Freescale Semiconductor, Inc.
Application Note

Document Number: AN5143
Rev. 0, 06/2015

Relocate Subroutines to PRAM for MC56F827xx DSC

Using the CodeWarrior Linker Command File

1. Introduction

There are two speed modes for MC56F827xx Digital Signal Controller (DSC). In normal mode, codes runs at a maximum frequency of 50 MHz in both Flash and RAM. In fast mode, codes runs at 50 MHz in Flash, but 100 MHz in RAM. This application note describes the details on how to relocate codes into RAM and speed up some time critical subroutines.

The relocation is realized by linker command file (LCF) and usage of pragma directives in a *.c source file.

The two scenarios of implementation are introduced.

- Relocating non-precompiled source code into RAM -- Assigns RAM and Flash space for code and data dynamically through well-designed linker command file.
- Relocating a compiled object, a library for example into RAM -- Dynamically assigns RAM space for code and data, but assigning a fixed Flash space is a must for code storage.

Contents

2. Dual-speed mode and memory map

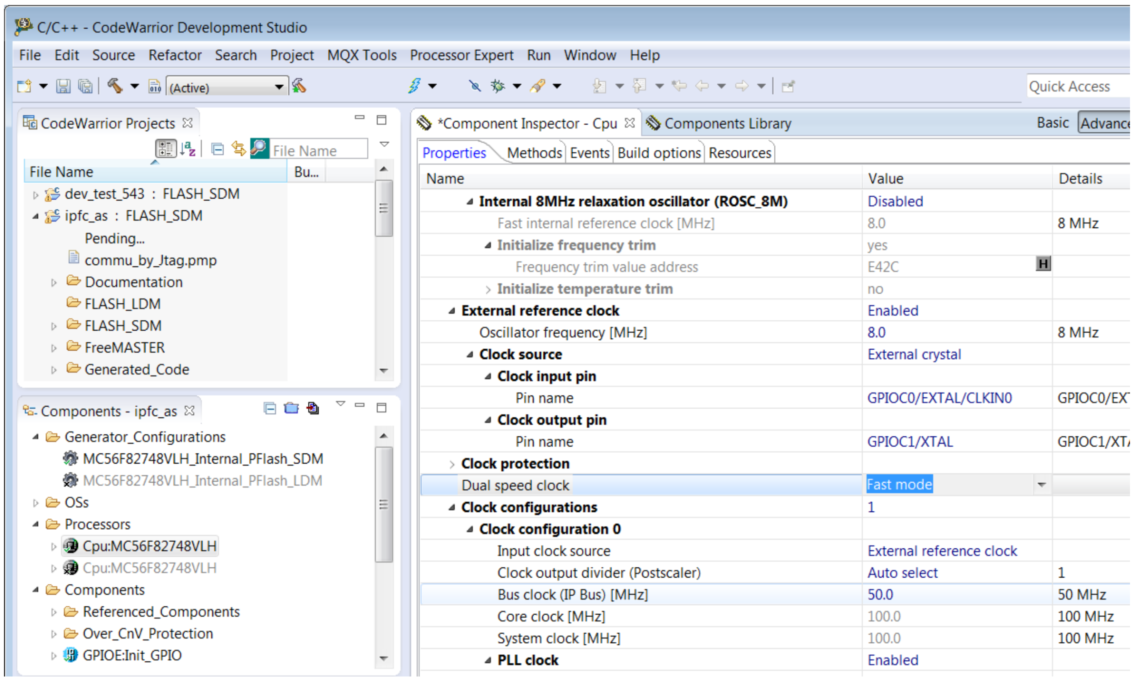
Table 1. MC56F827xx clock modes

Mode	Core	System	IP Bus
Normal	50	50	50
Fast	100	100	50

Table 1 lists operation frequency of dual-speed modes on MC56F827xx. The whole clock system is composed of three parts:

- Core clock -- Provides operation clock for 56800EX core.
- System clock -- Provides clock rate management to all on chip peripherals which in turn manages the IP bus frequency. System clock and system integration module (SIM) clock are synonymous.
- IP bus clock -- Provides read/write clocks for all peripherals.

Processor Expert (PEx) is a development assistant that helps you in rapidly configuring the Freescale microcontrollers. PEx provides a speed-mode configuration switch located in the CPU bean. In Figure 1, the configuration switches to fast mode, the estimated core, the system clock, and the IP bus frequencies update instantly. For more information, see the *MC56F827xx Reference Manual* (document MC56F827XXRM).



Name	Value	Details
Internal 8MHz relaxation oscillator (ROSC_8M)	Disabled	
Fast internal reference clock [MHz]	8.0	8 MHz
Initialize frequency trim	yes	
Frequency trim value address	E42C	
Initialize temperature trim	no	
External reference clock	Enabled	
Oscillator frequency [MHz]	8.0	8 MHz
Clock source	External crystal	
Clock input pin		
Pin name	GPIOC0/EXTAL/CLKIN0	GPIOC0/EX
Clock output pin		
Pin name	GPIOC1/XTAL	GPIOC1/XT
Clock protection		
Dual speed clock	Fast mode	
Clock configurations	1	
Clock configuration 0		
Input clock source	External reference clock	
Clock output divider (Postscaler)	Auto select	1
Bus clock (IP Bus) [MHz]	50.0	50 MHz
Core clock [MHz]	100.0	100 MHz
System clock [MHz]	100.0	100 MHz
PLL clock	Enabled	

Figure 1. Switching MC56F82748 into fast-mode in CodeWarrior for Microcontrollers v10.6

When 56F827xx is running under fast mode, the code runs at 50 MHz in Flash, but 100 MHz in RAM. Time critical subroutines can be relocated into RAM for better performance.

The 56800EX core of Freescale has a dual Harvard architecture. The Core bus are composed of program data bus (PDB) and primary/secondary data buses (XDB). It is possible to access the entire Flash and

RAM through PDB or XDB. These two addressing buses provide different functionality on Freescale DSC, which are listed below.

Program data bus, PDB:

- Core instruction fetch

Primary/secondary data buses, XDB:

- Data read/write
- Peripheral read/write. For example, DMA, ADC and others.

Relocating code into RAM requires individual sub-space in PDB mapped RAM. In Figure 2, the sub-space start from P:0xF000. The prefix “P” and “X” refer to the program data bus and the data bus respectively. Since the PDB RAM and the XDB RAM access the same on-chip RAM of MC56F82748, you must preserve the assigned sub-space in PDB RAM before the XDB RAM placement in link time. For more information, see the *MC56F827xx Reference Manual* (document MC56F827XXRM). *C56F827xx reference manual*.

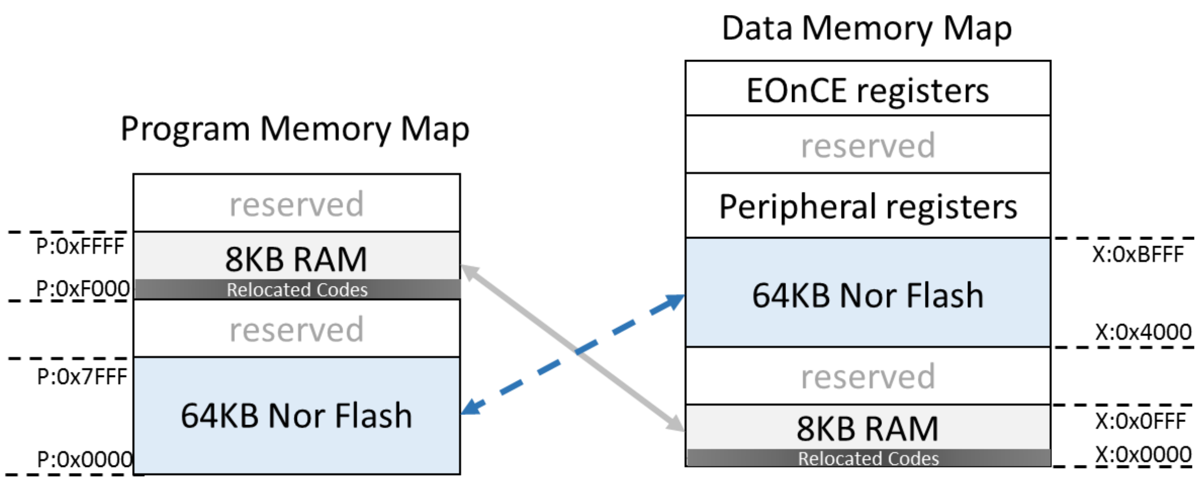
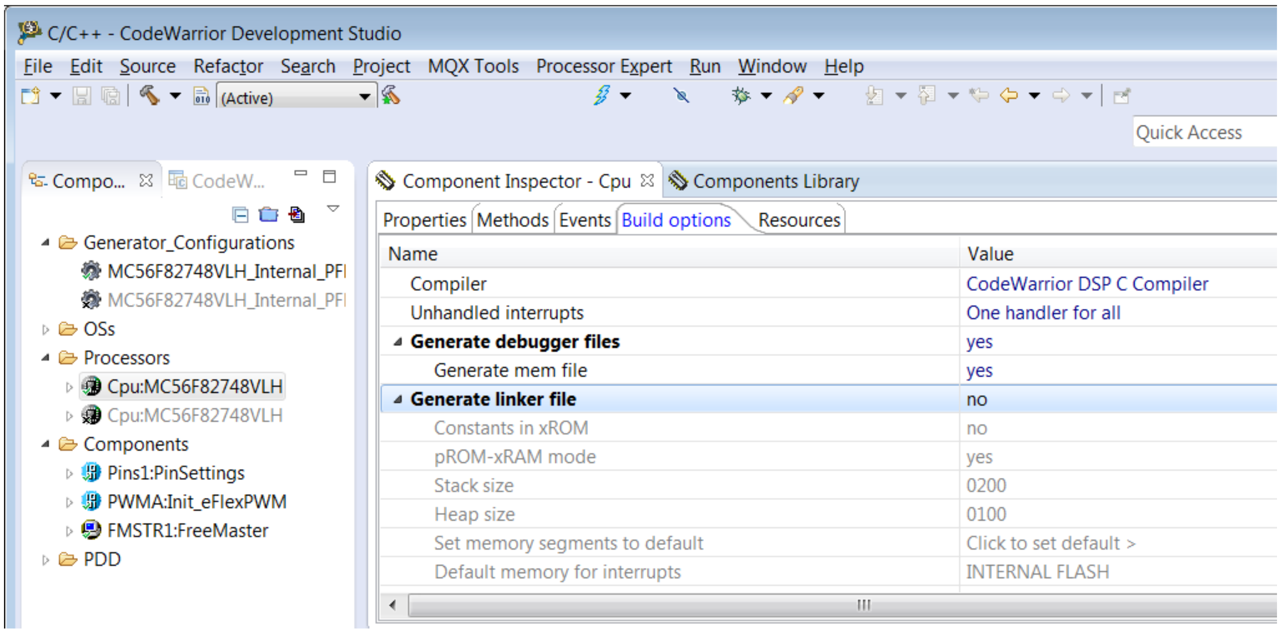


Figure 2. Memory Map of MC56F82748 DSC

3. Relocating code into internal RAM

This section provides guidance for relocating code into internal RAM with CodeWarrior for Microcontrollers v10.6 and PE. Because RAM is volatile, the relocated code must be stored in flash and copied into RAM during microcontroller start-up. The relocation of codes into RAM is realized by LCF (linker command file) and pragma directives source file. To preserve the customized linker command file you must disable the auto generate LCF by Processor Expert option, as shown in Figure 3.



Name	Value
Compiler	CodeWarrior DSP C Compiler
Unhandled interrupts	One handler for all
Generate debugger files	yes
Generate mem file	yes
Generate linker file	no
Constants in xROM	no
pROM-xRAM mode	yes
Stack size	0200
Heap size	0100
Set memory segments to default	Click to set default >
Default memory for interrupts	INTERNAL FLASH

Figure 3. Disable the Generate linker file option

The structure of a linker command file for DSC includes three segments.

- **Memory Segment**

  The memory segment in LCF divide available memory into sub segments as listed in Example 2, a sub segment description list name, attribute, start address and length sequentially.

- **Closure Segment**

  This is an optional block, which provides a way to assign symbols immune from dead stripping.

- **Section Segment**

  The section segment in LCF defines content of memory segments, is capable to store start/end address, and segment sizes into Global variables.

  For more information on these three segments, see Section 7.1 “Structure of Linker Command File” of “CodeWarrior Development Studio for Microcontrollers V10.x Digital Signal Controller Build Tools Reference Manual Rev. 10.6”.

3.1. Code relocating without libraries

This section describes how to relocate non-compiled source code into ram. Take a CodeWarrior project consist of a periodical interrupt service routine `PIT_100k_OnInterrupt()` for illustration. The following description shows how to use `#pragma` directive to define a unique code section, and relocate it to RAM by modify the default LCF generated by PEx.

3.1.1. Define code sections with pragma directive

To relocate code into RAM, it is necessary to tag desire codes for link time relocation. Example 1 shows how to create a new section `ramFunc`, and put the function `PIT_100k_OnInterrupt()` into it. All the codes in `ramFunc` section are referenced in LCF with the word `ramFunc.text`, and they have readable and executable attribution. Codes surround with `#pragma section ramArea` begin/end become members of section `ramFunc`.

Example 1. Use pragma directives to define a new code section

```
      #pragma define_section ramFunc "ramFunc.text" RX

      #pragma section ramArea begin
      #pragma interrupt alignsp
      void PIT_100k_OnInterrupt(void)
      {
        /*--------------------------------*/
        /* codes to be relocated into RAM */
        /*--------------------------------*/
      }
     #pragma section ramArea end
```

3.1.2. Relocate objects into RAM

The objects that are relocated are stored in Flash and copied into RAM during microcontroller initialization. Example 2 lists memory segments, the `.p_ramSpace` memory segment is the only difference from Processor Expert generated LCF.

In the LCF for DSC, memory segments with RWX attributes are placed into PDB and RW attributes area placed into XDB. The purpose of each memory area is listed below.

- Program memory
  - `.p_ramSpace` — PDB mapped RAM
  - `.p_Code` — User applications
  - `.p_reserverd_FCF` — Flash backdoor comparison key
  - `.p_Interrupts` — Interrupt vector tables
- Data memory
  - `.x_internal_ROM` — XDB mapped flash
  - `.x_Data` — XDB mapped RAM

Example 2. Memory segment for non-compiled code relocation

```
MEMORY {
        # I/O registers area for on-chip peripherals
        .x_Peripherals     (RW)   : ORIGIN=0xC000     , LENGTH=0

        # List of all sections specified in the "Build options" tab
        .p_Interrupts      (RWX)  : ORIGIN=0x00000000, LENGTH=0x000000DE
        .p_Code            (RWX)  : ORIGIN=0x00000208, LENGTH=0x00007DF8
        .x_Data            (RW)   : ORIGIN=0x00000000, LENGTH=0x00001000
        .p_reserved_FCF    (RWX)  : ORIGIN=0x00000200, LENGTH=0x00000008
        .x_internal_ROM    (RW)   : ORIGIN=0x000040DE, LENGTH=0x00000122
        .p_ramSpace        (RWX)  : ORIGIN=0x0000f000, LENGTH=0x00001000

        # p_flash_ROM_data mirrors x_Data, mapping to origin and length
        # the "X" flag in "RX" tells the debugger flash p-memory.
        # the p-memory flash is directed to the address determined by AT
        # in the data_in_p_flash_ROM section definition
        .p_flash_ROM_data  (RX)   : ORIGIN=0x00000000, LENGTH=0x00001000
}
```

In section segment, a section starts with a dot sign following unique section name and a semicolon. The braces surrounded block defines section content and variables, and end with a greater sign assigns mapped memory area.

- The content of `.p_Code` memory area define by section `.ApplicationCode`, which listed in Example 3, the `rtlib.text` had been commenting out with a leading pound character.
- The `rtlib.text` are routines for save/restore all registers, it comments out from section `.ApplicationCode` and will insert into section `.ramFunctions`, Runtime libraries take place when “saveall” parameter utilized for interrupt subroutines.
- The “ALIGN” syntax with parameter “2” moves location counter to next two words(4 bytes) alignment boundary, it’s also capable to move location counter to next n word boundary, where n is a power of 2.

Example 3. .p_Code content in LCF

```
.ApplicationCode :
        {
              F_Pcode_start_addr = .;
              # .text sections
              * (.text)
              #* (rtlib.text)
              * (startup.text)
              * (fp_engine.text)
              * (ll_engine.text)
              * (user.text)
              * (.data.pmem)
              F_Pcode_end_addr = .;

              # save address where for the data start in pROM
              . = ALIGN(2);
             __pROM_code_start = .;

        } > .p_Code
```

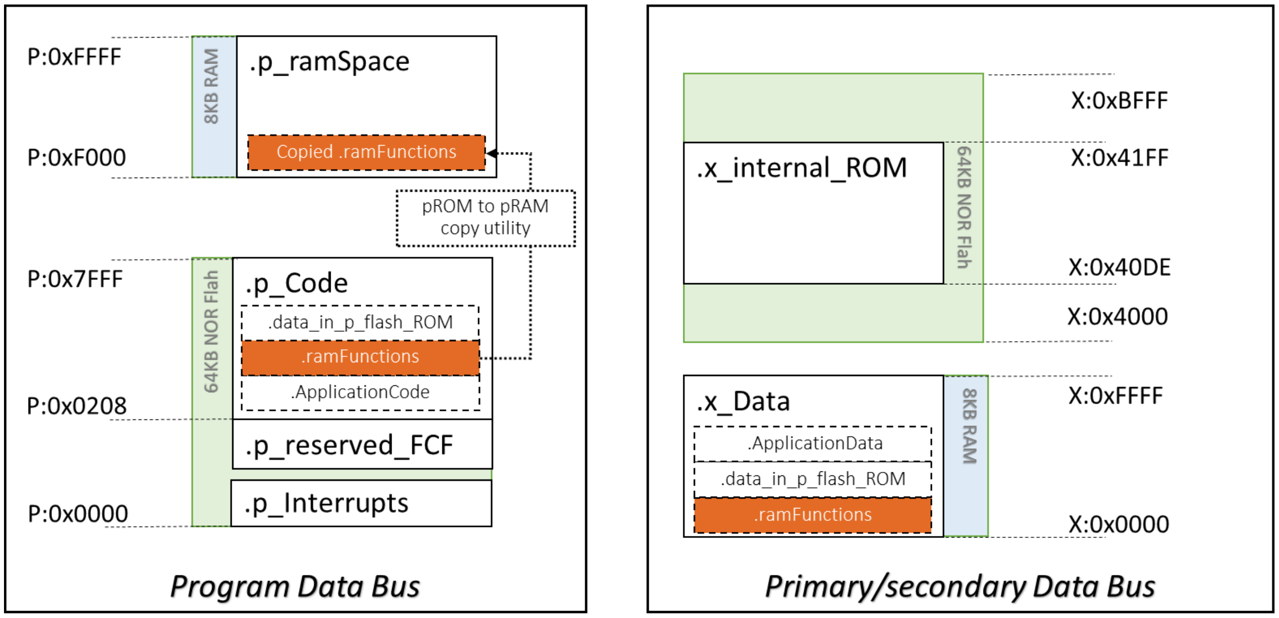


Figure 4. Memory planning for non-compiled code relocating

The content of memory segment .p_ramSpace defined by section segment .ramFunctions which listed in Example 4. An "AT" syntax defines optional load/resident address of a section segment. The codes inside .ramFunctions is going to resident in Flash memory with address __pROM_code_start and run on memory segment .p_ramSpace, this section will copied into memory segment .p_ramSpace during DSC initialization. The relationship between these section segments is illustrated in Example 4. And the codes relocated into RAM lists below.

- Objects resident in flash but execution on RAM
  - Runtime Libraries
  - Codes in section "ramArea.text", which defined by specialized pragma in Example 1

Example 4. .p_ramSpace area content of LCF

```
.ramFunctions : AT(__pROM_code_start)
        {
              F__pRAM_code_start = .;
              * (rtlib.text)
              * (ramArea.text)

              # save address where for the data start in pROM
              . = ALIGN(2);
              F__pRAM_code_end = .;
              __ramfunctions_size = F__pRAM_code_end - F__pRAM_code_start;
              __pROM_data_start = __pROM_code_start + __ramfunctions_size;

        } > .p_ramSpace
```

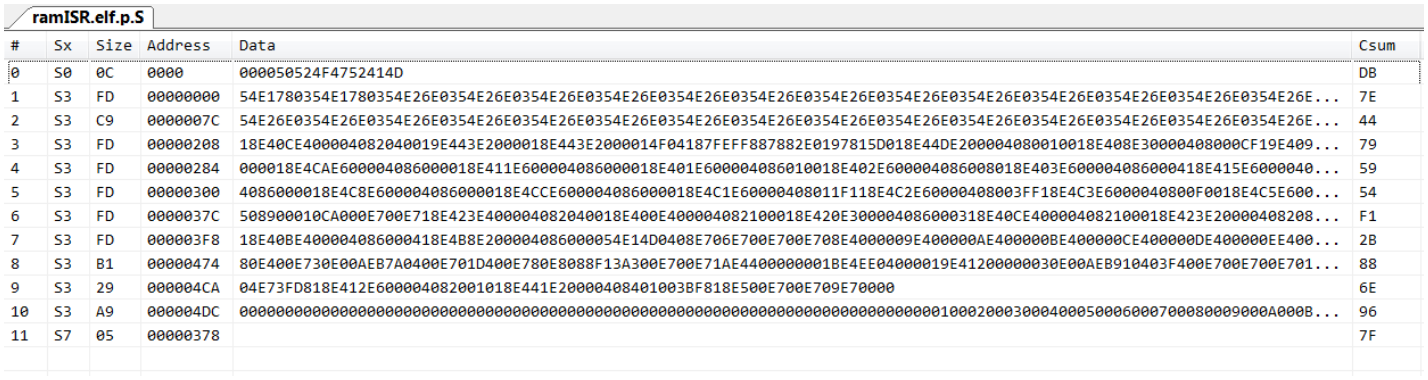
The section segment .p_flash_ROM_data in Example 5 provides predefined variable container, it placed at start of RAM by default. Since the section segment .ramFunctions had been placed at start of RAM as Example 4 shows. Section segments of .data_in_p_flash_ROM and .ApplicationData must stand aside in RAM, this achieved by moving location counter as shading text in Example 5.

Example 5. .p_flash_ROM_data area block of LCF

```
.data_in_p_flash_ROM : AT(__pROM_data_start)
        {
              # the data sections flashed to pROM
              # save data start so we can copy data later to xRAM
              . = . + __ramfunctions_size; #prevent overlap with  .ramFunctions
              __xRAM_data_start = .;

              # .data sections
              * (.const.data.char)     # used if "Emit Separate Char Data Section" enabled
              * (.const.data)

              * (fp_state.data)
              * (rtlib.data)
              * (.data.char)        # used if "Emit Separate Char Data Section" enabled
              * (.data)

              # save data end and calculate data block size
              . = ALIGN(2);
              __xRAM_data_end = .;
              __data_size = __xRAM_data_end - __xRAM_data_start;
             F_p_flash_ROM_data_loadAddr = __pROM_data_start + __ramfunctions_size;

        } > .p_flash_ROM_data
```

NOTE

The shifted location counter brings a content undefined sub-area in memory segment `.p_flash_ROM_data` shows in Figure 5, the undefined subarea will fill with 0x0000 automatically. It can be eliminated by assign two fixed RAM areas for code and data respectively. Generally, it's suggest keeping RAM space allocate dynamically while developing. The length of undefined area equals length of 1st section segment in RAM, place smaller one of `.ramfunctions` and `.data_in_p_flash_ROM` firstly makes it minimized.


ramISR.elf.p.S

#	Sx	Size	Address	Data	Csum
0	S0	0C	0000	000050524F4752414D	DB
1	S3	FD	00000000	54E1780354E1780354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E...	7E
2	S3	C9	0000007C	54E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E...	44
3	S3	FD	00000208	18E40CE400004082040019E443E2000018E443E2000014F04187FEFF887882E0197815D018E44DE200004080010018E408E30000408000CF19E409...	79
4	S3	FD	00000284	000018E4CAE600004086000018E411E600004086000018E401E600004086010018E402E600004086008018E403E600004086000418E415E6000040...	59
5	S3	FD	00000300	4086000018E4C8E600004086000018E4CCE600004086000018E4C1E60000408011F118E4C2E60000408003FF18E4C3E6000040800F0018E4C5E600...	54
6	S3	FD	0000037C	508900010CA000E700E718E423E400004082040018E400E400004082100018E420E300004086000318E40CE400004082100018E423E20000408208...	F1
7	S3	FD	000003F8	18E40BE400004086000418E4B8E200004086000054E14D0408E706E700E700E708E4000009E400000AE400000BE400000CE400000DE400000EE400...	2B
8	S3	B1	00000474	80E400E730E00AEB7A0400E701D400E780E8088F13A300E700E71AE4400000001BE4EE04000019E41200000030E00AEB910403F400E700E700E701...	88
9	S3	29	000004CA	04E73FD818E412E600004082001018E441E20000408401003BF818E500E700E709E70000	6E
10	S3	A9	000004DC	000000000000000000000000000000000000000000000000000000000000000000000000000000100020003000400050006000700080009000A000B...	96
11	S7	05	00000378		7F

Figure 5. Record File of source code relocated MC56F82748

In Example 6, a non-zero content of `F_pROM_to_pRAM` enables `pROM-to-pRAM` copy utility, and three variables `F_Livt_size`, `F_Livt_ROM_addr` and `F_Livt_RAM_addr` pass code sizes, resident address and runtime address of section `.ramFunctions` for `pROM to pRAM` utility. Memory copy utilities will be called while DSC start-up initialization and the user program activated right after start-up initialization finished.

Example 6. Parameters assignment for copy utility in section of .ApplicationData

```
        #parameters for .p_flash_ROM_data copy
              F_pROM_to_xRAM    = 0x0001; #enable pROM-to-xRAM copy utility
              F_Ldata_size      = __data_size;
              F_Ldata_RAM_addr  = __xRAM_data_start;
              F_Ldata_ROM_addr  = F_p_flash_ROM_data_loadAddr;

              #parameters for ramfunction copy
              F_pROM_to_pRAM    = 0x0001; #enable pROM-to-pRAM copy utility
              F_Livt_size      = __ramfunctions_size;
              F_Livt_RAM_addr  = F__pRAM_code_start;
              F_Livt_ROM_addr  = __pROM_code_start;
              F_start_bss     = _START_BSS;
              F_end_bss       = _END_BSS;

              __DATA_END=.;
         } > .x_Data
```

3.2. Code relocating with libraries

Previous section introduced a universal LCF framework for codes relocate into ram on MC56F82748. However, pre-compiled libraries hide source code and makes section pragma useless. The "OBJECT" keyword provides a way select compiled code (object) in LCF.

The modified memory map planning is illustrated in Figure 6. A size fixed flash area (`.p_FuncCode`) assigned for libraries storage, which is divided from the last 0x300 words (0x600 bytes) of 64KB flash. This area limits the size of codes/libraries execute on RAM. The flash size of 56F82748 is large enough. Therefore, the recommended setting size of `.p_ramFuncCode` is 8kb but shrinks when flash space is not enough.

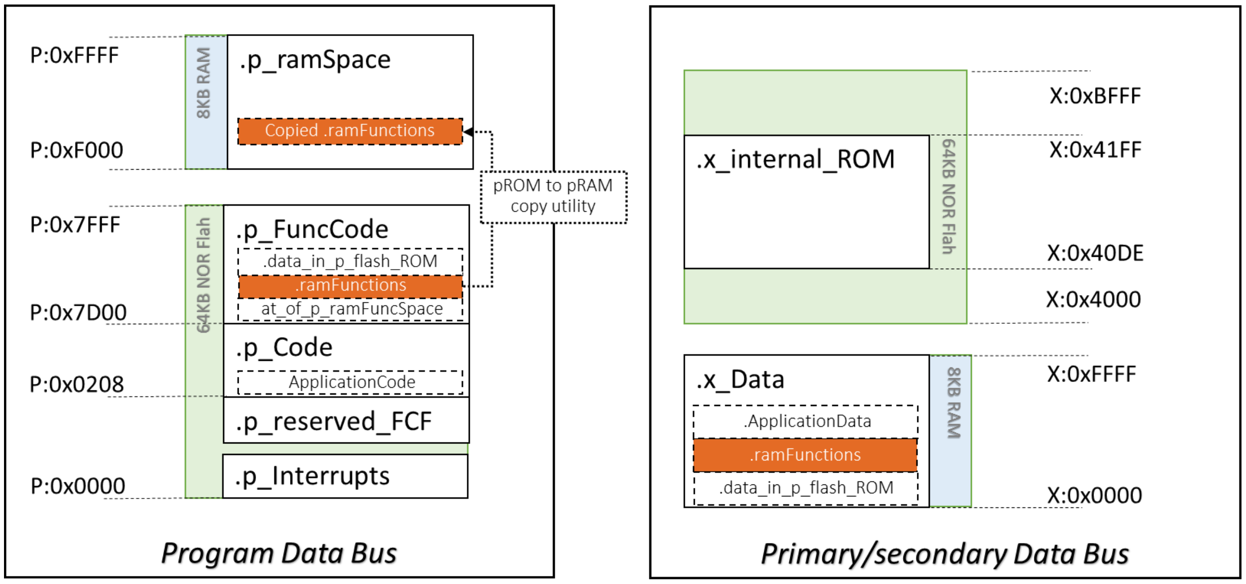


Figure 6. Memory planning for libraries relocating

In Example 7, the shaded text indicates modified text from processor expert generated LCF.

- The `.p_Code` memory segment length shrunken to 0x00007AF8 (62960 Bytes) from original 0x00007DF8.
- An extra memory segment `.p_ramFuncCode` added for codes/libraries storage (and execute on RAM).

Example 7. Memory segment for libraries/codes relocation on 56F82748

```
MEMORY {
        # I/O registers area for on-chip peripherals
        .x_Peripherals     (RW)  : ORIGIN = 0xC000, LENGTH = 0

        # List of all sections specified in the "Build options" tab
        .p_Interrupts      (RWX): ORIGIN = 0x00000000, LENGTH = 0x000000DE
        .p_Code            (RWX): ORIGIN = 0x00000208, LENGTH = 0x00007AF8 #PDB mapped flash (pFlash)
        .x_Data            (RW)  : ORIGIN = 0x00000000, LENGTH = 0x00001000
        .p_reserved_FCF    (RWX): ORIGIN = 0x00000200, LENGTH = 0x00000008
        .x_internal_ROM    (RW)  : ORIGIN = 0x000040DE, LENGTH = 0x00000122
        .p_ramFuncCode     (RWX): ORIGIN = 0x00007D00, LENGTH = 0x00000300 #pFlash subarea for ram-code
storage
        .p_ramFuncSpace    (RWX): ORIGIN = 0x0000F000, LENGTH = 0x00001000 #PDB mapped 8Kb RAM
        .p_flash_ROM_data  (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00001000
}
```

Section segment `.at_of_p_ramFuncSpace` in Example 8 writes end-address into variable `__pROM_data_start` for content insertion by other sections.

Example 8. Flash area for libraries resident

```
.at_of_p_ramFuncSpace :
        {
          WRITEH(0X2); # dummy insertion, prevent warning
          __pROM_data_start = .;
} > .p_ramFuncCode
```

Section segment .`data_in_p_flash_ROM` in Example 9.

- Resident at PDB address `__pROM_data_start` through keyword “AT”
- Variable `__ramFunc_code_start` assigns start of section segment `.ramFunctions`

Example 9. Selected content of .datain_p_flash_ROM Area

```
.data_in_p_flash_ROM : AT(__pROM_data_start)
        {
              # the data sections flashed to pROM
              # save data start so we can copy data later to xRAM

              __xRAM_data_start = .;

              # .data sections
              * (.const.data.char)     # used if "Emit Separate Char Data Section" enabled
              * (.const.data)

            ……

            ……

              . = ALIGN(2);
              __xRAM_data_end = .;
              __data_size = __xRAM_data_end - __xRAM_data_start;
              __ramFunc_code_start = __pROM_data_start + __data_size;

    } > .p_flash_ROM_data
```

The section segment .ramFunctions in Example 10 defines content of `.p_ramFuncSpace`.

- Keyword “AT” assign resident address for `.p_ramFuncSpace area`
- Shifted location counter “`.`” Prevents overlap with section segment `.data_in_p_flash_ROM`
- Keyword “`OBJECT`” selects functions, parameters can be find in Map file (Project_Name.elf.xMAP)
- Runtime routines `INTERRUPT_SAVEALL` and `INTERRUPT_RESTOREALL` relocate to `.p_ramFuncSpace`

Example 10. PDB Mapped RAM content definition

```
.ramFunctions : AT(__ramFunc_code_start)
        {
          . = . + __data_size;

          F_p_ramfunctions_start = .;
          #---- select function "PIT_100k_OnInterrupt()"               ----#
          OBJECT(FPIT_100k_OnInterrupt, Events_c.obj)
          #---- select library  "PCLIB_ControllerPIANDLPFILTERFAsm()"  ----#
          OBJECT(FPCLIB_ControllerPIANDLPFILTERFAsm, 56800Ex_PCLIB.lib)
          #---- routine of INTERRUPT_SAVEALL, INTERRUPT_RESTOREALL      ----#
          * (rtlib.text)
          . = ALIGN(2);
          F_p_ramfunctions_end = .;
          __ramFunctions_size = F_p_ramfunctions_end - F_p_ramfunctions_start;
          __ramFunctions_LdAddr = __ramFunc_code_start + __data_size;

        } > .p_ramFuncSpace
```

In section segment `.ApplicationData`,

- Location counter shifted, prevent overlap with section segment `.data_in_p_flash_ROM` and `.ramFunctions`
- Setup `pROM-to-xRAM` copy utility for section segment `.data_in_p_flash_ROM`
- Setup `pROM-to-pRAM` copy utility for section segment `.ramFunctions`

Example 11. Selected content of .x_Data area

```
.ApplicationData :
        {
              # save space for the pROM data copy
              . = __xRAM_data_start + __data_size + __ramFunctions_size;
              # runtime code __init_sections uses these globals:
              ……
              ……
              ……
              F_Ldata_size     = __data_size;         #pROM-to-xRAM copy utility
              F_Ldata_RAM_addr = __xRAM_data_start;
              F_Ldata_ROM_addr = __pROM_data_start;

              F_Livt_size     = __ramFunctions_size; #pROM-to-pRAM copy utility
              F_Livt_RAM_addr = F_p_ramfunctions_start;
              F_Livt_ROM_addr = __ramFunctions_LdAddr;

              F_xROM_to_xRAM    = 0x0000;
              F_pROM_to_xRAM    = 0x0001;
              F_pROM_to_pRAM    = 0x0001;

              F_start_bss    = _START_BSS;
              F_end_bss      = _END_BSS;

              __DATA_END=.;
} > .x_Data
```

3.3. Understanding the map report

After successfully built projects, linker logs object placement result in a map file. Example 12 lists map result of section `.ramFunctions` in Code relocating with libraries.

- A pound sign indicates start of comment.
- A non-pound started line lists map result.
- Start address, length, type, function name and source file sequentially.
- Function `PIT_100k_OnInterrupt` start from 0xF010 with length 0x0028 words (80 bytes).
- The size of `PIT_100k_OnInterrupt()`, `PCLIB_ControllerPIANDLPFILTERFAsm`, `INTERRUPT_SAVEALL` and `INTERRUPT_RESTOREALL` calculated in variable `__ramFunctions_size` , and it shows all functions in section `.ram_functions` takes 0x84 words (264 bytes).

Example 12. Map result of .ramFunctions

```
# .ramFunctions
#>0000F010          F_p_ramfunctions_start (linker command file)
```

```
  0000F010 00000028 .text   FPIT_100k_OnInterrupt    (Events_c.obj)
  0000F038 0000001A .text   FPCLIB_ControllerPIANDLPFILTERFAsm      (56800Ex_PCLIB.lib pclib_controlle)
  0000F052 00000042 rtlib.text rtlib.text    (runtime 56800E smm.lib save_reg.o     )
  0000F052 00000023 rtlib.text INTERRUPT_SAVEALL     (runtime 56800E smm.lib save_reg.o     )
  0000F075 0000001F rtlib.text INTERRUPT_RESTOREALL  (runtime 56800E smm.lib save_reg.o     )
#>0000F094          F_p_ramfunctions_end (linker command file)
#>00000084          __ramFunctions_size (linker command file)
#>00007D21          __ramFunctions_LdAddr (linker command file)
```

4. Conclusion and usage consideration

This application note gives guidance about how to relocate codes/libraries to RAM. The PROM-to-PRAM tool in start-up code provides a way copy codes to RAM while DSC initialization, if this tool is unavailable. A customized upload tool also provided(Example 13). Fast mode of MC56F82748 double clocking frequency of 56800EX core. By relocating codes into RAM, the flash access time won't limit DSC performance anymore.

Example 13. Memory copy tool

```
asm void mem_copy(long p_source_addr,long p_dest_addr,unsigned int cnt)
{
        move.l a10,r2
        move.l b10,r3
        do y0,>>end_pdbCpy // copy for 'cnt' times
                move.w p:(r2)+,x0 // fetch value at p-address
                nop
                nop
                nop
                move.w x0,p:(r3)+ // stash value at p-address
                end_pdbCpy:
                nop
                rts
}
```

Many conditions cause DSC runway or relocate fail while editing the linker command file. When this occurs:

- First, check the map report and fix incorrect object placement with linker command file.
- Second, check copy results in debug mode, and fix copy parameters when start-up copy failed.

In Relocate objects into RAM, the undefined area brings extra zero text in flash. It can be minimized by the smaller one of section `.data_in_p_flash_ROM` and `.ramFunctions` in start of RAM. An opposite example is framewok of Code relocating with libraries which places `.data_in_p_flash_ROM` at start of RAM area.

5. References

1. *MC56F827xx Reference Manual* (document MC56F827XXRM).
2. *Inclusion of DSC Freescale Embedded Software Libraries in CodeWarrior 10.2* (document AN4586).
3. *CodeWarrior Development Studio for Microcontrollers V10.x Digital Signal Controller Build Tools Reference Manual* (document CWMCUDSCCMPREF)

4. *Relocating Code and Data Using the CodeWarrior Linker Command File for ColdFire Architecture* (document AN4329)

6. Revision history

Revision number	Date	Substantive changes
0	06/2015	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN5143
Rev. 0
06/2015