文档编号：AN5143
第0版，2015年6月

将MC56F827xx DSC的子例程重定位到PRAM

使用CodeWarrior的链接器命令文件

1. 介绍

MC56F827xx数字信号控制器（DSC）有两种速度模式：正常模式和快速模式。在正常模式下，代码在Flash和RAM中均以最大频率50MHz运行。在快速模式下，代码在Flash中以频率为50MHz运行，而在RAM中则以100MHz运行。本应用笔记详细介绍了如何将代码重定位到RAM，并加速某些时间关键型的子例程。

重定位是通过链接器命令文件（LCF）和在*.c源文件中使用pragma指令来实现的。

下面介绍两种实现场景。

- 将非预编译的源代码重定位到RAM——通过精心设计的链接器命令文件，为代码和数据动态分配RAM和Flash空间。
- 将已编译对象（例如库）重定位到RAM——为代码和数据动态分配RAM空间，但也必须为代码存储分配一个固定的Flash空间。

目录

2. 双速模式和内存映射

表1. MC56F827xx时钟模式

模式	内核	系统	IP总线
正常	50	50	50
快速	100	100	50

表1列出了MC56F827xx双速模式的工作频率。整个时钟系统由三部分组成：

- 内核时钟——为56800EX内核提供运行时钟。
- 系统时钟——为所有片上外设提供时钟频率管理，进而管理IP总线频率。系统时钟和系统集成模块（SIM）时钟是同义词。
- IP总线时钟——为所有外设提供读/写时钟。

Processor Expert (PEx) 是一款开发助手，可用于快速配置恩智浦微控制器。PEx在CPU bean中提供了一个速度模式配置开关。在图1中，配置切换到快速模式时，估算的内核、系统时钟和IP总线频率也会立即更新。如需了解更多信息，请参阅《MC56F827xx参考手册》（文档MC56F827XXRM）。

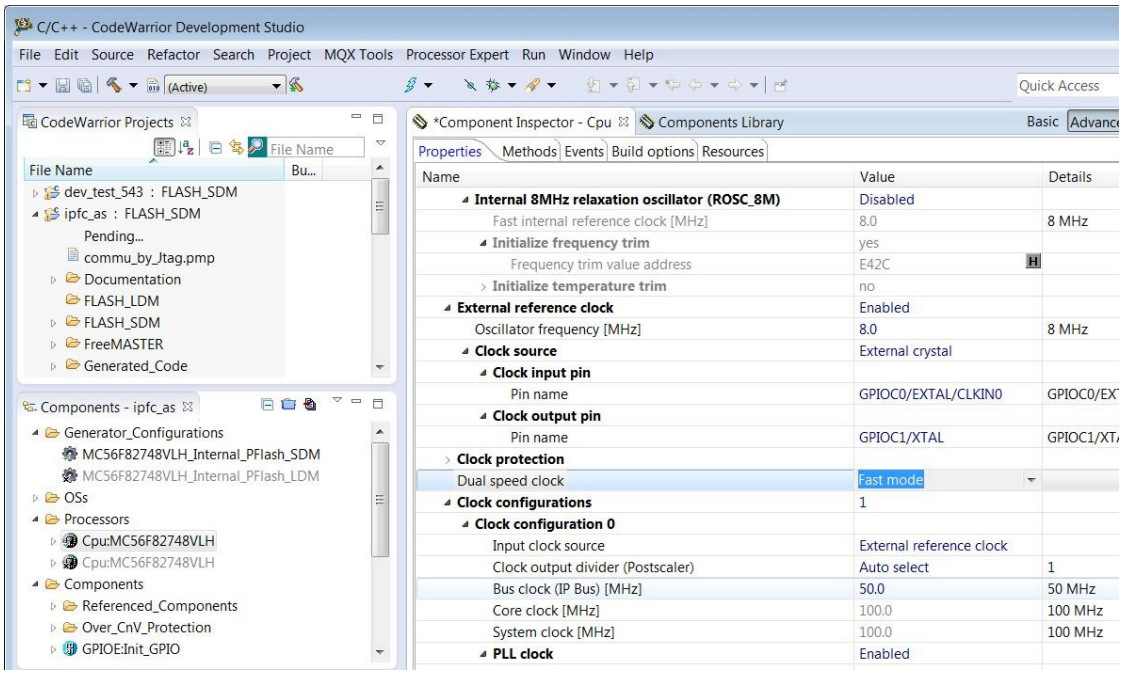

图1. 在CodeWarrior for Microcontrollers v10.6中将MC56F82748切换为快速模式

56F827xx在快速模式下运行时，代码在Flash中以频率50MHz运行，而在RAM中以频率100MHz运行。时间关键型子例程可以被重定位到RAM，从而提高性能。

恩智浦56800EX内核采用双哈佛架构。内核总线由程序数据总线（PDB）和主/辅数据总线（XDB）组成。可以通过PDB或XDB访问整个Flash和RAM。这两条寻址总线在恩智浦DSC上提供不同的功能，具体如下。

程序数据总线（PDB）：

- 内核指令获取

主/辅助数据总线（XDB）：

- 数据读/写
- 外设读/写。例如，DMA、ADC等。

将代码重定位到RAM需要单独的PDB映射的RAM中的子空间。在图2中，这些子空间从P:0xF000开始。前缀“P”和“X”分别表示程序数据总线和数据总线。由于PDB RAM和XDB RAM访问MC56F82748相同的片上RAM，因此必须在链接XDB RAM的位置之前，在PDB RAM中保留分配的子空间。如需了解更多信息，请查阅《MC56F827xx参考手册》（文档MC56F827XXRM）。

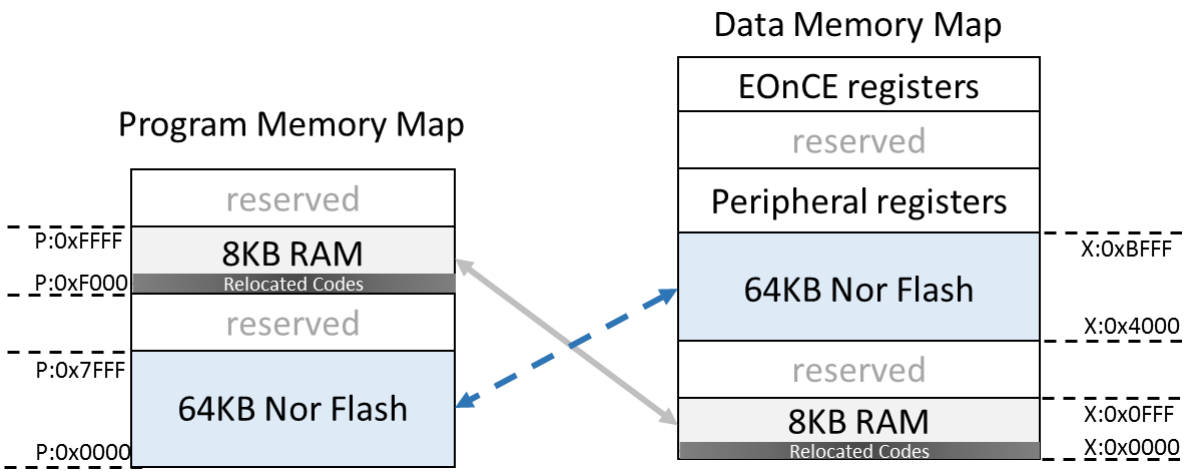

图2. MC56F82748 DSC内存映射

3. 将代码重定位到内部RAM

本节介绍如何使用CodeWarrior for Microcontrollers v10.6和PE将代码重定位到内部RAM。由于RAM是易失性存储器，重定位的代码必须首先存储在Flash中，然后在微控制器启动时复制到RAM。将代码重定位到RAM是通过LCF（链接器命令文件）和在源文件中使用pragma指令实现的。为了保留定制的链接器命令文件，必须禁用由Processor Expert (PE)自动生成LCF的选项，具体操作如图3所示。

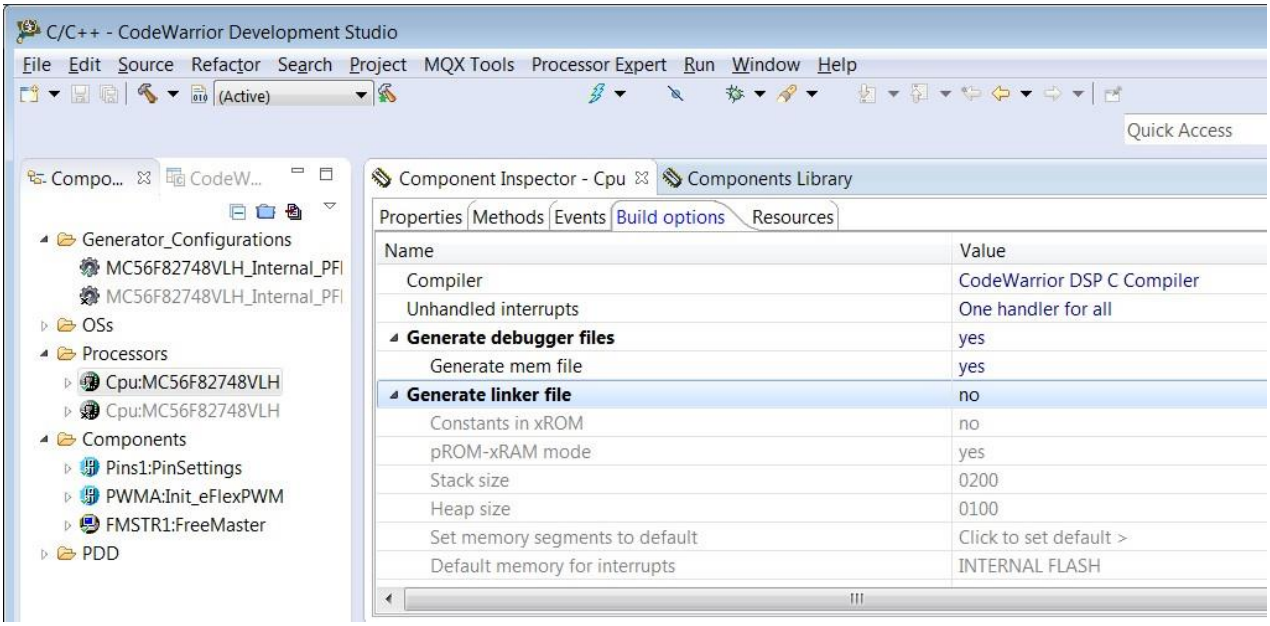

图3. 禁用Generate linker file选项

DSC的链接器命令文件结构包括三个段。

- **内存段**

  LCF中的内存段将可用内存划分为如例2所示的子段，这些子段依次列出了名称、属性、起始地址和长度。

- **关闭段**

  这是一个可选块，提供了一种不受死区剥离影响的符号分配方式。

- **节段**

  LCF中的节段定义了各内存段的内容，能够将起始/结束地址和段大小存储到全局变量中。有关这三个段的详细内容，请参阅《CodeWarrior Development Studio for Microcontrollers V10.x数字信号控制器构建工具参考手册第10.6版》第7.1节“链接器命令文件的结构”。

3.1. 不带库的代码重定位

本节介绍如何将非编译的源代码重定位到RAM中。以包含周期性中断服务例程`PIT_100k_OnInterrupt()`的CodeWarrior工程为例进行说明。下面将介绍如何使用`#pragma`指令定义唯一的代码段，并通过修改PEx生成的默认LCF，将其重定位到RAM中。

3.1.1. 使用pragma指令定义代码段

要将代码重定位到RAM中，就必须为链接时重定位标记所需的代码。例1展示了如何创建一个名为`ramFunc`的新段，并将函数`PIT_100k_OnInterrupt()`放入其中。`ramFunc`段中的所有代码在LCF中均以`ramFunc.text`引用，并且具有可读和可执行的属性。在`#pragma section ramArea`的`begin/end`之间的代码会成为`ramFunc`段的一部分。

例1. 使用pragma指令定义新代码段

```
#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. 将对象重定位到RAM

重定位的对象首先存储在Flash中，然后在微控制器初始化时复制到RAM中。例2列出了内存段，.p_ramSpace内存段是与Processor Expert生成的LCF的唯一不同之处。

在DSC的LCF中，具有RWX属性的内存段放置在PDB中，而具有RW属性的区域放置在XDB中。下面列出了每个内存区域的用途。

- 程序内存
  - .p_ramSpace ——PDB映射的RAM
  - .p_Code ——用户应用程序
  - .p_reserverd_FCF——Flash后门比较密钥
  - .p_Interrupts——中断向量表
- 数据内存
  - .x_internal_ROM ——XDB映射的flash
  - .x_Data——XDB映射的RAM

例2. 用于非编译代码重定位的内存段

```
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
}
```

在节段中，一个节以点号开头，后跟唯一的节名称和分号。花括号之间的代码块定义了节段的内容和变量，并以大于号结束，指定所映射的内存区域。

- .p_Code内存区域的内容由.ApplicationCode节定义，如例3所列，rtlib.text已用井号(#)注释掉。
- rtlib.text是用于保存/恢复所有寄存器的例程，它从.ApplicationCode节里注释掉，并插入到.ramFunctions节，当中断子例程使用“saveall”参数时，运行时库会发挥作用。
- 参数为“2”的“ALIGN”语法将位置计数器移动到下一个双字（4字节）的对齐边界，它还能够将位置计数器移动到下一个n字对齐边界，其中n是2的幂。

例3. LCF中的.p_Code内容

```
.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
```

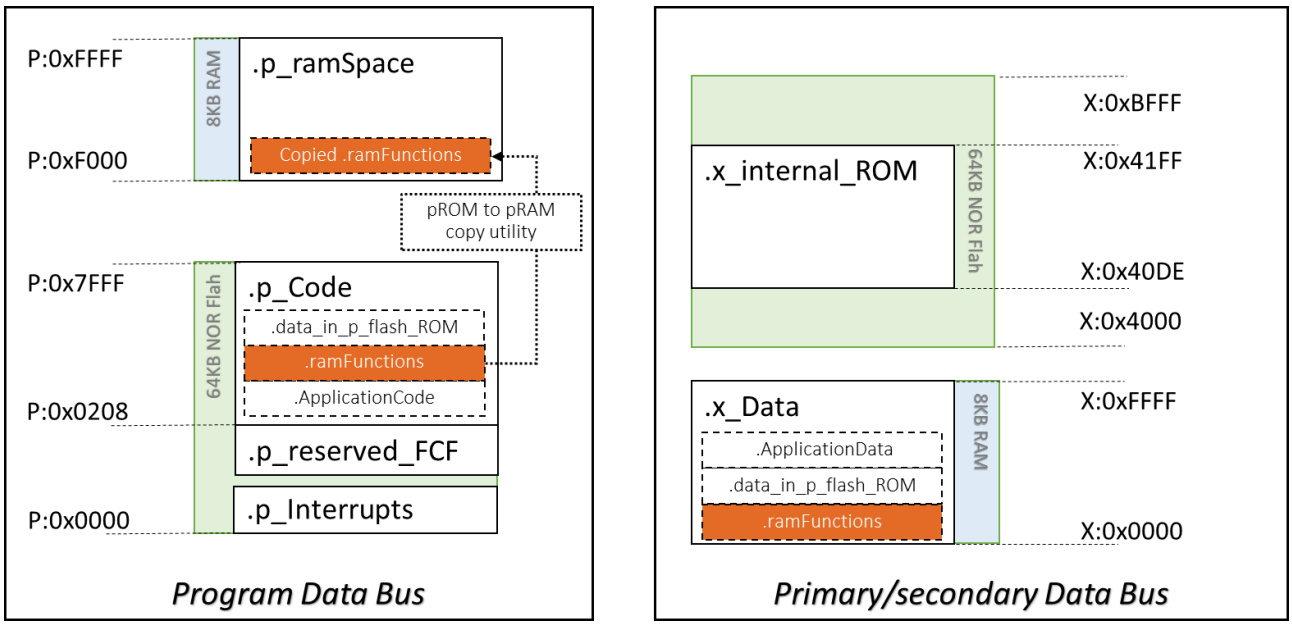

图4. 用于非编译代码重定位的内存规划

.p_ramSpace内存段的内容由例4中所列的节段.ramFunctions来定义。“AT”语法定义节段的可选加载/驻留地址。.ramFunctions中的代码首先会驻留在Flash中，地址为__pROM_code_start，然后在.p_ramSpace内存段上运行。在DSC初始化时，这些代码将被复制到.p_ramSpace内存段中。这些节段之间的关系如例4所示。重定位到RAM中的代码如下所列。

- 对象驻留在flash中，但在RAM上执行
  - 运行时库
  - “ramArea.text”节中的代码，由例1中专用pragma指令定义

例4. LCF的.p_ramSpace区域内容

```
.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
```

在例5中，.p_flash_ROM_data节段提供了预定义的变量容器，默认放置在RAM的开头。由于.ramFunctions节段已放置在RAM的开头（如例4所示），因此.data_in_p_flash_ROM和.ApplicationData节段必须放置在RAM中.ramFunctions节段之后，这是通过移动位置计数器来实现的，如例5中的阴影文本所示。

例5. LCF的.p_flash_ROM_data区域块

```
.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
```

注意

移位计数器会在.p_flash_ROM_data内存段中生成内容未定义的分区，如图5所示，这些未定义的分区会自动填充0x0000。为了消除它，可以为代码和数据分别分配两个固定的RAM区域。一般来说，建议在开发时保持RAM空间的动态分配。未定义区域的长度等于RAM中第一节段的长度，因此应先放置.ramfunctions和.data_in_p_flash_ROM中较小的那个节段，使其最小化。

ramISR.elf.p.S

#	Sx	Size	Address	Data	Csum
0	S0	0C	0000	000050524F4752414D	DB
1	S3	FD	00000000	54E1780354E1780354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E...	7E
2	S3	C9	0000007C	54E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E0354E26E...	44
3	S3	FD	00000208	18E40CE400004082040019E443E2000018E443E2000014F04187FEFF887882E0197815D018E44DE200004080010018E408E30000408000CF19E409...	79
4	S3	FD	00000284	000018E4CAE600004086000018E411E600004086000018E401E600004086010018E402E600004086008018E403E600004086000418E415E6000040...	59
5	S3	FD	00000300	4086000018E4C8E600004086000018E4CCE600004086000018E4C1E600004080 11F118E4C2E600004080003FF18E4C3E6000040800F0018E4C5E600...	54
6	S3	FD	0000037C	508900010CA000E700E718E423E400004082040018E400E400004082100018E420E300004086000318E40CE400004082100018E423E20000408208...	F1
7	S3	FD	000003F8	18E40BE400004086000418E4B8E200004086000054E14D0408E706E700E700E708E4000009E400000AE400000BE400000CE400000DE400000EE400...	2B
8	S3	B1	00000474	80E400E730E00AEB7A0400E701D400E780E8088F13A300E700E71AE4400000001BE4EE04000019E4120000030E00AEB910403F400E700E700E701...	88
9	S3	29	000004CA	04E73FD818E412E600004082001018E441E200004084010 03BF818E500E700E709E70000	6E
10	S3	A9	000004DC	0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010002000300040005000600070008000900 0A000B...	96
11	S7	05	00000378		7F

图5. 重定位MC56F82748源代码的记录文件

如例6所示，F_pROM_to_pRAM中的非零内容会使能pROM-to-pRAM的复制工具,三个变量F_Livt_size、F_Livt_ROM_addr和F_Livt_RAM_addr将.ramFunctions节的代码大小、驻留地址和运行时地址传递给pROM-to-pRAM工具。DSC启动初始化时将调用内存复制工具，启动初始化完成后用户程序将立即激活。

例6. .ApplicationData节中复制实用程序的参数赋值

```
        #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. 带库的代码重定位

上一节介绍了在MC56F82748上将代码重定位到RAM中的通用LCF框架。然而，预编译库隐藏了源代码，使得pragma段指令无法使用。为了解决这个问题，我们在LCF中使用“OBJECT”关键字来选择已编译代码（对象）。

修改后的内存映射规划如图6所示。为库的存储分配了一个固定大小的flash区域（.p_FuncCode），它从64KB flash的末端划出0x300个字（0x600个字节）。该区域限制了能在RAM上执行的代码/库的大小。由于56F82748的flash空间足够大，我们建议将.p_ramFuncCode的大小设置为8kb，但如果flash空间不足，可适当减少。

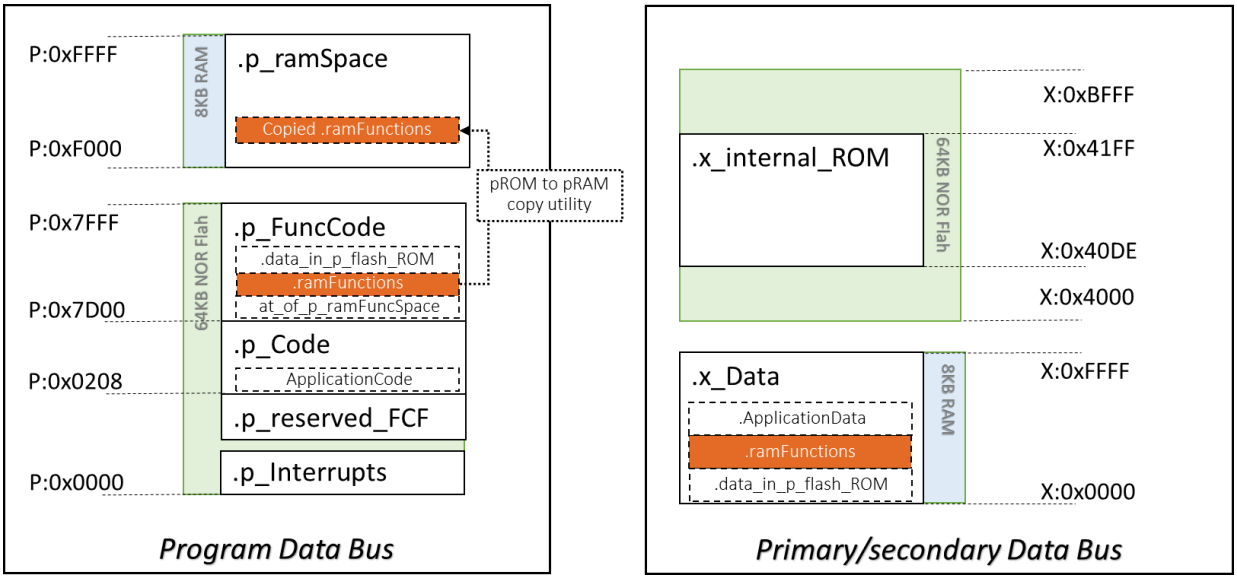

图6. 库的重定位的内存规划

在例7中，阴影文本表示Processor Expert生成的LCF的修改内容。

- .p_Code内存段的长度从原先的0x00007DF8缩减为0x00007AF8（62960字节）。
- 添加了额外的内存段.p_ramFuncCode，用于存储代码/库（并在RAM上执行）。

例7. 56F82748上库/代码重定位的内存段

```
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
}
```

例8中.at_of_p_ramFuncSpace节段将结束地址写入变量__pROM_data_start，供其他节插入内容。

例8. 库驻留的Flash区域

```
.at_of_p_ramFuncSpace :
```

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

例9中的.data_in_p_flash_ROM节段。

- 通过关键字“AT”驻留在PDB地址__pROM_data_start
- __ramFunc_code_start变量分配.ramFunctions节段的开头

例9. .datain_p_flash_ROM区域的选定内容

```
.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
```

在例10中，.ramFunctions节段定义了.p_ramFuncSpace的内容。

- 使用关键字“AT”为.p_ramFuncSpace area分配驻留地址
- 移位计数器“.”可防止与.data_in_p_flash_ROM节段重叠
- 使用“OBJECT”关键字选择函数，其参数可在映射文件（Project_Name.elf.xMAP）中找到。
- 运行时例程INTERRUPT_SAVEALL和INTERRUPT_RESTOREALL也被重定位到.p_ramFuncSpace

例10. PDB映射的RAM内容定义

```
.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
```

在.ApplicationData节段中，

- 将位置计数器移位，防止与.data_in_p_flash_ROM和.ramFunctions节段重叠
- 为.data_in_p_flash_ROM节段设置pROM-to-xRAM复制工具
- 为.ramFunctions节段设置pROM-to-pRAM复制工具

例11. .x_Data区域的选定内容

```
.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. 理解映射报告

成功构建工程后，链接器会在映射文件中记录对象放置的结果。例12列出了带库的代码重定位时，.ramFunctions节的映射结果。

- 井号(#)表示注释的开始。
- 非井号开头的行列出实际的映射结果。
- 按顺序列出起始地址、长度、类型、函数名称和源文件。
- PIT_100k_OnInterrupt函数从0xF010开始，长度为0x0028个字（80字节）。
- PIT_100k_OnInterrupt()、PCLIB_ControllerPIANDLPFILTERFAsm、INTERRUPT_SAVEALL和INTERRUPT_RESTOREALL的大小在__ramFunctions_size变量中进行计算，它显示了.ram_functions节中所有函数占用0x84个字（264字节）。

例12. .ramFunctions的映射结果

```
# .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.结论和使用注意事项

本应用笔记介绍了如何将代码/库重定位到RAM。如果没有可用的PROM-to-PRAM工具，可以在DSC初始化时使用启动代码中的内存复制工具将代码复制到RAM。我们还提供了一个自定义的上传工具（如例13所示）。MC56F82748的快速模式能够将56800EX内核时钟加倍。通过将代码重定位到RAM，可以确保flash的访问时间不会成为限制DSC性能的瓶颈。

例13. 内存复制工具

```
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
}
```

在编辑链接器命令文件时，多种情况可能导致DSC运行异常或重定位失败。在这种情况下：

- 首先，检查映射报告并纠正链接器命令文件中不正确的对象位置。
- 其次，在调试模式下检查复制结果，并在启动复制失败时，调整复制参数。

在将对象重定位到RAM时，未定义区域会在flash中产生额外的空白文本。为了减少这种情况，可以将.data_in_p_flash_ROM节和.ramFunctions节中较小的一个放置在RAM的起始部分。相反，对于重定位库代码的框架，可以将.data_in_p_flash_ROM放置在RAM区域的起始位置。

5.参考资料

1. 《MC56F827xx参考手册》（文档MC56F827XXRM）。
2. 《CodeWarrior 10.2中的 DSC Freescale嵌入式软件库》（文档AN4586）。
3. 《CodeWarrior Development Studio for Microcontrollers V10.x数字信号控制器构建工具参考手册》（文档CWMCUDSCCMPREF）。

4. 《ColdFire架构使用CodeWarrior链接器命令文件重定位代码和数据》（文档AN4329）。

6.修订历史

版本号	日期	实质性变更
第0版	2015年6月	初版发布

How to Reach Us:

Home Page:

nxp.com.cn

Web Support:

nxp.com.cn/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

NXP reserves the right to make changes without further notice to any products herein. NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com.cn/SalesTermsandConditions.

Document Number: AN5143
Rev. 0
06/2015