关于Codewarrior 中的 .prm 文件

要讨论单片机的地址映射，就必须要接触.prm文件，本篇的讨论基于 Codewarrior 5.0 编译器，单片机采用MC9S12XS128。

通过项目模板建立的新项目中都有一个名字为“project.prm”的文件，位于Project Settings->Linker Files文件夹下。一个标准的基于XS128的.prm文件起始内容如下：

.prm文件范例：

```
/* This is a linker parameter file for the MC9S12XS128 */
/*This file is setup to use the HCS12X core only.If you plan to also use the XGATE in your project, best create a new
   project with the'New Project Wizard' (File|New... menu in the CodeWarrior IDE) and choose the appropriateproject
   parameters.*/

NAMES       /* CodeWarrior will pass all the needed files to the linker by command line. But here you may add
                your ddditional files */
END

SEGMENTS    /* here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. All addresses are
                logical' */

/* Register space   */
/*      IO_SEG          = PAGED                                 0x0000   TO    0x07FF; intentionally not
   defined */

/* non-paged RAM */
       RAM              = READ_WRITE    DATA_NEAR               0x2000   TO    0x3FFF;

/* non-banked FLASH */
       ROM_4000         = READ_ONLY     DATA_NEAR   IBCC_NEAR   0x4000   TO    0x7FFF;
       ROM_C000         = READ_ONLY     DATA_NEAR   IBCC_NEAR   0xC000   TO    0xFEFF;
/*     VECTORS          = READ_ONLY                             0xFF00   TO    0xFFFF; intentionally not
   defined: used for VECTOR commands below */
//OSVECTORS       = READ_ONLY                                   0xFF10   TO    0xFFFF;   /* OSEK interrupt
vectors (use your vector.o) */

/* paged EEPROM                                                 0x0800   TO    0x0BFF; addressed through
   EPAGE */
       EEPROM_00        = READ_ONLY     DATA_FAR    IBCC_FAR    0x000800 TO    0x000BFF;
```

```
        EEPROM_01       = READ_ONLY   DATA_FAR   IBCC_FAR    0x010800   TO   0x010BFF;
        EEPROM_02       = READ_ONLY   DATA_FAR   IBCC_FAR    0x020800   TO   0x020BFF;
        EEPROM_03       = READ_ONLY   DATA_FAR   IBCC_FAR    0x030800   TO   0x030BFF;
        EEPROM_04       = READ_ONLY   DATA_FAR   IBCC_FAR    0x040800   TO   0x040BFF;
        EEPROM_05       = READ_ONLY   DATA_FAR   IBCC_FAR    0x050800   TO   0x050BFF;
        EEPROM_06       = READ_ONLY   DATA_FAR   IBCC_FAR    0x060800   TO   0x060BFF;
        EEPROM_07       = READ_ONLY   DATA_FAR   IBCC_FAR    0x070800   TO   0x070BFF;

/* paged RAM:                                                 0x1000     TO    0x1FFF; addressed through
   RPAGE */
/* RAM_FE                = READ_WRITE                         0xFE1000   TO    0xFE1FFF; intentionally not
   defined: equivalent to RAM: 0x2000..0x2FFF */
/*      RAM_FF           = READ_WRITE                         0xFF1000   TO    0xFF1FFF; intentionally not
   defined: equivalent to RAM: 0x3000..0x3FFF */
/* paged FLASH:                                               0x8000     TO   0xBFFF; addressed through
   PPAGE */
        PAGE_F8         = READ_ONLY   DATA_FAR   IBCC_FAR    0xF88000   TO   0xF8BFFF;
        PAGE_F9         = READ_ONLY   DATA_FAR   IBCC_FAR    0xF98000   TO   0xF9BFFF;
        PAGE_FA         = READ_ONLY   DATA_FAR   IBCC_FAR    0xFA8000   TO   0xFABFFF;
        PAGE_FB         = READ_ONLY   DATA_FAR   IBCC_FAR    0xFB8000   TO   0xFBBFFF;
        PAGE_FC         = READ_ONLY   DATA_FAR   IBCC_FAR    0xFC8000   TO   0xFCBFFF;
/*      PAGE_FD         = READ_ONLY                          0xFD8000   TO   0xFDBFFF; intentionally not
   defined: equivalent to ROM_4000 */
        PAGE_FE         = READ_ONLY   DATA_FAR   IBCC_FAR    0xFE8000   TO   0xFEBFFF;
/*      PAGE_FF         = READ_ONLY                          0xFF8000   TO   0xFFBFFF; intentionally not
   defined: equivalent to ROM_C000 */
END

PLACEMENT /* here all predefined and user segments are placed into the SEGMENTS defined above. */

        _PRESTART,                      /* Used in HIWARE format: jump to _Startup at the code start */
        STARTUP,                        /* startup data structures */
        ROM_VAR,                        /* constant variables */
        STRINGS,                        /* string literals */
        VIRTUAL_TABLE_SEGMENT,          /* C++ virtual table segment */
        //.ostext,                      /* eventually OSEK code    */
        DEFAULT_ROM, NON_BANKED,        /* runtime routines which must not be banked */
        COPY                            /* copy down information: how to initialize variables */
                                        /* in case you want to use ROM_4000 here as well, make sure
                                           that all files (incl. library files) are compiled with the
                                           option: -OnB=b */
                        INTO            ROM_C000 /*, ROM_4000*/ ;

        OTHER_ROM       INTO            PAGE_FE, PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8;
```

```
    //.stackstart,              /* eventually used for OSEK kernel awareness: Main-Stack Start */
      SSTACK,                   /* allocate stack first to avoid overwriting variables on overflow */
    //.stackend,                /* eventually used for OSEK kernel awareness: Main-Stack End */
      PAGED_RAM,                /* there is no need for paged data accesses on this derivative */

      DEFAULT_RAM               /* all variables, the default RAM location */
                        INTO                RAM;

      DISTRIBUTE        DISTRIBUTE_INTO
                                            ROM_4000, PAGE_FE, PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8;
      CONST_DISTRIBUTE  DISTRIBUTE_INTO
                                            ROM_4000, PAGE_FE, PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8;
      DATA_DISTRIBUTE   DISTRIBUTE_INTO
                                            RAM;
       //.vectors         INTO   OSVECTORS; /* OSEK vector table */
END

ENTRIES   /* keep the following unreferenced variables */
          /* OSEK: always allocate the vector table and all dependent objects */
          //_vectab OsBuildNumber _OsOrtiStackStart _OsOrtiStart
END

STACKSIZE   0x100   /* size of the stack (will be allocated in DEFAULT_RAM) */

/* use these definitions in plane of the vector table ('vectors') above */
VECTOR 0 _Startup   /* reset vector: this is the default entry point for a C/C++ application. */
                    //VECTOR 0 Entry   /* reset vector: this is the default entry point for an Assembly application. */
                    //INIT Entry       /* for assembly applications: that this is as well the initialization entry point */
```

1 .prm 文件组成结构

按所含的信息的不同. prm文件有六个组成部分构成，这里仅讨论和内存空间映射关系紧密的三个部分，其他的不做讨论。

- **SEGMENTS … END**

定义和划分芯片所有可用的内存资源，包括程序空间和数据空间。一般我们将程序空间定义成ROM，把数据空间定义成RAM，但这些名字都不是系统保留的关键词，可以由用户随意修改。用户也可以把内存空间按地址和属性随意分割成大小不同的块，每块可以自由命名。例如同样是RAM，可以使用不同的属性，使其有复位后变量清零和不清零之分。

关于内存划分的具体方法在后面详解。

- **PLACEMENT … END**

将指派源程序中所定义的各种段，如数据段DATA_SEG、CONST_SEG和代码段CODE_SEG被具体放置到哪一个内存块中。它是将源程序中的定义描述和实际物理内存挂钩的桥梁。

- **STACKSIZE**

定义系统堆栈长度，其后给出的长度字节数可以根据实际应用需要进行修改。堆栈的实际定位取决于RAM内存的划分和使用情况。默认的情况下，堆栈放在RAM区域的起始部分。当然，堆栈的定义不只有这种方式，还可以使用STACKTOP关键字。后面将详细讨论。

2 内存划分的具体方式

由SEGMENTS开始到END为止，中间可以添加任意多行内存划分的定义，每一行用分号结尾。定义行的语法型式为：

[块名] = [属性1] [属性2] ， … ， [属性n] [起始地址] TO [结束地址];

其中，

- “块名”的定义和C语言变量定义相同，是以英文字母开头的一个字符串，用户可以自己任意定义块名。

- “属性”用户是不能自己定义的，因为属性名指定了上面所说的“块名”所对应的不同的内存类型和访问方式，而不同物理内存的类型和访问方式是一定的。

对于“属性1”，Codewarrior 5.0中可以有三种不同的类型，对于只读的Flash-ROM区属性一定是READ_ONLY，对于可读写的RAM区属性可以是READ_WRITE，也可以是NO_INIT。它们两者的关键区别是ANSI-C的初始化代码会把定位在READ_WRITE块中的所有全局和静态变量自动清零，而NO_INIT块中的变量将不会被自动清零。当然只是复位时不清零，掉电时还是清零的，但是对于单片机系统，变量在复位时不被自动清零这一特性有时是很关键的，在某些应用中有特殊的用途。

对于“属性2 … 属性n”，根据上面给出的.prm的范例文件可以看出来，可能的形式有“DATA_FAR”、“DATA_NEAR”、“IBCC_FAR”、“IBCC_NEAR”四种类型。其中，“DATA_FAR”

和“DATA_NEAR”相对应，当内存区域包含变量或者是常量时（通常是RAM、Flash和EEPROM），必须指明上面两种属性中的一种，由于涉及到内存的分页，可以这样理解：“DATA_FAR”属性指定的内存块为可以保存数据的非固定页，而“DATA_NEAR”属性指定的内存块为可以保存数据的固定页；同理“IBCC_FAR”和“IBCC_NEAR”相对应，当内存区域包含代码时（Flash和EEPROM），必须指明上面两种属性中的一种，“IBCC_FAR”属性指定的内存块为可以保存代码的非固定页，而“IBCC_NEAR”属性指定的内存块为可以保存代码的固定页

讨论到这里，细心的读者已经发现，在上面的.prm文件范例中，RAM的属性有“DATA_FAR”和“DATA_NEAR”两种，Flash的属性中也是四种都有，但是EEPROM中却只有“DATA_FAR”和“IBCC_FAR”两种，这正好验证了上一篇文章（飞思卡尔16位单片机的资源配置）中所提到的，RAM、Flash中都有固定页，但是EEPROM中全部是非固定页。

- 起始地址和结束地址决定了一内存块的物理位置，对于固定页，用4位16进制数表示，而对于非固定页，则用6位16进制表示，多出来的两位其实是寄存器EPAGE、RPAGE或PPAGE的值，可见，对于分页的资源，是通过寄存器（EPAGE、RPAGE或PPAGE）和16位的地址总线的组合来进行寻址的。

“TO”是系统保留的关键字，必须大写。

下面，根据上面范例提供的内容，举几个例子：

例1　RAM　= READ_WRITE　DATA_NEAR　0x2000 TO　0x3FFF;

上面这句话的意思是：分配0x2000-0x3FFF的区域的块名为“RAM”（当然可以定义别的名称），由上一篇文章而知，这一区域的物理内存的性质为RAM，属性应该为“READ_WRITE”，并且这一区域中的两页都为固定页，所以为“DATA_NEAR”。

例2　将8K字节RAM的后面4K字节定义成非自动清零的数据保留区，则应如下定义：

```
SEGMENTS
    ......
    RAM           = READ_WRITE   DATA_NEAR    0x2000 TO   0x2FFF;
    RAM_NO_INIT   = NO_INIT      DATA_NEAR    0x3000 TO   0x3FFF;
    ......
END
```

注意，各部分RAM的分配地址不应该存在重叠的部分，否则会发生错误。

例3　EEPROM_00　= READ_ONLY　DATA_FAR IBCC_FAR　0x000800 TO 0x000BFF;

XS128单片机中的EEPROM由Data-Flash模拟，所以属性为READ_ONLY。EEPROM全部为非固定页，所以用“DATA_FAR”、“IBCC_FAR”。后面的起始地址和结束地址分别为6位的16进制数，前两位的“00”实质指的是EEPROM分页寄存器EPAGE的值为0x00。

用SEGMENTS只是从单片机的物理内存这一角度对其进行空间划分。源程序本身并不知道物理内存被分割和属性定义的这些细节。它们两者之间必须通过下面的PLACEMENT建立联系。

3 程序段和数据段的放置

PLACEMENT-END内所描述的信息是告诉连接器源程序中所定义的各类段应该被具体放置到哪一个内存块中去。其语法型式为：

[段名1], [段名2]，… ，　[段名n]　INTO [内存块名1]，[内存块名2]，… ，[内存块名n];

和

[段名1], [段名2]，… ，　[段名n]　DISTRIBUTE_INTO [内存块名1]，[内存块名2]，… ，[内存块名n];

其中

- 段名就是在源程序中用“#pragma”声明的数据段、常数段或代码段的名字。如果用缺省名“DEFAULT”， 则默认的数据段名为DEFAULT_RAM，代码段和常数段名为DEFAULT_ROM。若程序中定义的段名没有在PLACEMENT中提及，则将被视同为DEFAULT。几个相同性质但不同名字的段可以被放置到同一个内存块中，相互之间用逗号分隔。

- INTO 是系统保留的关键词，在这里为“放入”的意思。

- DISTRIBUTE_INTO 也是系统的保留关键字。Codewarrior 具有内存自动优化的功能，但是在“Small memory”模式中，这种功能不会被启用，只有当16-bit的地址空间不能存放下所有的变量和代码时，才会启用这种功能。

在SEGMENTS-END区域中，当在内存模块的属性中加上“DATA_FAR”、“DATA_NEAR”、“IBCC_FAR”、“IBCC_NEAR”四种属性中的任何一种时，那么在PLACEMENT-END区域中，就需要指定段名“DISTRIBUTE”，“CONST_DISTRIBUTE”，“DATA_DISTRIBUTE”（系统默认的，非关键字，用户可以自行更改）所分配的内存空间，这就需要使用“DISTRIBUTE_INTO”关键字。

关于内存自动优化功能，可以参考freescale的官方技术手册“TN 262.pdf”。

- 内存块名就是前面介绍的用SEGMENTS划分好的不同的内存块名字。

利用这样直观的定位描述文本可以方便灵活的将数据或代码定位到芯片内存任意可能的位置，实现某些特殊目的的应用。

下面的例子，说明了各种段名、PLACEMENT 和SEGMENTS之间的对应关系。

例4 定义非自动清零的数据段

```
SEGMENTS
    ......
     RAM             = READ_WRITE    DATA_NEAR     0x2000 TO    0x2FFF;
    RAM_NO_INIT    = NO_INIT         DATA_NEAR     0x3000 TO    0x3FFF;
    ......
END

PLACEMENT
    ......
DATA_PERSISTENT    INTO    RAM_NO_INIT;
    ......
END
```

```
//源程序编写：
#pragma   DATA_SEG   DATA_PERSISTENT //定义复位时非自定清零数据段
byte sysState;
#pragma   DATA_SEG   DEFAULT
```

4 堆栈的设置

关于堆栈的设置，Codewarrior提供了两种方式：“STACKSIZE”命令方式和“STACKTOP”命令方式。这两种方式在同一个.prm文件中，不能同时存在。当用户只关心堆栈的大小而不关心堆栈的存放位置时，推荐使用STACKSIZE方式。

系统默认的方式为使用STACKSIZE方式。

STACKSIZE命令方式：

当使用STACKSIZE命令方式时，如果在PLACEMENT-END部分声明了“SSTACK INTO RAM”，这样的话，堆栈区就被放在RAM区域的起始部分，下面的例子说明了这种方式：

例5

```
SEGMENTS
    ......
    RAM             =  READ_WRITE    DATA_NEAR      0x2000  TO    0x3FFF;
    ......
END

PLACEMENT
    ......
    SSTACK,  PAGED_RAM,  DEFAULT_RAM   INTO   RAM;
    ......
END
STACKSIZE   0x100
```

上面的例子将堆栈区域存放的地址为0x20FF-0x2000, 初始的堆栈指针指向栈顶地址0x20FF。

相反，如果在PLACEMENT-END部分没有声明“SSTACK INTO RAM”，则堆栈被分配在RAM区域中已分配空间的后面。请参见例6。

例6

```
SEGMENTS
    ......
    RAM             = READ_WRITE    DATA_NEAR      0x2000  TO    0x3FFF;
    ......
END
```

```
PLACEMENT
    ......
    PAGED_RAM,   DEFAULT_RAM    INTO   RAM;
    ......
END
STACKSIZE   0x100
```

在这个例子中，如果RAM区域中已经分配的变量占用了4个字节（从0x2000到0x2003），则堆栈放在这四个字节的后面，从0x2103到0x2004，初始的堆栈指针指向0x2103。

STACKYOP命令方式：

当使用STACKTOP命令方式时，如果在PLACEMENT-END部分声明了“SSTACK INTO RAM”，同样，堆栈区就被放在RAM区域的起始部分，初始的栈顶则由STACKTOP指定。若没有相应的声明，则初始的栈顶由STACKTOP指定，而堆栈的大小则根据处理器的不同由编译器自行设定，其大小足够装下处理器的PC寄存器的值。