Freescale Semiconductor
Application Note

Document Number: AN4329

Relocating Code and Data Using the CodeWarrior Linker Command File (LCF) for ColdFire Architecture

By: Carlos Musich

1 Introduction

This document provides guidance for relocating code and data within the Microcontroller memory map. It also explains how to create new memory segments and sections by editing the CodeWarrior Linker Command File (LCF) for ColdFire Architectures.

For detailed information on the ColdFire Architectures Compiler and Linker Command File (LCF) refer to the CodeWarrior Development Studio for Microcontrollers V10.x ColdFire Architectures Build Tools Reference Manual. You can find this document in *{CW10.x installation path}\MCU\Help\PDF\MCU_ColdFire_Compiler.pdf*.

2 Preliminary Background

The Linker Command File (LCF) along with other compiler directives, places pieces of code and data into RAM and ROM. You can do this by creating specific sections in the LCF and then matching them to the source code using pragma directives.

Contents

The base project in this document uses a stationary-based project for the ColdFire MCF52259EVB in CodeWarrior 10.x, with a modified LCF in the Flash target.

3 LCF Overview

Other than making a program file from your project's object files, the linker has several extended functions for manipulating program code in different ways. You can access these functions through commands in the linker command file (LCF).

Linker command files consist of three kinds of segments, which must be in this order:

- A memory segment, which begins with the MEMORY{} directive,
- An optional closure segment, which begins with the FORCE_ACTIVE{}, KEEP_SECTION{}, or REF_INCLUDE{} directives, and
- A sections segment, which begins with the SECTIONS{} directive.

3.1 Memory Segment

The LCF memory segment is used to divide the Microcontroller memory into segments. Listing 1 shows the LCF memory segment of the MCF52259 stationary project.

Listing 1. LCF Memory Segment of MCF52259 Stationary Project

```
MEMORY {
   vectorrom    (RX)   : ORIGIN = 0x00000000, LENGTH = 0x00000400
   cfmprotrom   (RX)   : ORIGIN = 0x00000400, LENGTH = 0x00000020
   code         (RX)   : ORIGIN = 0x00000500, LENGTH = 0x0007FB00
   vectorram    (RWX)  : ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram      (RWX)  : ORIGIN = 0x20000400, LENGTH = 0x00005C00
}
```

You can notice that the LCF defines five Microcontroller memory segments. Each default segment allocates the following information:

- `Vectorrom` — Exception handler vectors.
- `Cfmprotrom` — Flash backdoor key.
- `Code` — Application's instructions.
- `Vectorram` — During startup routine, exception handler vectors are copied from vectorrom to vectorram.
- `Userram` — This memory segment is reserved for the user´s application.

NOTE For more information, refer *LCF Structure > Memory Segment* and *Commands, Directives, and Keywords > MEMORY directive* topics in the *ELF Linker and Command Language chapter* of *Microcontrollers V10.x ColdFire Architectures Build Tools Reference Manual*.

3.2 Sections Segment

Next to the memory segment you will find the sections segment. One of the functions of the sections segments is to define any global symbols that you want to use in your output file.

Listing 2. Section Segment Labels

```
# Memory Mapped Registers IPSBAR= 0x40000000)
  ___IPSBAR          = 0x40000000;

# 64 Kbytes Internal SRAM
  ___RAMBAR         = 0x20000000;
  ___RAMBAR_SIZE    = 0x00010000;

# 512 KByte Internal Flash Memory
  ___FLASHBAR       = 0x00000000;
  ___FLASHBAR_SIZE  = 0x00080000;

  ___SP_AFTER_RESET = ___RAMBAR + ___RAMBAR_SIZE - 4;
```

In Listing 2 you can find six labels. Labels can refer to a physical address or a determined quantity of bytes. From Listing 2 you get the following information:

- ____IPSBAR — Internal Peripheral System Base Address Register indicates the base address where all the Microcontroller registers are located. When a register is read or written the address is calculated with this address as base plus the offset of the register.
- ____RAMBAR — RAM Base Address Register is set to address 0x20000000. This is the address where RAM begins. Every time you refer to ___RAMBAR you will be referencing address 0x20000000.
- ____RAMBAR_SIZE — RAM Base Address Register Size indicates the number of bytes contained in RAM. 0x00010000 is 64Kb.
- ____FLASHBAR — Flash Base Address Register is the address where Flash begins which is address 0x00000000.
- ____FLASHBAR_SIZE — Flash Base Address Register Size Indicates the Flash size. 0x00080000 is 512Kb.
- ____SP_AFTER_RESET — Stack Pointer After Reset will be set to address 0x20000FFFB which is the last address in RAM where an integer (4 bytes) can be allocated. Stack Pointer will decrease its value by four memory addresses each time data is pushed into the stack.

Sections segment defines the contents of the target-memory sections. In a default project, `userram`, `code` and `vectorram` segments are initialized with empty sections as shown in Listing 3.

NOTE For more information, refer *LCF Structure > Sections Segment and Commands, Directives, and Keywords > SECTIONS directive* topics in the *ELF Linker and Command Language chapter of Microcontrollers V10.x ColdFire Architectures Build Tools Reference Manual*.

Listing 3. Empty Sections in Default Project

```
  .userram    : {} > userram
  .code       : {} > code
  .vectorram  : {} > vectorram
```

A section can be named as you want. In the code above the sections have the same names as the initialized segments but they can be renamed as you wish.

The sections segment of a LCF created by a MCF52259 stationary project contains seven default sections.

- `.text` — Contains all the code and constants of the application. This section is located in segment code.
- `.vectors` — Is placed in segment vectorrom and contains the exceptions vector table.
- `.cfmprotect` — Contains the flash backdoor key. It is contained in cfmprotrom.
- `.data` — Contains all the data needed by the program while its execution. This section is located in segment userram.
- `.bss` — Contains non-initialized data and is located in segment userram.
- `.custom` — Includes all the labels that define stack and heap starting and ending addresses as well as their sizes. It does not contain code or data.
- `.romp` — When a project starts running, all the variables are loaded into RAM in order to be used. This data must be located in ROM and then it must be copied to RAM in runtime. This section contains a structure that is used in startup routine to copy data in section .data from ROM to RAM.

To understand the content of sections following are the three LCF sections that are used and modified in order to relocate code and data.

Listing 4. Section .text

```
.text :
{
   *(.text)
   . = ALIGN (0x4);
   *(.rodata)
   . = ALIGN (0x4);
   ___ROM_AT = .;
   ___DATA_ROM = .;
} >> code
```

The name of the first section is `.text`. The section can be named anyway the user wants. It is very important not to confuse the name of this section with the instruction `filename(.text)` which is used inside the section's content. With this instruction LCF tells the linker that the code of the specified file is

going to be allocated in this section. When an asterisk is used at the beginning instead of a file name it refers to all the source files of the project.

After all the code is placed into memory and the memory gets aligned, the read only data is placed next to the code with the instruction `*(.rodata)`.

The labels `___ROM_AT` and `___DATA_ROM` point to the first empty address after *(.text) and *(.rodata) are placed. Both labels point to the same address.

Finally, `>> code` means that all the content of this section is going to be placed in memory segment `code`.

> **NOTE** For more information, refer *Commands, Directives, and Keywords* > *ALIGN* directive topic in the *ELF Linker and Command Language chapter of Microcontrollers V10.x ColdFire Architectures Build Tools Reference Manual*.

Listing 5. Section .data

```
.data : AT(___ROM_AT)
        {
        ___DATA_RAM = .;
        . = ALIGN(0x4);
        *(.exception)
        . = ALIGN(0x4);
        __exception_table_start__ = .;
        EXCEPTION
        __exception_table_end__ = .;

        ___sinit__ = .;

        STATICINIT
        __START_DATA = .;

        *(.data)
        . = ALIGN (0x4);
        __END_DATA = .;

        __START_SDATA = .;
        *(.sdata)
        . = ALIGN (0x4);
        __END_SDATA = .;

        ___DATA_END = .;
        __SDA_BASE = .;
         = ALIGN (0x4);
        } >> userram
```

The section `.data` is placed in the segment `userram`. RAM sections are more complicated because content in RAM cannot be saved when turning power off, then you first need to save the code and data in flash and then copy it to RAM.

The first line of the Listing 5 contains the instruction `AT` with the label `___ROM_AT` as parameter. This means that the content of section `.data` is going to be resident in the flash address pointed by label

____ROM_AT. Then, the startup code performs a routine to copy this data from Flash to RAM using the information provided in section .romp. This routine can be found in the startcf.c file.

Listing 6. Section .romp

```
_romp_at = ___ROM_AT + SIZEOF(.data);
.romp : AT(_romp_at)
{
        __S_romp = _romp_at;
        WRITEW(___ROM_AT);
        WRITEW(ADDR(.data));
        WRITEW(SIZEOF(.data));
        WRITEW(0);
        WRITEW(0);
        WRITEW(0);
}
```

4 Relocating Code in ROM

To place data and code in a specific memory location there are two general steps that must be performed.

1. Use pragma compiler directives to tell the compiler which part of the code is going to be relocated.
2. Tell the linker where is going to be placed this code within the memory map using LCF definitions.

4.1 Relocating Function in ROM

To put code in a specific memory section it is needed first to create the section using the define_section pragma directive. In Listing 7 a new section called mySectionInROM is created. All the content in this section is going to be referenced in the LCF with the word .romsymbols.

After defining a new section you can place code in this section by using the __declspec() directive. In Listing 7, __declspec() directive is used to tell the compiler that function funcInROM() is going to be placed in section mySectionInROM.

Create a stationary project for MCF52259EVB and add the following code to your main.c file before the main() function.

Listing 7. Code to add in the main.c

```
#pragma define_section mySectionInROM ".romsymbols" far_absolute RX
__declspec(mySectionInROM) void funcInROM(int flag); //Fcn Prototype
__declspec(mySectionInROM) void funcInROM(int flag){
   if (flag > 0)
    {
        printf("Option 1 selected \n\r");
        printf("Executing funcInROM() \n\r");
        printf("This function is executed from section myROM \n\r");
     }
}
```

4.2 Placing Code in ROM

You have just edited a source file to tell the compiler which code will be relocated. Next, the LCF needs to be edited to tell the linker the memory addresses where these sections are going to be allocated.

First you need to define a new Microcontroller memory segment where new sections will be allocated. You can have just one memory segment for all the new sections or one segment for each section.

4.2.1 Create New ROM Segment

Below you can find the memory segment of a LCF. Notice that the segment code has been edited and its length has been reduced by `0x10000` from its original size. This memory space is taken to create the new segment. In Listing 8 the new segment is called `myrom`, it will be located next to segment code and its length is going to be `0x10000`. You can calculate the address where segment code ends by adding its length plus the origin address.

Edit your LCF as shown in Listing 8. Ensure you edit M52259_Internal_Flash.lcf.

Listing 8. Memory Segment of LCF

```
MEMORY {
   vectorrom    (RX)   : ORIGIN = 0x00000000, LENGTH = 0x00000400
   cfmprotrom   (RX)   : ORIGIN = 0x00000400, LENGTH = 0x00000020
   code         (RX)   : ORIGIN = 0x00000500, LENGTH = 0x0006FB00
   myrom        (RX)   : ORIGIN = 0x00070000, LENGTH = 0x00010000
   vectorram    (RWX)  : ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram      (RWX)  : ORIGIN = 0x20000400, LENGTH = 0x00005C00
}
```

4.2.2 Create New ROM Section

The next step is to add the content of the new section into the Microcontroller memory segment you have reserved. This is done in the sections segment of the LCF.

The code below creates a new section called `.rom_symbols`, then the label `__ROM_SYMBOLS` points to the address where the section begins. Then `*(.romsymbols)` instruction is used to tell the linker that all the code referenced with this word is going to be placed in section .rom_symbols.

Finally you close the section telling the linker that this content is going to be located in segment `myrom`.

Add the code in Listing 9 just after section .text and before section .data in your M52259_Internal_Flash.lcf.

Listing 9. Code to Add after Section .text

```
.rom_symbols :
{
        __ROM_SYMBOLS = . ; #start address of the new symbol area
       . = ALIGN (0x4);
       *(.romsymbols) #actual data matching pragma directives.
       . = ALIGN (0x4);
} > myrom
```

Inside your project folder you will find a subfolder called M52259EVB_Internal_Flash. In this folder you will find your .xMAP file. If you compile at this point you will find in the xMAP file that section `.rom_symbols` was created in address `0x70000` as shown in Listing 10.

Listing 10. .rom_symbols Created in Address 0x70000

```
# .rom_symbols
#>00070000     __ROM_SYMBOLS (linker command file)
```

As you can notice the section is empty. Note that only the label is pointing to the start address and funcInROM has not been linked. This is because the compiler is optimizing the function. To avoid this optimization the function must be called more than once in the program.

Lets create some code to call the function `funcInROM()` any number of times in runtime. Copy the code in Listing 11 inside your `main()` function.

Listing 11. Code for main() Function

```
char counter = 0;

printf("***Modifying LCF Example***");

/***Infinite loop**/
while(TRUE){

    printf("\n\n\n\r Select one option: \n\n\r");
    printf("1: Run function from Flash: \n\r");

    counter = uart_getchar(0);
    myTest(counter);

    printf("\n\n\r Press any key to continue... \n\n\r");
    uart_getchar(0);
}
```

Now add this line at the top of your main.c file: `#include "uart_support.h"`

Finally add the function in Listing 12 above `main()` function in your `main.c` file.

Listing 12. Function to add before main() Function

```
void myTest(int x);   // Function prototype
void myTest(int x){

    if(x == '1') funcInROM(x);
    else printf("\n\rInvalid option!");
}
```

NOTE If you compile you may get **"Error: Illegal 16-bit PC relative"**. To avoid this error please change your **Code Model** and **Data Model** options from **Near 16-bit** to **Far 32-bi**t. You can find these options in **Project >**

Properties > C/C++ Build > Settings > ColdFire Compiler > Processor.

Now that the project is built correctly you will find in the .xMAP file that the section has been relocated in the desired address as shown in Listing 13. Select **Project > Clean** option before compiling.

Listing 13. Section Relocated at Desired Address

```
# .rom_symbols
#>00070000           __ROM_SYMBOLS     (linker command file)
  00070000 0000003E .romsymbols funcInROM      (main.obj)
```

At this point you can test your project by connecting MCF52259EVB's UART0 port to your PC and opening a HyperTerminal at `19200 bauds`.

5 Relocating Code and Data in Internal RAM

Since it is not possible to write a variable in ROM, data must be relocated in RAM. Code can be also relocated in RAM. Another reason to relocate code in RAM is that it is twice as fast as in Flash.

In Section 4 you have seen how to put one single function in a new section using `__declspec()` directive. As it was only one function it was not a problem. In big projects it may be complicated to use this directive with each of the functions you want to relocate. Besides this, it is not possible to use `__declspec()` to relocate data.

5.1 Relocating Code and Data in Internal RAM

Another way to tell the compiler to relocate code and also data is to create a new section using define_section pragma directive and writing the code inside `#pragma section <Section Name>` begin and `#pragma section <Section Name> end` directives as shown in the Listing 14 below.

Listing 14. Using Pragma Directives to Define a Section

```
#pragma define_section mySectionInRAM  ".myCodeInRAM" far_absolute RX
#pragma section mySectionInRAM begin

struct {
unsigned char data0;
unsigned char data1;
unsigned char data2;
unsigned char data3;
unsigned char data4;
unsigned char data5;
unsigned char data6;
unsigned char data7;
} CTMData = { 0x82, 0x65, 0x77, 0x32, 0x84, 0x69, 0x83, 0x84 };

void funcInRAM(int flag); //Function Prototype
void funcInRAM(int flag){
    if (flag > 0)
    {
        printf("Option 2 selected \n\r");
        printf("Executing funcInRAM() \n\r");
```

```
        printf("This function is executed from section myRAM \n\r");
        printf("CTMData: data0 = %x, data1 = %x, data2 = %x, data3 = %x, data4 = %x, data5 =
%x, data6 = %x, data7 = %x \n\r",CTMData.data0, CTMData.data1, CTMData.data2, CTMData.data3,
CTMData.data4, CTMData.data5, CTMData.data6, CTMData.data7);
        printf("The Address where this data is located is: 0x%lx\r\n\n\n",__myRAMStart);
     }
}

#pragma section mySectionInRAM end
```

Notice that in this case it is not necessary to use the `__declspec()` directive.

Add the code in Listing 14 in your `main.c` file after `funcInROM()`.

5.2 Placing Code and Data in RAM

Placing code and data into RAM is more complicated. As the content in RAM cannot be saved when turning power off, you first need to save the code and data in flash and then make a copy to RAM in runtime.

Next are described the steps to relocate code and data in a new RAM segment.

5.2.1 Create New RAM Segment

As it was made for the new ROM segment, a piece of the `userram` memory segment is taken to create a new memory segment called `myram`.

Edit your LCF as shown in Listing 15.

Listing 15. Edit LCF

```
MEMORY {
     vectorrom     (RX)    : ORIGIN = 0x00000000, LENGTH = 0x00000400
     cfmprotrom    (RX)    : ORIGIN = 0x00000400, LENGTH = 0x00000020
     code          (RX)    : ORIGIN = 0x00000500, LENGTH = 0x0006FB00
     myrom         (RX)    : ORIGIN = 0x00070000, LENGTH = 0x00010000
     vectorram     (RWX)   : ORIGIN = 0x20000000, LENGTH = 0x00000400
     userram       (RWX)   : ORIGIN = 0x20000400, LENGTH = 0x00004C00
     myram         (RWX)   : ORIGIN = 0x20005000, LENGTH = 0x00001000
}
```

5.2.2 Create New RAM Section

The memory segment specifies the intended location in RAM. The sections segment specifies the resident location in ROM, via its AT (address) parameter.

The code below shows a new section called `.my_ram` which is going to be linked in segment myram but is going to be resident in the Flash memory address calculated by label `___CodeStart`. This label is intended to find the first address available in flash.

In Listing 4 the linker places in the segment code all the code and then the read only data. After this it sets a label called `__ROM_AT`. Section .data is allocated in the address pointed by this label. You can place the

new code after section .data, so you calculate the address where section .data ends by using the instruction ___ROM_AT + SIZEOF(.data).

Add the code in Listing 16 to the LCF. You can put this code just after section .data.

Listing 16. Add Code to LCF

```
___CodeStart   =  ___ROM_AT + SIZEOF(.data);

.my_ram : AT(___CodeStart)
{
        . = ALIGN (0x4);
        ___myRAMStart = .;
        *(.myCodeInRAM)
        ___myRAMEnd = .;
        . = ALIGN (0x4);
} > myram

___CodeSize = ___myRAMEnd - ___myRAMStart;
```

You need to know the size of the code to indicate the amount of bytes that are going to be copied from Flash to RAM in runtime. To do this three labels are defined in the code above.

- ___myRAMStart — Is set in the point just before the new code is placed
- ___myRAMEnd — Is set after the code indicating the address where it ends
- ___CodeSize — Subtraction of both the addresses give us the size of the code and this value is assigned to the label ___CodeSize.

In order to let the compiler know the value of these labels the following defines must be declared in the source files. Copy the code in Listing 17 in main.c file before all the functions.

Listing 17. Code to Copy in main.c

```
    extern unsigned long __CodeStart[];
    #define CodeStart   (unsigned long)__CodeStart

    extern unsigned long __CodeSize[];
    #define CodeSize    (unsigned long)__CodeSize

    extern unsigned long __myRAMStart[];
    #define StartAddr   (unsigned long)__myRAMStart
```

5.2.3 Final Adjustments

Section .romp provides to the startup routine the information needed to copy the content of section .data from Flash to RAM. Notice that this section is placed by default after section .data. This is indicated by the instruction AT(_romp_at).

Label _romp_at points to the address where section .data ends, but now you have placed the section .my_ram in that memory space. As you can notice labels ___CodeStart and _romp_at point to the same address. So you need to move section .romp after section .my_ram by adding the space occupied by this section. This is done with the instruction SIZEOF(.my_ram).

Modify your LCF as shown in Listing 18.

Listing 18. Modify the LCF

```
_romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.my_ram);
    .romp : AT(_romp_at)
    {
       __S_romp = _romp_at;
       WRITEW(___ROM_AT);
       WRITEW(ADDR(.data));
       WRITEW(SIZEOF(.data));
       WRITEW(0);
       WRITEW(0);
       WRITEW(0);
       }
```

Now you have relocated code and data in RAM. But the function will be optimized as it is not called more than once in the code. Edit your `main()` function and `myTest()` function as shown in Listing 19 and Listing 20 so the compiler will not optimize it.

Listing 19. Edit main() Function

```
char counter = 0;

printf("***Modifying LCF Example***");

/***Infinite loop**/
while(TRUE){

    printf("\n\n\n\r Select one option: \n\n\r");
    printf("1: Run function from Flash: \n\r");
    printf("2: Run function from Internal RAM: \n\r");

    counter = uart_getchar(0);
    myTest(counter);

    printf("\n\n\r Press any key to continue... \n\n\r");
    uart_getchar(0);
}
```

Listing 20. Edit myTest() Function

```
void myTest(int x){

    if(x == '1') funcInROM(x);
    else if(x == '2') funcInRAM(x);
    else printf("\n\rInvalid option!");
}
```

Now the project will compile and you will find in the .xMAP file the next information showing that the code and data were relocated in the intended location. Select **Project > Clean** option before compiling.

Listing 21. Code and Data Relocated at the Intended Location

```
# .my_ram
#>20005000          ___myRAMStart (linker command file)
```

```
  20005000 00000008 .myCodeInRAM CTMData(main.obj)
  20005008 000000B0 .myCodeInRAM funcInRAM(main.obj)
#>200050B8           ___myRAMEnd (linker command file)
#>000000B8          ___CodeSize (linker command file)
```

5.3 Copying Code and Data from Flash to RAM

At this point you have relocated successfully the code and data in RAM. But the Microcontroller will not find them as they are still not copied into RAM. You can use the routine in Listing 22 to copy the code and data from Flash to RAM in runtime.

Listing 22. Copy Code and Data from Flash to RAM in Runtime

```
uint8 *Source;       /* use this pointer to get the begining of the Code */
uint8 *Destiny;      /* use this pointer to get the RAM destination address */
uint32 MemorySize;   /* Gets the size of the code that is being copied */

void copyToMyRAM();//Function prototype
void copyToMyRAM(){

/* Initialize the pointers to start the copy from Flash to RAM */
Source = (unsigned char *)(CodeStart);
Destiny = (unsigned char *)(StartAddr);
MemorySize = (unsigned long)(CodeSize);

/* Copying the code from Flash to RAM */
while(MemorySize--)
{
*Destiny++ = *Source++;
}
}
```

Copy the code in Listing 22 in your `main.c` file before all the functions you already have. You must call `copyToMyRAM()` function before the while loop inside `main()` function.

Now you can run your code with code and data relocated in ROM and RAM by connecting MCF52259EVB's UART0 port to your PC and opening a HyperTerminal at `19200` bauds.

6 Relocating Code and Data in External MRAM

Many times the internal RAM in the Microcontroller you are using is not enough for the application. For this reason it is needed to use external memories as part of the solution. The process to relocate code and data in external memories is exactly the same as you did for internal RAM. The only difference is that the external device needs to be communicated by an interface controller.

This example is made for MCF52259EVB which has an onboard external MRAM. To communicate with this memory MCF52259 provides miniFlexBus module. The scope of this document is not to explain miniFlexBus configuration, but next you can find a routine that you can use to initialize miniFlexBus and set it to communicate with external MRAM.

NOTE For more information on miniFlexBus configurations, refer to AN3854 at `http://www.freescale.com/files/32bit/doc/app_note/AN3854.pdf`.

6.1 Set miniFlexbus

The function in Listing 23 can be called from `main()` function to configure the miniFlexBus in order to communicate with External MRAM.

Copy the code in Listing 23 in your `main.c` file and call `initFlexBus()` function from `main()` function before the while loop.

Listing 23. Copy Code to main.c

```
void initFlexBus();//Function Prototype
void initFlexBus(){

     //initialize MiniBus pins
     MCF_GPIO_PTEPAR = 0xFF;
     MCF_GPIO_PTFPAR = 0xFF;
     MCF_GPIO_PTGPAR = 0xFF;
     MCF_GPIO_PTHPAR = 0x5555;
     MCF_GPIO_PASPAR = MCF_GPIO_PASPAR_MB_ALE_MB_CS1;

     //MRAM
     MCF_FBCS0_CSAR = MCF_FBCS_CSAR_BA(0x80000000);

     MCF_FBCS0_CSCR =   MCF_FBCS_CSCR_WS(1)
                      | MCF_FBCS_CSCR_AA
                      | MCF_FBCS_CSCR_PS_8;

     MCF_FBCS0_CSMR =   MCF_FBCS_CSMR_BAM_512K
                      | MCF_FBCS_CSMR_V;

}
```

NOTE Ensure that the jumper J22 in EVB is set.

6.2 Relocating Code and Data in External MRAM

After setting the miniFlexBus the Microcontroller must be able to communicate with External MRAM. Code and data can be relocated in External MRAM. This can be done both ways; as it was made for ROM in Section 4.1 or for internal RAM in Section 5.1. In Listing 24 the first option is used.

Copy the code in Listing 24 in your `main.c` file after `funcInRAM()`.

Listing 24. Copy Code before funcInRAM() Function

```
#pragma define_section ExtMRAM ".myCodeInExtMRAM" far_absolute RX

__declspec(ExtMRAM) void funcInExtMRAM(int flag); //Function prototype
```

```
__declspec(ExtMRAM) void funcInExtMRAM(int flag){
    if (flag > 0)
    {
        printf("Option 3 selected \n\r");
        printf("Executing funcInExtMRAM() \n\r");
        printf("This function is executed from External MRAM \n\r");
        printf("The Address where this function is located is: 0x%lx\r\n\n\n",__MRAMStart);
     }
}
```

6.3 Placing Code in External MRAM

The rest of the process is exactly the same as it was for placing code in internal RAM.

6.3.1 Create New Segment

MiniFlexBus was configured to use as base address `0x80000000`. So you need to add this information to the memory segment. The length of the segment will be the size of the external memory.

Modify the memory segment of your linker command file as shown in Listing 25.

Listing 25. Modify the Memory Segment of Linker Command File

```
MEMORY {
   vectorrom    (RX)    :  ORIGIN = 0x00000000, LENGTH = 0x00000400
   cfmprotrom   (RX)    :  ORIGIN = 0x00000400, LENGTH = 0x00000020
   code         (RX)    :  ORIGIN = 0x00000500, LENGTH = 0x0006FB00
   myrom       (RX)    :  ORIGIN = 0x00070000, LENGTH = 0x00010000
   vectorram    (RWX)   :  ORIGIN = 0x20000000, LENGTH = 0x00000400
   userram     (RWX)    :  ORIGIN = 0x20000400, LENGTH = 0x00004C00
   myram       (RWX)    :  ORIGIN = 0x20005000, LENGTH = 0x00001000
   extmram     (RWX)    :  ORIGIN = 0x80000000, LENGTH = 0x00002000
}
```

6.3.2 Create New Section

The content of the external MRAM will also be resident in Flash and then it is going to be copied to the MRAM in runtime. The format of the new section will be the same as section `.my_ram` which was created to relocate the code in internal RAM.

Copy the new section shown in Listing 26 in your LCF below section .my_ram

Listing 26. Create New Section

```
___MRAMCodeStart    =  ___CodeStart + SIZEOF(.my_ram);

.my_mram : AT(___MRAMCodeStart)
{
        . = ALIGN (0x4);
        ___MRAMStart = .;
        *(.myCodeInExtMRAM)
        ___MRAMEnd = .;
        . = ALIGN (0x4);
} > extmram
```

```
___MRAMCodeSize = ___MRAMEnd - ___MRAMStart;
```

As it was made for section .my_ram three labels are defined to make the copy from Flash to MRAM. These labels provide the addresses where the code begins, where it ends and its size. The Flash address where this code will be resident is after section .my_ram and it is calculated by label ___MRAMCodeStart.

In order to let the compiler know the value of these labels the following defines must be declared in main.c file before all the functions.

Listing 27. Declaring Defines in main.c File

```
extern unsigned long __MRAMCodeStart[];
#define MRAMCodeStart  (unsigned long)__MRAMCodeStart

extern unsigned long __MRAMCodeSize[];
#define MRAMCodeSize  (unsigned long)__MRAMCodeSize

extern unsigned long __MRAMStart[];
#define MRAMStartAddr  (unsigned long)__MRAMStart
```

6.3.3 Final Adjustments

As the section .my_mram was placed in the memory space where section .romp was allocated you need to change again .romp location in order not to overwrite section .my_mram. You can see in Listing 28 how to modify the calculation of the address where section .romp is placed.

Listing 28. Modify the Address Calculation where Section .romp is Placed

```
_romp_at = ___ROM_AT + SIZEOF(.data) + SIZEOF(.my_ram) + SIZEOF(.my_mram);
         .romp : AT(_romp_at)
         {
         __S_romp = _romp_at;
         WRITEW(___ROM_AT);
         WRITEW(ADDR(.data));
        WRITEW(SIZEOF(.data));
        WRITEW(0);
         WRITEW(0);
         WRITEW(0);
        }
```

Now you have relocated a function in external MRAM. But it will be optimized as it is not called more than once in the code. Edit your main() function and myTest() function as shown in Listing 29 and Listing 30 so the compiler will not optimize it.

Listing 29. Edit main() Function

```
char counter = 0;

printf("***Modifying LCF Example***");

/***Infinite loop**/
```

```
while(TRUE){

    printf("\n\n\n\r Select one option: \n\n\r");
    printf("1: Run function from Flash: \n\r");
    printf("2: Run function from Internal RAM: \n\r");
    printf("3: Run function from External MRAM: \n\n\n\r");

    counter = uart_getchar(0);
    myTest(counter);

    printf("\n\n\r Press any key to continue... \n\n\r");
    uart_getchar(0);
}
```

Listing 30. Edit myTest() Function

```
void myTest(int x){

   if(x == '1') funcInROM(x);
   else if(x == '2') funcInRAM(x);
   else if(x == '3') funcInExtMRAM(x);
   else printf("\n\rInvalid option!");
}
```

Now the project will compile and you will find in the .xMAP file the next information showing that the code was relocated in the intended location. Select **Project > Clean** option before compiling.

Listing 31. Code Relocated to Intended Location

```
# .my_mram
#>80000000          ___MRAMStart (linker command file)
  80000000 00000058 .myCodeInExtMRAM funcInExtMRAM(main.obj)
#>80000058          ___MRAMEnd (linker command file)
#>00000058          ___MRAMCodeSize (linker command file)
```

6.4 Copying Code and Data from Flash to RAM

At this point you have relocated successfully the code in External MRAM. But the Microcontroller will not find it as it is still not copied into RAM. You can use the routine in Listing 32 to copy the code from Flash to External MRAM in runtime.

Listing 32. Routine to Copy from Flash to External MRAM

```
    uint8 *MRAMSource;
    uint8 *MRAMDestiny;
    uint32 MRAMSize;
    void copyToExtMRAM()
    void copyToExtMRAM(){

    /* Initialize the pointers to start the copy from Flash to RAM */
            MRAMSource = (unsigned char *)(MRAMCodeStart);
            MRAMDestiny = (unsigned char *)(MRAMStartAddr);
            MRAMSize = (unsigned long)(MRAMCodeSize);

            /* Copying the code from Flash to External RAM */
            while(MRAMSize--)
```

```
            {
                 *MRAMDestiny++ = *MRAMSource++;
             }

    }
```

Copy the code in Listing 32 in your main.c file before all the functions you already have. You must call `copyToExtRAM()` before the while loop inside `main()` function but after `initFlexBus()`.

Now you can run your code with functions relocated in ROM and RAM by connecting the MCF52259EVB'2 UART0 port to your PC and opening a HyperTerminal at `19200`.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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)
support@freescale.com

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4329

12 April 2013