F28335调试笔记

编制： 于海

时间：2017.12.15

版本：

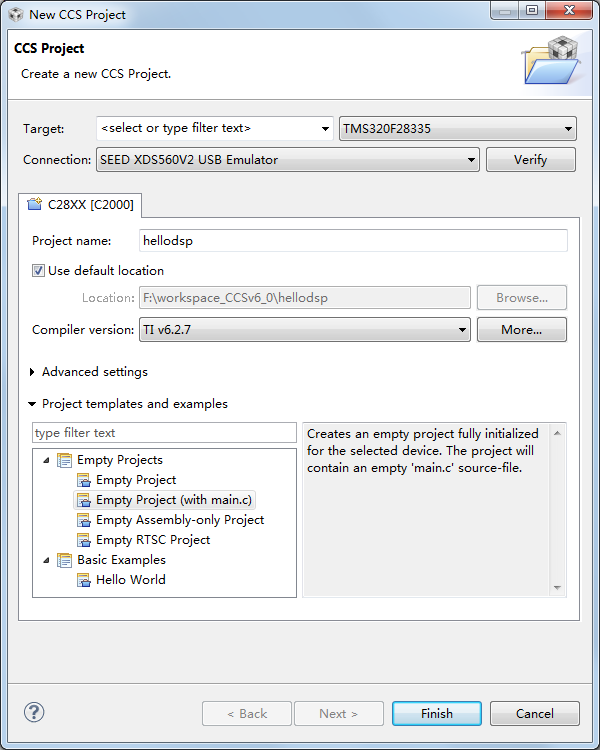
审批：

# **一、安装环境搭建**

*分项描述测试此板所有功能的测试用例：*

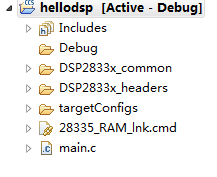
## 1、安装环境搭建：

首先下载CCSV6软件以及艾睿合众的仿真器驱动，SEED-XDS560V2，之后打开CCS软件建立CCS Project，按照下面选择：

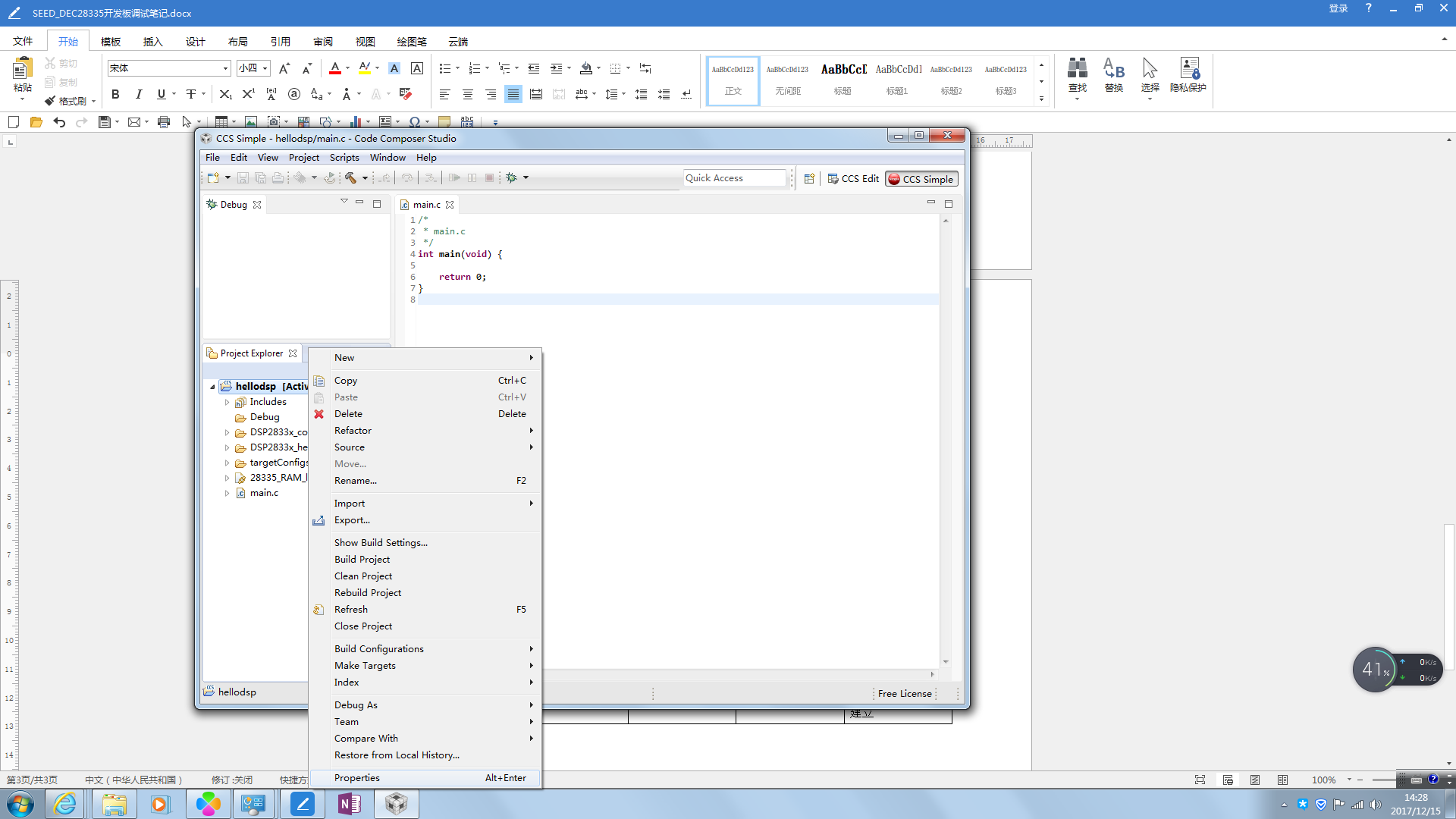


我使用的是seed-xds560v2仿真器，装好驱动后，电脑的设备管理器已经识别了，打开ccsv6后， 进行target configuration，connection一栏没有出现seed-xds560 USB v2，然后选择其它xds560 usb v2连接不上，后来将seed-xds560v2仿真器重新安装了一遍，路径安装在ccsv6/ccs\_base/目录下，之后必须重启CCS，就能发现SEED仿真器。原来是仿真器的安装路径安装不对，不能安装默认的路径安装。

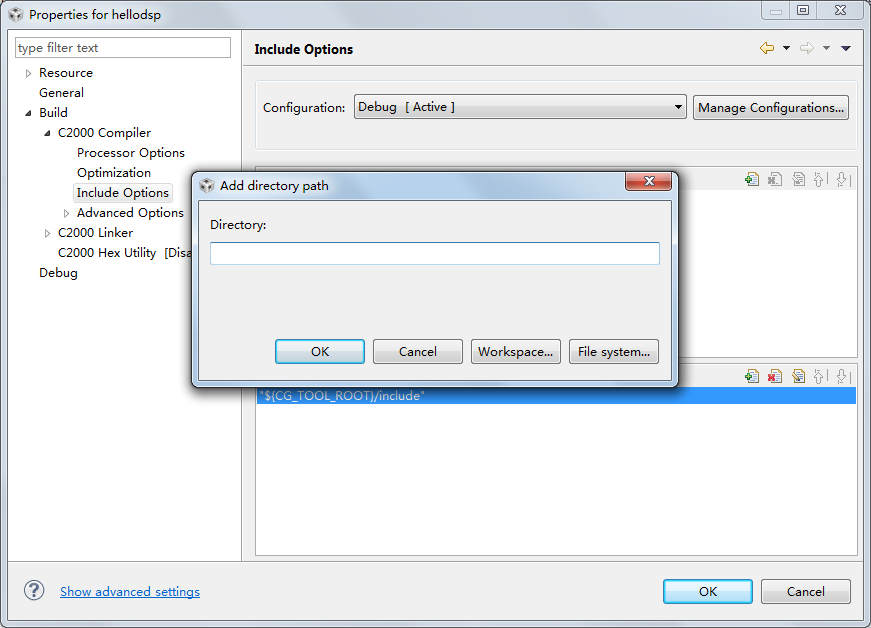
之后将这两个文件夹拷贝到工程目录下，这时工程会自动添加这两个文件夹，



这两个文件夹为TI官网上下载的28335所需的cmd、lib、c以及h文件，之后将这个文件夹的include文件夹添加到工程路径。



右键工程名-“properties”-“Include Optitions”之后点击右下方“绿色加号”在弹出界面中点击“workspaces”添加includes文件夹路径，



对于如何修改工程名？

CCS识别的不是工程文件文件夹的名字，而是具体的工程文件.project，而这个文件名不可改，想要文件名，你可以在CCS的工程列表中选择你的工程，右键->copy，然后在工程列表的空白处右键->paste，在弹出的Copy project对话框中填写你的新工程名和保存路径就可以了。

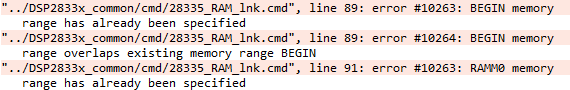
# **二、裸机程序移植常见错误**

到“DSP2833x\_examples”文件夹下找一个DAC的例程，调试出现如下错误：



错误1

DSP2833x\_Headers\_nonBIOS.cmd这个cmd文件出现错误;



错误2

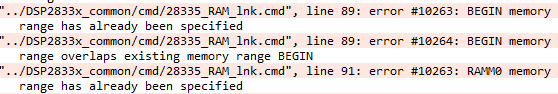
28335\_RAM\_lnk.cmd这个cmd文件出现错误;

百度一下错误，发现是cmd文件出现重复定义造成的，

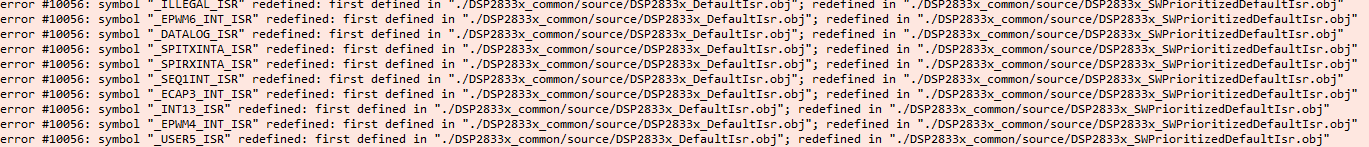
在DSP28335工程文件里（不用BIOS产生CMD文件），手写CMD文件一般有两个，在RAM里调试时用的两个CMD文件分别为DSP2833x\_Headers\_nonBIOS.cmd和28335\_RAM\_lnk.cmd，烧写到flash里时用的两个CMD文件分别为DSP2833x\_Headers\_nonBIOS.cmd和F28335.cmd，其中DSP2833x\_Headers\_nonBIOS.cmd文件可以在所有工程文件中通用，主要作用是把外设寄存器产生的数据段映射到对应的存储空间，可以跟DSP2833x\_GlobalVariableDefs.c文件对照一下看看。

如果修改就参考datasheet中F28335的memory map来修改CMD文件；

因此删除F28335.cmd和DSP2833x\_Headers\_BIOS.cmd这两个文件。之后编译又出现下面一些错误和警告：



错误3



错误4

错误4发现是DSP2833x\_common/source文件夹下的c文件存在变量重复定义的情况；

DSP2833x\_SWPrioritizedDefaultIsr.c与DSP2833x\_DefaultIsr.c这两个源文件不能同时使用；DSP2833x\_PieVect.c与DSP2833x\_SWPiroritizedPieVect.c这两个源文件也不能同时使用

DSP2833x\_DefaultIsr.c和DSP2833x\_PieVect.c只是使用了硬件优先级;

DSP2833x\_SWPrioritizedDefaultIsr.c

和DSP2833x\_SWPiroritizedPieVect.c还可以应用于软件优先级设置。

我这次使用DSP2833x\_DefaultIsr.c和DSP2833x\_PieVect.c这种硬件优先级；

DSP281x\_PieVect.c文件中包含初始化PIE中断(peripheral interrupt expansion)向量表的初始化子程序，给每一个PIE级中断向量初始化一个值，指向一个空的中断服务程序；DSP281x\_PieVect.h文件中定义与中断向量有关结构体类型DSP281x\_DefaultIsr.c中包含的是一些空的中断服务程序，这些空的中断服务

程序可以用来进行中断测试。用户也可以将自己的代码放在这些子程序中，进

行中断操作，还可以编写新的中断服务程序再重载该终端的中断向量。

在工程中删除DSP2833x\_SWPrioritizedDefaultIsr.c和DSP2833x\_SWPiroritizedPieVect.c这两个源文件错误4就没了。

下面排查错误3。

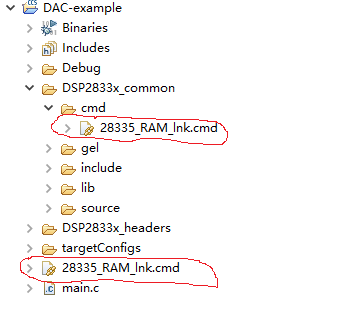
我将main.c程序注销还是出现错误3，

终于找到错误3的原因了：这个错误由两个错误组成：error #10263: BEGIN memory range has already been specified error #10264: BEGIN memory range overlaps existing memory range BEGIN

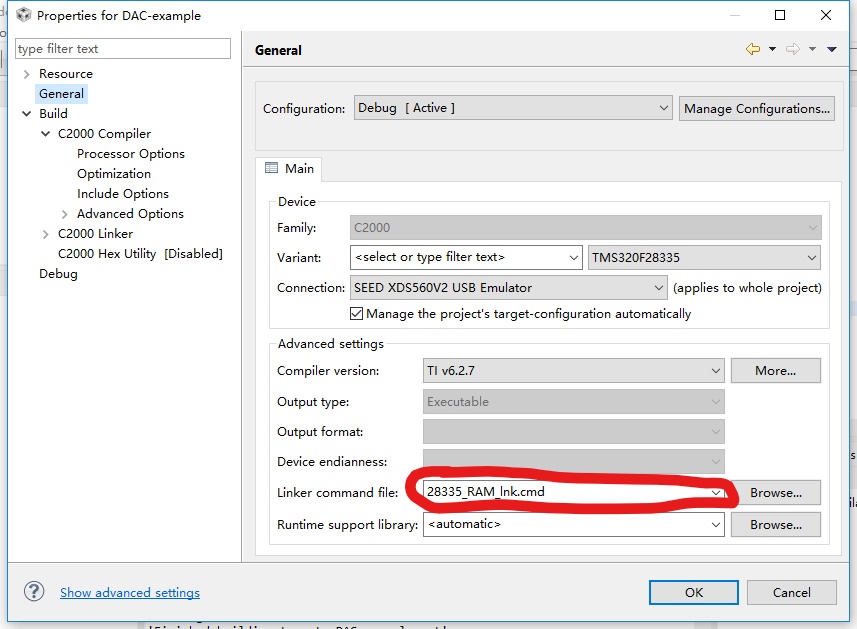
error #10263:是存储空间重复定义，肯定有连个cmd文件重复定义了一个存储区域了

error #10264:是存储空间不足，这是由error #10263引起的，解决了error #10263也就解决了error #10264.

后来发现CCS默认加载了28335\_RAM\_lnk.cmd文件，如下图所示：



工程中的D:\workspaceCCSv6\_0\DAC-example\DSP2833x\_common\cmd文件夹已经加载了28335\_RAM\_lnk.cmd文件，但是系统默认加载了28335\_RAM\_lnk.cmd文件，造成了重复加载。右键工程名，选择“properties”-“General”下面界面可以看到链接的cmd文件：



因此删除掉DSP2833x\_common\cmd文件夹，因为系统加载cmd文可以方便更改cmd文件，系统调试用28335\_RAM\_lnk.cmd文件，往flash下载程序需要F28335.cmd文件。

<http://www.docin.com/p-189398655.html>

# **三、FLYCON飞控板软件调试**

## 1、通用GPIO口调试

飞控板的GPIO口主要用于LED、蜂鸣器以及DS18B20驱动。

F28335共有88个GPIO口，分为A\B\C三组，A组包括GPIO0~GPIO31,B组包括GPIO32~GPIO63,C组包括GPIO64~GPIO87。每个引脚都复用了多个功能，但同一时刻每个引脚只能用其中一个功能。

下面的程序是将GPIO配置为输出：

void DQ\_OUT(void) // GPIO85配置为输出口

{

EALLOW;

GpioCtrlRegs.GPCMUX2.bit.GPIO85 = 0; // GPIO85通用的IO

GpioCtrlRegs.GPCDIR.bit.GPIO85 = 1; // GPIO85配置为输出口

EDIS;

}

下面的程序是将GPIO配置为带上拉输入：

void DQ\_IN(void) // GPIO85配置为输入口

{

EALLOW;

GpioCtrlRegs.GPCMUX2.bit.GPIO85 = 0; // GPIO85通用的IO

GpioCtrlRegs.GPCDIR.bit.GPIO85 = 0; //配置为输入

GpioCtrlRegs.GPCPUD.bit.GPIO85 = 0; //开启上拉

EDIS;

}

可以通过下面的四种方式对IO口进行操作：

1）可以通过GPXDAT寄存器独立读/写I/O信号；

2）利用GPXSET寄存器写1（写0无效）对I/O口置位；

3）利用GPXCLEAR寄存器写1（写0无效）对I/O口清零；

3）利用GPXTOGGLE寄存器写1（写0无效）对I/O口翻转；

## 2、Mcbsp\_SPI驱动程序调试

开始连接16488发现程序无法下载，发现可能是复位引脚处于低电平状态，导致DSP处于复位状态。后将16488的reset引脚断开就可以正常下载程序了。

ADIS16488惯性传感器采用SPI通信，F28335只有1路SPI，还可以利用Mcbspa和Mcbspb配置成两路SPI。

ADIS16488使用Mcbspb\_SPI进行通信的设置流程：

1、首先利用InitMcbspbGpio();函数将GPIO配置为Mcbspb的通信引脚；

2、利用init\_mcbspb\_spi()函数将Mcbspb的寄存器配置为SPI所需的值；

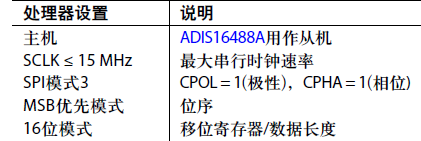
3、利用ADIS16488\_init();函数将ADIS16488初始化；

4、利用void Mcbspa\_SPI\_TX(Uint16 a)；和Uint16 Mcbspa\_SPI\_RX(void)对ADIS16488的寄存器进行写和读操作。

这其中比较容易出的问题如下所示：

### 1）主机SPI的通信配置；

ADIS16488要求SPI的通信配置如下图所示：



通用主机SPI配置

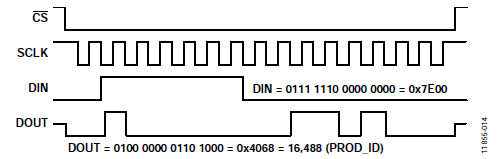
**SPI配置如下所示：**

**McbspbRegs.SPCR1.bit.CLKSTP = 2; // 开启时钟停止模式无延时**

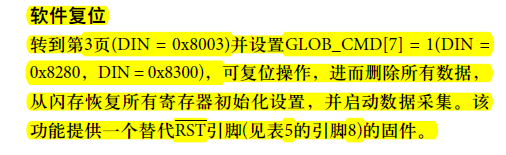
**McbspbRegs.PCR.bit.CLKXP = 1; // CPOL = 1, CPHA = 1 falling edge**

之前将McbspbRegs.SPCR1.bit.CLKSTP配置为3带延时的模式，通过示波器看时序不对，第一个上升沿开始输入，第二个下降沿输出。





### 2）初始化ADIS16488



程序如下所示：

**void ADIS16488\_init(void)**

**{**

**Mcbspb\_SPI\_TX(0x8003); // 转到第3页**

**Mcbspb\_SPI\_TX(0x8280); // 软件复位配置**

**Mcbspb\_SPI\_TX(0x8300);**

**Mcbspb\_SPI\_TX(0x8000); // 转到第0页 读取数据**

**DELAY\_US(1000);**

**}**

### 3）读取数据

因为写入地址之后读取的数据是上一次的数值，只有读取之后才能

Mcbspb\_SPI\_TX(ADIS16488\_REG\_PROD\_ID);

DELAY\_US(1000);

Mcbspb\_SPI\_TX(ADIS16488\_REG\_X\_GYRO\_OUT);

PROD\_ID = Mcbspb\_SPI\_RX();

DELAY\_US(1000);

2018.03.05发现ADIS16488突然调试不通，测试很多遍输出一直是0xffff。

2018.03.07将reset引脚连接上程序也可以正常下载，说明16488没有复位成功。

拿了一块ADIS16405模块，开始测试输出也始终是0xffff，后来怀疑可能是模块休眠，在网上的ADI的论坛里面也有提出ADIS16488输出数据不稳定，有时候读数据不通，放几天调试又有数据了。

看资料发现如果模块休眠需要将CS引脚，置高之后再置低，开始在

void InitMcbspbGpio(void)函数里实现发现，输出没变化还是0xffff，后来在主函数中循环实现如下所示：

while(1)

{

GpioDataRegs.GPACLEAR.bit.GPIO27 = 1; // iSensor\_CS拉低

DELAY\_US(1000);

Mcbspb\_SPI\_TX(0x5600);

DELAY\_US(1000);

PROD\_ID = Mcbspb\_SPI\_RX();

GpioDataRegs.GPASET.bit.GPIO27 = 1; // iSensor\_CS拉高

DELAY\_US(1000);

scia\_xmit16(PROD\_ID);

}

}

这个时候串口输出0x4015也就是16405，这个ID。

之后将

GpioDataRegs.GPACLEAR.bit.GPIO27 = 1; // iSensor\_CS拉低

GpioDataRegs.GPASET.bit.GPIO27 = 1; // iSensor\_CS拉高

这两个配置都注释掉也能够正常输出，说明模块正常启动后就不需要循环将CS拉高和置低了。

但是上面的程序对于ADIS16488不好用。

## 3、GPS SP驱动程序调试

ublox M8N使用SPI进行通信调试，注意如下问题：

**SPI初始化之前必须将/CS片选端置高，初始化之后再将/CS片选端拉低。**

**为了使接收器上电的时候能正确地读取配置（因为SPI的MOSI、MISO、SCK启动的时候用作配置脚），必须保证在启动的时候使SS\_N为高。之后，SPI功能不会影响配置脚。**

**如果不按照上述配置，SPI读取的数据为乱码。**

1,01,02,u-blox AG - www.u-blox.com\*4E

$GNTXT,01,01,02,HW UBX-M8030 00080000\*60

$GNTXT,01,01,02,ROM CORE 3.01 (107888)\*2B

$GNTXT,01,01,02,FWVER=SPG 3.01\*46

$GNTXT,01,01,02,PROTVER=18.00\*11

$GNTXT,01,01,02,GPS;GLO;GAL;BDS\*77

$GNTXT,01,01,02,SBAS;IMES;QZSS\*49

$GNTXT,01,01,02,GNSS OTP=GPS;GLO\*37

$GNTXT,01,01,02,LLC=FFFFFFFF-FFFFFFFF-FFFFFFFF-FFFFFFFF-FFFFFFFD\*2F

$GNTXT,01,01,02,ANTSUPERV=AC SD PDoS SR\*3E

$GNTXT,01,01,02,ANTSTATUS=DONTKNOW\*2D

$GNTXT,01,01,02,PF=3FF\*4B

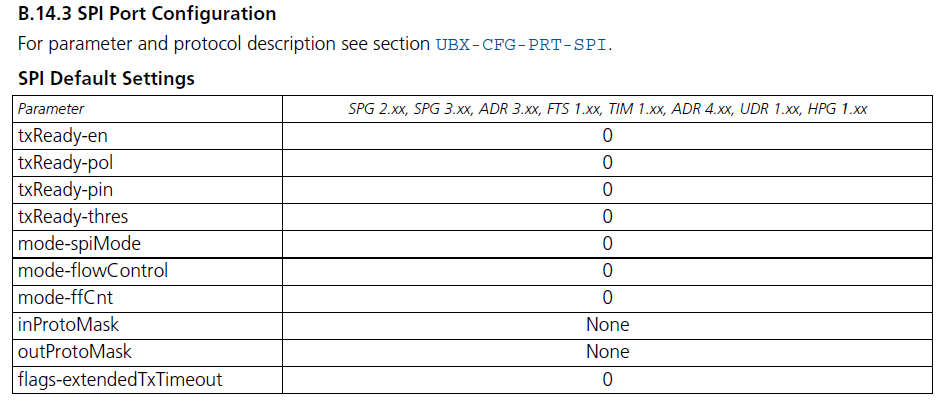
上面是重新上电后才有的数据。

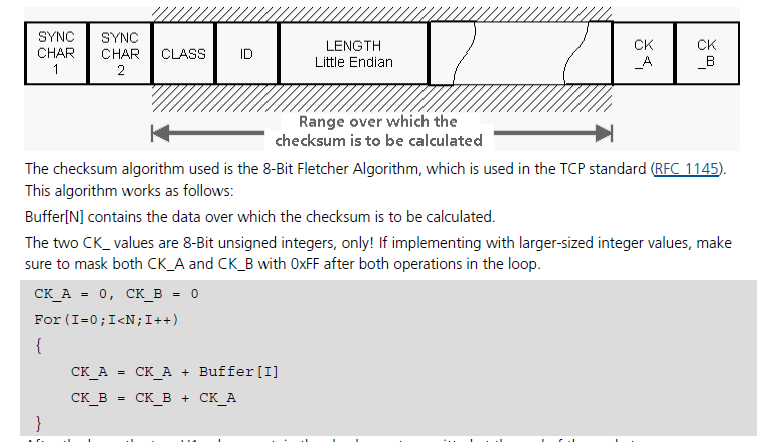
SpiaRegs.SPIPRI.bit.FREE = 0; // 标准SPI模式

这个位决定了仿真暂停时，SPI将继续执行或立即停止还是当前操作完成后停止，有时候重新上电没有数据需要更改这个位的值。

下面这篇博客写的很细：

http://blog.csdn.net/shanghaiqianlun/article/details/7253520





8位校验和计算程序如下所示：

**void CheckSUM(void)**

**{**

**uchar SPIPORT\_command[24] = {0x06,0x00,0x14,0x00,0x04,0x00,0x00,0x02,0x00,0x00,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};**

**Uint16 CK\_A = 0;**

**Uint16 CK\_B = 0;**

**Uint16 i;**

**for(i=0; i<24; i++)**

**{**

**CK\_A = CK\_A+SPIPORT\_command[i];**

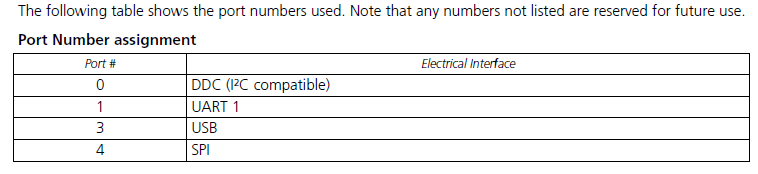
**CK\_B = CK\_B+CK\_A;**

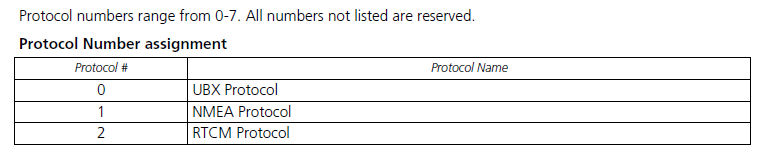
**}**

**CK\_A = CK\_A & 0xff;**

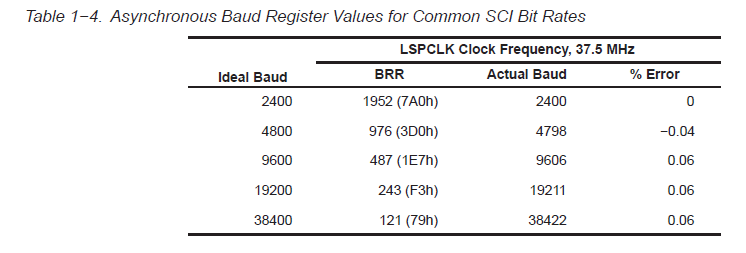
**CK\_B = CK\_B & 0xff;**

**}**





## 4、SCI驱动程序调试





BRR为16位寄存器值

## 5、EPWM驱动程序调试

ePWM模块的7个模块就像一条生产线，一级一级的经过，但DSP更高级，可以实现通过配置，使得ePWM只经过我选择的生产线，没有被选择上的就不要经过。例如，死区控制和斩波模块可以需要也可以不需要，这就看实际系统需不需要了。在实际使用ePWM时，正常的发出PWM波往往只要要配置TB、CC、AQ、DB、ET五个模块。

网上找到的例程是时间基准子模块配置程序，程序移植到工程出现下面的错误：

"../28335\_RAM\_lnk.cmd", line 125: error #10099-D: program will not fit into available memory. placement with alignment/blocking fails for section ".text" size 0x1034 page 0. Available memory ranges:

需要将RAML1 的长度修改为大于0x1034，因此将其改为0x001100；

修改后如下所示：

RAML0 : origin = 0x008000, length = 0x001000

RAML1 : origin = 0x009000, length = 0x001100

修改为：

RAML0 : origin = 0x008000, length = 0x000F00

RAML1 : origin = 0x008F00, length = 0x001100

修改后没有上面的错误了。

由于网上的例程只是单独子模块的程序，不能产生PWM，后来找到艾睿合众的例程中Example\_EpwmSetup.c这个文件里面EPwmSetup()函数含有epwm1产生PWM波的完整配置。

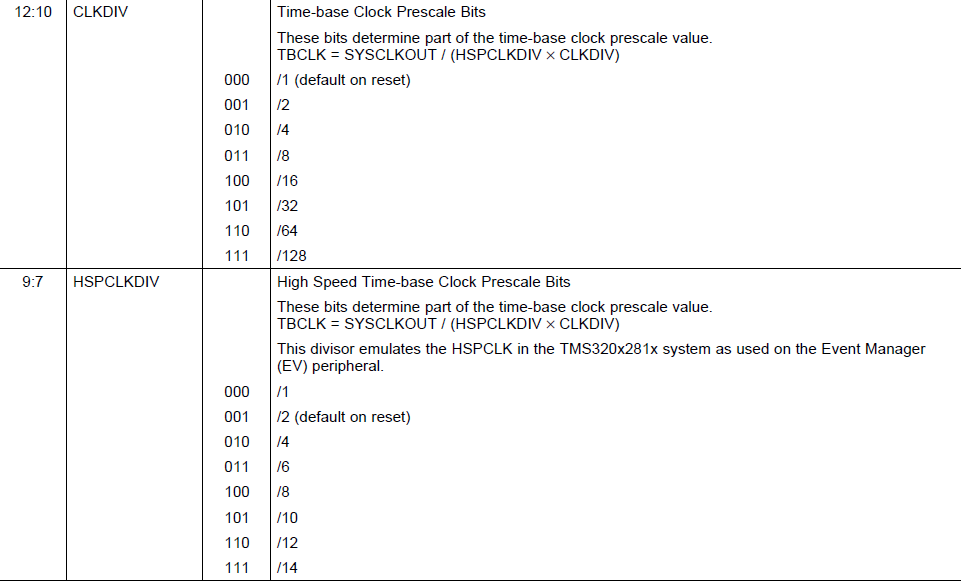
EPwm1Regs.TBPRD=SP;用于设置PWM的频率；

TBPRD是16位的寄存器，最大值是65535，如果所需频率较低，需要将TBLCK分频，

EPwm1Regs.TBCTL.bit.HSPCLKDIV=1; // 2分频

EPwm1Regs.TBCTL.bit.CLKDIV=6; // 64分频

TBCLK = SYSCLKOUT/(HSPCLKDIV\*CLKDIV);



**注意：修改某个EPWMx的时间基准，必须将前面的时间基准修改，因为epwm时间基准同步信号是由前面的epwm模块产生的，除了EPWM1。**

**因为EPWM模块共有六组，每一组包含两路EPWMxA和EPWMxB，其中EPWMxA和EPWMxB的频率是同时修改的，占空比可以互补输出，好像也可以单独修改，现在还没实现。**

上面一条说的是修改时间基准的方法，下面是修改EPWM周期与占空比的方法：

1）频率配置

EPwm1Regs.TBPRD=SQ;

上面的程序是修改频率配置，

#define PWM\_CLK 50 // PWM频率设置50Hz

#define SQ TBCLK/(2\*PWM\_CLK)

TBCLK = CPU\_CLK/(HSPCLKDIV\*CLKDIV);

其中由于CPU\_CLK是150MHz，PWM\_CLK是你要设置的PWM的频率值，HSPCLKDIV和CLKDIV是高速时基时钟分频位和时基时钟分频位，可以设置。

2）占空比配置

EPwm1Regs.CMPA.half.CMPA =SQ/2; // 占空比50%

## 6、SBUS功能调试

一、协议说明：

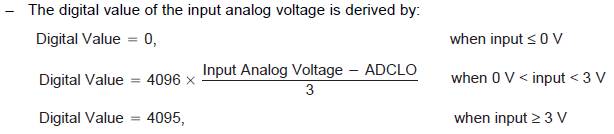
串口配置为波特率100kbps（ 100000 ），8位数据，偶校验(even)，2位停止位，无流控。

6、ADC功能调试

BATT6S\_VOLT = (((AdcRegs.ADCRESULT0)>>4)\*3.0)/4096;

BATT6S\_VOLT 为float型，如果把3.0改为3，BATT6S\_VOLT 为int型。

模拟输入转换为数字输出公式如下所示：



ADCLO为参考输入负端，通常接地；3为内部参考电压值。

测试发现AD采样值始终为4095，转换为实际输出一直是3V，实际输入为0，检查发现，DSP的ADCREFP与AGND短路，未发现是什么原因。

6、eCAN功能调试

ECanaRegs不支持位操作，TI的例程中将ECanaRegs赋给ECanaShadow，对影子寄存器进行位操作，提高代码的可读性。

ECanbShadow.CANMC.all = ECanbRegs.CANMC.all;

ECanbShadow.CANMC.bit.STM = 1; // 自测模式

// ECanbShadow.CANMC.bit.STM = 0; // 正常模式

ECanbShadow.CANMC.bit.SCB = 1; // eCAN mode (reqd to access 32 mailboxes)

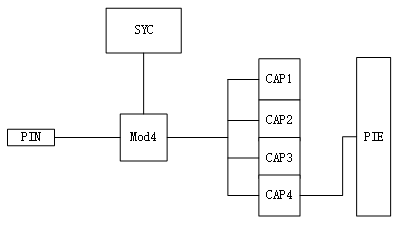
ECanbRegs.CANMC.all = ECanbShadow.CANMC.all;

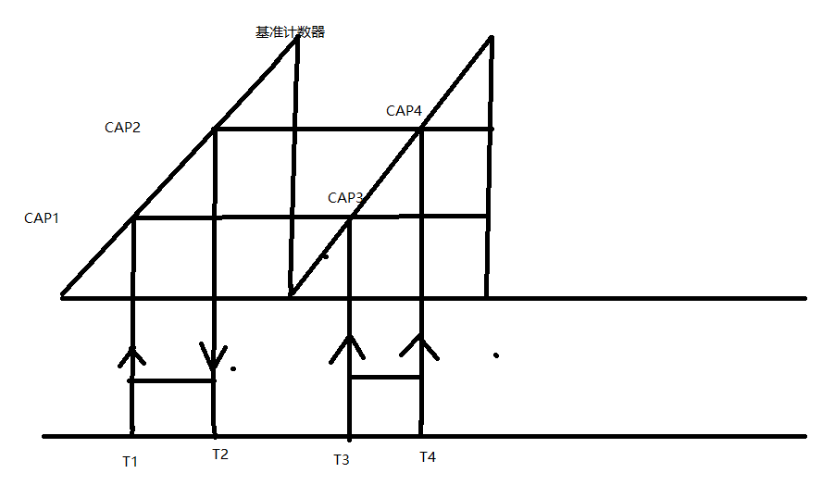
EDIS;

## 7、eCAP功能调试

CAP有两个功能，一个是输入捕获，另一个是APWM，即输出PWM波。虽然可以输出PWM，但是我觉得好像没啥用。

在DSP中有一个MOD4寄存器，这个寄存器是不断对数据进行余4，即0-1-2-3-0-1-2-3…….不断的进行循环。比如现在MOD4的值为0，然后基准计数器不断地累加，当捕获到上升沿（或者下降沿，看自己的设定）时，将此时基准计数器的值存入CAP1中，然后MOD4加一，然后当捕获到下一个下降沿时，将此时的基准寄存器的值存入CAP2，然后MOD4加一，依次循环。





## 8、Futaba SBUS功能调试

S-bus为futaba使用的串行通信协议。实际上为串口通信。

跟TTL串口信号相比，S-bus的逻辑电平是反的，需用如下电路对电平反相，再借到串口接收的Rx管脚就可以了；注意只能硬件取反不能软件取反，因为只有硬件取反才能将空闲状态时变为高电平。

一、协议说明：

串口配置为波特率100kbps（ 100000 ），8位数据，偶校验(even)，2位停止位，无流控。

ScibRegs.SCIHBAUD = 0x00;

ScibRegs.SCILBAUD = 0x2e;

每帧25个字节，按照如下顺序排列：

[startbyte] [data1] [data2] .... [data22] [flags][endbyte]；

SBUS最多包含18通道，其中前面16个为模拟通道，还有2个数字通道在flags字节里面，数字通道只能输出0和1。

flags字节组成：

bit7 = ch17 = digital channel (0x80)

bit6 = ch18 = digital channel (0x40)

bit5 = Frame lost, equivalent red LED on receiver (0x20)

bit4 = failsafe activated (0x10)

bit3 = n/a

bit2 = n/a

bit1 = n/a

bit0 = n/a

起始字节是0x0F。中间22个字节就是16个通道的数据了，为什么是16个通道？因为22x8=11x16，每个通道用11bit表示，范围是0-2047。

具体每个通道的11位数据怎样解析可以直接看程序update\_channels()子函数；，原理目前我也没太明白。

因为舵机或者电机PWM驱动信号为50Hz，即周期为20ms，高电平时间为1000-2000us，其中高电平时间1500us时，舵机或电机处于中间位置，1000us为一侧最小位置，2000us为一侧最大位置。

最后将得到的通道0数据乘以系数后赋值给PWM占空比赋值语句。

EPwm3Regs.CMPA.half.CMPA = x\*3/5;

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 编制 | 更改内容 |
|  |  |  | 建立 |