第 3 章

使用 C 语言操作 DSP 的寄存器

嵌入式系统开发常用的语言通常有两种，即汇编语言和 C 语言。绝大多数的工程师对汇编语言的感觉肯定都是类似的，就是难以理解，想到用汇编语言来开发一个复杂的工程肯定有点担心，但是如果换成 C 语言肯定就会好多了，毕竟 C 语言是面向对象的，更贴近平时的语言习惯。幸好，DSP 的开发既支持汇编语言，也支持 C 语言。在开发 DSP 程序时用得比较多的还是 C 语言，只有在对于时间要求非常严格的地方才会插入汇编语言。开发时，需要频繁地对 DSP 的寄存器进行配置。本章将以 F28335 的外设 SCI 为例，介绍如何使用 C 语言中结构体和位定义的方法来实现对寄存器的操作，并了解 F28335 的头文件是如何编写的。

3.1 寄存器的 C 语言访问

由于 DSP 的寄存器能够实现对系统和外设功能的配置与控制，因此在 DSP 的开发过程中，对于寄存器的操作是极为重要的，也是很频繁的，也就是说，对寄存器的操作是否方便会直接影响到 DSP 的开发是否方便。幸好，F28335 为用户提供了位定义和寄存器结构体的方式，能够很方便地访问和控制内部寄存器。接下来，将以外设串行通信接口（Serial Communication Interface，SCI）为例，详细介绍如何使用 C 语言的位定义和寄存器结构体的方式来实现对 SCI 寄存器的访问，在这个过程中，大家也可以了解 F28335 头文件的编写方法。

3.1.1 了解 SCI 的寄存器

这部分内容本身应该是介绍 SCI 的时候才讲的，为了让大家对 SCI 的寄存器能够提前有所了解，故在此也稍作介绍。F28335 的 SCI 模块具有相同功能的串行通信接口 SCI-A、SCI-B 和 SCI-C，也就是说，体现到硬件上，F28335 可支持 3 个串口。SCI-A、SCI-B 和 SCI-C 就像三胞胎一样，具有相同的寄存器，如表 3-1 所示。

表 3-1 SCI 相关寄存器

寄存器名	地址单元格			大小(×16bit)	功能描述
	SCI-A	SCI-B	SCI-C		
SCICCR	0x0000 7050	0x0000 7750	0x0000 7770	1	通信控制寄存器
SCICTL1	0x0000 7051	0x0000 7751	0x0000 7771	1	控制寄存器 1
SCIHBAUD	0x0000 7052	0x0000 7752	0x0000 7772	1	数据传输速率寄存器高字节
SCILBAUD	0x0000 7053	0x0000 7753	0x0000 7773	1	数据传输速率寄存器低字节
SCICTL2	0x0000 7054	0x0000 7754	0x0000 7774	1	控制寄存器 2
SCIRXST	0x0000 7055	0x0000 7755	0x0000 7775	1	接收状态寄存器
SCIRXEMU	0x0000 7056	0x0000 7756	0x0000 7776	1	仿真缓冲寄存器
SCIRXBUF	0x0000 7057	0x0000 7757	0x0000 7777	1	接收数据缓冲寄存器
SCITXBUF	0x0000 7059	0x0000 7759	0x0000 7779	1	发送数据缓冲寄存器
SCIFFTX	0x0000 705A	0x0000 775A	0x0000 777A	1	FIFO 发送寄存器
SCIFFRX	0x0000 705B	0x0000 775B	0x0000 777B	1	FIFO 接收寄存器
SCIFFCT	0x0000 705C	0x0000 775C	0x0000 777C	1	FIFO 控制寄存器
SCIPRI	0x0000 705F	0x0000 775F	0x0000 777F	1	优先权控制寄存器

从表 3-1 可以看到,外设 SCI 的每一个寄存器都占据 1 字节,即 16 位宽度。从其地址分布来看,SCI-A 的寄存器地址从 0x0000 7050 到 0x0000 705F,中间缺少了 0x0000 7058、0x0000 705D、0x0000 705E。SCI-B 的寄存器地址从 0x0000 7750 到 0x0000 775F,中间缺少了 0x0000 7758、0x0000 775D、0x0000 775E。SCI-C 的寄存器地址从 0x0000 7770 到 0x0000 777F,中间缺少了 0x0000 7758、0x0000 775D、0x0000 775E。中间缺少的这些地址为系统保留的寄存器空间,暂时还没有使用。表 3-1 所列出的寄存器位于 F28335 存储器空间的外设帧 2 内,是在物理上实际存在的存储器单元。实际上,这些寄存器就是预定义了具体功能的存储单元,系统会根据这些存储单元具体的配置来进行工作。

在自然语言中去描述 SCI-A 寄存器 SCICCR 的某个位时,可以读作“SCIA 的通信控制寄存器 SCICCR 的第 x 位”。这么读的时候第一反应是什么?这不是和 C 语言结构体成员的表述方式一样吗?这说明 SCI-A 的寄存器可以采用结构体的方式来表示。

3.1.2 使用位定义的方法定义寄存器

先来介绍一下 C 语言中一种被称为“位域”或者“位段”的数据结构。所谓“位域”,就是把一个字节中的二进制位划分为几个不同的区域,并说明每个区域的位数。每个域都有一个域名,允许在程序中按域名进行操作。位域的定义和位域变量的说明同结构体定义和其成员说明类似,其语法格式为:

```
Struct 位域结构名
{
 类型说明符 位域名 1:位域长度;
 类型说明符 位域名 2:位域长度;
```

```
 …
 类型说明符 位域名 n:位域长度;
};
```

其中，类型说明符就是基本的数据类型，可以是 int、char 型等。位域名可以任意取，能够反映其位域的功能即可，位域长度是指这个位域是由多少个位组成的。和结构体定义一样，花括号最后的“；”不可缺少，否则会出错。

图 3-1 是将一个名为 bs 字的 16 位划分成 3 个位域，其中 D0～D7 共 8 位为位域 a，D8～D9 共 2 位为位域 b，D10～D15 共 6 位为位域 c。若用位域的方式来定义，则如例 3-1 所示。

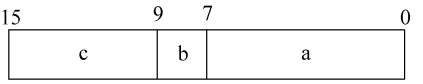

图 3-1　bs 的位域定义

【例 3-1】 位域定义。

```
struct bs                                    //定义位域 bs
 {
 int a:8;
 int b:2;
 int c:6;
 };
struct bs bs1;                               //声明 bs 型变量 bs1
```

位域也是 C 语言中的一种数据结构，因此需要遵循先声明后使用的原则。在例 3-1 中，声明了 bs1，说明 bs1 是 bs 型的变量，共占 2 字节，其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

关于位域的定义还有以下几点说明：

(1) 位域的定义必须按从右往左的顺序，也就是说得从最低位开始定义。

(2) 一个位域必须存储在同一个字节中，不能跨两个字节。如果一个字节所剩空间不够放另一位域时，应该从下一个单元起存放该位域，如下所示：

```
struct bs
{
 int a:4;
 int :0;                                     //空域
 int b:5;                                    //从第二个字节开始存放
 int c:3;
};
```

在这个位域定义中，第一个位域 a 占第一个字节的 4 位，而第二个位域 b 占 5 位。很显然第一个字节剩下的 4 位不能够完全容纳位域 b，所以第一个字节的后 4 位写 0 留空，b 从第二个字节开始存放。

(3) 位域的长度不能大于一个字节的长度，也就是说一个位域不能超过 8 位。

(4) 位域可以无位域名，这时，它只用作填充或调整位置。无名的位域是不能使用的，

如下所示：

```
struct bs
{
 int a:4;
 int :2;                                    //这 2 位不能使用
 int b:2;
 int c:5;
 int d:3;
};
```

掌握了 C 语言中位域的知识后，下面以 SCI-A 的通信控制寄存器 SCICCR 为例说明如何使用位域的方法来定义寄存器。图 3-2 为 SCI-A 通信控制寄存器 SCICCR 的具体定义。

7	6	5	4	3	2	1	0
STOP BITS	EVEN/ODD PARITY	PARITY ENABLE	LOOPBACK ENA	ADDR/IDLE MODE	SCICHAR2	SCICHAR1	SCICHAR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

图 3-2　SCI-A 通信控制寄存器 SCICCR

SCI-A 模块所有的寄存器都是 8 位的，当一个寄存器被访问时，寄存器数据位于低 8 位，高 8 位为 0。SCICCR 的 D0～D2 为字符长度控制位 SCICHAR，占据 3 位。D3 为 SCI 多处理器模式控制位 ADDRIDLE_MODE，占据 1 位。D4 为 SCI 回送从测试模式使能位 LOOPBKENA，占据 1 位。D5 为 SCI 极性使能位 PARITYENA，占据 1 位。D6 为 SCI 奇/偶极性使能位 PARITY，占据 1 位。D7 为 SCI 结束位的个数 STOPBITS，也占据 1 位。D8～D15 保留，共 8 位。因此，可以将寄存器 SCICCR 用位域的方式表示为如例 3-2 所示的数据结构。

【例 3-2】 用位域方式定义 SCICCR。

```
struct SCICCR_BITS
{
  Uint16 SCICHAR:3;                         //2:0  字符长度控制位
  Uint16 ADDRIDLE_MODE:1;                   //3  多处理器模式控制位
  Uint16 LOOPBKENA:1;                       //4  回送测试模式使能位
  Uint16 PARITYENA:1;                       //5  极性使能位
  Uint16 PARITY:1;                          //6  奇/偶极性选择位
  Uint16 STOPBITS:1;                        //7  结束位个数
  Uint16 rsvd1:8;                           //15:8  保留
};

struct SCICCR_BITS bit;
bit.SCICHAR = 7;                            //SCI 字符长度控制位为 8 位
```

在寄存器中，被保留的空间也要在位域中定义，只是定义的变量不会被调用，如例 3-2 中的 rsvd1，为 8 位保留的空间。一般位域中的元素是按地址的顺序来定义的，所以中间如果有空间保留，那么需要一个变量来代替，虽然变量并不会被调用，但是必须要添加，以防后续寄存器位的地址混乱。

例 3-2 还声明了一个 SCICCR_BITS 的变量 bit，这样就可以通过 bit 来实现对寄存器的位的访问了。例子中是对位域 SCICHAR 赋值，配置 SCI 字符控制长度为 8 位（SCICHAR 的值为 7，对应于字符长度为 8 位）。

3.1.3　声明共同体

使用位定义的方法定义寄存器可以方便地实现对寄存器的功能位进行操作，但是有时如果需要对整个寄存器进行操作，那么位操作是不是就显得有些麻烦了呢？所以很有必要引入能够对寄存器整体进行操作的方式，这样想要进行整体操作时就用整体操作的方式，想要进行位操作时就用位操作的方式。这种二选一的方式是不是让人想起 C 语言的共同体了呢？例 3-3 为对 SCI 的通信控制寄存器 SCICCR 进行共同体的定义，使得用户便于选择对位或者寄存器整体进行操作。

【例 3-3】 SCICCR 的共同体定义。

```
 union SCICCR_REG
 {
Uint16 all;                                  //可实现对寄存器整体操作
struct SCICCR_BITS bit;                      //可实现位操作
 };
 union SCICCR_REG SCICCR;
 SCICCR.all = 0x007F;
 SCICCR.bit.SCICHAR = 5;
```

例 3-3 先是定义了一个共同体 SCICCR_REG，然后声明了一个 SCICCR_REG 变量 SCICCR，接下来变量 SCICCR 就可以对寄存器实现整体操作或者进行位操作，很方便。在例 3-3 中，先是通过整体操作，对寄存器的各个位进行了配置，SCICHAR 位被赋值为 7，也就是说，SCI 数据位长度为 8；紧接着，变量 SCICCR 通过位操作的方式，将 SCICHAR 的值改为 5，即 SCI 数据的长度最终被设置为 6。

3.1.4　创建结构体文件

从表 3-1 可以看出，SCI 模块除了寄存器 SCICCR 之外，还有许多的寄存器。为了便于管理，需要创建一个结构体，用来包含 SCI 模块的所有的寄存器，如例 3-4 所示。

【例 3-4】 SCI 寄存器的结构体文件。

```
struct SCI_REGS
{
```

```
    union SCICCR_REG   SCICCR;                    //通信控制寄存器
    union SCICTL1_REG  SCICTL1;                   //控制寄存器 1
    Uint16       SCIHBAUD;                        //数据传输速率寄存器(高字节)
    Uint16       SCILBAUD;                        //数据传输速率寄存器(低字节)
    union SCICTL2_REG SCICTL2;                    //控制寄存器 2
    union SCIRXST_REG SCIRXST;                    //接收状态寄存器
    Uint16       SCIRXEMU;                        //接收仿真缓冲寄存器
    union SCIRXBUF_REG SCIRXBUF;                  //接收数据寄存器
    Uint16 rsvd1;                                 //保留
    Uint16       SCITXBUF;                        //发送数据缓冲寄存器
    union SCIFFTX_REG  SCIFFTX;                   //FIFO 发送寄存器
    union SCIFFRX_REG  SCIFFRX;                   //FIFO 接收寄存器
    union SCIFFCT_REG  SCIFFCT;                   //FIFO 控制寄存器
    Uint16 rsvd2;                                 //保留
    Uint16 rsvd3;                                 //保留
    union SCIPRI_REG   SCIPRI;                    //FIFO 优先级控制寄存器
};
extern volatile struct SCI_REGS SciaRegs;
extern volatile struct SCI_REGS ScibRegs;
extern volatile struct SCI_REGS ScicRegs;
```

在例 3-4 所示的 SCI 寄存器结构体 SCI_REGS 中，有的成员是 union 形式的，有的是 Uint16 形式的，定义为 union 形式的成员既可以实现对寄存器的整体操作，也可以实现对寄存器进行位操作，而定义为 Uint16 的成员只能直接对寄存器进行操作。

在 3.1.1 节中提到过，无论是 SCI-A、SCI-B，还是 SCI-C，在其寄存器存储空间中，有 3 个存储单元是被保留的，在对 SCI 的寄存器进行结构体定义时，也要将其保留。如例 3-4 所示，保留的寄存器空间采用变量来代替，但是该变量不会被调用，如 rsvd1、rsvd2、rsvd3。

在定义了结构体 SCI_REGS 之后，需要声明 SCI_REGS 型的变量 SciaRegs、ScibRegs、ScicRegs，分别用于代表 SCI-A 的寄存器、SCI-B 的寄存器和 SCI-C 的寄存器。关键字 extern 的意思是“外部的”，表明这个变量在外部文件中被调用，是一个全局变量。关键字 volatile 的意思是“易变的”，使得寄存器的值能够被外部代码任意改变，例如可以被外部硬件或者中断任意改变，如果不使用关键字 volatile，则寄存器的值只能被程序代码所改变。

前面是以 SCICCR 为例来介绍如何使用位定义的方式表示某个寄存器，又以 SCI 模块为例来讲解如何用结构体文件来表示一个外设模块的所有寄存器。如果根据前面的介绍，将 SCI 所有的寄存器用位定义的方式来表示，然后根据需要来定义共同体，最后定义寄存器结构体文件，可以发现，原来这就是 F28335 的头文件 DSP2833x_Sci.h 的内容。因为 F28335 的寄存器结构是固定的，所以，系统的头文件可以拿现成的来使用，一般情况下不需要再做修改了。

当如例 3-4 所示，定义了结构体 SCI_REGS 型的变量 SciaRegs、ScibRegs 和 ScicRegs 之后，就可以方便地实现对寄存器的操作了。下面以对 SCI-A 的寄存器 SCICCR 的操作为

例，介绍如何开发程序。

【例 3-5】 对 SCICCR 按位进行操作。

```
SciaRegs.SCICCR.bit.STOPBITS = 0;                    //1 位结束位
      SciaRegs.SCICCR.bit.PARITYENA = 0;             //禁止极性功能
      SciaRegs.SCICCR.bit.LOOPBKENA = 0;             //禁止回送测试模式功能
      SciaRegs.SCICCR.bit.ADDRIDLE_MODE = 0;         //空闲线模式
      SciaRegs.SCICCR.bit.SCICHAR = 7;               //8 位数据位
```

【例 3-6】 对 SCICCR 整体进行操作。

```
SciaRegs.SCICCR.all = 0x0007;
```

例 3-5 和例 3-6 的作用是一样的，都是对 SCI-A 的寄存器 SCICCR 进行初始化操作，只不过例 3-5 是对 SCICCR 按位进行操作的，而例 3-6 是对 SCICCR 整体进行操作的。还有一些寄存器，例如，SCIHBAUD 和 SCILBAUD 在结构体 SCI_REGS 的定义中是 Uint16 型的，那么如何对这类寄存器操作呢？如例 3-7 所示。

【例 3-7】 对 SCIHBAUD 和 SCILBAUD 进行操作。

```
SciaRegs.SCIHBAUD = 0;
SciaRegs.SCILBAUD = 0xF3;
```

由于 SCIHBAUD 和 SCILBAUD 定义时是 Uint16 型的，所以不能使用.all 或者.bit 的方式来访问了，只能直接给寄存器整体进行赋值。上面介绍的 3 种操作几乎涵盖了在 F28335 开发过程中对寄存器操作的所有方式，也就是说，掌握了这 3 种方式，就可以实现对 F28335 各种寄存器的操作了。

无论是 SCI-A、SCI-B，还是 SCI-C，都有好多寄存器，每个寄存器又都有若干位域，每个位域又都有自己的名字和功能，如此复杂的寄存器，是否要全部记住呢？否则如何写程序呢？答案显然是否定的。很多初学者把很多精力都花在了记忆寄存器上了，以至于看了后面就忘了前面的，到头来依然是一头雾水。

其实 CCS 为用户书写程序时提供了非常方便的功能，譬如书写语句 SciaRegs.SCICCR.bit.STOPBITS=0，先在 CCS 中输入 SciaRegs，然后输入“.”，就会弹出一个下拉列表框，将 SCI-A 模块下所有的寄存器列了出来，如图 3-3 所示。单击列表框中寄存器 SCICCR，便输入了寄存器 SCICCR。在这里一定要注意，必须输入 SciaRegs，每个字母的大小写都必须符合，否则是不会出现下拉列表框的。在输入 SCICCR 之后，继续输入成员操作符“.”，弹出新的下拉列表框，如图 3-4 所示。列表框中是共同体变量 SCICCR 的两个成员 all 或者 bit。如果要对寄存器进行整体操作，就单击 all，如果对寄存器进行位操作，就单击 bit。在这里，选择单击 bit，然后继续输入“.”，会弹出一个下拉列表框，里面列出了寄存器 SCICCR 的所有位域，也就是 bit 的所有成员，单击列表框中的 STOPBITS，便完成了输入，如图 3-5 所示。

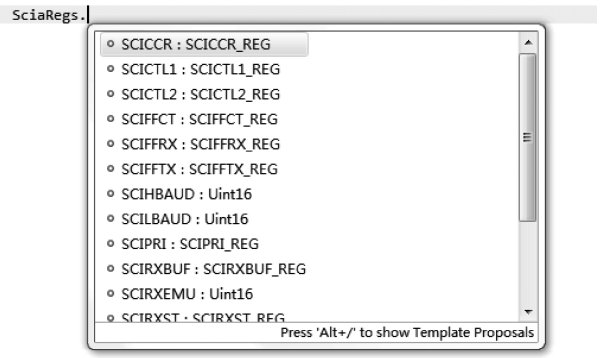

图 3-3 输入寄存器 SCICCR

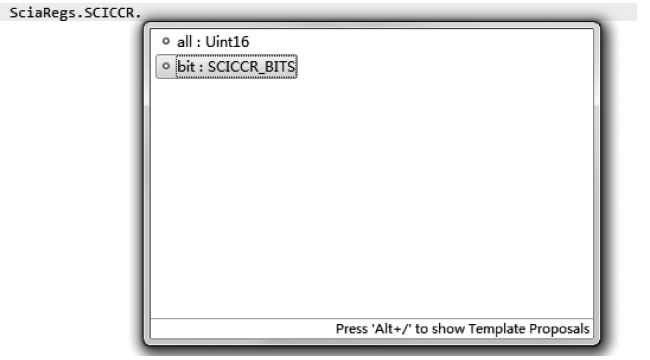

图 3-4 输入 bit

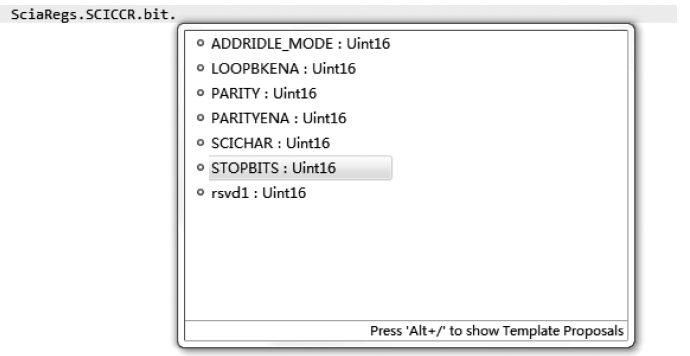

图 3-5 输入位域 STOPBITS

这是 CCS 的感应功能,很显然,使用感应功能的前提是工程加载了 F28335 的头文件,其下拉列表框中的内容都是头文件中所定义的结构体或者共同体的成员。因为 C 语言是区分大小写的,所以在最先手动输入外设寄存器名字的时候一定要注意字母的大小写,否则 CCS 也无法感应。能够对寄存器的位域进行提示和操作是使用位定义和寄存器结构体方

式访问寄存器最显著的优点。

如果选中 SciaRegs. SCICCR，便可以观察到寄存器 SCICCR 的每个位域的值，如图 3-6 所示，这也是使用位定义和寄存器结构体方式访问寄存器的优点，当然前提是程序已经执行了这条赋值语句。

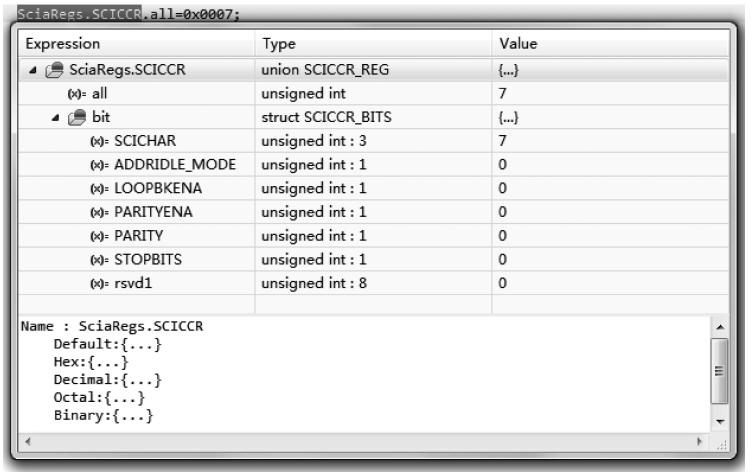

图 3-6　在 CCS 中观察 SCICCR

3.2　寄存器文件的空间分配

值得注意的是，之前所做的工作只是将 F28335 的寄存器按照 C 语言中位域定义和寄存器结构体的方式组织了数据结构，当编译时，编译器会把这些变量分配到存储空间中，但是很显然还有一个问题需要解决，就是如何将这些代表寄存器数据的变量同实际的物理寄存器结合起来呢？

这个工作需要两步来完成：第一步使用 DATA_SECTION 的方法将寄存器文件分配到数据空间中的某个数据段；第二步在 CMD 文件中，将这个数据段直接映射到这个外设寄存器所占的存储空间。通过这两步，就可以将寄存器文件同物理寄存器相结合起来了，下面详细讲解。

1. 使用 DATA_SECTION 方法将寄存器文件分配到数据空间

编译器产生可重新定位的数据和代码模块，这些模块就称为段。这些段可以根据不同的系统配置分配到相应的地址空间，各段的具体分配方式在 CMD 文件中定义。关于 CMD 文件，将在第 4 章中详细讲解。在采用硬件抽象层设计方法的情况下，变量可以采用“# pragma DATA_SECTION”命令分配到特殊的数据空间。在 C 语言中，“# pragma DATA_SECTION”的编程方式如下：

```
# pragma DATA_SECTION (symbol,"section name");
```

其中,symbol 是变量名,而 section name 是数据段名。下面以变量 SciaRegs 和 ScibRegs 为例,将这两个变量分配到名字为 SciaRegsFile 和 ScibRegsFile 的数据段。

【例 3-8】 将变量分配到数据段。

```
#pragma DATA_SECTION(SciaRegs,"SciaRegsFile");
volatile struct SCI_REGS SciaRegs;
#pragma DATA_SECTION(ScibRegs,"ScibRegsFile");
volatile struct SCI_REGS ScibRegs;
```

例 3-8 其实是 DSP2833x_GlobalVariableDefs.c 文件中的一段,其作用就是将 SciaRegs 和 ScibRegs 分配到名字为 SciaRegsFile 和 ScibRegsFile 的数据段。CMD 文件会将每个数据段直接映射到相应的存储空间中。表 3-1 说明了 SCI_A 寄存器映射到起始地址为 0x0000 7050 的存储空间。使用分配好的数据段,变量 SciaRegs 就会分配到起始地址为 0x0000 7050 的存储空间。那么如何将数据段映射到寄存器对应的存储空间呢?这就要研究一下 CMD 文件中的内容了。

2. 将数据段映射到寄存器对应的存储空间

【例 3-9】 将数据段映射到寄存器对应的存储空间。

```
/*************************************************************************
 * 存储器 SRAM.CMD 文件
 * 将 SCI 寄存器文件结构分配到相应的存储空间
 *************************************************************************/
 MEMORY
 {
PAGE 1 :
SCI_A     : origin = 0x007050, length = 0x000010
SCI_B     : origin = 0x007750, length = 0x000010
 }
 SECTIONS
 {
  SciaRegsFile     : > SCI_A,     PAGE = 1
  ScibRegsFile     : > SCI_B,     PAGE = 1
 }
```

从例 3-9 可以看到,首先在 MEMORY 部分,SCI_A 寄存器的物理地址从 0x007050 开始,长度为 16,SCI_B 寄存器的物理地址从 0x007750 开始,长度也为 16。然后在 SECTIONS 部分,数据段 SciaRegsFile 被映射到了 SCI_A,而 ScibRegsFile 被映射到了 SCI_B,实现了数据段映射到相应的存储器空间。

通过以上两部分的操作,才完成了将外设寄存器的文件映射到寄存器的物理地址空间上,这样才可以通过 C 语言来实现对 F28335 寄存器的操作。

本章带大家一步一步地实现了如何使用 C 语言对 F28335 的寄存器进行操作,也提到了存储器空间和 CMD 文件,但并没有细讲,在接下来的章节里,就要详细介绍 F28335 的存

储器结构、映像，以及如何编写 CMD 文件等内容。

习题

3-1　用位域的方式定义 SCI 的控制寄存器 SCICTL1，SCICTL1 的位如图 3-7 所示。

7	6	5	4	3	2	1	0
Reserved	RX ERR INT ENA	SW RESET	Reserved	TXWAKE	SLEEP	TXENA	RXENA
R-0	R/W-0	R/W-0	R-0	R/S-0	R/W-0	R/W-0	R/W-0

图 3-7　SCI 控制寄存器 SCICTL1

3-2　对 SCI-B 的控制寄存器 SCICTL1 整体赋值 0x003。

3-3　对 SCI-B 的控制寄存器 SCICTL1 按位赋值，RXENA 赋值为 1，TXENA 赋值为 1。