(PFLASH_SSD_CONFIG PSSDConfig) __relocate_code__;

void __relocate_code__ FlashLaunchCommand (PFLASH_SSD_CONFIG PSSDConfig)
{
/* Flash launch command */

    /* clear CCIF bit */
    REG_WRITE(PSSDConfig->ftfxRegBase + FTFx_SSD_FSTAT_OFFSET,
              FTFx_SSD_FSTAT_CCIF);

    /* check CCIF bit */
    while(FALSE == (REG_BIT_TEST(PSSDConfig->ftfxRegBase +
          FTFx_SSD_FSTAT_OFFSET, FTFx_SSD_FSTAT_CCIF)))
    {
        /* wait till CCIF bit is set */
    }
}
```

5. Call the function to launch the flash command from within the `FlashCommandSequence` function of the C90TFS flash software drivers. Call the function just after the CCOB registers are loaded and before the error flags are checked.

```
// Call RAM function to launch command
FlashLaunchCommand (PSSDConfig);
```

7 References

The following reference materials can be used for this application note.

- KINETIS512_SC: Bare-metal sample code projects for Kinetis 100MHz V1 microcontroller family, available on **freescale.com**
- C90TFS_FLASH_DRIVER: TFS Flash Driver Software for Kinetis and ColdFire+ Microcontrollers, available on **freescale.com**
- AN4329: Relocating Code and Data Using the CodeWarrior Linker Command File (LCF), available on **freescale.com**
- IAR Technical Note 11578: Execute in RAM after copying from flash/ROM (v5.20 and later) , available on **iar.com**
- IAR Technical Note 75500. Using "__ramfunc" on assembly source (EWARM 5.x & 6.x):, available on **iar.com**
- Porting Freescale ARM Compiler-based Projects to use ARM GCC (included in the CodeWarrior 10.3 build at <CodeWarrior root directory >\MCU\Help\PDF\Porting_ARM_GCC.pdf)

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4695
Rev. 0, 04/2013

freescale