

**通过设置寄存器编写代码**

|  |  |
| --- | --- |
| **姓 名** | **冯浩然** |
| **学 号** | **202100800378** |
| **专 业** | **电子科学与技术** |
| **班 级** | **2021级1班** |
| **学 院** | **机电与信息工程学院** |
| **指导教师** | **王小利** |
| **课程名称** | **DSP原理与应用** |
| **课 程 号** | **28033160** |
| **学年学期** | **2023–2024学年第2学期** |
| **实验日期** | **2024年5月16日** |

# 一 完成SCI的接收、发送代码。

**1 初始化系统控制模块**

1. #include "DSP2833x\_Device.h"     *// Headerfile Include File*
2. #include "DSP2833x\_Examples.h"   *// Examples Include File*
3. void InitSysCtrl(void)
4. {
5. *// 禁用看门狗*
6. DisableDog();
7. *// 初始化PLL控制：PLLCR和DIVSEL*
8. *// DSP28\_PLLCR 和 DSP28\_DIVSEL 定义在 DSP2833x\_Examples.h 中*
9. InitPll(DSP28\_PLLCR, DSP28\_DIVSEL);
10. *// 初始化外设时钟*
11. InitPeripheralClocks();
12. }
13. void DisableDog(void) *//禁止看门狗*
14. {
15. EALLOW;
16. SysCtrlRegs.WDCR= 0x0068;
17. EDIS;
18. }
19. void InitPll(Uint16 val, Uint16 divsel)
20. {
21. *// 确保PLL没有在备用模式下运行*
22. if (SysCtrlRegs.PLLSTS.bit.MCLKSTS != 0)
23. {
24. *// 检测到外部时钟丢失*
25. *// 用适当的系统关闭函数替换这行代码*
26. asm("        ESTOP0");
27. }
28. *// 在PLLCR从0x0000更改之前，DIVSEL必须为0*
29. *// 它由外部复位XRSn设置为0*
30. *// 这使我们处于1/4*
31. if (SysCtrlRegs.PLLSTS.bit.DIVSEL != 0)
32. {
33. EALLOW;
34. SysCtrlRegs.PLLSTS.bit.DIVSEL = 0;
35. EDIS;
36. }
37. *// 更改PLLCR*
38. if (SysCtrlRegs.PLLCR.bit.DIV != val)
39. {
40. EALLOW;
41. *// 在设置PLLCR之前关闭缺失时钟检测逻辑*
42. SysCtrlRegs.PLLSTS.bit.MCLKOFF = 1;
43. SysCtrlRegs.PLLCR.bit.DIV = val;
44. EDIS;
45. *// 可选：等待PLL锁定*
46. *// 在此期间，CPU将切换到OSCCLK/2直到PLL稳定*
47. *// 一旦PLL稳定，CPU将切换到新的PLL值*
48. *//*
49. *// 这个锁定时间由PLL锁定计数器监控*
50. *//*
51. *// 代码不需要坐等PLL锁定*
52. *// 但是，如果代码有任何时间关键的操作*
53. *// 并且需要正确的时钟锁定，那么最好等待切换完成*
54. *// 等待PLL锁定位被设置*
55. *// 在这个循环之前应该禁用看门狗，或者在循环中通过ServiceDog()喂养*
56. *// 取消注释以禁用看门狗*
57. DisableDog();
58. while(SysCtrlRegs.PLLSTS.bit.PLLLOCKS != 1)
59. {
60. *// 取消注释以喂养看门狗*
61. *// ServiceDog();*
62. }
63. EALLOW;
64. SysCtrlRegs.PLLSTS.bit.MCLKOFF = 0;
65. EDIS;
66. }
67. *// 如果切换到1/2*
68. if((divsel == 1)||(divsel == 2))
69. {
70. EALLOW;
71. SysCtrlRegs.PLLSTS.bit.DIVSEL = divsel;
72. EDIS;
73. }
74. *// 注意：仅在PLL被旁路（即PLLCR = 0）或关闭时使用此设置*
75. *// 如果切换到1/1*
76. *// \* 首先切换到1/2并让电源稳定*
77. *//   所需时间将取决于系统，这只是一个示例*
78. *// \* 然后切换到1/1*
79. if (divsel == 3)
80. {
81. EALLOW;
82. SysCtrlRegs.PLLSTS.bit.DIVSEL = 2;
83. DELAY\_US(50L);  *// 延迟50微秒*
84. SysCtrlRegs.PLLSTS.bit.DIVSEL = 3;
85. EDIS;
86. }
87. }
88. void InitPeripheralClocks(void)
89. {
90. EALLOW;
91. *// HISPCP/LOSPCP 预分频寄存器设置，通常会设置为默认值*
92. SysCtrlRegs.HISPCP.all = 0x0001;
93. SysCtrlRegs.LOSPCP.all = 0x0002;
94. *// XCLKOUT 与 SYSCLKOUT 的比率。默认情况下 XCLKOUT = 1/4 SYSCLKOUT*
95. *// XTIMCLK = SYSCLKOUT/2*
96. XintfRegs.XINTCNF2.bit.XTIMCLK = 1;
97. *// XCLKOUT = XTIMCLK/2*
98. XintfRegs.XINTCNF2.bit.CLKMODE = 1;
99. *// 启用 XCLKOUT*
100. XintfRegs.XINTCNF2.bit.CLKOFF = 0;
101. *// 外设时钟使能设置为所选的外设。*
102. *// 如果不使用某个外设，请关闭其时钟以节省电力。*
103. *//*
104. SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;    *// ADC*
105. ADC\_cal();
106. SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 1;   *// I2C*
107. SysCtrlRegs.PCLKCR0.bit.SCIAENCLK = 1;   *// SCI-A*
108. SysCtrlRegs.PCLKCR0.bit.SCIBENCLK = 1;   *// SCI-B*
109. SysCtrlRegs.PCLKCR0.bit.SCICENCLK = 1;   *// SCI-C*
110. SysCtrlRegs.PCLKCR0.bit.SPIAENCLK = 1;   *// SPI-A*
111. SysCtrlRegs.PCLKCR0.bit.MCBSPAENCLK = 1; *// McBSP-A*
112. SysCtrlRegs.PCLKCR0.bit.MCBSPBENCLK = 1; *// McBSP-B*
113. SysCtrlRegs.PCLKCR0.bit.ECANAENCLK = 1;  *// eCAN-A*
114. SysCtrlRegs.PCLKCR0.bit.ECANBENCLK = 1;  *// eCAN-B*
115. SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;   *// 禁用 ePWM 中的 TBCLK*
116. SysCtrlRegs.PCLKCR1.bit.EPWM1ENCLK = 1;  *// 启用 ePWM1 时钟（添加此行以完成代码）*
118. EDIS;
119. }
     1. **初始化SCI引脚**
120. void InitSciaGpio()
121. {
122. EALLOW;
123. */\* 启用所选引脚的内部上拉 \*/*
124. *// 用户可以启用或禁用上拉。*
125. *// 这将启用指定引脚的上拉。*
126. GpioCtrlRegs.GPAPUD.bit.GPIO28 = 0;    *// 启用 GPIO28 (SCIRXDA) 的上拉*
127. GpioCtrlRegs.GPAPUD.bit.GPIO29 = 0;    *// 启用 GPIO29 (SCITXDA) 的上拉*
128. */\* 将所选引脚的限定设置为仅异步 \*/*
129. *// 输入默认情况下与 SYSCLKOUT 同步。*
130. *// 这将为所选引脚选择异步（无限定）。*
131. GpioCtrlRegs.GPAQSEL2.bit.GPIO28 = 3;  *// 异步输入 GPIO28 (SCIRXDA)*
132. */\* 使用 GPIO 寄存器配置 SCI-A 引脚 \*/*
133. GpioCtrlRegs.GPAMUX2.bit.GPIO28 = 1;   *// 配置 GPIO28 以进行 SCIRXDA 操作*
134. GpioCtrlRegs.GPAMUX2.bit.GPIO29 = 1;   *// 配置 GPIO29 以进行 SCITXDA 操作*
135. EDIS;
136. }

**3 初始化SCIB**

1. void scia\_echoback\_init()
2. {
3. *//  在 InitSysCtrl() 函数中已开启 SCIA 外设的时钟*
4. SciaRegs.SCICCR.all = 0x0007;   *// 1个停止位，无回环*
5. *// 无奇偶校验，8位字符，*
6. *// 异步模式，空闲线协议*
7. SciaRegs.SCICTL1.all = 0x0003;  *// 启用 TX, RX, 内部 SCICLK,*
8. *// 禁用 RX 错误, 睡眠, TX 唤醒*
9. SciaRegs.SCICTL2.all = 0x0003;
10. SciaRegs.SCICTL2.bit.TXINTENA = 1;  *// 启用 TX 中断*
11. SciaRegs.SCICTL2.bit.RXBKINTENA = 1; *// 启用 RX/BK 中断*
12. #if (CPU\_FRQ\_150MHZ)
13. SciaRegs.SCIHBAUD = 0x0001;  *// 9600 波特率 @LSPCLK = 37.5MHz*
14. SciaRegs.SCILBAUD = 0x00E7;
15. #endif
16. #if (CPU\_FRQ\_100MHZ)
17. SciaRegs.SCIHBAUD = 0x0001;  *// 9600 波特率 @LSPCLK = 20MHz*
18. SciaRegs.SCILBAUD = 0x0044;
19. #endif
20. SciaRegs.SCICTL1.all = 0x0023;  *// 从复位中释放 SCI*
21. }
22. void scia\_fifo\_init()                            *//SCIB FIFO 初始化*
23. {
24. SciaRegs.SCIFFTX.all=0xE040;
25. SciaRegs.SCIFFRX.all=0x204f;
26. SciaRegs.SCIFFCT.all=0x0;
27. }

**4 主程序函数**

1. #include "DSP28x\_Project.h"
2. void scia\_echoback\_init(void);
3. void scia\_fifo\_init(void);
4. void scia\_xmit(int a);
5. void scia\_msg(char \*msg);
6. Uint16 LoopCount;
7. Uint16 ErrorCount;
8. void main(void)
9. {
10. Uint16 ReceivedChar;
11. char \*msg;
12. *// 步骤 1. 初始化系统控制:*
13. *// PLL, 看门狗, 启用外围时钟*
14. InitSysCtrl();
15. *// 步骤 2. 初始化 GPIO:*
16. *//仅初始化 SCI-A 端口的引脚。*
17. InitSciaGpio();
18. *// 步骤 3. 清除所有中断并初始化 PIE 向量表:*
19. *// 禁用 CPU 中断*
20. DINT;
21. *// 将 PIE 控制寄存器初始化为默认状态。*
22. *// 默认状态是禁用所有 PIE 中断并清除标志。*
23. InitPieCtrl();
24. *// 禁用 CPU 中断并清除所有 CPU 中断标志:*
25. IER = 0x0000;
26. IFR = 0x0000;
27. *// 使用指向壳中断服务例程 (ISR) 的指针初始化 PIE 向量表。*
28. InitPieVectTable();
29. *// 步骤 4. 初始化所有设备外围设备:*
30. scia\_echoback\_init(); *// 初始化 SCI-A 以进行回显*
31. *// 主程序循环*
32. while(1)
33. {
34. *// 等待接收字符*
35. while(SciaRegs.SCIRXST.bit.RXRDY == 0) { }
36. *// 读取接收到的字符*
37. ReceivedChar = SciaRegs.SCIRXBUF.all;
38. *// 回显接收到的字符*
39. while(SciaRegs.SCICTL2.bit.TXRDY == 0) { }
40. SciaRegs.SCITXBUF = ReceivedChar;
41. }
42. }
43. *// 步骤 5. 用户特定代码:*
44. LoopCount = 0;
45. ErrorCount = 0;
46. scia\_fifo\_init();     *// 初始化 SCI FIFO*
47. scia\_echoback\_init(); *// 初始化 SCI 以进行回显*
48. msg = "\r\n\n\nHello World!\0";
49. scia\_msg(msg);
50. msg = "\r\nYou will enter a character, and the DSP will echo it back! \n\0";
51. scia\_msg(msg);
52. for(;;)
53. {
54. msg = "\r\nEnter a character: \0";
55. scia\_msg(msg);
56. *// 等待输入字符*
57. while(SciaRegs.SCIFFRX.bit.RXFFST != 1) { } *// 等待 XRDY = 1 表示空状态*
58. *// 获取字符*
59. ReceivedChar = SciaRegs.SCIRXBUF.all;
60. *// 回显字符*
61. msg = "  You sent: \0";
62. scia\_msg(msg);
63. scia\_xmit(ReceivedChar);
64. LoopCount++;
65. }
66. }
67. *// 从 SCI 发送一个字符*
68. void scia\_xmit(int a)
69. {
70. while (SciaRegs.SCIFFTX.bit.TXFFST != 0) {}
71. SciaRegs.SCITXBUF = a;
72. }
73. *// 发送一个字符串消息*
74. void scia\_msg(char \*msg)
75. {
76. int i;
77. i = 0;
78. while(msg[i] != '\0')
79. {
80. scia\_xmit(msg[i]);
81. i++;
82. }
83. }

# 二 C语言结构体、联合知识

**结构体（struct）**

结构体是一种用户定义的数据类型，它允许你将不同类型的数据组合成一个单一的复合类型。在DSP编程中，结构体常用于表示复杂的数据结构，如控制参数、滤波器系数、信号样本等。

示例：

1. typedef struct {
2. float a;
3. float b;
4. float c;
5. int16\_t d;
6. } FilterCoefficients;
7. FilterCoefficients myFilter;
8. myFilter.a = 1.0f;
9. myFilter.b = 0.5f;
10. myFilter.c = 0.25f;
11. myFilter.d = 100;

在上面的示例中，我们定义了一个名为`FilterCoefficients`的结构体，它包含三个浮点数和一个16位整数。然后，我们创建了一个该类型的变量`myFilter`，并为其成员赋值。

**联合（union）**

联合也是一种用户定义的数据类型，但它与结构体不同。在联合中，所有的成员都共享同一块内存空间。这意味着在任何时候，只有一个成员是有效的。联合通常用于节省内存空间，或者在需要以不同方式解释同一内存块时使用。

示例：

1. typedef union {
2. struct {
3. uint16\_t high;
4. uint16\_t low;
5. } words;
6. uint32\_t dword;
7. } WordUnion;
8. WordUnion myUnion;
9. myUnion.dword = 0x12345678;
10. printf("High word: 0x%04X\n", myUnion.words.high); *// 输出: High word: 0x5678*
11. printf("Low word: 0x%04X\n", myUnion.words.low);  *// 输出: Low word: 0x1234*

在上面的示例中，我们定义了一个名为`WordUnion`的联合，它包含一个匿名结构体和一个32位无符号整数。这个联合允许我们以两种不同的方式访问和解释同一个32位值：作为两个16位的“字”，或作为一个完整的32位“双字”。

# 三 C语言位域知识

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;**

**};**

有如下规则：

* 1. 位域的定义必须按从右往左的顺，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
  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. 位域可以无谓域名，这时他只用作填充或调整位置，无名的位域是不能够使用的。

示例：

1. struct bs  //定义位域bs
2. {
3. int a:4;
4. int :0; *//空域*
5. int b:5; *//从第二个字节开始存放*
6. int :3; *//这三位不能用*
7. int c:5;
8. int d:3;
9. };
10. struct bs bs1; *//声明bs型变量bs1*

# 四 对寄存器进行部分位操作、全部位操作的知识。

**寄存器定义**

首先，我们需要定义一个寄存器的结构体和联合，以便能够方便地访问寄存器的各个位域。假设我们有一个16位的寄存器，其中包含多个不同的位域。

1. typedef union {
2. struct {
3. uint16\_t BIT0 : 1;
4. uint16\_t BIT1 : 1;
5. uint16\_t BIT2 : 1;
6. uint16\_t BIT3 : 1;
7. uint16\_t BIT4 : 1;
8. uint16\_t BIT5 : 1;
9. uint16\_t BIT6 : 1;
10. uint16\_t BIT7 : 1;
11. uint16\_t BIT8 : 1;
12. uint16\_t BIT9 : 1;
13. uint16\_t BIT10 : 1;
14. uint16\_t BIT11 : 1;
15. uint16\_t BIT12 : 1;
16. uint16\_t BIT13 : 1;
17. uint16\_t BIT14 : 1;
18. uint16\_t BIT15 : 1;
19. } bits;
20. uint16\_t all;
21. } MY\_REGISTER;
22. volatile MY\_REGISTER \*myRegister = (volatile MY\_REGISTER \*)0x0000; *// 假设寄存器地址为0x0000*

**部分位操作**

部分位操作是指只修改寄存器中的某些特定位，而不影响其他位。通过使用结构体中的位域，可以方便地进行这种操作。

示例：

1. *// 设置BIT0和BIT1为1*
2. myRegister->bits.BIT0 = 1;
3. myRegister->bits.BIT1 = 1;
4. *// 清除BIT2和BIT3*
5. myRegister->bits.BIT2 = 0;
6. myRegister->bits.BIT3 = 0;
7. *// 读取BIT4的值*
8. uint16\_t bit4\_value = myRegister->bits.BIT4;

**全部位操作**

全部位操作是指一次性读取或写入整个寄存器的值。这通常用于初始化或重置寄存器。

示例：

1. *// 将寄存器的所有位设置为0xFFFF*
2. myRegister->all = 0xFFFF;
3. *// 将寄存器的所有位清零*
4. myRegister->all = 0x0000;
5. *// 读取寄存器的所有位*
6. uint16\_t reg\_value = myRegister->all;

**结合使用**

在实际应用中，部分位操作和全部位操作通常结合使用。例如，你可能需要先读取寄存器的当前值，然后只修改其中的某些位，最后将修改后的值写回寄存器。

示例：

1. *// 读取当前寄存器值*
2. uint16\_t current\_value = myRegister->all;
3. *// 修改BIT5和BIT6的值*
4. current\_value |= (1 << 5);  *// 设置BIT5*
5. current\_value &= ~(1 << 6); *// 清除BIT6*
6. *// 将修改后的值写回寄存器*
7. myRegister->all = current\_value;

通过这种方式，我们可以灵活地对寄存器进行部分位操作和全部位操作，从而实现对硬件的精确控制。