

2016

宙斯盾版之 SXD28335_QFP 用户手册



玻尔电子

2016-10-1

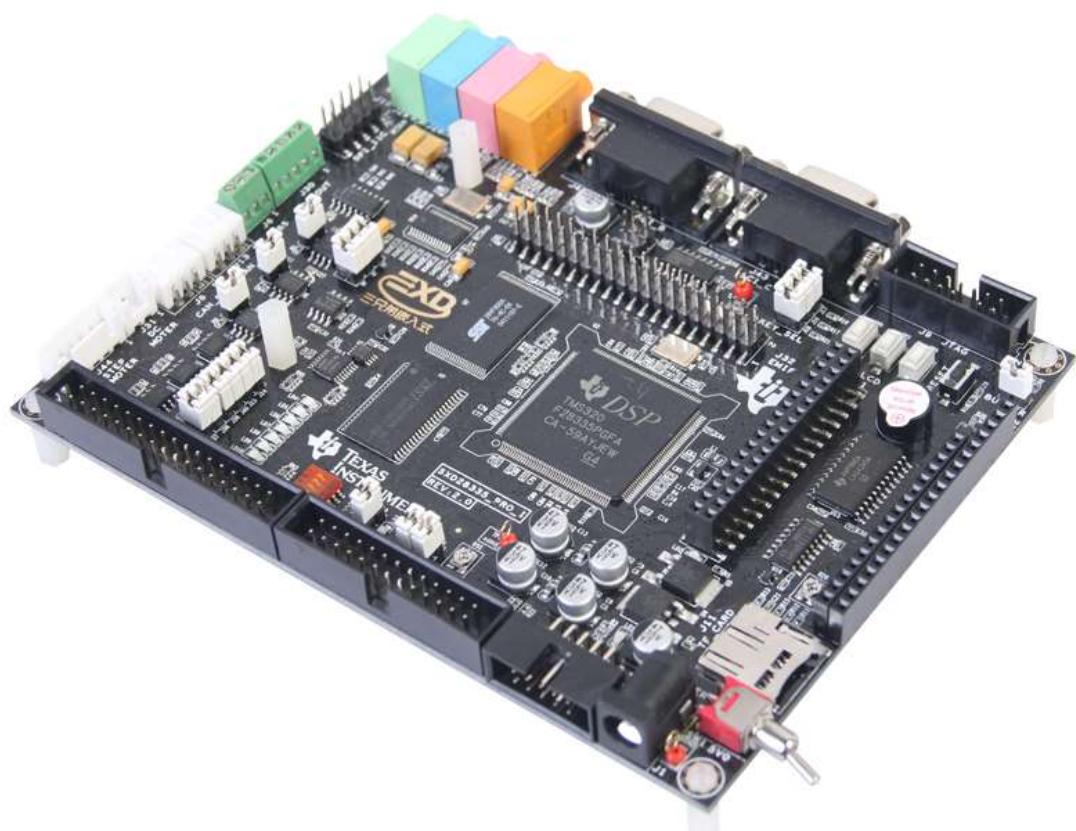
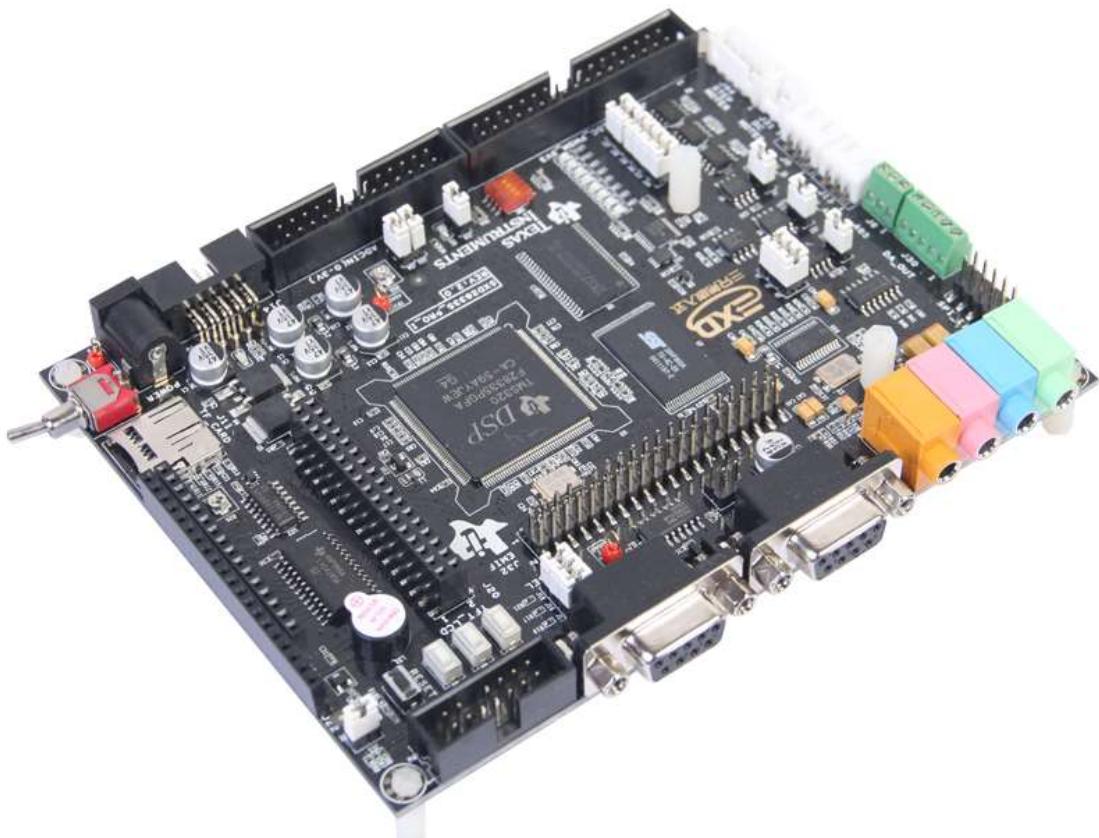
目录

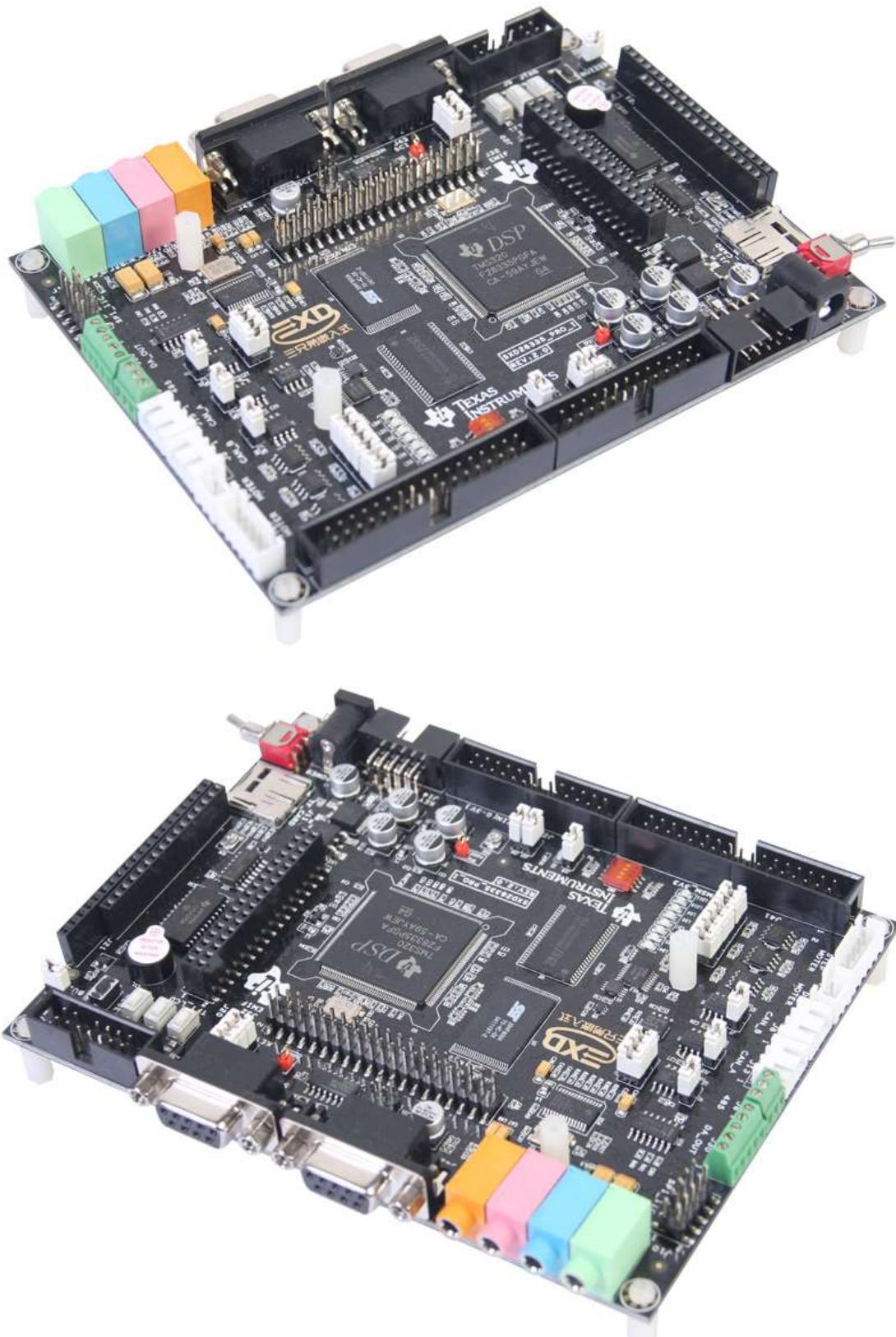
第一章 宙斯盾版之 SXD28335_QFP 产品图片	3
第二章 宙斯盾版之 SXD28335_QFP 实验方法	6
实验 1：GPIO 控制 LED 及蜂鸣器实验 (Example_2833x_LEDBlink)	6
实验 2：通用输入输出接口(GPIO)实验 (Example_2833x_GpioToggle)	15
实验 3：步进电机实验 (Example_2833x_Step_Motor)	18
实验 4：外部中断试验 (Example_2833x_ExternalInterrupt)	21
实验 5：定时器实验 (Example_2833x_CpuTimer)	30
实验 6：看门狗实验 (Example_2833x_Watchdog)	34
实验 7：利用 XINTF 操作 SRAM 实验 (Example_2833xCodeRunFromXintf)	39
实验 8：外部 FLASH 及 SRAM 实验 (Example_2833x_ExFlash_Sram_Core)	42
实验 9：LCD1602 液晶屏实验 (Example_2833x_lcd1602)	51
实验 10：LCD12864 液晶屏实验 (Example_2833x_Lcd12864)	59
实验 11：ADC 循序采样模式测试 (Example_2833xadc_seqmode_test)	67
实验 12：AD 转换实验 (Example_2833x_AdcSeq_ovdTest)	70
实验 13：DMA 方式存取 ADC 转换结果 (Example_2833xadc_dma)	77
实验 14：PWM 启动 ad 转换试验 (Example_2833x_AdcSoc)	79
实验 15：Flash 版点灯程序实验 (Example_2833x_Flash)	85
实验 16：SCI 串口实验 (Example_2833x_Sci_Echoback)	90
实验 17：FIFO 模式下串口实验 (Example_2833xScia_FFDLB)	100
实验 18：FIFO 中断模式下串口实验 (Example_2833xSci_FFDLB_int)	102
实验 19：MAX485 总线实验 (Example_2833x_Max485)	106
实验 20：SPI_DA 程序实验 (Example_2833x_Daout)	111
实验 21：DA 与 AD 联合试验 (Example_2833x_Dac_adc)	117
实验 22：SPI 内循环程序实验 (Example_2833xspi_loopback)	124
实验 23：FIFO 模式下 SPI 程序实验 (Example_2833xSpi_FFDLB)	126
实验 24：SPI 工作于 FIFO 中断方式下实验 (Example_2833xSpi_FFDLB_int)	130
实验 25：IIC EEPROM 实验 (Example_2833x_I2C_eeprom)	135
实验 26：IIC 流水灯实验 (Example_2833x_iicliushui)	141
实验 27：CH452 按键实验 (Example_F28335_Ch452)	144
实验 28：音频实验之警报实验 (Example_2833x_mcbsp_bell)	146
实验 29：音频实验之录音实验 (Example_2833x_mcbsp_rec)	151
实验 30：Mcbsp 模拟 SPI 总线实验 (Example_2833xMcBSP_SPI_DLDB)	155
实验 31：Mcbsp 自测试模式实验 (Example_2833xMcBSP_DLDB)	157
实验 32：McBSP 自测试模式中断实验 (Example_2833xMcBSP_DLDB_int)	159
实验 33：Mcbsp 联合 DMA 实验 (Example_2833xMcBSP_DLDB_DMA)	161
实验 34：片内 DMA 数据搬移试验 (Example_2833xDMA_ram_to_ram)	163
实验 35：片内外 RAM 通过 DMA 进行数据搬移 (Example_2833xDMA_xintf_to_ram)	165
实验 36：Can 发送试验 (Example_Sxd28335_ECan_Tx)	168
实验 37：Can 接收试验 (Example_2833x_Can_Rx)	173
实验 38：CANA 发送 CANB 接收试验 (Example_2833x_EcanA_to_B_Xmit)	179
实验 39：eCAN 自测试模式 (Example_2833x_ecan_back2back)	189
实验 40：直流电机实验 (Example_Sxd28335_Dcmotor_Ch452)	192

实验 41: PWM 计数方式之 UpDown 模式实验 (Example_2833xEPwmUpDownAQ)	198
实验 42: PWM 计数方式之 Up 计数实验 (Example_2833xEPwmUpAQ)	202
实验 43: PWM 用于定时功能实验 (Example_2833xEPwmTimerInt)	207
实验 44: 带死区的 PWM 实验 (Example_2833xEPwmDeadBand)	210
实验 45: CAP 捕捉 PWM 边沿实验 (Example_2833xECap_Capture_Pwm)	217
实验 46: CAP 输出 PWM 波实验 (Example_2833xECap_apwm)	223
实验 47: EQEP 频率测量 (Example_2833xEqep_freqcal)	227
实验 48: EQEP 位置速度测量 (Example_2833xEqep_pos_speed)	229
实验 49: SXD28335 开发板 TF 卡实验 (Example_2833x_SD_FAT32)	232
实验 50: 单精度浮点运算软件实现 (Example_2833xFPU_software)	235
实验 51: 单精度浮点单元硬件实现 (Example_2833xFPU_hardware)	240
实验 52: 高分辨率 PWM 演示示例 (Example_2833xHRPWM)	244
实验 53: High Resolution PWM SFO V5	244
实验 54: High Resolution PWM Symmetric Duty Cycle SFO V5	246
实验 55: FFT 例子 (Example_2833x_FFT)	248
实验 56: 有限长单位冲击响应 Example_2833x_FIR	252
实验 57: 无限长脉冲响应 (Example_2833x_IIR)	260
实验 58: 自适应滤波器 (Example_2833xFIRLMS)	265
实验 59: FIR 滤波器实验 (Example_2833xFIR_V2)	269
实验 60: 卷积实验 (Example_2833x_Convolution)	274
实验 61: UCOS 操作系统例程 (Example_2833x_ucosii_run_in_flash_nonBIOS)	277
实验 62: TFT3.2 寸屏实验 (Example_2833x_TFT32)	282
第三章 宙斯盾版之 SXD28335_QFP 标配及选配清单	284
第四章 宙斯盾版之 SXD28335_QFP 包装	287
第五章 宙斯盾版之 SXD28335_QFP 其它说明	288

第一章 宙斯盾版之 SXD28335_QFP 产品图片







第二章 宙斯盾版之 SXD28335_QFP 实验方法

实验 1：GPIO 控制 LED 及蜂鸣器实验 (Example_2833x_LEDBlink)

一 实验目的：

- ✧ 熟悉 TMS320F28335 的 GPIO 配置过程；
- ✧ 学会用 GPIO 控制 Led、蜂鸣器等简单功能，并深入学习 F28335 的 GPIO 模块；

二 实验设备

- ✧ 计算机（已安装 CCSv6.0 开发环境）
- ✧ SXD28335_SYS_PRO_III 开发板
- ✧ 5V 2A (或 3A)DC 电源
- ✧ 仿真器（本手册都是以三兄弟嵌入式生产的 XDS100V3 仿真器为例，其余仿真器类似）

三 实验原理

通过定时器定时控制 Led 灯和数码管。第一次实验可以先不用关心定时器功能。把重点放在 GPIO 功能上，后期会介绍定时器功能；

在 SXD28335_SYS_PRO_I 的开发板中，DSP 的 GPIO0/EPWM1A 管脚有两个功能：GPIO(普通输入输出 IO 口功能)和 PWM 功能。这里配置为 GPIO 功能。

GPIO53/EQEP1I/XD26 管脚有三个功能：GPIO(普通输入输出 IO 口功能)、EQEP1I 功能和数据总线功能，这里配置为 GPIO 功能。

GPIO0/EPWM1A 管脚通过插针 J19 (实验时需要用跳线帽将其短路) 与 LD1 (LED 灯) 链接。GPIO0/EPWM1A 管脚链接到了 LD1 的阴极，如果想让灯亮则需要控制 GPIO0/EPWM1A 管脚为输出并且是低电平。如果想让灯灭则需要控制 GPIO0/EPWM1A 管脚为输出并且是高电平；



图 1 LED 灯的硬件连接图

GPIO53 管脚通过插针 J15 与三级管链接控制蜂鸣器。如果想让蜂鸣器响则需要控制 GPIO53 管脚为输出并且是低电平(三极管的集电极和发射极导通有电流功能蜂鸣器)。否则需要控制 GPIO53 管脚为输出并且是高电平；

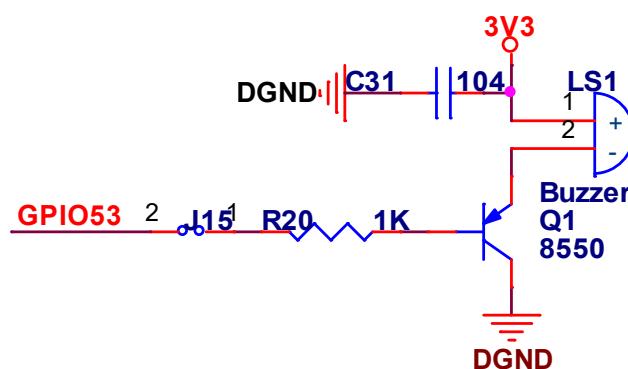


图 2 蜂鸣器的硬件连接图

四 实验步骤

玻尔电子致力于 C2000 全系列开发平台及应用方案的推广

由于我们在如何建立第一个工程文档中，详细演示了工程建立的步骤，这里我们不演示此工程如何建立(步骤类似)，在此不赘述，我们将导入现成的工程文件。

1. 连接实验设备：
 - ◆ 连接仿真器和开发板
 - ◆ 插入外部+5V 电源
2. 打开 CCS6.0；

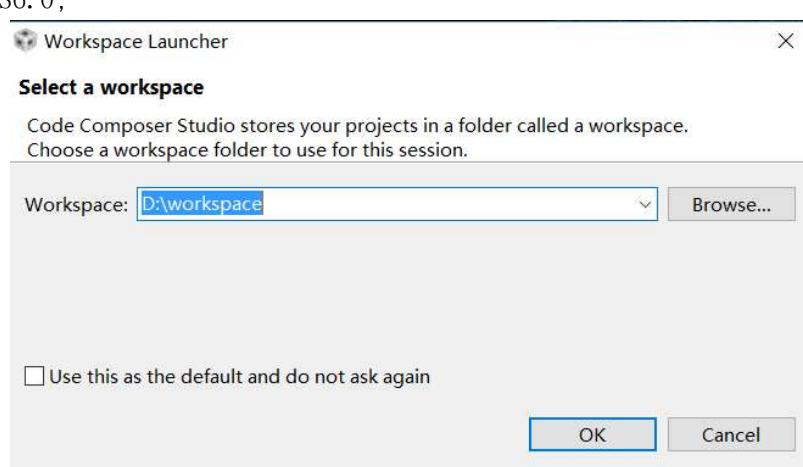


图 3 CCS 工程目录

3. 点击菜单 Project->Import CCS Projects...；点击 browse；选择 D:\workspace\Example_2833x_LEDblink，点击确定，(默认工程为 RAM 模式，不建议老选择 FLASH 模式，在调试阶段尽量采用 RAM 模式，请在工程发行时选择 FLASH 模式：flash 方式后期会介绍)

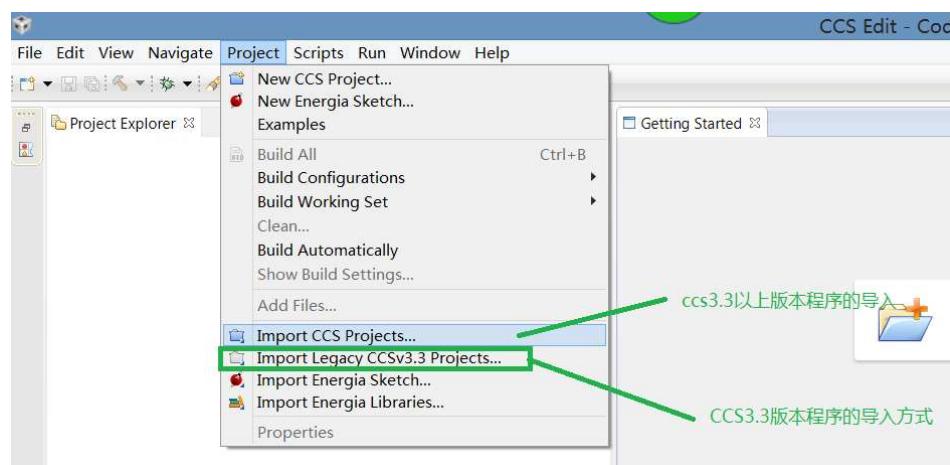


图 4 CCS 导入工程方法

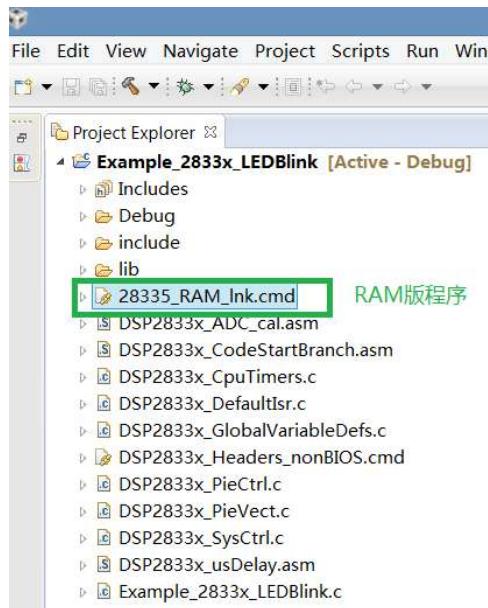


图 5 RAM 版本的程序(由 cmd 文件决定)

注：Flash 版程序的烧写将在后面的章节介绍

4. 连接仿真器与设备并上电；
5. 前面例程没有讲解如何进行仿真器配置，由于 CCS 支持很多处理器和仿真器，它不知道用户目前用的是哪个仿真器和 DSP，所以要通过配置文件告诉 CCS 软件。在此详细讲解下仿真器的配置步骤。这里以 XDS100V3 为例，其余仿真器型号配置与此类似：

File->New Target Configuration File, 设置 File Name: XDS100V3_28335.ccxml (名字可以随便起，但是最好能一眼看出仿真器型号和 DSP 型号的。并且要保证配置文件仅有一个，否则会出现莫名其妙的错误。一般我们只建立一个全局的配置文件，这样所有工程都用这一个配置文件，我们可以不用每个工程都建立配置文件了)

Use Share Location: 复选，点 Finish。

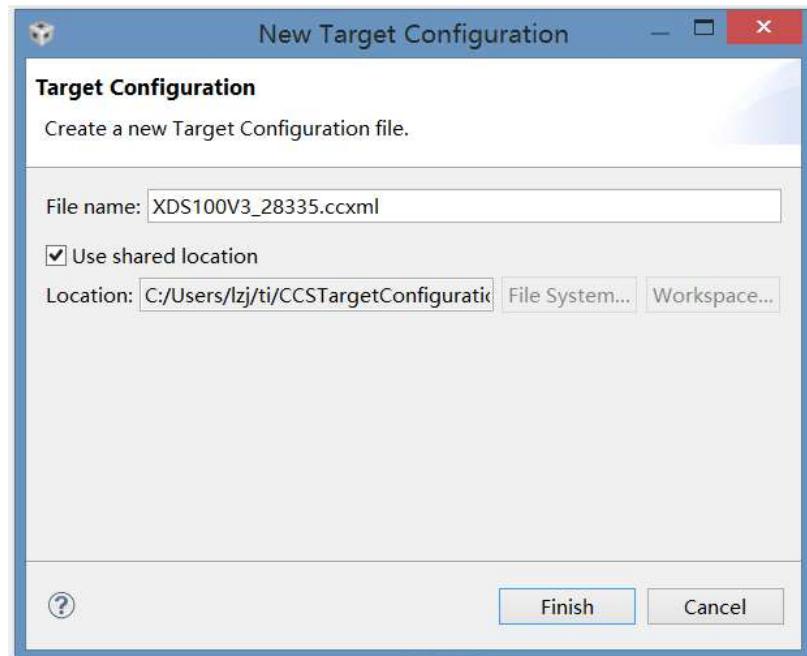


图 6 配置文件的命名

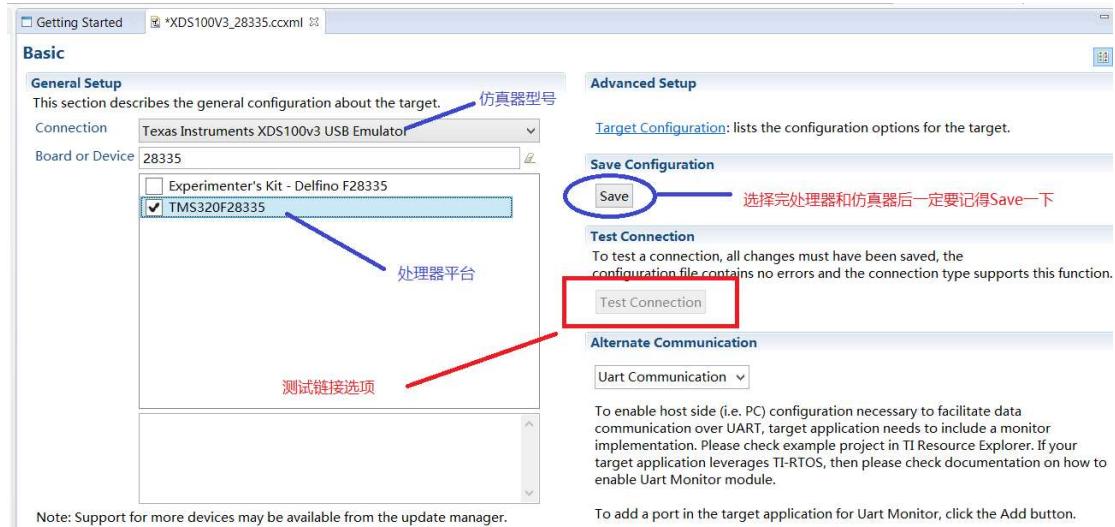


图 7 配置文件的设置

进行如上设置，点击右侧的 Save，其变为灰色后，点击高亮的 Test Connect，在弹出的 TXT 文件的最后一行为：The JTAG DR Integrity scan-test has succeeded.，则说明仿真器正常连接目标板；

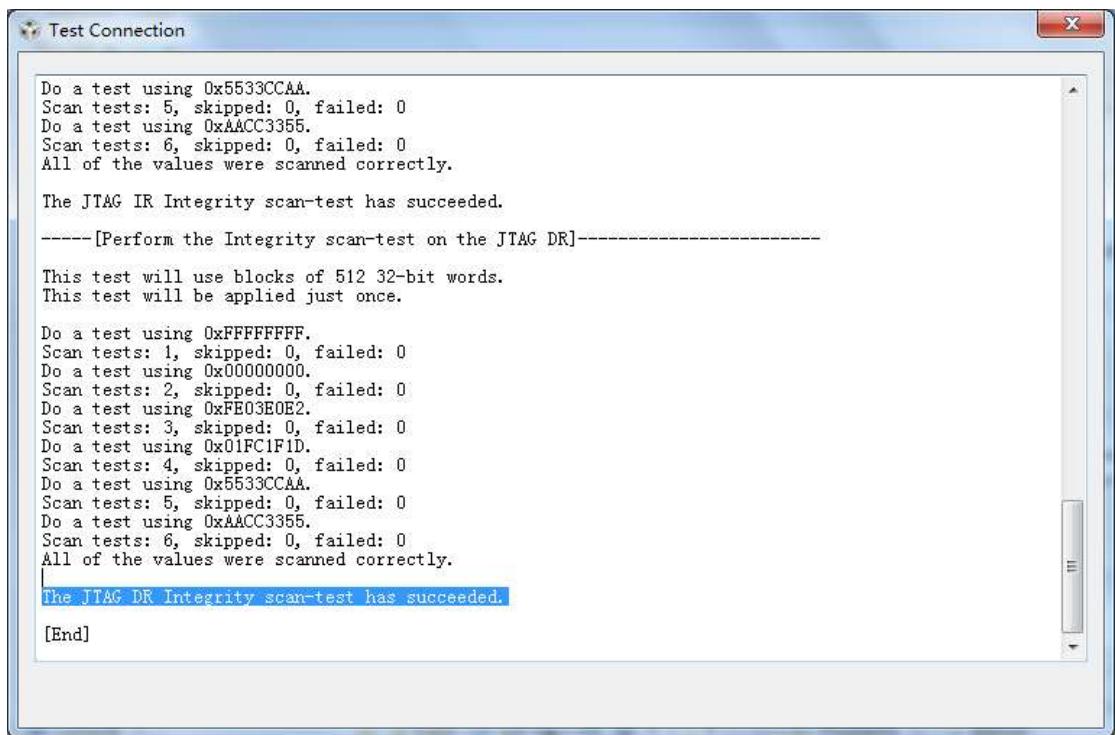
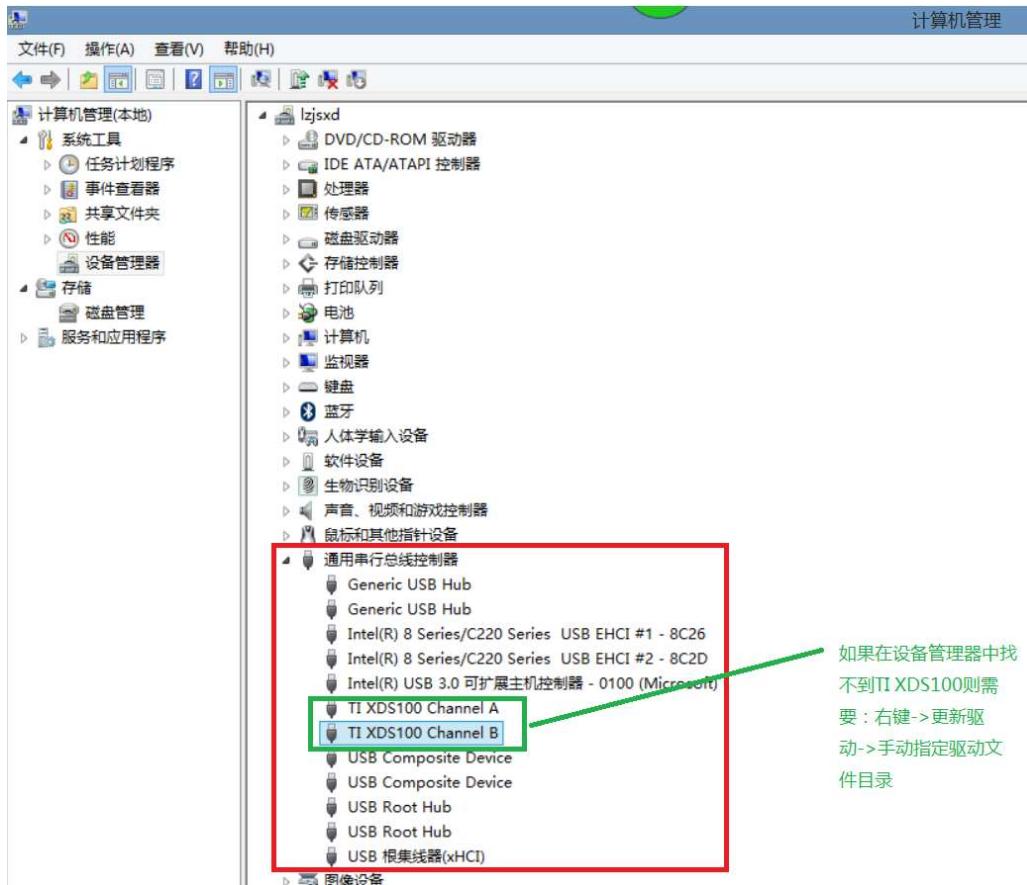


图 8 测试链接成功

注：如果没有测试链接成功，则需求去设备管理器中看一下仿真器的驱动是否安装成功。XDS100 系列的仿真器驱动程序在：C:\ti\ccsv6\ccs_base\emulation\windows\xds100_drivers\ftdi 下。



6. 在 CCS6.0 的菜单栏 View->Target Configuration，在 Target Configuration 栏下有 User Defined，选择 XDS100V3_28335，右键点击 Set As Default，自此我们建立了一个针对 28335 的配置，以后不需要每个工程都进行此配置；

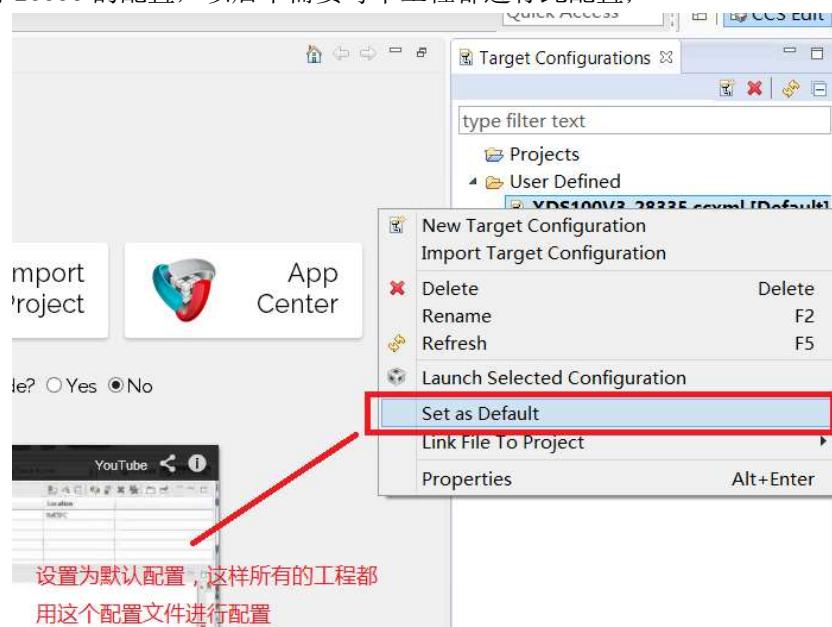


图 9 设置为默认的配置文件

7. 在线调试工程，载入调试界面；

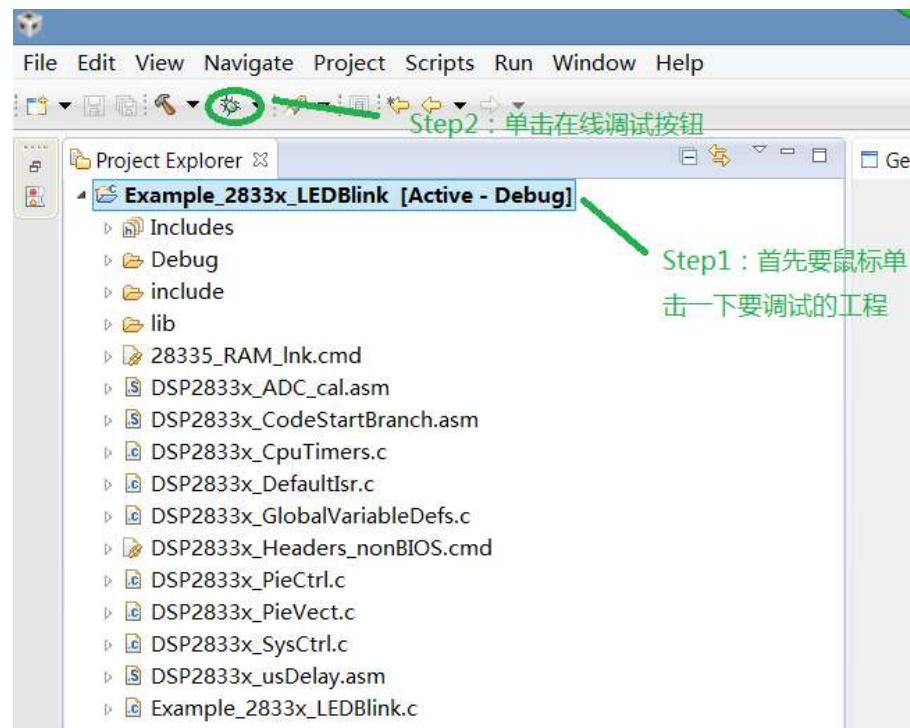


图 10 调试工程

五. 实验结果与分析

实验结果：

Led灯闪烁，蜂鸣器发出滴滴的声音；

GPIO 外设简介：

图 11 为 GPIO0~GPIO27 模块的内部功能图。由于 28335 的大部分 IO 口都有复用功能，也就是 GPIO 口只是它的一个普通功能，有的 IO 口还有 PWM 功能、SCI 功能和 SPI 功能等。这里首先关注 GPIO 功能；

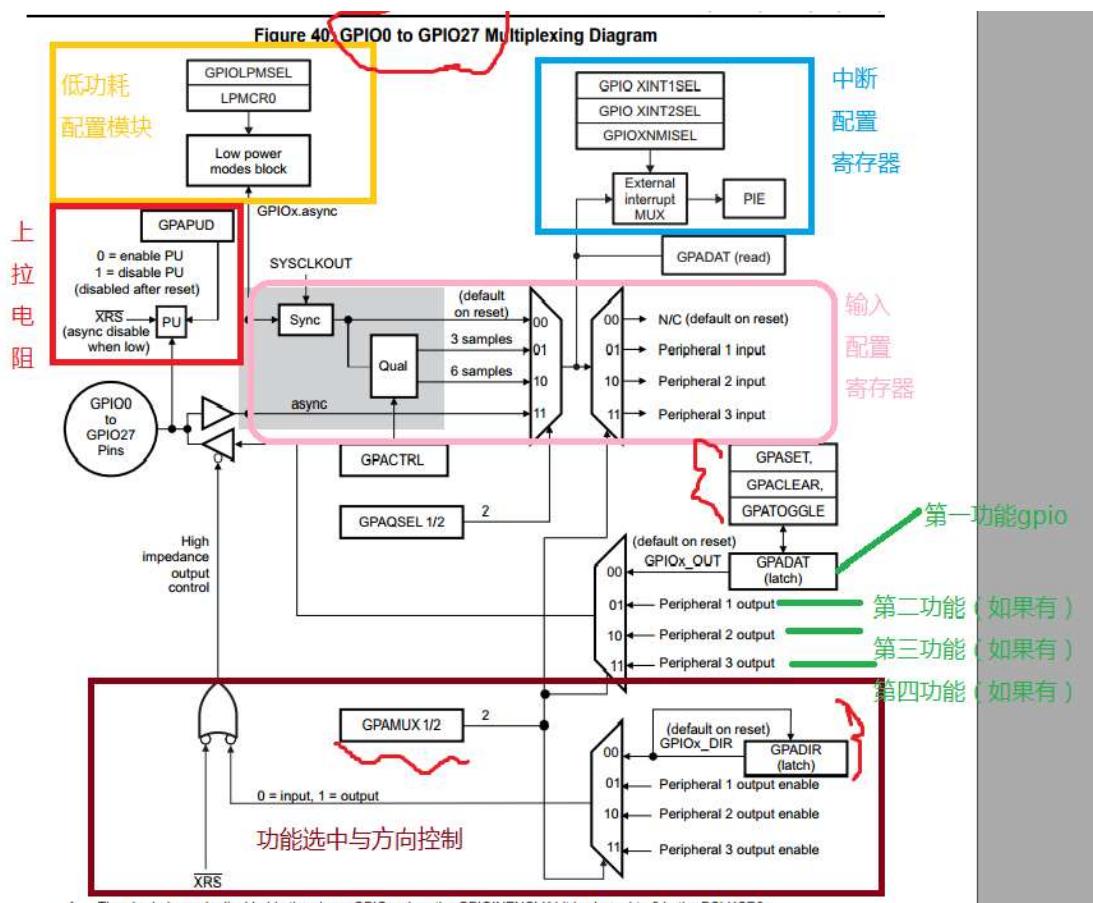


图 11 GPIO 功能外设

这里涉及到几个要用到的寄存器：

1、功能选择寄存器 GPAMUX1/2；每组 I/O 一般有 32 个 I/O 口可以配置。GPAMUX1 对应每组的低 16 个 I/O 口，GPAMUX2 对应高 16 个 I/O 口。

EXAS INSTRUMENTS		General-Purpose Input/Output (GPIO)			
com	如果是00则对应普通 IO口功能	如果是01则对应第一 种外设功能		10则对应第二种外设 功能	
GPAMUX1 Register Bits	Default at Reset Primary I/O Function	Peripheral Selection	Peripheral Selection 2	Peripheral Selection 3	第三种外设功能
1-0	(GPAMUX1 bits = 00)	(GPAMUX1 bits = 01)	(GPAMUX1 bits = 10)	(GPAMUX1 bits = 11)	
3-2	GPIO0	EPWM1A (O)	Reserved ⁽¹⁾	Reserved ⁽¹⁾	
5-4	GPIO1	EPWM1B (O)	ECAP6 (I/O)	MFSRB (I/O) ⁽¹⁾	
7-6	GPIO2	EPWM2A (O)	Reserved ⁽¹⁾	Reserved ⁽¹⁾	
9-8	GPIO3	EPWM2B (O)	ECAP5 (I/O)	MCLKRB (I/O) ⁽¹⁾	
11-10	GPIO4	EPWM3A (O)	Reserved ⁽¹⁾	Reserved ⁽¹⁾	
13-12	GPIO5	EPWM3B (O)	MFSRA (I/O)	ECAP1 (I/O)	
15-14	GPIO6	EPWM4A (O)	EPWMSYNC1 (I)	EPWMSYNC0 (O)	
17-16	GPIO7	EPWM4B (O)	MCLKRA (I/O)	ECAP2 (I/O)	
19-18	GPIO8	EPWM5A (O)	CANTXB (O)	ADCSCAO (O)	
21-20	GPIO9	EPWM5B (O)	SCITXDB (O)	ECAP3 (I/O)	
23-22	GPIO10	EPWM6A (O)	CANRXB (I)	ADCSCBO (O)	
25-24	GPIO11	EPWM6B (O)	SCIRXDB (I)	ECAP4 (I/O)	
27-26	GPIO12	TZ1 (I)	CANTXB (O)	MDXB (O)	
29-28	GPIO13	TZ2 (I)	CANRXB (I)	MDRB (I)	
31-30	GPIO14	TZ3/XHOLD (I)	SCITXDB (O)	MCLKXB (I/O)	
	GPIO15	TZ4/XHOLD (O)	SCIRXDB (I)	MFSXB (I/O)	

图 12 GPAMUX 配置

- 2、方向控制寄存器 GPADIR：如果对应的位为 1 则配置为输出，否则则配置为输入。
- 3、置位寄存器 GPASET：如果对应的位为 1 则将对应的 IO 口拉高(输出高电平)。
- 4、强制拉低管脚 GPACLEAR：如果对应的位为 1 则将对应的 IO 口拉低(输出低电平)。
- 5、输出状态翻转寄存器 GPATOGGLE：如果 GPATOGGLE 的某位为 1 则将相应的 IO 口输出状态进行翻转。

程序解析：

```
//#####
//这个表要看懂，这是 boot 模式选择的表。这四个 IO 口对应着 SW2(4 位拨码开关)；如果为：
// 1 1 1 1 则是开发板一上电就从 Flash 里加载程序；板子默认出场状态就是 Flash 模式：
//      GPIO87    GPIO86    GPIO85    GPIO84
//      XA15      XA14      XA13      XA12
//      PU        PU        PU        PU
//      =====
//      1         1         1         1      Jump to Flash
//      1         1         1         0      SCI-A boot
//      1         1         0         1      SPI-A boot
//      1         1         0         0      I2C-A boot
//      1         0         1         1      eCAN-A boot
//      1         0         1         0      McBSP-A boot
//      1         0         0         1      Jump to XINTF x16
//      1         0         0         0      Jump to XINTF x32
//      0         1         1         1      Jump to OTP
//      0         1         1         0      Parallel GPIO I/O boot
//      0         1         0         1      Parallel XINTF boot
//      0         1         0         0      Jump to SARAM      <- "boot to
//SARAM"
//      0         0         1         1      Branch to check boot mode
//      0         0         1         0      Boot to flash, bypass ADC cal
//      0         0         0         1      Boot to SARAM, bypass ADC cal
//      0         0         0         0      Boot to SCI-A, bypass ADC cal
//      Boot_Table_End$ 
//#####
// $TI Release: 2833x/2823x Header Files and Peripheral Examples V133 $
// $Release Date: June 8, 2012 $
//#####
#include "DSP28x_Project.h"      // 头文件，一般把工程里的 include 目录下的文件包含进来了
interrupt void cpu_timer0_isr(void); // 定时器 0 中断函数
void main(void)
{
// 初始化系统函数，在系统时钟那块将对此函数进行分析
    InitSysCtrl();
// 禁止 cpu 中断
    DINT;
```

```
// 初始化 PIE 控制, 这个会在 PIE 模块进行介绍
    InitPieCtrl();
// Disable CPU interrupts and clear all CPU interrupt flags:
    IER = 0x0000;
    IFR = 0x0000;
// 初始化 PIE 向量表, 这个会在 PIE 模块进行介绍
    InitPieVectTable();
// 将定时器中断函数映射到中断向量表中。只有这样, 当发生中断时 CPU 通过中断向量表才会
// 跳到定时器 0 的中断函数中去: cpu_timer0_isr
    EALLOW; // 开启操作模式, 有些寄存器比较重要, 为了防止误操作, 所以加了这两个语句
    PieVectTable.TINTO = &cpu_timer0_isr;
    EDIS; // 禁止操作, EDIS 和 EALLOW 成对出现
    InitCpuTimers(); // 配置定时器
#ifndef (CPU_FRQ_150MHZ)
// 配置定时器中断周期
    ConfigCpuTimer(&CpuTimer0, 150, 500000);
#endif
#ifndef (CPU_FRQ_100MHZ)
// Configure CPU-Timer 0 to interrupt every 500 milliseconds:
// 100MHz CPU Freq, 50 millisecond Period (in uSeconds)
    ConfigCpuTimer(&CpuTimer0, 100, 500000);
#endif
    CpuTimer0Regs.TCR.all = 0x4001; //开启定时器
// 配置我们要用的GPIO 口, Led 灯和蜂鸣器的控制口
    EALLOW;
    GpioCtrlRegs.GPBMUX2.bit.GPIO53 = 0; //选择为普通 IO 口功能
    GpioCtrlRegs.GPBDIR.bit.GPIO53 = 1; //配置方向为输出
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0; //选择为普通 IO 口功能
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1; //配置方向为输出
    EDIS;
// 使能 CPU 级中断
    IER |= M_INT1;
// 使能 PIE 级中断
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
// 使能全局中断
    EINT; // Enable Global interrupt INTM
    ERTM; // Enable Global realtime interrupt DBGM
//死循环, 如果规定的时间到了, 则会触发中断。从而进入定时器中断, 处理中断函数
    for(;;);
}
interrupt void cpu_timer0_isr(void)
{
    CpuTimer0.InterruptCount++; // 中断次数计数器
```

```
// 每次进入定时器中断下面三个 IO 口的电平状态都会发生翻转
GpioDataRegs.GPBToggle.bit.GPIO53= 1;
GpioDataRegs.GPBToggle.bit.GPIO62= 1;
GpioDataRegs.GPAToggle.bit.GPIO0 = 1;
// Acknowledge this interrupt to receive more interrupts from group 1
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}

=====
// No more.
=====
```

实验 2：通用输入输出接口(GPIO)实验 (Example_2833x_GpioToggle)

一、实验目的：

- ✧ 熟悉 TMS320F28335 的 GPIO 配置过程;
- ✧ 了解 SXD28335 开发板上面的 LED 资源;

二、实验设备：

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ 28335 开发板一套;

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开;

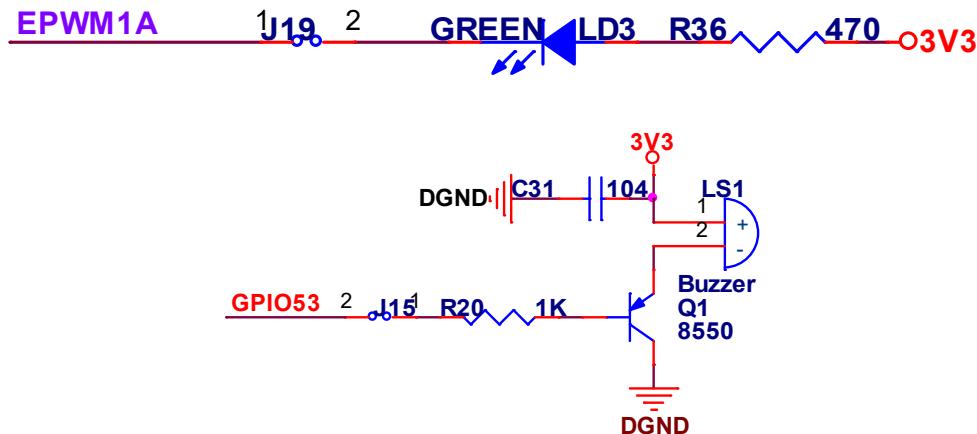


图 1 GPIO 硬件连接图

从上图中可知需要分别用跳线帽将 J62 和 J61 分别连接。

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处（注意仿真器插入方向，请仔细核对防差错针的位置，JTAG 接口是用来连接仿真器的。开发板上对应的是 1~4 针的座子。
注：不要用开发板上 2~0 针或 1~8 针的座子连接仿真器，因为这个是 ad 模拟信号的输入接口）；

- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：
- ◆ 然后单击图中红色的方框处的调试按钮，进行调试。

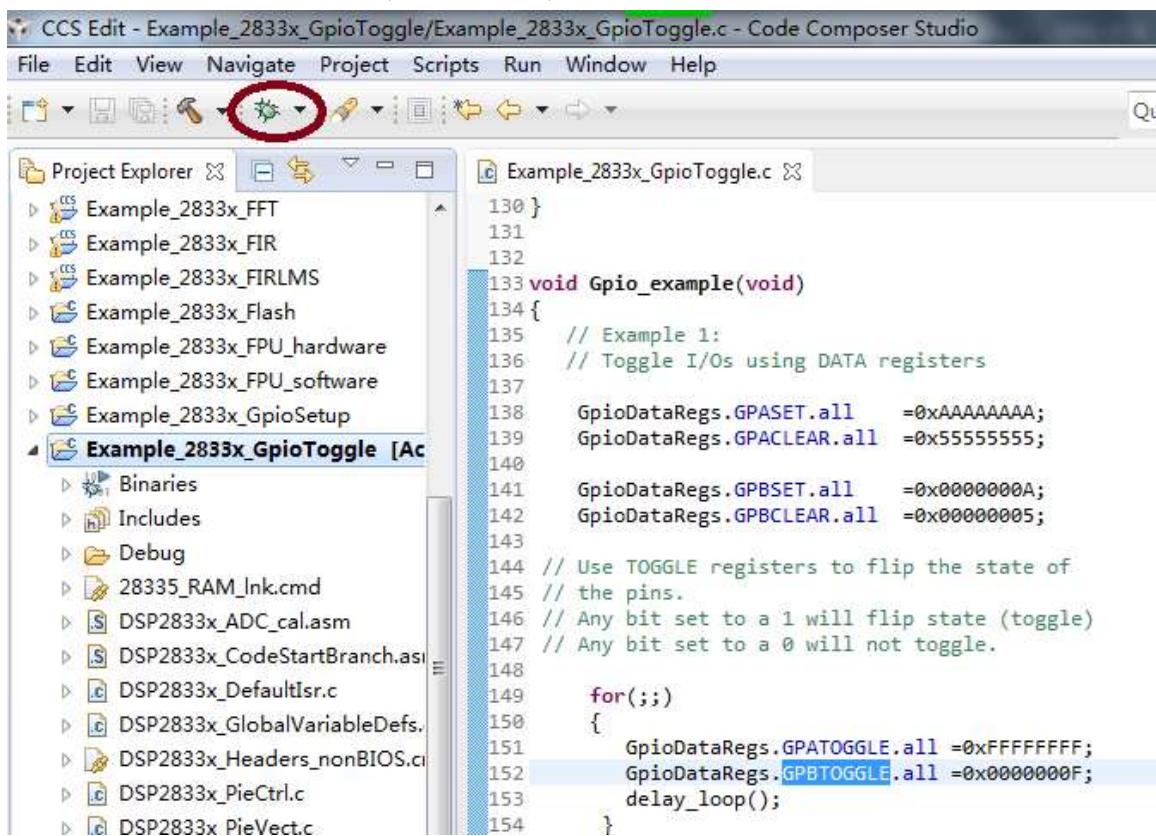


图 2 要调试的工程

- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

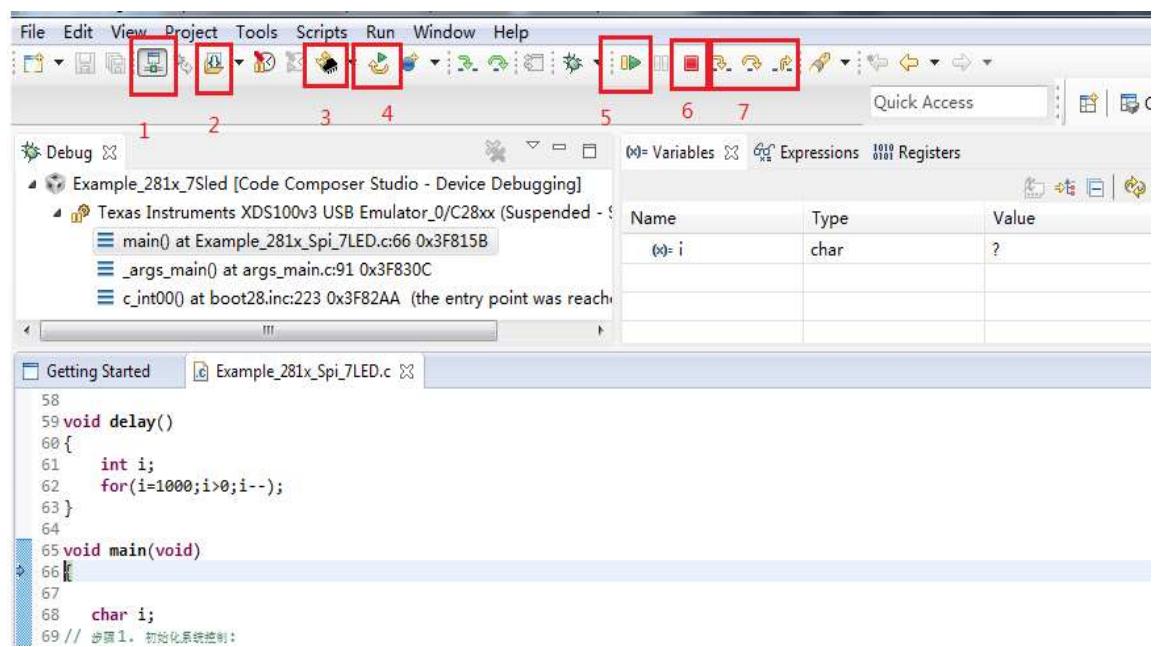


图 3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的

- ◆ 图中 3 是 C P U 软 R e s e t；
 - ◆ 图中 4 是调试时恢复到程序的开始处。
 - ◆ 图中 5 是全速运行；
 - ◆ 图中 6 是停止调试；
 - ◆ 图中 7 是用于单步调试的；
- ◆ 这时我们点击上图中第 5 个图标。全速运行

四、试验现象：

- ◆ 我们可以观察到有两个 LED 灯在不断的闪烁并且蜂鸣器也间断的响着。

五、程序解析：

```
/ 初始化系统控制
// PLL 的配置，看门狗的禁止，外设时钟的使能
    InitSysCtrl();
// 禁止 cpu 中断
    DINT;
// 初始化 PIE 控制寄存器到默认的配置
    InitPieCtrl();
Gpio_select();      //配置要控制的 IO 口
// 禁止中断使能和清除中断标志寄存器
    IER = 0x0000;
    IFR = 0x0000;
// 初始化 PIE 向量表
    InitPieVectTable();
    // 进行 IO 口控制
#ifndef EXAMPLE1

    // This example uses DATA registers to toggle I/O's
    Gpio_example1();

#endif

void Gpio_example1(void)
{
    // Example 1:
    //用 TOGGLE 寄存器改变 IO 口输出状态

    for(;;)
    {

        GpioDataRegs.GPADAT.bit.GPIO0    =1;
        GpioDataRegs.GPBDAT.bit.GPIO53  =1;
    }
}
```

```

delay_loop();

GpioDataRegs.GPADAT.bit.GPIO0      =0;
GpioDataRegs.GPBDAT.bit.GPIO53     =0;
delay_loop();
}

}

```

实验 3：步进电机实验 (Example_2833x_Step_Motor)

一 实验目的：

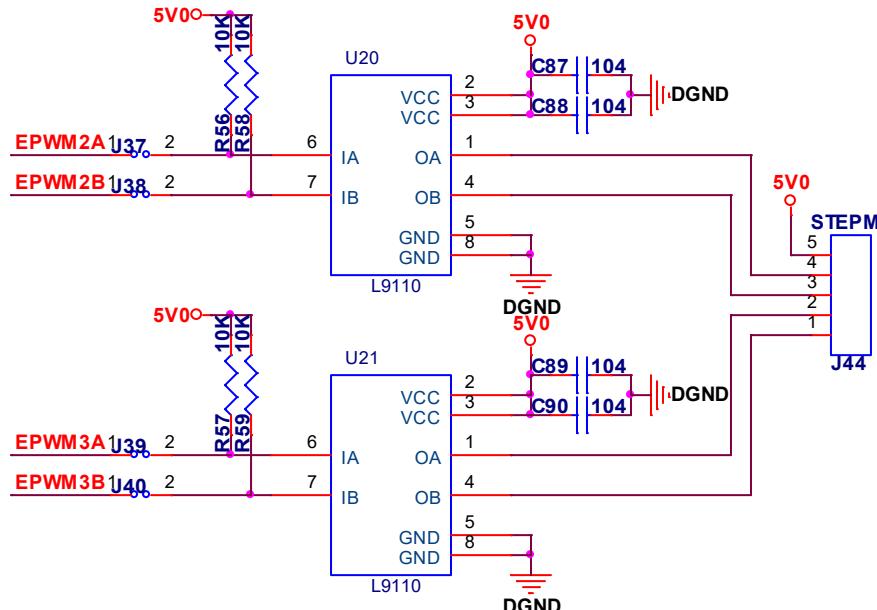
- ✧ 了解 GPIO 口工作原理；
- ✧ 了解步进电机 28BYJ48 电机详细使用说明；

二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ 28335 开发板一套，步进电机一个；

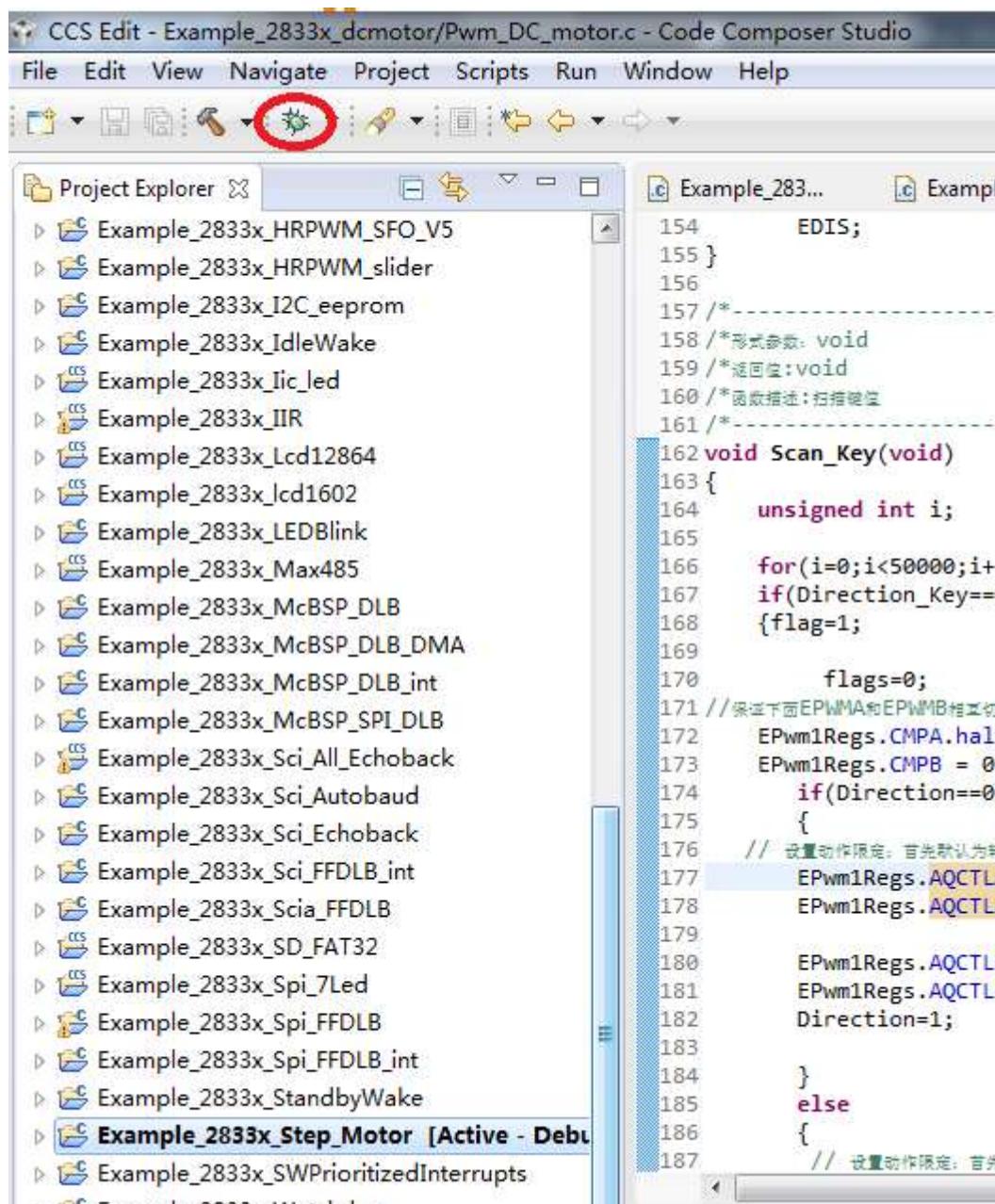
三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

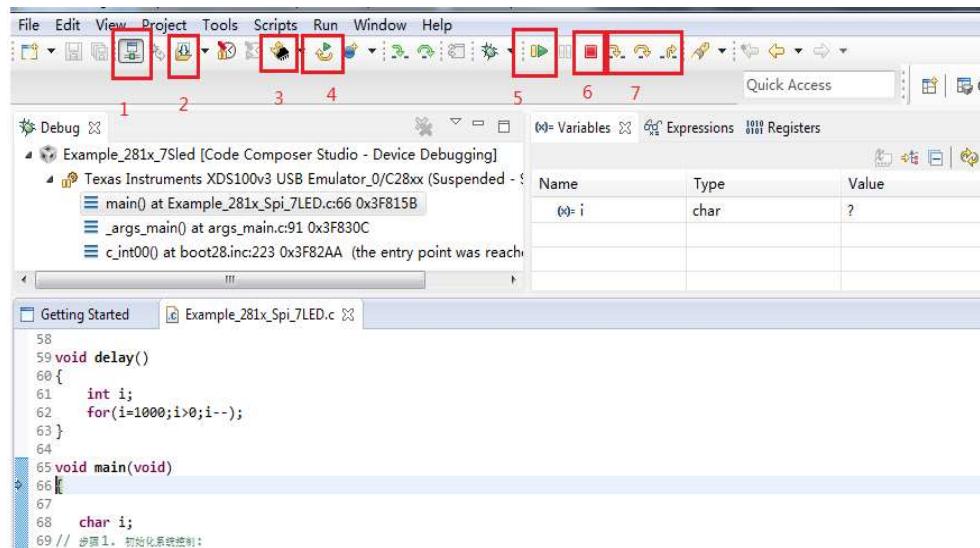


从原理图可知需要用四个跳线帽分别将 J37、J38、J39 和 J40 短路。SXD28335 开发板用两个 L9110 芯片代替了三级管搭建的驱动电路。这个芯片的功能说简单一些就是电压的转换。因为 DSP28335 的 IO 口输出的电压一般是 3V 左右。而驱动电机需要 5V。

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处（注意仿真器插入方向，请仔细核对防差错针的位置，JTAG 接口是用来连接仿真器的。开发板上对应的是 1~4 针的座子。
注：不要用开发板上 2~0 针或 1~8 针的座子连接仿真器，因为这是 ad 模拟信号的输入端口，而不是 JTAG 下载程序接口）；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：
然后单击图中红色的方框处的调试按钮，进行调试。



- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。



图中1图标是用来进行与开发板进行连接的按钮；

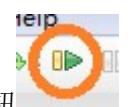
图中2是用来下载 Debug 文件下的.out 文件的

图中3是CPU软Reset；

图中5是全速运行；

图中6是停止调试；

图中7是用于单步调试的；



我们将电机的插座插到 J44(5 个插针的座子上)，然后单击全速运行按钮。

四 试验现象：

电机会进行正向旋转和反向旋转。

24BYJ48 步进电机简介：

步进电机是将电脉冲信号转变为角位移或线位移的开环控制元件。在非超载的情况下，电机的转速、停止的位置只取决于脉冲信号的频率和脉冲数，而不受负载变化的影响，即给电机加一个脉冲信号，电机则转过一个步距角。这一线性关系的存在，加上步进电机只有周期性的误差而无累积误差等特点。使得在速度、位置等控制领域用步进电机来控制变的非常的简单。为此，三兄弟开发板套件中首次引入了步进电机技术，采用扩展的方式，方便用户应用掌握。

虽然步进电机已被广泛地应用，但步进电机并不能象普通的直流电机，交流电机在常规下使用。它必须由双环形脉冲信号、功率驱动电路等组成控制系统方可使用。因此用好步进电机却非易事，它涉及到机械、电机、电子及计算机等许多专业知识。

步进电机的主要特性：

1、步进电机必须加驱动才可以运转，驱动信号必须为脉冲信号，没有脉冲的时候，

步进电机静止，如果加入适当的脉冲信号，就会以一定的角度（称为步角）转动。转动的速度和脉冲的频率成正比。

2、我们用的是 28BYJ48 5V 驱动的 4 相 5 线的步进电机，而且是减速步进电机，减速比为 1: 64，步进角为 $5.625/64$ 度。如果需要转动 1 圈，那么需要 $360/5.625*64=4096$ 个脉冲信号。

3、步进电机具有瞬间启动和急速停止的优越特性。

4、改变脉冲的顺序，可以方便的改变转动的方向。

因此，目前打印机，绘图仪，机器人，等等设备都以步进电机为动力核心。

```
unsigned char Forward[]={0x0004, 0x0008, 0x0010, 0x0020};//正向旋转给定脉冲
```

```
unsigned char Step_table1[]={0x0020, 0x0010, 0x0008, 0x0004};//反向旋转给定脉冲
```

```
void Step_dianji_test(void)
```

```
{
```

```
    Uint16 j, i;
```

```
    step_Gpio_select();
```

```
    j=512;
```

```
    while(j--)
```

```
{
```

```
        for(i=0; i<4; i++)
```

```
{
```

```
            GpioDataRegs.GPATD.all=Step_table[i]; //GPIO2-GPI05 依次输出高平
```

```
            delay_loop2(80000);
```

```
}
```

```
}
```

```
    j=512;
```

```
    GpioDataRegs.GPATD.all=0x0000;
```

```
    while(j--)
```

```
{
```

```
        for(i=0; i<4; i++)
```

```
{
```

```
            GpioDataRegs.GPATD.all=Step_table1[i]; //GPIO2-GPI05 依次输出高平
```

```
            delay_loop2(80000);
```

```
}
```

```
}
```

```
}
```

实验 4：外部中断试验 (Example_2833x_ExternalInterrupt)

一 实验目的：

✧ 了解 PIE 外设中断的使用方法

✧ 了解 TMS320F28335 的中断设置；

二 实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

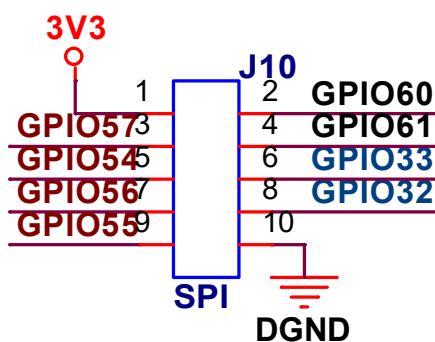


图 1 按键模块

本实验是通过杜邦线将 GPIO60/GPIO61 与 DGND 链接触发中断，在中断函数里点亮 Led 灯。

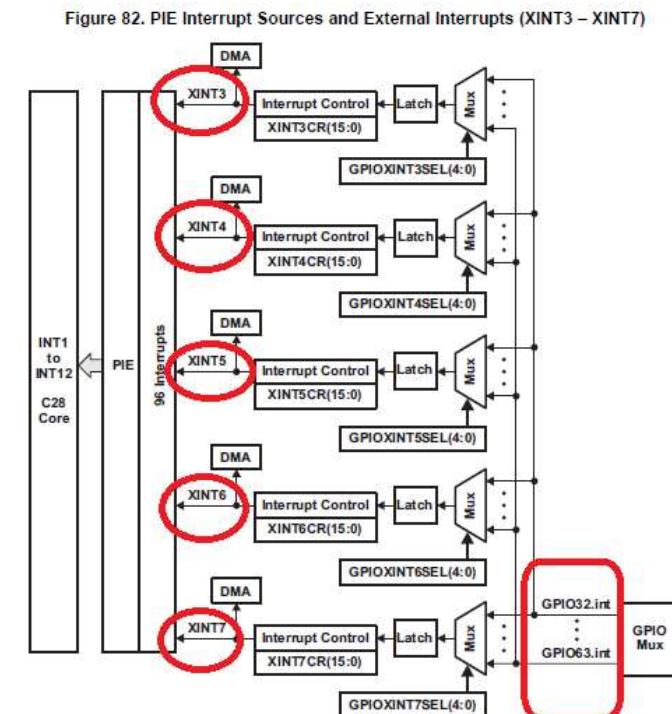
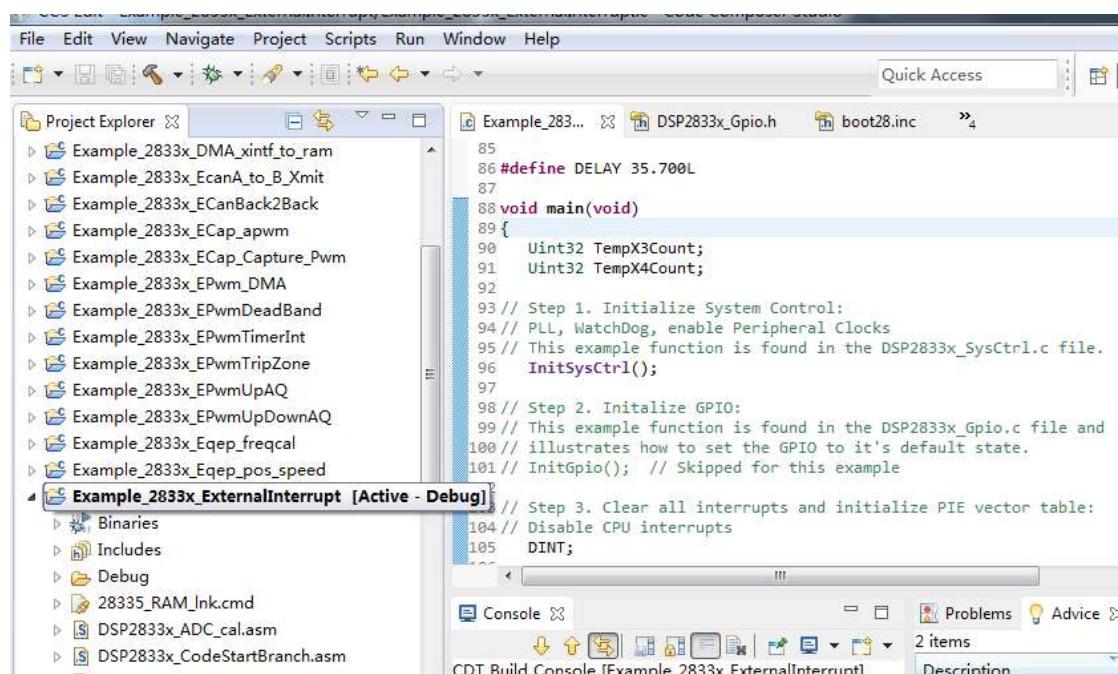


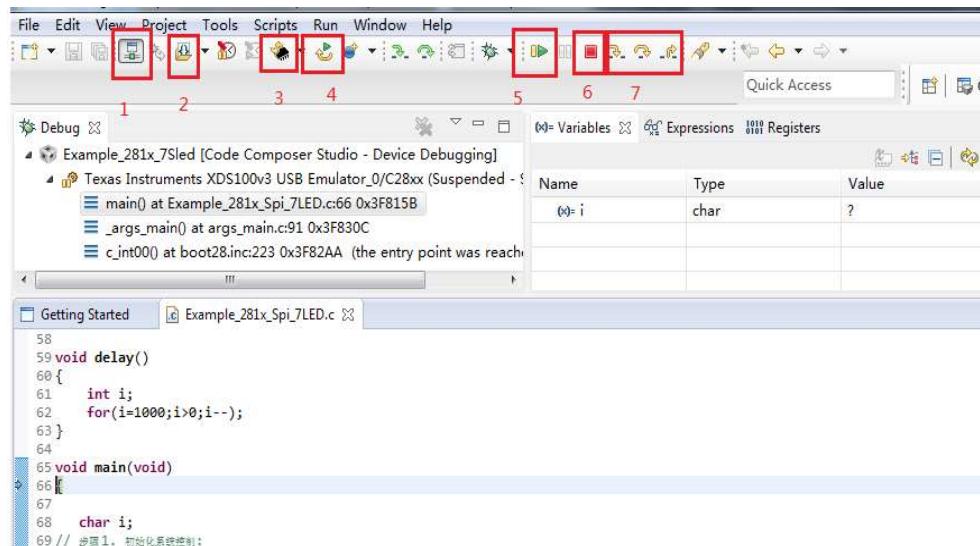
图 2 中断源

从上图中可以看到 GPIO60、61 如果要触发外部中断，则需要配置为外部中断 3 至 7 其中之 1。本程序配置 GPIO60 为 XINT3，GPIO61 为 XINT4；

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：
- ◆ 然后单击图中红色的方框处的调试按钮，进行调试。（注：此时认为兄弟你已经会设置配置文件，并将其设置为默认配置）。



- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。



图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

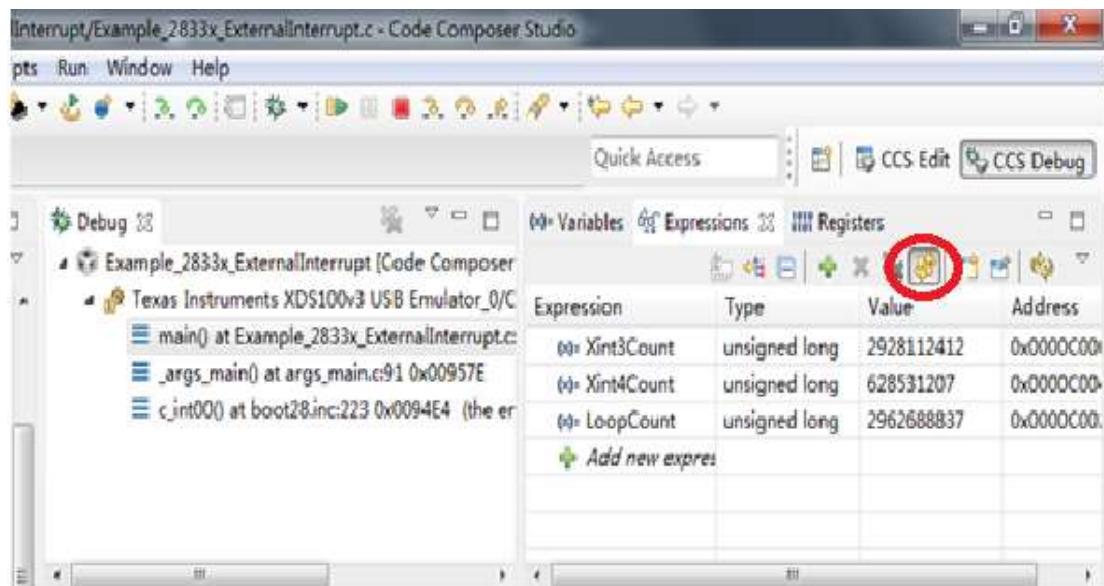
图中 7 是用于单步调试的；

四 试验现象：

首先将 Xint3Count (中断 3 发生中断的次数计数器) 和 Xint4Count

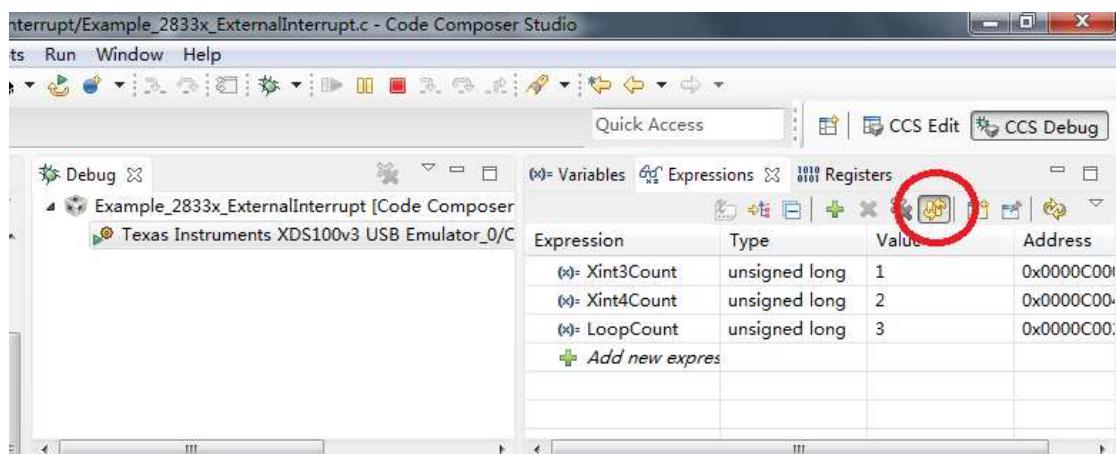
(中断 4 发生中断的次数计数器) 还有空循环计数器 LoopCount 添加

到观察窗口中，如下图所示：



这里。我们直接用鼠标左键单击全速运行，这时程序会全速运行。

现象: 我们依次按下 SW4 、 SW5，会发现每次按下按键时中断计数器会加 1



实验原理及程序说明:

8 个 PIE 块中断被组合进一个 CPU 中断中。总共 12 个 CPU 中断组，每组 8 个中断，等于 96 个中断。在 2833x/2823x 器件上，这些中断中被外设使用的 58 个中断显示在下表中。

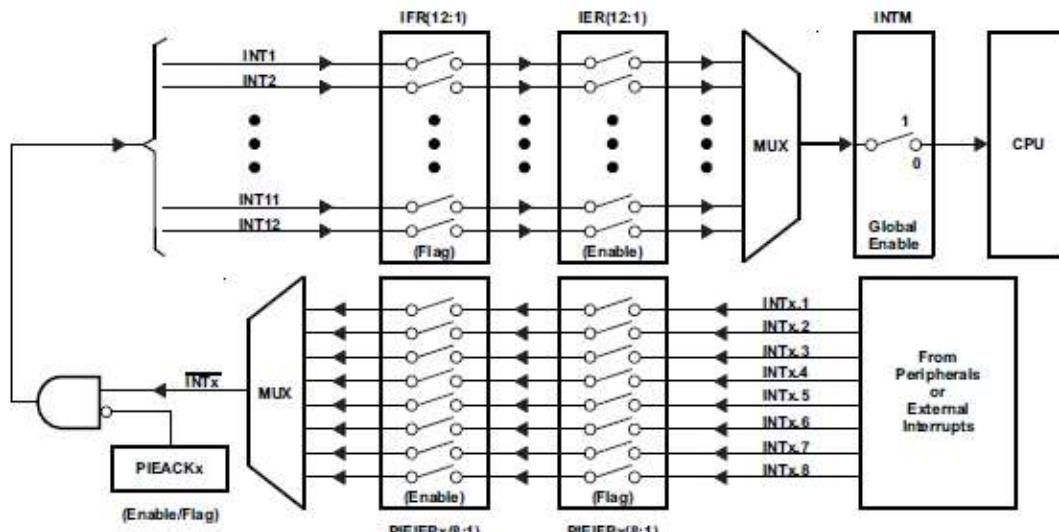


图 3-7. 使用 PIE 块的中断复用

表 3-13. PIE 外设中断⁽¹⁾

CPU 中断	PIE 中断							
	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT (LPM/WD)	TINT0 (定时器 0)	ADCINT (ADC)	XINT2	XINT1	被保留	SEQ2INT (ADC)	SEQ1INT (ADC)
INT2	被保留	被保留	EPWM6_TZINT (ePWM6)	EPWM5_TZINT (ePWM5)	EPWM4_TZINT (ePWM4)	EPWM3_TZINT (ePWM3)	EPWM2_TZINT (ePWM2)	EPWM1_TZINT (ePWM1)
INT3	被保留	被保留	EPWM8_INT (ePWM8)	EPWM5_INT (ePWM5)	EPWM4_INT (ePWM4)	EPWM3_INT (ePWM3)	EPWM2_INT (ePWM2)	EPWM1_INT (ePWM1)
INT4	被保留	被保留	ECAP6_INT (eCAP6)	ECAP5_INT (eCAP5)	ECAP4_INT (eCAP4)	ECAP3_INT (eCAP3)	ECAP2_INT (eCAP2)	ECAP1_INT (eCAP1)
INT5	被保留	被保留	被保留	被保留	被保留	被保留	EQEP2_INT (eQEP2)	EQEP1_INT (eQEP1)
INT6	被保留	被保留	MXINTA (McBSP-A)	MRINTA (McBSP-A)	MXINTB (McBSP-B)	MRINTB (McBSP-B)	SPITXINTA (SPI-A)	SPIRXINTA (SPI-A)
INT7	被保留	被保留	DINTCH6 (DMA)	DINTCH5 (DMA)	DINTCH4 (DMA)	DINTCH3 (DMA)	DINTCH2 (DMA)	DINTCH1 (DMA)
INT8	被保留	被保留	SCITXINTC (SCI-C)	SCIRXINTC (SCI-C)	被保留	被保留	I2CINT2A (I2C-A)	I2CINT1A (I2C-A)
INT9	ECAN1_INTB (CAN-B)	ECANO_INTB (CAN-B)	ECAN1_INTA (CAN-A)	ECANO_INTA (CAN-A)	SCITXINTB (SCI-B)	SCIRXINTB (SCI-B)	SCITXINTA (SCI-A)	SCIRXINTA (SCI-A)
INT10	被保留	被保留	被保留	被保留	被保留	被保留	被保留	被保留
INT11	被保留	被保留	被保留	被保留	被保留	被保留	被保留	被保留
INT12	LUF (FPU)	LVF (FPU)	被保留	XINT7	XINT6	XINT5	XINT4	XINT3

(1) 96 个可能中断中，目前有 58 个正在使用。其余中断保留供未来的器件使用。如果它们在 PIEIFRx 级被启用并且这个组中的中断没有一个被外设使用，这些中断可被用作软件中断。否则，在意外地清除它们的标志同时修改 PIEIFR 的情况下，来自外设的中断也许会丢失。总的来说，在两个安全情况下，被保留的中断可被用作软件中断：

1) 组内没有外设使中断有效。
2) 没有外设中断被分配到这个组（例如：PIE 组 11）。

```
#include "DSP28x_Project.h"      // Device Headerfile and Examples Include File

// Prototype statements for functions found within this file.
__interrupt void xint3_isr(void);
__interrupt void xint4_isr(void);

// Global variables for this example
volatile Uint32 Xint3Count;
volatile Uint32 Xint4Count;
Uint32 LoopCount;

#define DELAY 35.700L
```

```
void main(void)
{
    // Step 1. Initialize System Control:
    // PLL, WatchDog, enable Peripheral Clocks
    // This example function is found in the DSP2833x_SysCtrl.c file.
    InitSysCtrl();

    // Step 2. Initialize GPIO:
    // This example function is found in the DSP2833x_Gpio.c file and
    // illustrates how to set the GPIO to it's default state.
    // InitGpio(); // Skipped for this example

    // Step 3. Clear all interrupts and initialize PIE vector table:
    // Disable CPU interrupts
    DINT;

    // Initialize PIE control registers to their default state.
    // The default state is all PIE interrupts disabled and flags
    // are cleared.
    // This function is found in the DSP2833x_PieCtrl.c file.
    InitPieCtrl();

    // Disable CPU interrupts and clear all CPU interrupt flags:
    IER = 0x0000;
    IFR = 0x0000;

    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    // This will populate the entire table, even if the interrupt
    // is not used in this example. This is useful for debug purposes.
    // The shell ISR routines are found in DSP2833x_DefaultIsr.c.
    // This function is found in DSP2833x_PieVect.c.
    InitPieVectTable();

    // Interrupts that are used in this example are re-mapped to
    // ISR functions found within this file.
    EALLOW; // This is needed to write to EALLOW protected registers
    PieVectTable.XINT3 = &xint3_isr;
    PieVectTable.XINT4 = &xint4_isr;
    EDIS; // This is needed to disable write to EALLOW protected registers
```

```
// Step 4. Initialize all the Device Peripherals:  
// This function is found in DSP2833x_InitPeripherals.c  
// InitPeripherals(); // Not required for this example  
  
// Step 5. User specific code, enable interrupts:  
  
// Clear the counters  
Xint3Count = 0; // Count Xint1 interrupts  
Xint4Count = 0; // Count XINT2 interrupts  
LoopCount = 0; // Count times through idle loop  
  
// Enable Xint1 and XINT2 in the PIE: Group 1 interrupt 4 & 5  
// Enable int1 which is connected to WAKEINT:  
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; // Enable the PIE block  
PieCtrlRegs.PIEIER12.bit.INTx1= 1; // Enable PIE Group 12 INT1  
PieCtrlRegs.PIEIER12.bit.INTx2 = 1; // Enable PIE Group 12 INT2  
IER |= M_INT12; // Enable CPU int1  
EINT; // Enable Global Interrupts  
  
  
// GPIO0 and GPIO1 are inputs  
EALLOW;  
GpioCtrlRegs.GPBMUX2.bit.GPIO60 = 0; // GPIO  
GpioCtrlRegs.GPBDIR.bit.GPIO60 = 0; // input  
GpioCtrlRegs.GPBQSEL2.bit.GPIO60 = 0; // Xint1 Sync to SYSCLKOUT  
only  
  
GpioCtrlRegs.GPBMUX2.bit.GPIO61 = 0; // GPIO  
GpioCtrlRegs.GPBDIR.bit.GPIO61 = 0; // input  
GpioCtrlRegs.GPBQSEL2.bit.GPIO61 = 2; // XINT2 Qual using 6  
samples  
GpioCtrlRegs.GPBCTRL.bit.QUALPRD0 = 0xFF; // Each sampling window is  
510*SYSCLKOUT  
EDIS;  
  
  
EALLOW;  
GpioIntRegs.GPIOXINT3SEL.bit.GPIOSEL = 0x1c; //  
GpioIntRegs.GPIOXINT4SEL.bit.GPIOSEL = 0x1d; //  
EDIS;  
  
// Configure XINT1  
XIIntruptRegs.XINT3CR.bit.POLARITY = 0; // Falling edge interrupt  
XIIntruptRegs.XINT4CR.bit.POLARITY = 1; // Rising edge interrupt
```

```
// Enable XINT1 and XINT2
XIntruptRegs.XINT3CR.bit.ENABLE = 1;           // 使能外部中断 3
XIntruptRegs.XINT4CR.bit.ENABLE = 1;           // 使能外部中断 4

// 用来指示发生了中断，在中断函数中进行翻转
EALLOW;
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;             // GPIO
GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;               // output
GpioDataRegs.GPATDAT.bit.GPIO0=1;                //付初值
EDIS;
LoopCount=0;

// Step 6. IDLE loop:
for(;;)
{
    LoopCount++;
}

// Step 7. Insert all local Interrupt Service Routines (ISRs) and functions here:
// If local ISRs are used, reassign vector addresses in vector table as
// shown in Step 5

__interrupt void xint3_isr(void)
{
    GpioDataRegs.GPATGGLE.bit.GPIO0 = 1;      // GPIO0 翻转
    Xint3Count++;

    // Acknowledge this interrupt to get more from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP12;
}

__interrupt void xint4_isr(void)
{
    GpioDataRegs.GPATGGLE.bit.GPIO0 = 1;      // GPIO0 翻转
    Xint4Count++;

    // Acknowledge this interrupt to get more from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP12;
}
```

实验 5:定时器实验 (Example_2833x_CpuTimer)

一 实验目的:

- ✧ 了解 TMS320F28335 的定时器工作原理;
- ✧ 了解 TMS320F28335 的中断设置;

二 实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三、实验步骤:

- ✧ 首先打开 CCS6.0 开发环境;
- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处 (注意仿真器插入方向, 请仔细核对防差错针的位置, JTAG 接口是用来连接仿真器的。开发板上对应的是 1~4 针的座子。
注: 不要用开发板上 2~0 针或 1~8 针的座子连接仿真器, 因为这是 ad 模拟信号的输入端口, 而不是 JTAG 下载程序接口);
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:
然后单击图中红色椭圆处的调试按钮, 进行调试。

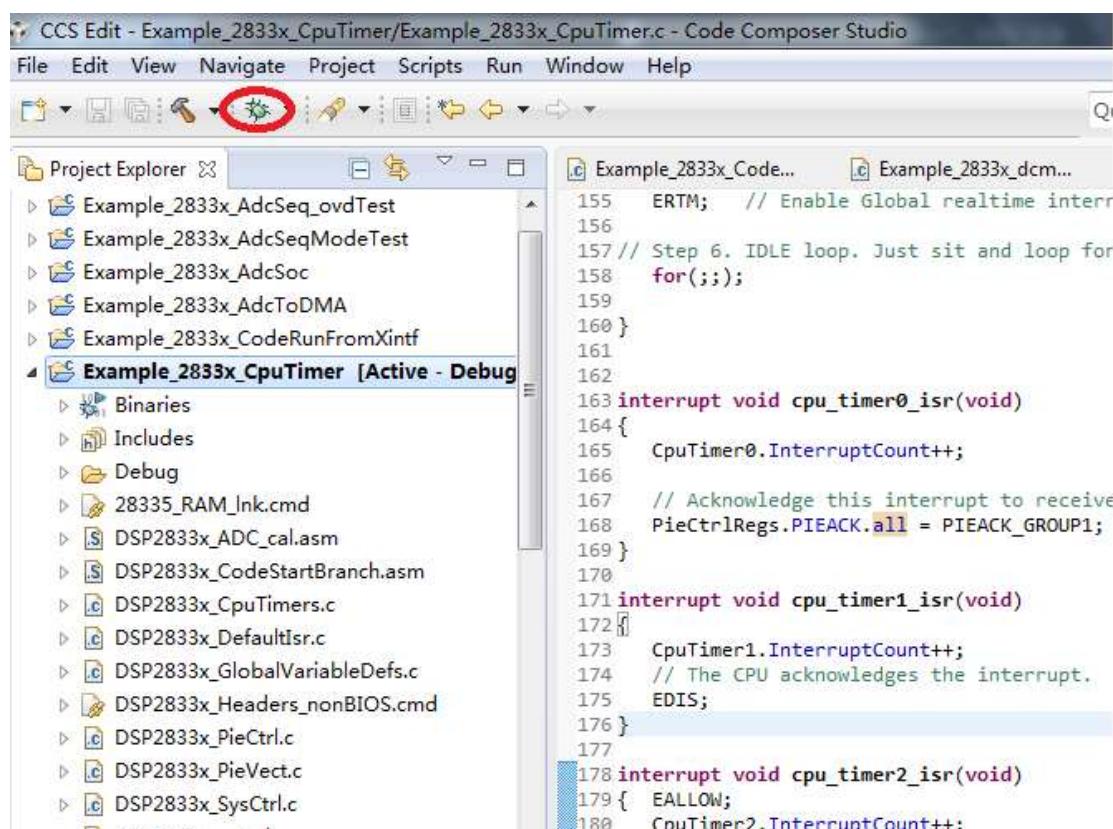


图 1 调试

- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。此时会出现下面的界面。

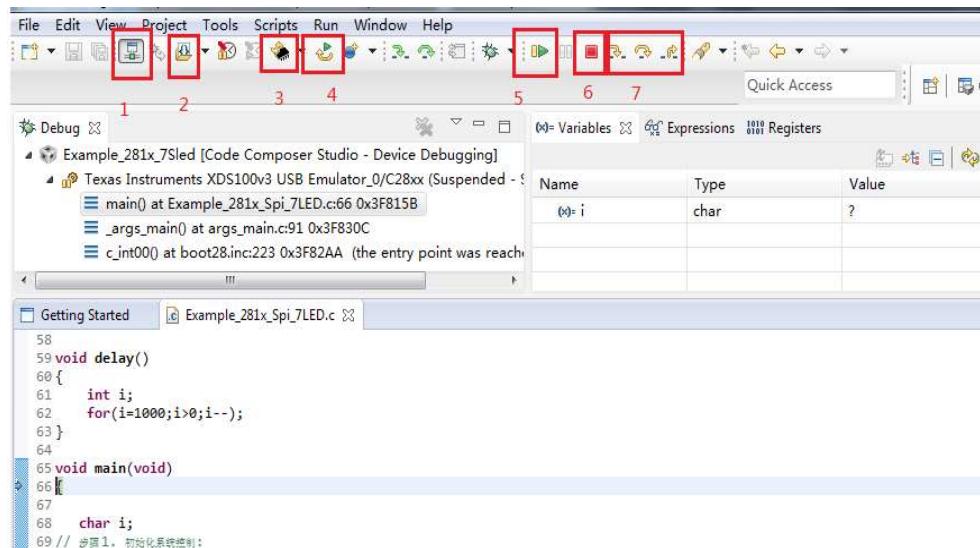


图 2 常用调试命令

图中 1 图标是用来进行与开发板进行连接的按钮;

图中 2 是用来下载 Debug 文件下的.out 文件的;

图中 3 是 C P U 软 R e s e t ;

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行;

图中 6 是停止调试;

图中 7 是用于单步调试的;

四、实验现象：

首先将 CpuTimer0.InterruptCount 、 CpuTimer1.InterruptCount 、 CpuTimer2.InterruptCount 添加到观察窗口中，如下图所示：

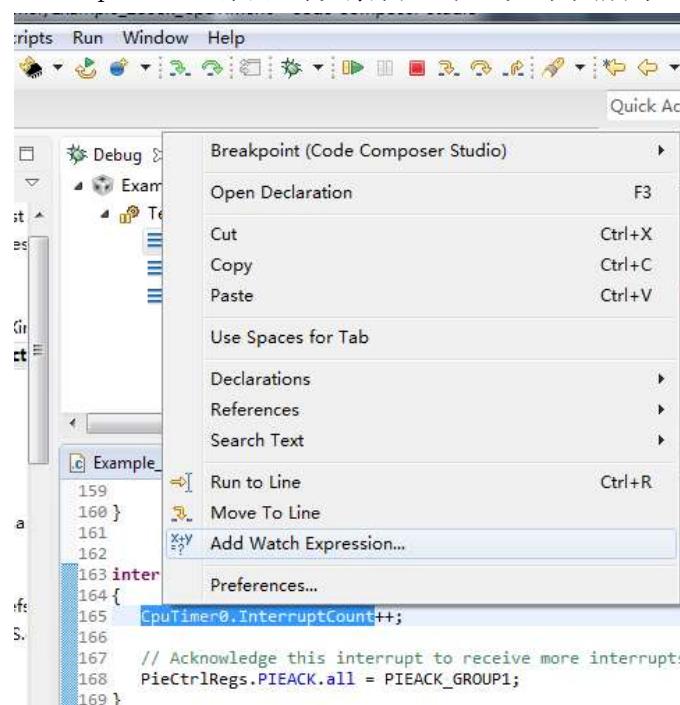


图 3 添加变量到观察窗口

◆ 设置连续刷新图标

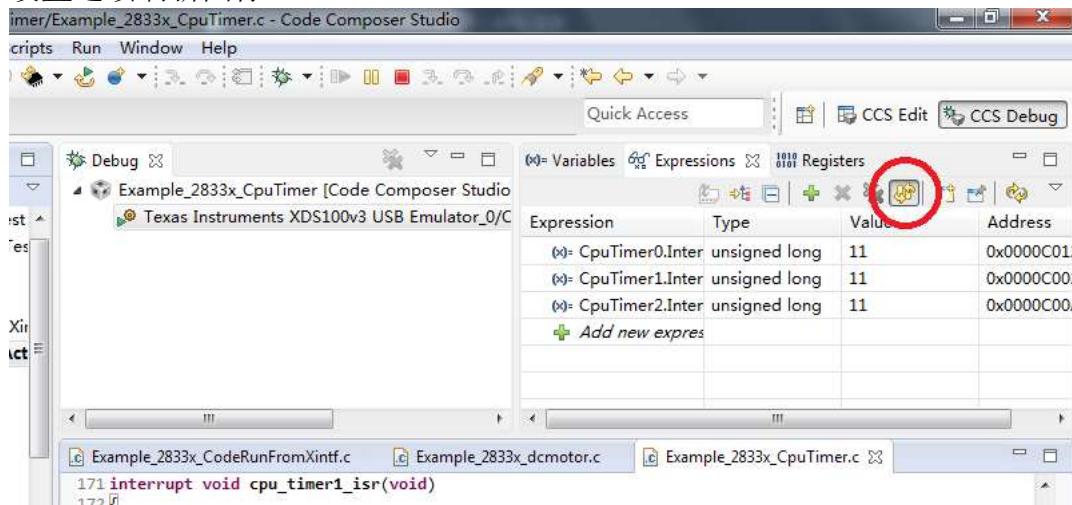


图 4 设置连续刷新

程序说明：

```

EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.TINT0 = &cpu_timer0_isr;//将中断函数添加到中断向量表中
PieVectTable.XINT13 = &cpu_timer1_isr;//将中断函数添加到中断向量表中
PieVectTable.TINT2 = &cpu_timer2_isr;//将中断函数添加到中断向量表中
EDIS;

// 配置定时器 0、1、2 每隔 1s 产生中断一次
ConfigCpuTimer(&CpuTimer0, 150, 1000000);
ConfigCpuTimer(&CpuTimer1, 150, 1000000);
ConfigCpuTimer(&CpuTimer2, 150, 1000000);

//使能定时器中断
IER |= M_INT1;
IER |= M_INT13;
IER |= M_INT14;

//中断函数，进行计数器加 1 操作
interrupt void cpu_timer0_isr(void)
{
    // 定时器中断标志，每次进入中断后都进行加 1 操作
    CpuTimer0 InterruptCount++;
    // Acknowledge this interrupt to receive more interrupts from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}

interrupt void cpu_timer1_isr(void)
{
    CpuTimer1 InterruptCount++;
    // The CPU acknowledges the interrupt.
    EDIS;
}

```

```
interrupt void cpu_timer2_isr(void)
{
    EALLOW;
    CpuTimer2.InterruptCount++;
    // The CPU acknowledges the interrupt.
    EDIS;
}

//定时器配置函数
void ConfigCpuTimer(struct CPUTIMER_VARS *Timer, float Freq, float Period)
{
    //三个参数，第一个表示哪个定时器，第二个表示定时器频率，第三个表示定时器周期值
    Uint32 temp;
    //初始化定时器周期寄存器：
    Timer->CPUFreqInMHz = Freq;
    Timer->PeriodInUSec = Period;
    temp = (long) (Freq * Period);
    Timer->RegsAddr->PRD.all = temp; // Freq * Period 的值给周期寄存器
    Timer->RegsAddr->TPR.all = 0; // Set pre-scale counter to divide by 1
    (SYSCLKOUT):
    Timer->RegsAddr->TPRH.all = 0;
    初始化定时器控制寄存器：
    Timer->RegsAddr->TCR.bit.TSS = 1; // 1 = Stop timer, 0 = Start/Restart Timer
    Timer->RegsAddr->TCR.bit.TRB = 1; // 1 = reload timer
    Timer->RegsAddr->TCR.bit.SOFT = 0;
    Timer->RegsAddr->TCR.bit.FREE = 0; // Timer Free Run Disabled
    Timer->RegsAddr->TCR.bit.TIE = 1; // 0 = Disable/ 1 = Enable Timer Interrupt
    中断计数复位：
    Timer->InterruptCount = 0;
}
```

通过以上程序就可以让定时器 0 每隔一段时间产生一次中断，这段时间的计算公式为： $\Delta T = Freq * Period / 150000000$ (s)；（其中 150000000 是 CPU 的时钟频率，即 150MHz 的时钟频率）针对此实验，Freq 为 150，Period 为 1000000，那么 $\Delta T=1s$ 。

中断向量表的初始化函数如下：

```
void InitPieVectTable(void)
// 此函数初始化中断向量表，将中断服务函数与向量表关联
{
    int16 i;
    Uint32 *Source = (void *) &PieVectTableInit; // 中断服务函数入口地址
    Uint32 *Dest = (void *) &PieVectTable; // 中断向量表
    EALLOW;
    for (i=0; i < 128; i++)
        *Dest++ = *Source++; // 把中断入口地址送给中断向量表，达到关联的目的
    EDIS;
    // Enable the PIE Vector Table
```

```
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; // 使能 PIE 模块的总中断
}

下面的一句话就是告诉定时器 0 的中断入口地址为中断向量表的 INTO
EALLOW;
PieVectTable.TINT0 = &ISRTimer0;

EDIS;
```

下面的两句程序是告诉 CPU 第一组中断将会产生，并使能第一组中断的第 7 个小中断

```
IER |= M_INT1;
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
```

```
_interrupt void cpu_timer0_isr(void)
{
    CpuTimer0.InterruptCount++;
    // Acknowledge this interrupt to receive more interrupts from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

实验 6：看门狗实验（Example_2833x_Watchdog）

一 实验目的：

- ✧ 熟悉 TMS320F28335 的 Watchdog 配置过程；
- ✧ 学会用 Watchdog 中断函数监测程序跑飞；

二 实验设备

- ✧ 计算机（已安装 CCSv6.0 开发环境）
- ✧ SXD28335 或 SXD28335B 开发板
- ✧ 5V 2A (或 3A)DC 电源
- ✧ 仿真器（本手册都是以三兄弟嵌入式生产的 XDS100V3 仿真器为例，其余仿真器类似）

三 实验原理

实验一：定期产生看门狗中断，在中断中 WakeCount++；

实验二：定期喂狗，程序不再进入看门狗中断函数；

四 实验步骤

由于我们在如何建立第一个工程文档中，详细演示了工程建立的步骤，这里我们不演示此工程如何建立(步骤类似)，在此不赘述，我们将导入现成的工程文件。

- (1) 首先将 CCS6.0 开发环境打开；
- (2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处（注意仿真器插入方向，请仔细核对防差错针的位置，JTAG 接口是用来连接仿真器的。开发板上对应的是 1~4 针的座子。注：不要用开发板上 2~0 针或 1~8 针的座子连接仿真器，因为这是 ad 模拟信号的输

入端口，而不是 JTAG 下载程序接口）；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如图 1 所示：

然后单击图中红色的方框处的调试按钮，进行调试。

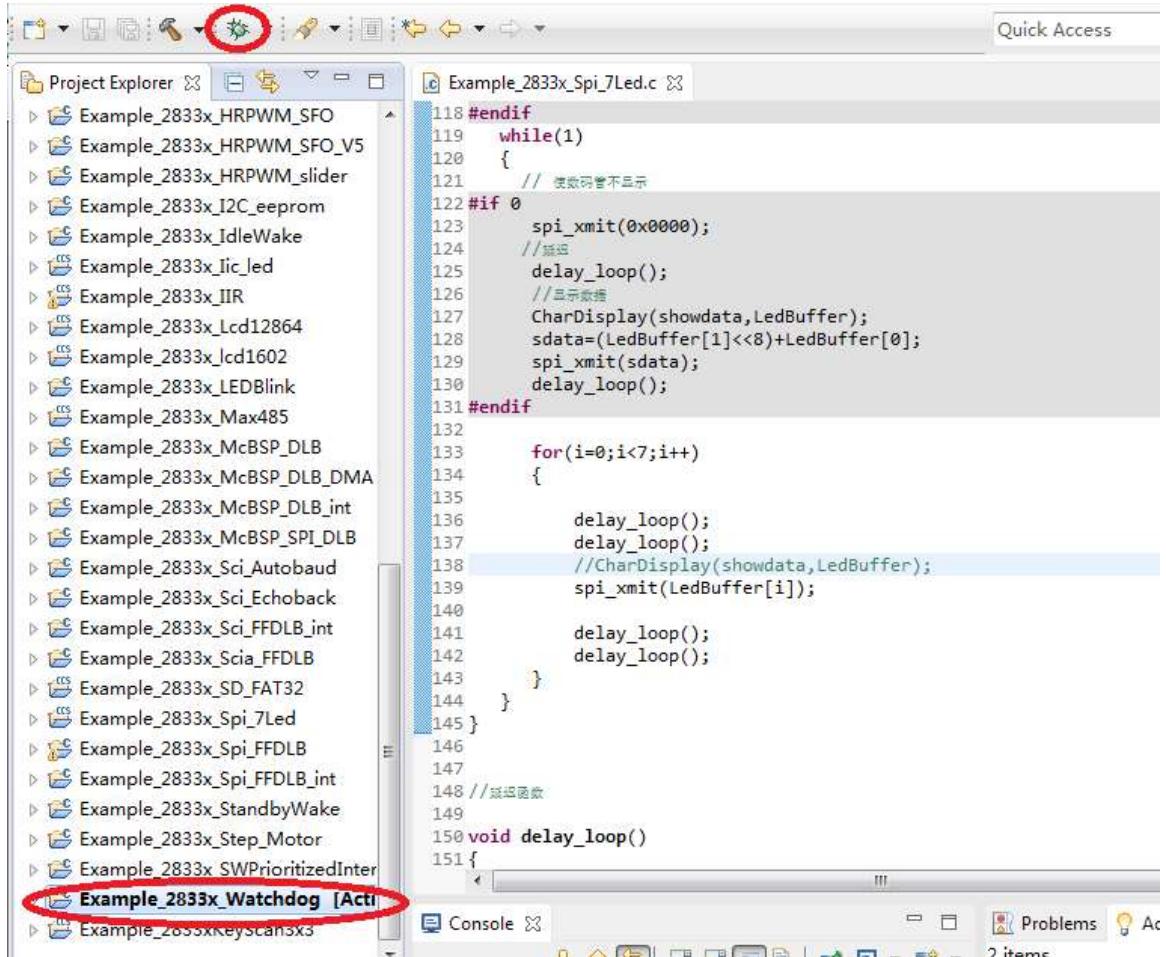


图 1 在线调试步骤

(4) 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现图 2 的界面。

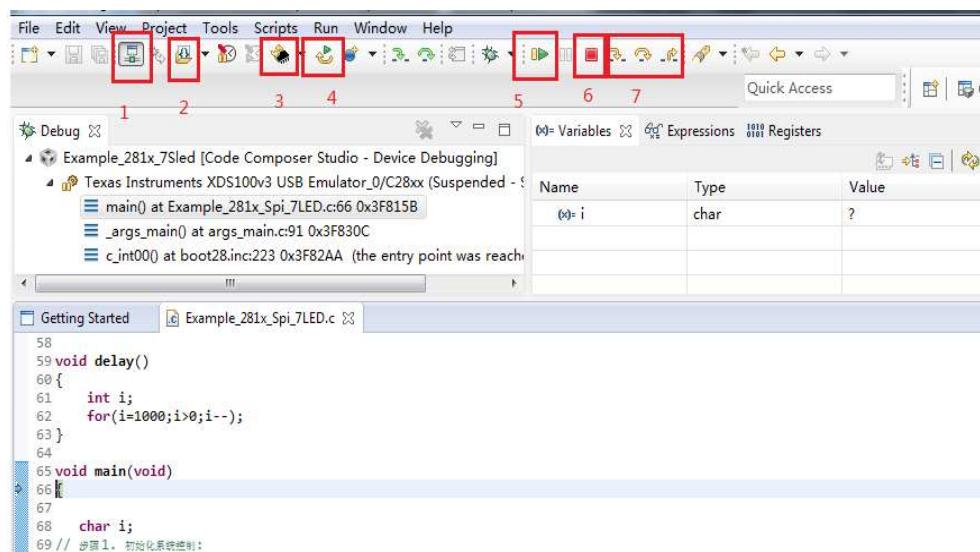


图 2 调试界面简介

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 CPU 软 Reset；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

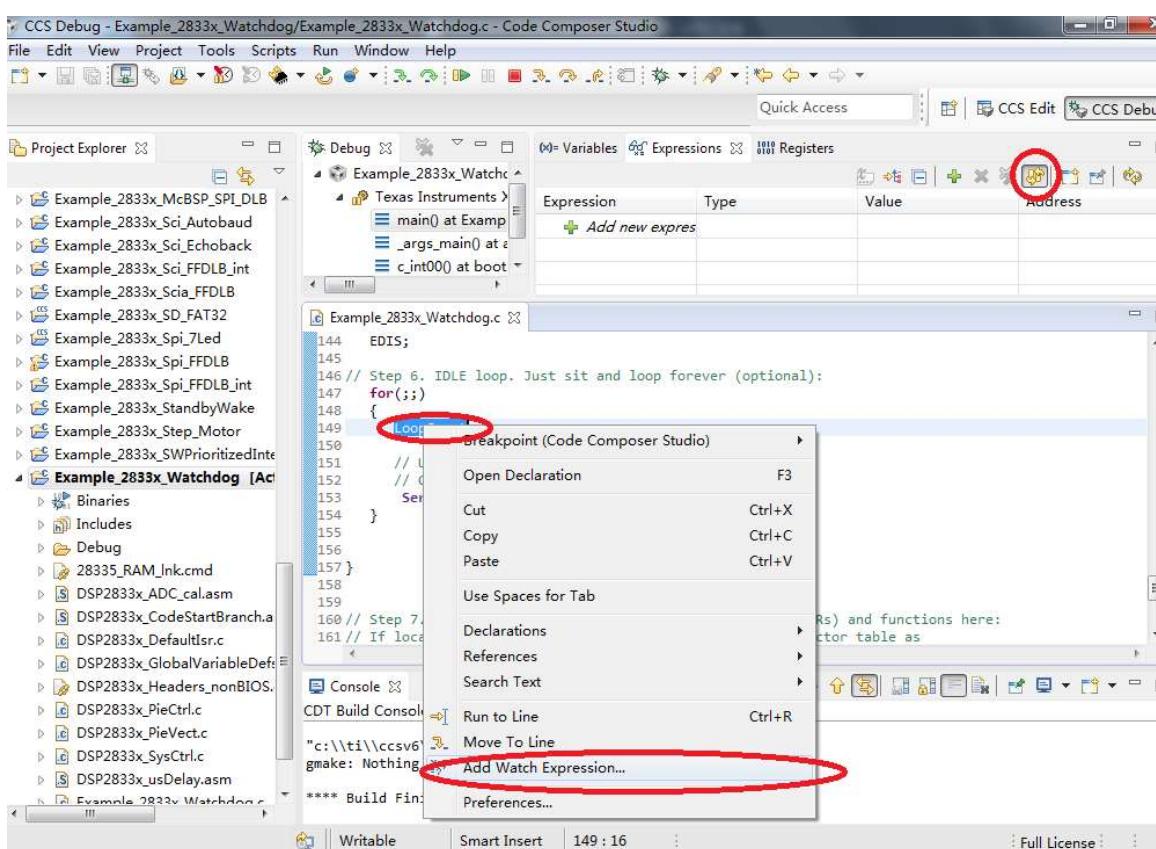
五 实验步骤

情况 1：

```
for(;;)
{
    LoopCount++;

    // Uncomment ServiceDog to just loop here
    // Comment ServiceDog to take the WAKEINT instead
    ServiceDog();    //喂狗操作, 防止狗咬(复位)
}
```

现象：将两个变量 LoopCount 和 WakeCount 加载到观察窗口

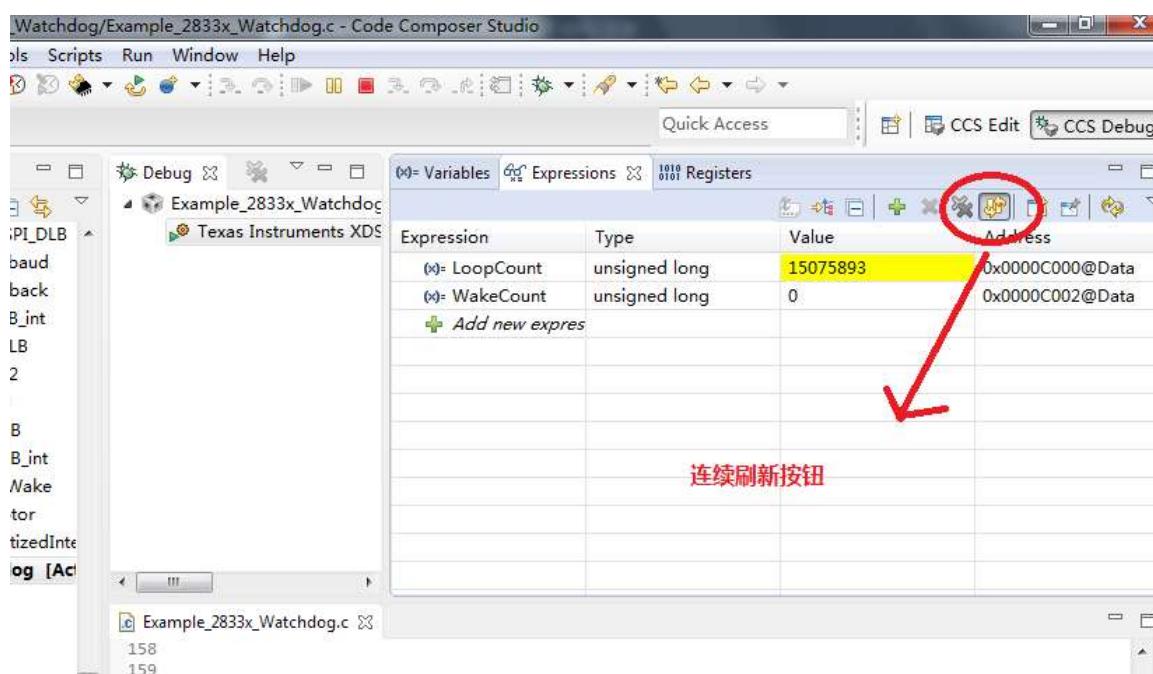


```

158
159
160 // Step 7. Insert all local Interrupt Service Routines (ISRs) and functions here:
161 // If local ISRs are used, reassign vector addresses in vector table as
162 // shown in Step 5
163
164 interrupt void wakeint_isr(void)
165 {
166     WakeCount++;
167
168     // Acknowledge this interrupt to get more from group 1
169     PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
170 }
171
172 //=====
173 // No more.

```

观察两个变量:WakeCount(复位次数)计数器没有进行加 1. 说明没有进入看门狗中断;



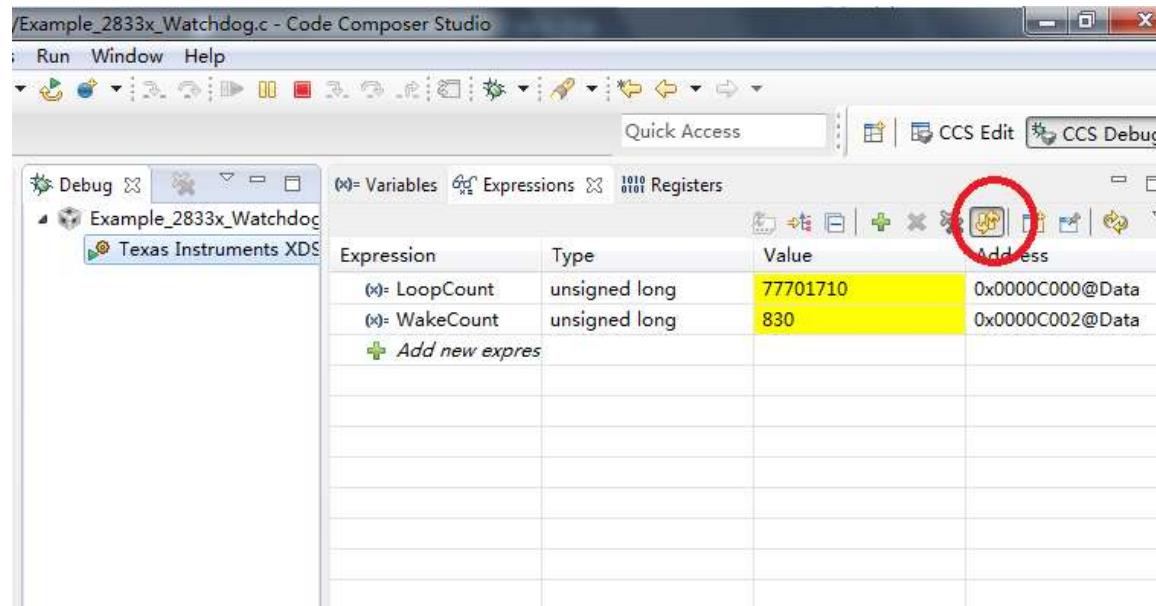
情况 2:

```

for(;;)
{
    LoopCount++;

    // Uncomment ServiceDog to just loop here
    // Comment ServiceDog to take the WAKEINT instead
    // ServiceDog(); //喂狗操作, 防止狗咬(复位), 注视掉喂狗函数
}

```

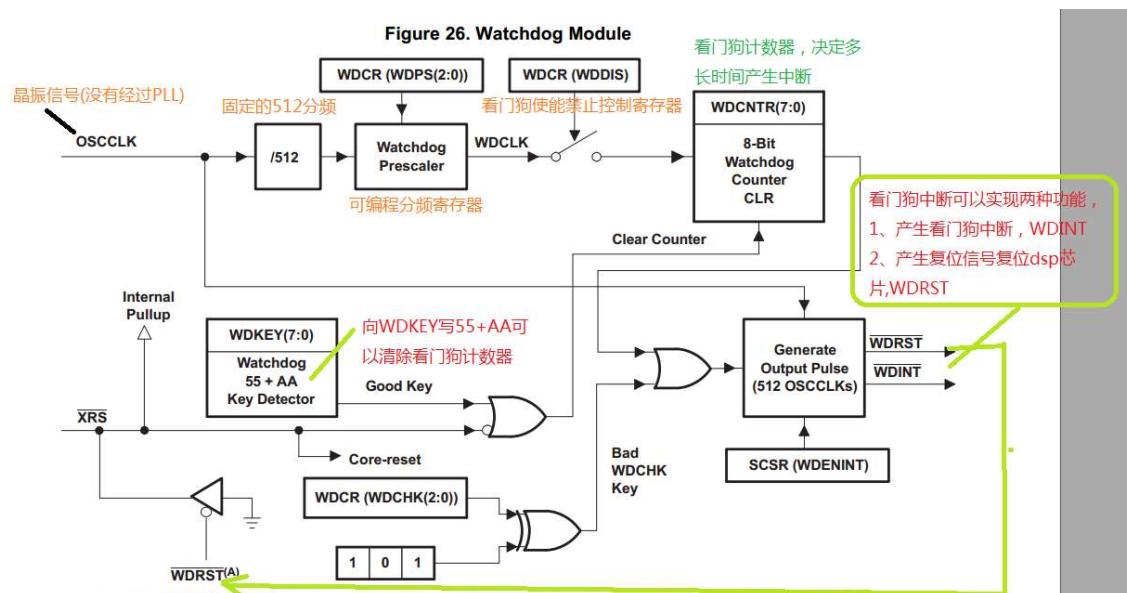


从图中可以看到 WakeCount 计数器已经不再是 0, 说明被狗咬了(复位了)

看门狗功能简介

由于看门狗的时钟不经过 PLL。而是直接由晶振提供的。所以不受 PLL 控制。如果中断条件满足则会产生一个 $512 * \text{OSCCLKs}$ 时间长度的中断信号。看门狗中断可以实现两种功能，一种是简单的进入看门狗中断，但是不复位 DSP 芯片。第二种是产生复位信号拉低 XRS (复位管脚)。

通过公式: $512 * \text{OSCCLKs} = 512 * 1 / 30000000$ 我们可以得知只要我们能提供这么长的一个低脉冲信号 DSP 就可以实现复位，但是为了留有余量，一般会比这个时间大一些;



程序解析

```
// 使能看门狗
EALLOW;
```

```

SysCtrlRegs.WDCR = 0x0028;
EDIS;
for(;;)
{
    LoopCount++;

    //是否喂狗,注视掉就不喂狗,狗饿了就要进入看门狗中断
    // ServiceDog();
}

```

实验 7：利用 XINTF 操作 SRAM 实验（Example_2833xCodeRunFromXintf）

一 实验目的：

- ✧ 了解 DSP28335 的 Flash 启动方式；
- ✧ 了解 DSP28335 的 Flash 烧写方法；

二 实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板一套；

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开；找到片外 Sram。如下图所示。从图中可知 XRDn 为读控制信号、XWEn 为写控制信号、R_CSn 为片选信号。

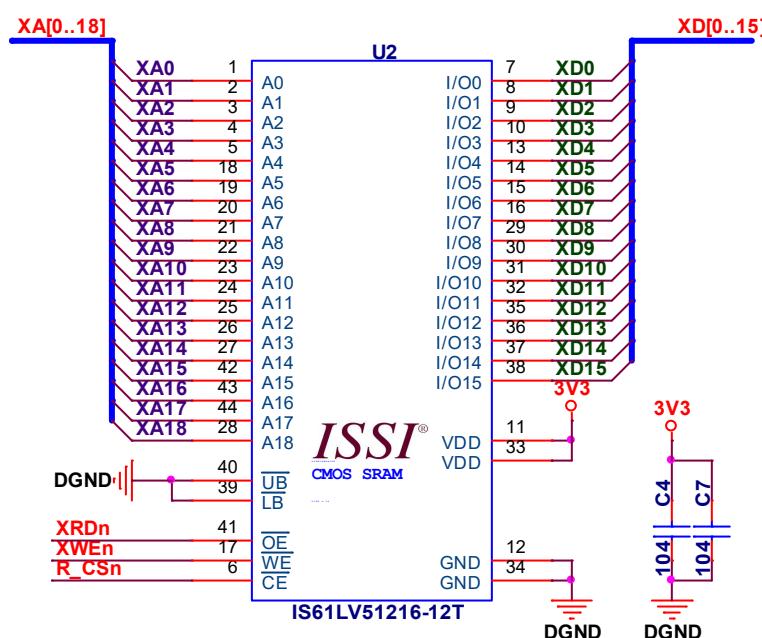


图 1 外部 SRAM 接口图

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：然后单击图中红色的方框处的调试按钮，进行调试。

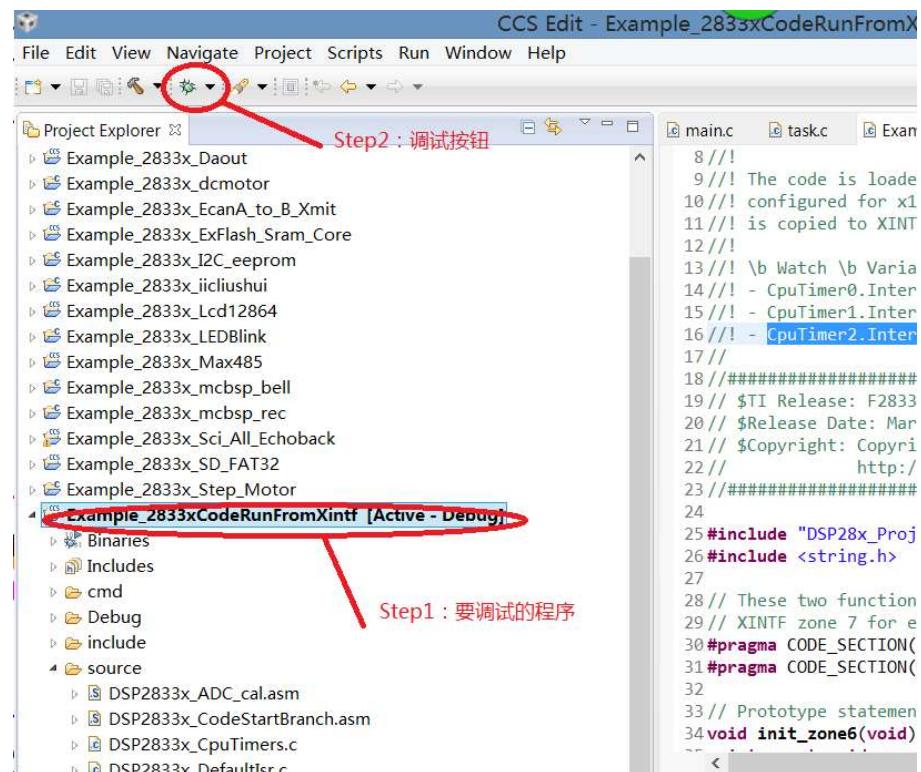


图 2 调试方法

◇ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

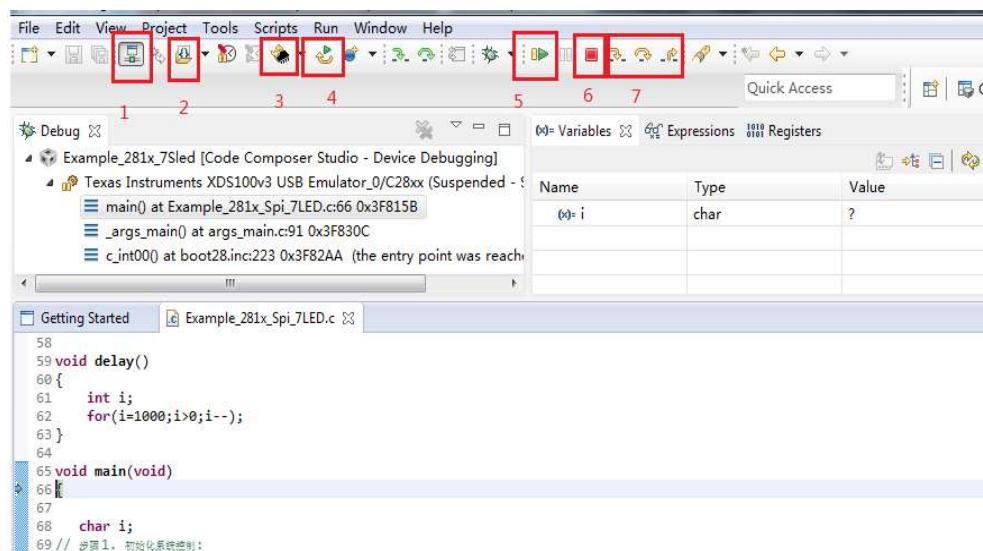


图 3 调试界面

- ① 图中 1 图标是用来进行与开发板进行连接的按钮；
- ② 图中 2 是用来下载 Debug 文件下的.out 文件的
- ③ 图中 3 是 C P U 软 R e s e t；
- ④ 图中 4 是调试时恢复到程序的开始处。
- ⑤ 图中 5 是全速运行；
- ⑥ 图中 6 是停止调试；
- ⑦ 图中 7 是用于单步调试的；

四 实验现象

首先将用于中断计数的 CpuTimer0. InterruptCount、CpuTimer1. InterruptCount 和 CpuTimer2. InterruptCount 添加到观察窗口中。并设置窗口为连续刷新。开始全速运行，可以看到这三个计数器会累加计数。



图 5 添加变量界面

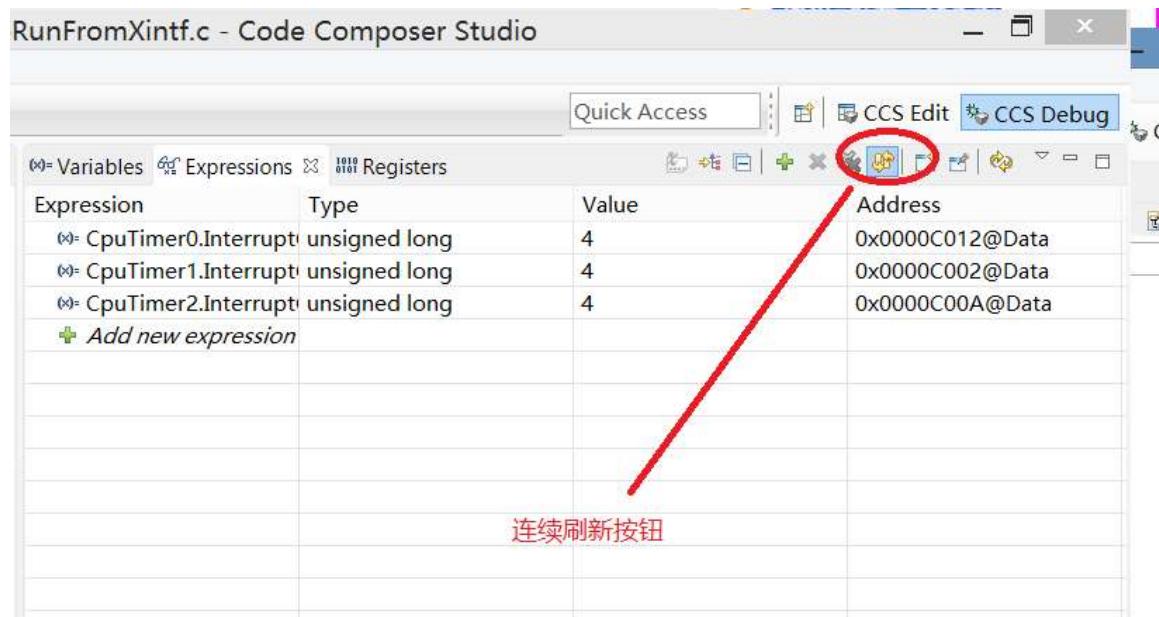


图 6 连续刷新按钮的设置

五 程序解析

◇ 下面是 28335_RAM_xintf_lnk.cmd 的内容

```
xintffuncs      : LOAD = RAML1,
                  RUN = ZONE6,
                  LOAD_START(_XintffuncsLoadStart),
                  LOAD_END(_XintffuncsLoadEnd),
                  RUN_START(_XintffuncsRunStart),
                  LOAD_SIZE(_XintffuncsLoadSize),
                  PAGE = 0
```

这段程序的作用是将程序加载到片内 RAML1 空间中，运行时搬移到片外 ZONE6 去执行。`_XintffuncsLoadStart` 为变量加载的起始地址、`_XintffuncsLoadEnd` 变

量加载的结束地址、_XintffuncsRunStart 运行的起始地址、_XintffuncsLoadSize 加载代码的大小。这四个变量都是 CCS 自己算的。我们不用关心这些。

主函数一开始用 `#pragma CODE_SECTION` 将中断函数 `cpu_timer0_isr` 和 `cpu_timer1_isr` 指定到片外 SRam 空间。

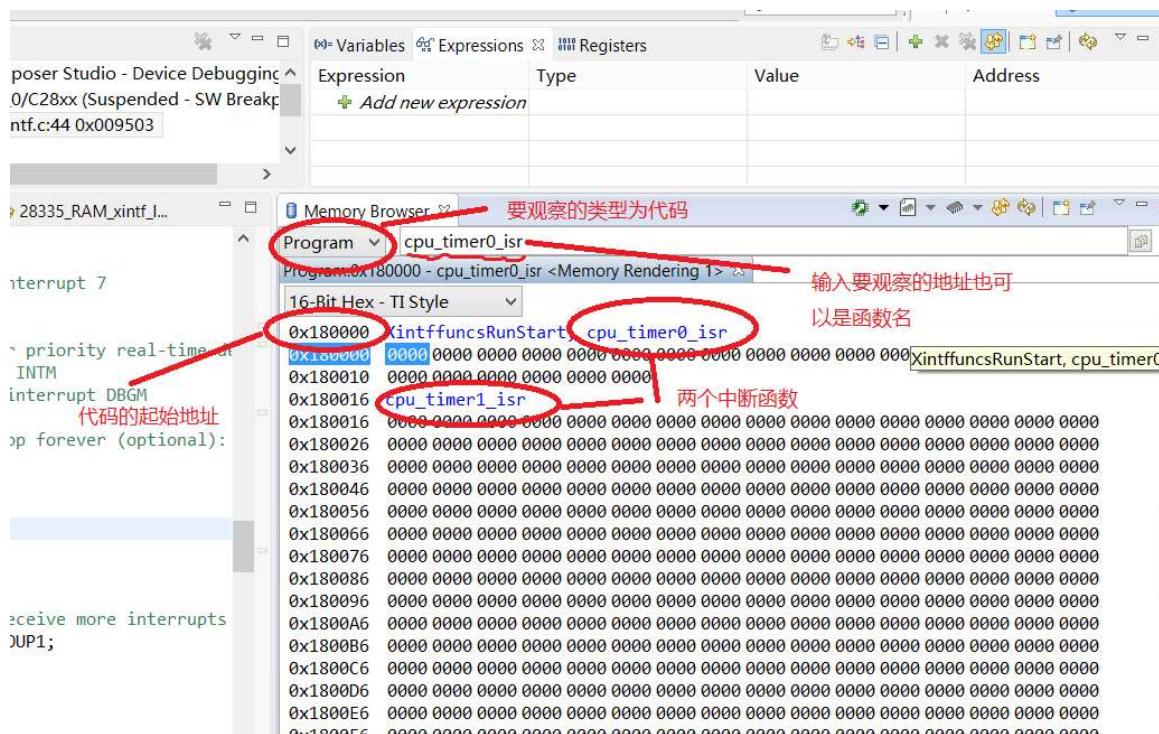
```
#pragma CODE_SECTION(cpu_timer0_isr, "xintffuncs");
#pragma CODE_SECTION(cpu_timer1_isr, "xintffuncs");
```

主函数中

```
// 首先要初始化访问 zone6 空间的时序
init_zone6();

// 开始 COPY 代码到片外
memcpy(&XintffuncsRunStart, &XintffuncsLoadStart,
(Uint32)&XintffuncsLoadSize);
```

也可以通过 View->Memory Browser 查看 `cpu_timer0_isr` 和 `cpu_timer1_isr` 所在空间的地址。



实验 8：外部 FLASH 及 SRAM 实验 (Example_2833x_ExFlash_Sram_Core)

一 实验目的：

- ✧ 了解 DSP28335 的 XINTF 外设的使用；
- ✧ 了解 SST39VF800A 芯片的总线时序和读写命令；
- ✧ 了解 IS61LV51216-12T 芯片的总线时序和读写命令；

二 实验设备

- ◆ PC 机一台；
- ◆ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ◆ SXD28335 开发板一套；

三 实验步骤

- ◆ 首先将 CCS6.0 开发环境打开；找到片外 Sram。如下图所示。从图中可知 XRDn 为读控制信号、XWEn 为写控制信号、R_CSsn 为片选信号。

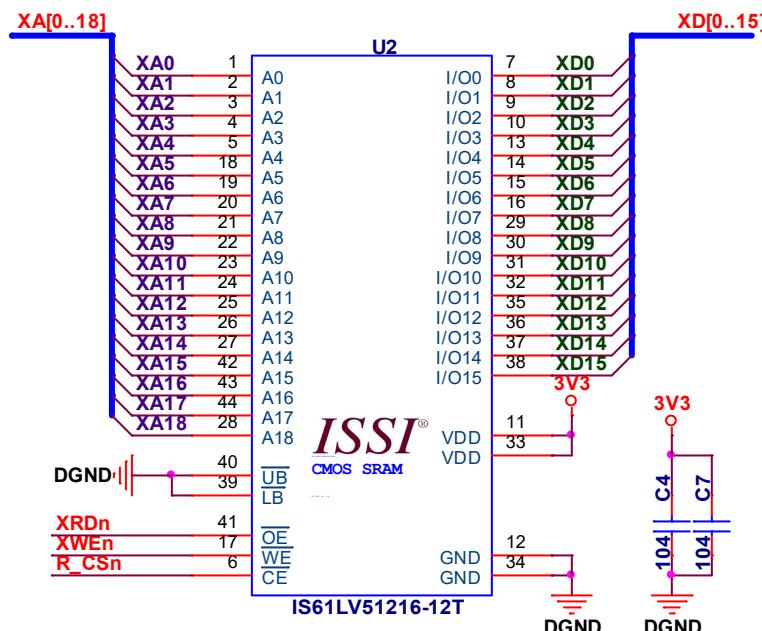


图 1 外部 SRAM 接口图

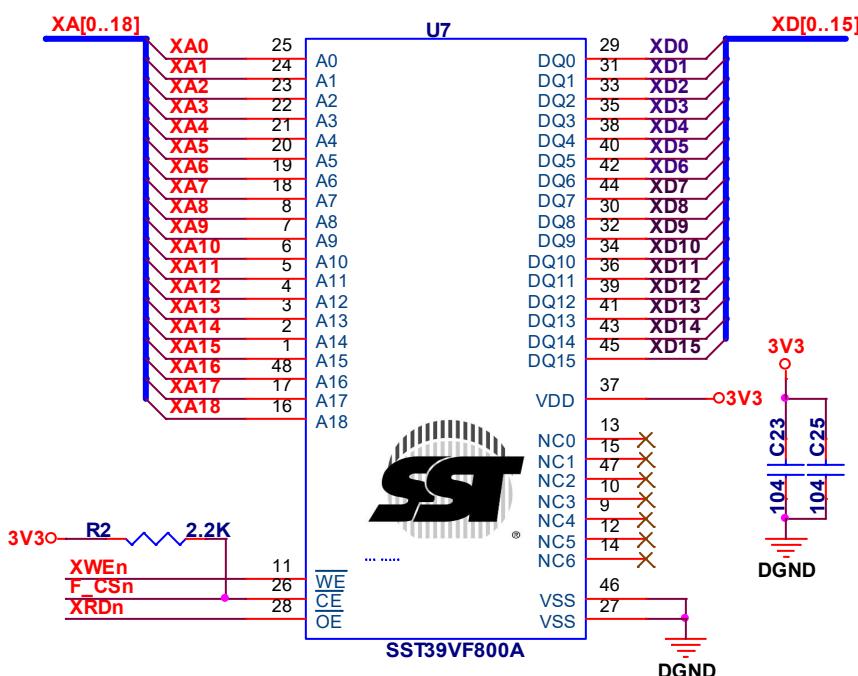


图 2 外部 Flash 接口图

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。

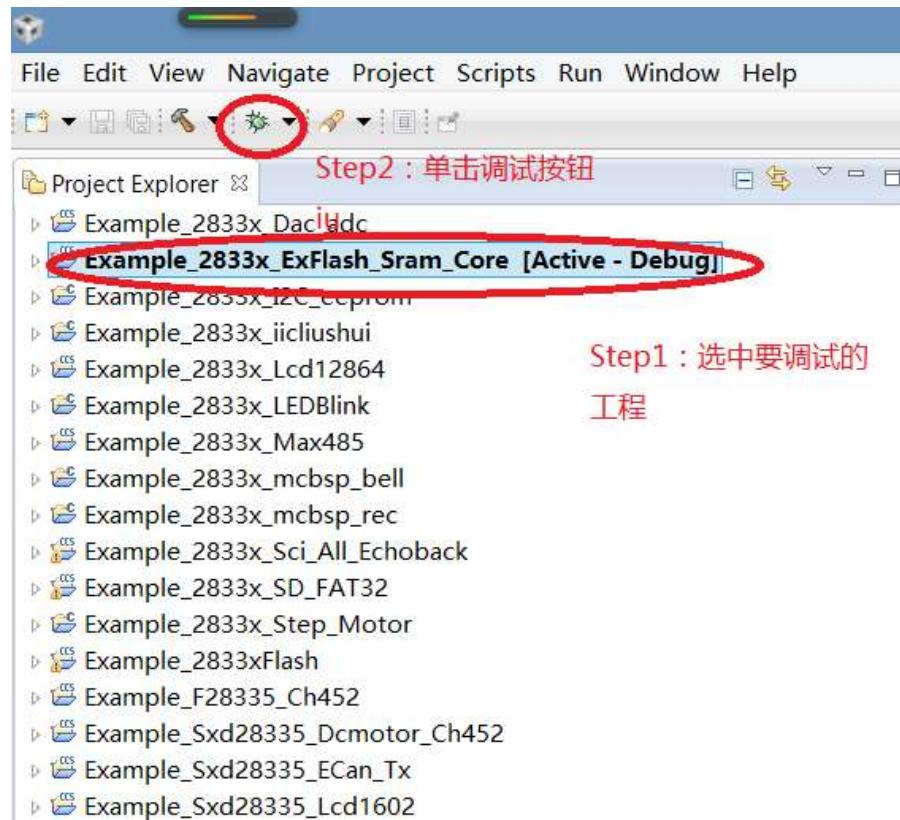


图 3 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

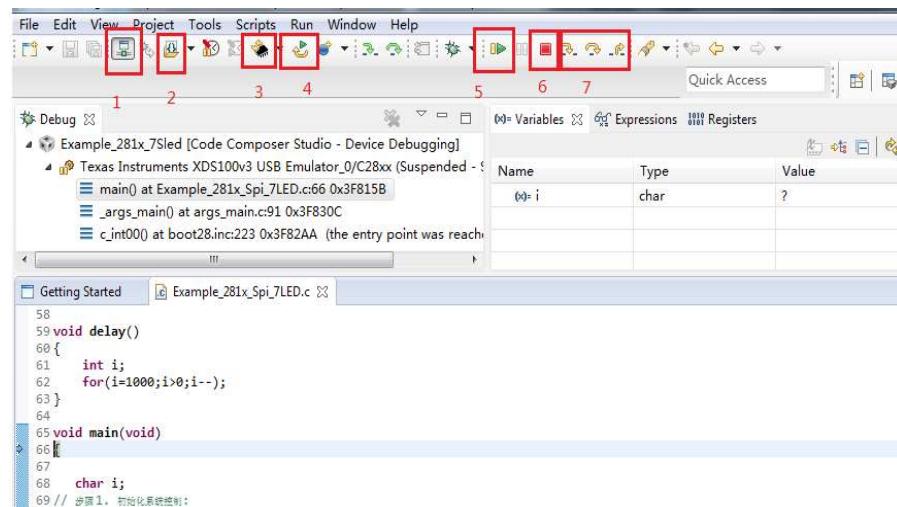


图 4 调试界面

- (8) 图中 1 图标是用来进行与开发板进行连接的按钮；
- (9) 图中 2 是用来下载 Debug 文件下的.out 文件的；
- (10) 图中 3 是 C P U 软 R e s e t；
- 11 图中 4 是调试时恢复到程序的开始处。
- 12 图中 5 是全速运行；
- 13 图中 6 是停止调试；
- 14 图中 7 是用于单步调试的；

- ◆ 断点的设置方法是：在蓝色条框上双击即可加上断点。加上断点后会出现如下图中椭圆里的小点。

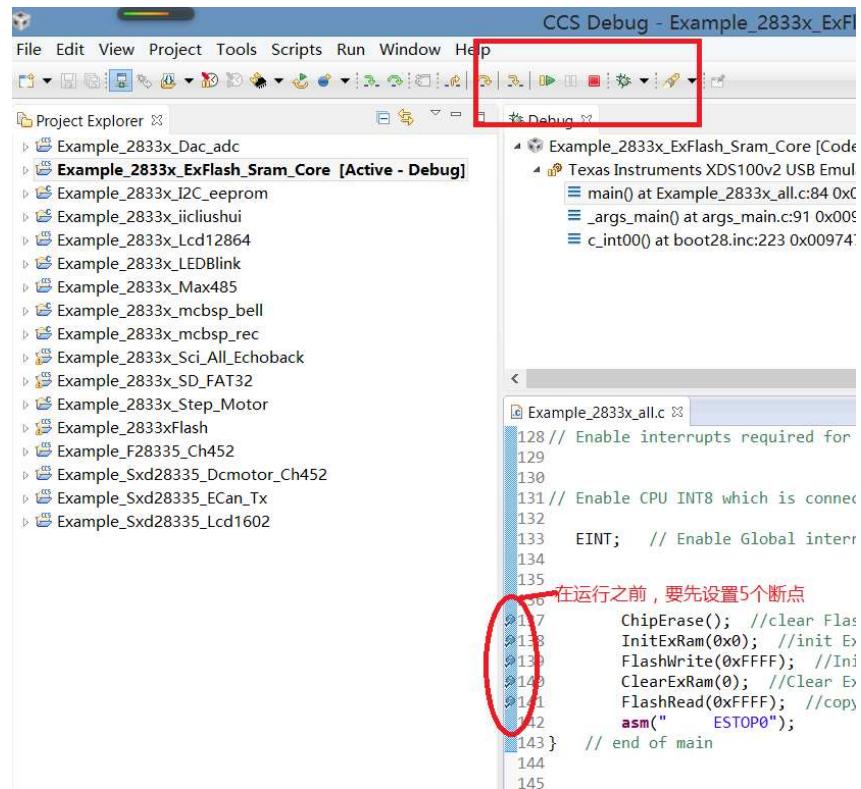


图 5 断点的设置

```

CCS Debug - Example_2833x_ExFlash_Sram_Core/39VF800a.c - Code Composer Studio
Help

Debug
Example_2833x_ExFlash_Sram_Core [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Emulator_0/C28xx (Suspended - SW Breakpoint)
main() at Example_2833x_all.c:84 0x009771
_args_main() at args_main.c:91 0x0097CC
c_int00() at boot28.inc:223 0x009747 (the entry point was reached)

Example_2833x_all.c boot28.inc 39VF800a.c
1 #include "DSP2833X_Device.h" 起始地址所在的文件名
2
3 // Definitions for the SST 39VF400A part
4 #define SST_ID 0x00BF /* SST Manufacturer's ID code */
5 #define SST_39VF800A 0x2780 /* SST39VF800/SST39VF800A device code */
6 #define TimeOutErr 1
7 #define VerifyErr 2
8 #define WriteOK 0
9 #define EraseErr 3
10 #define EraseOK 0
11 #define SectorSize 0x800
12 #define BlockSize 0x8000
13 unsigned int *FlashStart = (unsigned int *)0x00100000;//flash的首地址
14 unsigned int *ExRamStart = (unsigned int *)0x00180000;//sram的首地址
15
16 Uint16 SectorErase(Uint16 SectorNum)
17{
18     Uint16 i,Data;

```

图 6 片外存储器起始地址

ChipErase(); //擦除 Flash, 所以我们要观察 Flash 起始地址开始的内容。如果擦除成功则出现以 Flash 为起始地址的内容为 0xffff

InitExRam(0x0); //初始化片外 Sram 的程序。可以通过 SRAM 的起始地址来看 SRam 的初始化结果

FlashWrite(0xFFFF); //向片外 Flash 写数据, 所以我们要观察 Flash 起始地址开始的内容

ClearExRam(0); //清除片外 SRam 为 0. 所以我们要观察 Sram 起始地址开始的内容
FlashRead(0xFFFF); //从片外 Flash 里 copy 内容到片外 Sram 中, 这时片外 Flash 和 Sram 内容一样

观察存储器内容的方法是:

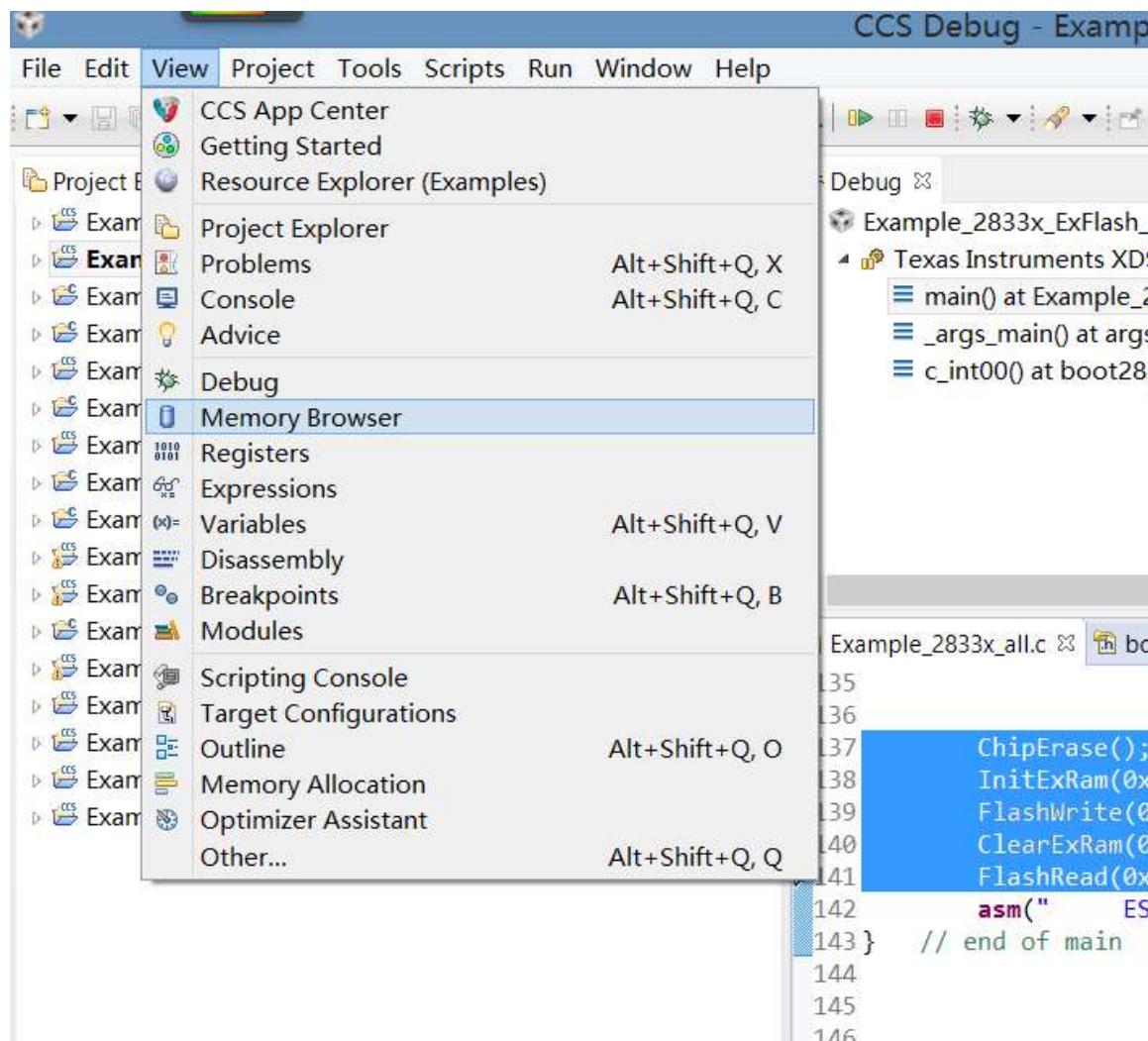


图 7 打开 Memory Browser

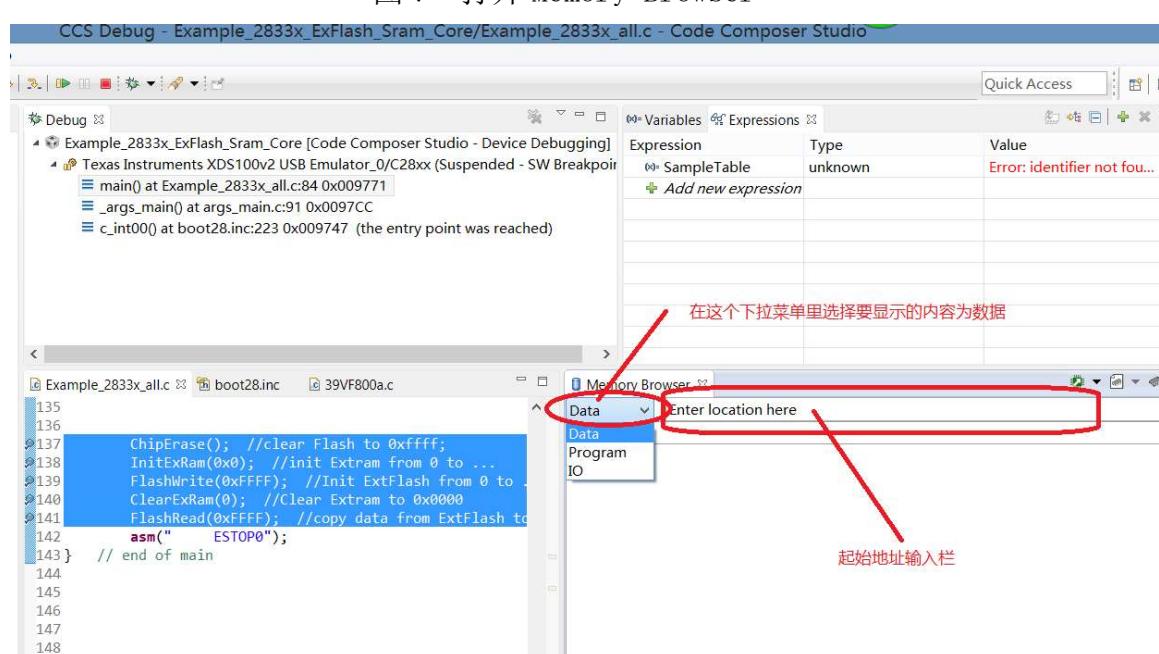


图 8 Memory Browser 界面

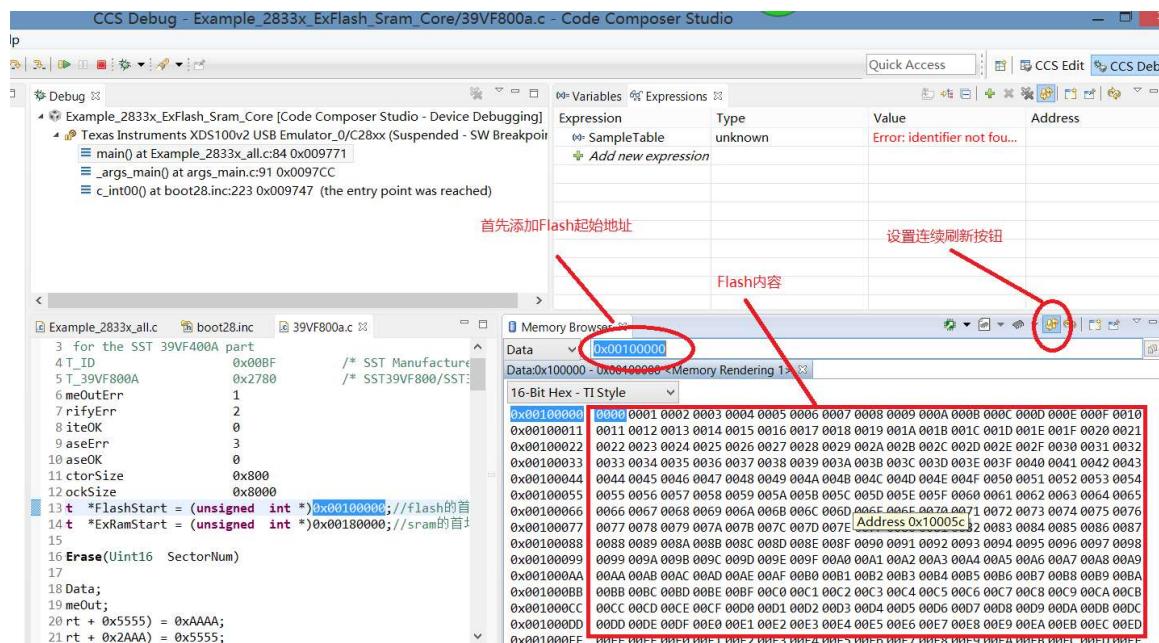


图 9 显示 Flash 内容

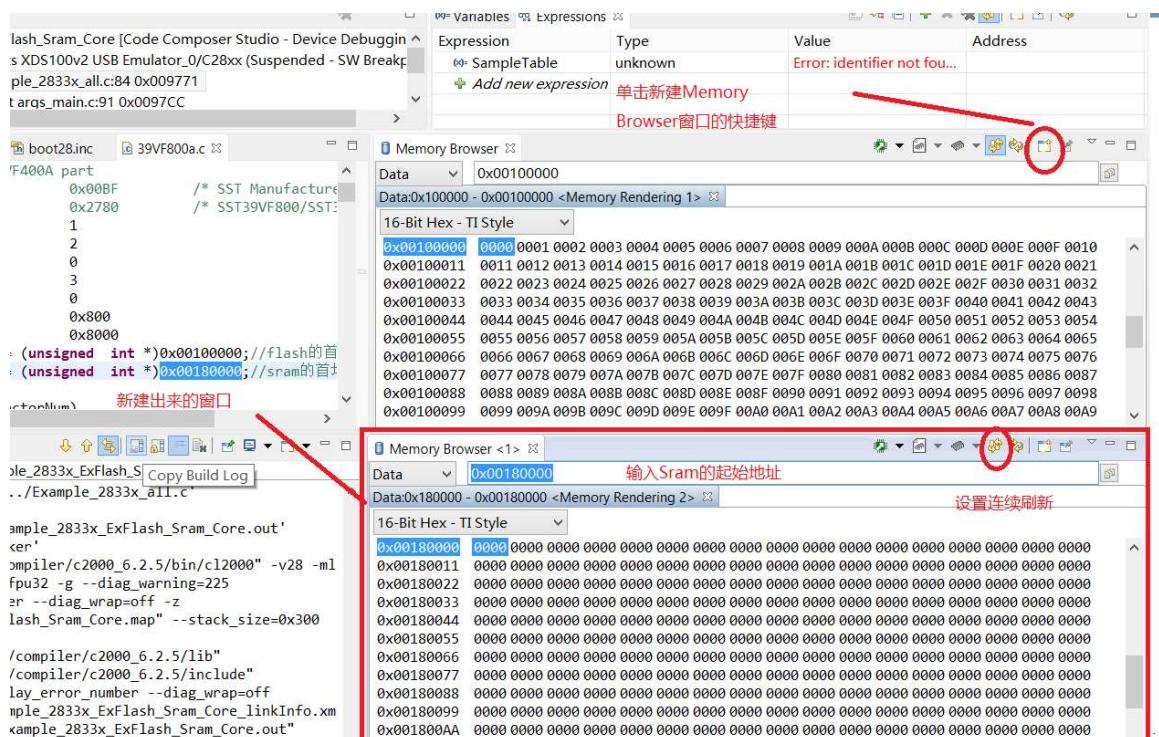


图 10 显示 Sram 内容

到目前为止实验的前期工作我们已经准备好了，接下来我们就全速运行我们的程序，看看这 5 个断点处片外 FFlash 和 Sram 的结果：

四 实验现象

实验现象就在下面这段话里，相信你们已经知道怎么看结果了：

```
ChipErase(); //擦除 Flash，所以我们要观察 Flash 起始地址开始的内容。如果擦除成功则
出现以 Flash 为起始地址的内容为 0xffff
```

InitExRam(0x0); // 初始化片外 Sram 的程序。可以通过 SRAM 的起始地址来看 SRam 的初始化结果

FlashWrite(0xFFFF); // 向片外 Flash 写数据，所以我们要观察 Flash 起始地址开始的内容

ClearExRam(0); // 清除片外 SRam 为 0. 所以我们要观察 Sram 起始地址开始的内容

FlashRead(0xFFFF); // 从片外 Flash 里 copy 内容到片外 Sram 中，这时片外 Flash 和 Sram 内容一样

五 程序解析

InitXintf16Gpio(); // 初始化 IO 口为 XINTF 口

由于没有初始化 XINTF 的时序，这时会用 CPU 上电时默认的配置，也就是 XINTF 工作在最慢的条件下：

此实验也需要对 DSP 的总线进行配置，详细配置过程请参照本实验。此实验需要重点了解的是 FLASH 的擦除、写、读操作。FLASH 的操作相对 SDRAM 的操作要复杂一些，其操作过程及命令如下图所示：

Command Sequence	1st Bus Write Cycle		2nd Bus Write Cycle		3rd Bus Write Cycle		4th Bus Write Cycle		5th Bus Write Cycle		6th Bus Write Cycle	
	Addr ⁽¹⁾	Data	Addr ⁽¹⁾	Data								
Word-Program	5555H	AAH	2AAA	55H	5555H	A0H	WA ⁽³⁾	Data				
Sector-Erase	5555H	AAH	2AAA	55H	5555H	80H	5555H	AAH	2AAA	55H	SA _x ⁽²⁾	30H
Block-Erase	5555H	AAH	2AAA	55H	5555H	80H	5555H	AAH	2AAA	55H	BA _x ⁽²⁾	50H
Chip-Erase	5555H	AAH	2AAA	55H	5555H	80H	5555H	AAH	2AAA	55H	5555H	10H
Software ID Entry	5555H	AAH	2AAA	55H	5555H	90H						
CFI Query Entry	5555H	AAH	2AAA	55H	5555H	98H						
Software ID Exit/CFI Exit	XXH	F0H										
Software ID Exit/CFI Exit	5555H	AAH	2AAA	55H	5555H	F0H						

342 PGM T4.0

Notes: (1) Address format A₁₄-A₀ (Hex), Addresses A₁₅, A₁₆, and A₁₇ are "Don't Care" for Command sequence.

(2) SA_x for Sector-Erase; uses A₁₇-A₁₁ address lines

BA_x, for Block-Erase; uses A₁₇-A₁₅ address lines

(3) WA = Program word address

(4) Both Software ID Exit operations are equivalent

(5) DQ₁₅ - DQ₈ are "Don't Care" for Command sequence

Notes for Software ID Entry Command Sequence

- With A₁₇-A₁=0; SST Manufacturer Code = 00BFH, is read with A₀ = 0,
SST39VF400 Device Code = 2780H, is read with A₀ = 1.
- The device does not remain in Software Product ID Mode if powered down.

下面通过程序介绍 FLASH 的擦除、写、读操作过程（其详细原理需要参照其数据手册）。

FLASH 擦除程序：

```
Uint16 ChipErase(void)
```

```
{
```

```
Uint16 Data;
```

```
Uint32 TimeOut, i;
```

以下过程需要严格遵守

```
* (FlashStart + 0x5555) = 0xAAAA; // 需要对 FLASH 的 0x5555 单元写 0xAAAA
```

```
* (FlashStart + 0x2AAA) = 0x5555; // 需要对 FLASH 的 0x2AAA 单元写 0x5555
```

```
*(FlashStart + 0x5555) = 0x8080; // 随后对 FLASH 的 0x5555 单元写 0x8080;
*(FlashStart + 0x5555) = 0xAAAA; // 之后对 FLASH 的 0x5555 单元写 0xAAAA
*(FlashStart + 0x2AAA) = 0x5555; // 需要对 FLASH 的 0x2AAA 单元写 0x5555
*(FlashStart + 0x5555) = 0x1010; // 需要对 FLASH 的 0x5555 单元写 0x1010
i = 0;
TimeOut = 0;
while(i<5)
{
Data = *(FlashStart + 0x3FFF);
if (Data == 0xFFFF)
i++;
else
i=0;
if ( ++TimeOut>0x1000000)
return (TimeOutErr);
}
for (i=0;i<0x40000;i++)
{
Data = *(FlashStart + i);
if (Data !=0xFFFF)
return (EraseErr);
}
return (EraseOK);
以上部分检测 FLASH 是否擦除正确，正确的话返回 EraseOK.；否则返回 EraseErr，标明
擦除失败
}
```

FLASH 写操作程序：

```
Uint16 FlashWrite(Uint32 RamStart, Uint32 RomStart, Uint16 Length)
```

FLASH 写函数里面有 3 个参数，分别是源地址、目的地址、所传地址长度

```
{
Uint32 i,TimeOut;
Uint16 Data1,Data2, j;
for (i=0;i<Length;i++)
{
以下 3 行过程需要严格遵守
*(FlashStart + 0x5555) = 0x00AA;
*(FlashStart + 0x2AAA) = 0x0055;
*(FlashStart + 0x5555) = 0x00A0;
*(FlashStart + RomStart + i) = *(ExRamStart + RamStart + i);
将源地址（SDRAM）数据送给目的地址（FLASH）各个单元
TimeOut = 0;
j=0;
while(j<5)
```

```
{  
Data1 = *(FlashStart + RomStart + i);  
Data2 = *(FlashStart + RomStart + i);  
if (Data1 == Data2) j++;  
else j=0;  
if ( ++TimeOut>0x1000000) return  
(TimeOutErr);  
}  
}  
for (i=0;i<Length;i++)  
{  
Data1 = *(FlashStart + RomStart +i);  
Data2 = *(ExRamStart + RamStart +i);  
if (Data1 != Data2) return (VerifyErr);  
}  
return (WriteOK);  
以上部分同样是检测 FLASH 写入的数据和读出的数据是否一样，一样的话返回 WriteOK，  
否则返回 VerifyErr，标明操作失败  
}
```

FLASH 的读操作比较简单，和 SDRAM 一样，具体程序如下所示：

```
void FlashRead(Uint32 RamStart, Uint32 RomStart, Uint16 Length)  
{  
Uint32 i;  
Uint16 Temp;  
for (i=0;i<Length;i++)  
{  
Temp = *(FlashStart + RomStart +i);  
*(ExRamStart + RamStart +i) = Temp;  
}  
}
```

实验 9：LCD1602 液晶屏实验（Example_2833x_lcd1602）

一 实验目的：

- ✧ 了解 LCD1602 总线时序；
- ✧ 了解如何使用 XINTF；

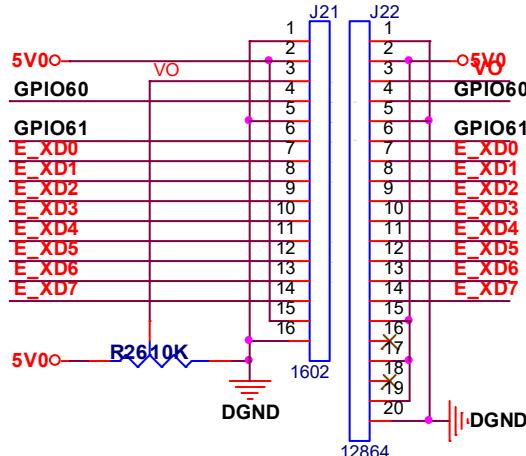
二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套, 1602 液晶屏一块；

三 实验步骤：

首先，我们从图中可以看到要控制 LCD1602 首先要了解 LCD1602 的控制时序图。

(注: GPIO60-->对应的是 LCD 的 RS、GPIO61-->对应的是 LCD 1 6 0 2 的 EN, LCD1602 的 RW 信号被固定为低电平, 只向 LCD1602 写数据, 而不读数据)。由于三兄弟选用的是 5V 的 LCD1602 液晶屏。所以要将 3V3 的 IO 口电压转换为 5V 的电压来驱动 LCD1602。



- ◇ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处 (注意仿真器插入方向, 请仔细核对防差错针的位置, JTAG 接口是用来连接仿真器的。开发板上对应的是 1~4 针的座子, 不要用开发板上 2~0 针或 1~8 针的座子连接仿真器) ;
- ◇ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示: 然后单击图中红色的方框处的调试按钮, 进行调试。

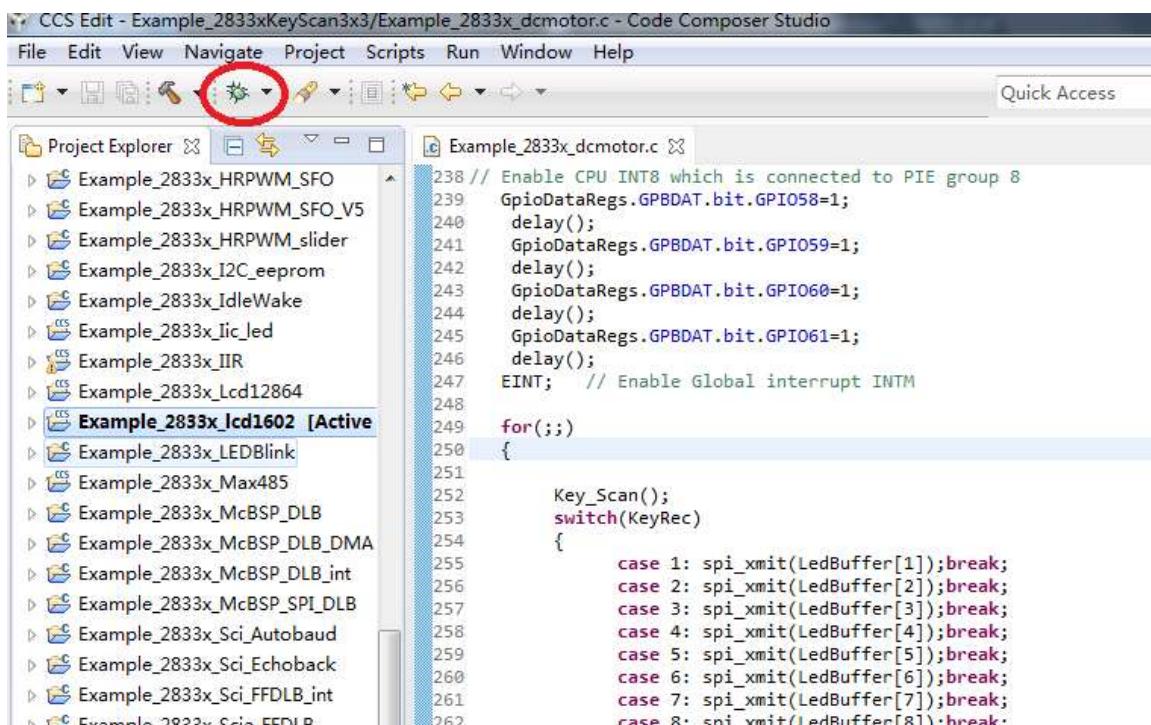


图 2 调试方法

- ◇ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

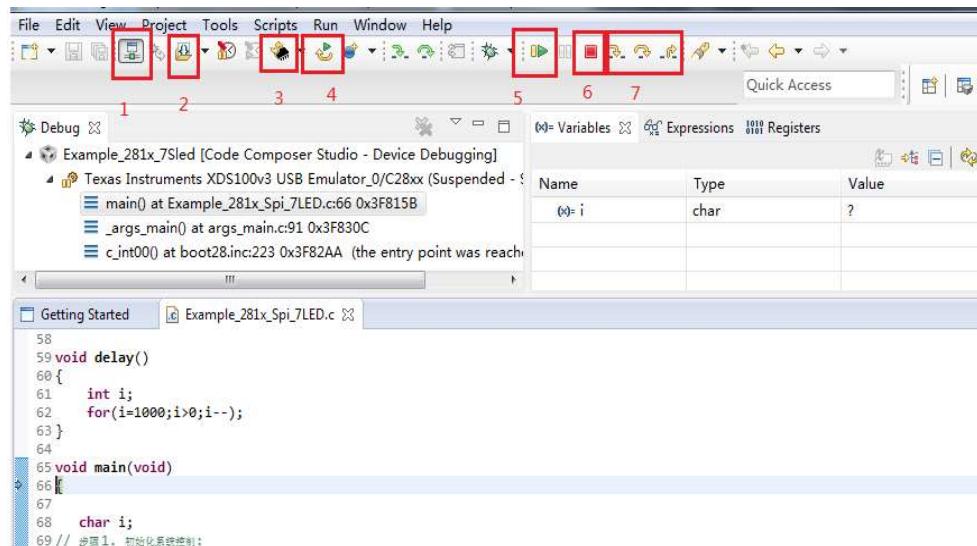


图 3 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的；

图中 3 是 C P U 软 R e s e t；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行。

四 试验现象：

L c d 1 6 0 2 液晶屏上会显示两行英文：

Sxdembed

Matlab sysbios

如果液晶屏没有显示，可以用螺丝刀旋转滑动变阻器 R26，来调节液晶屏的背光。

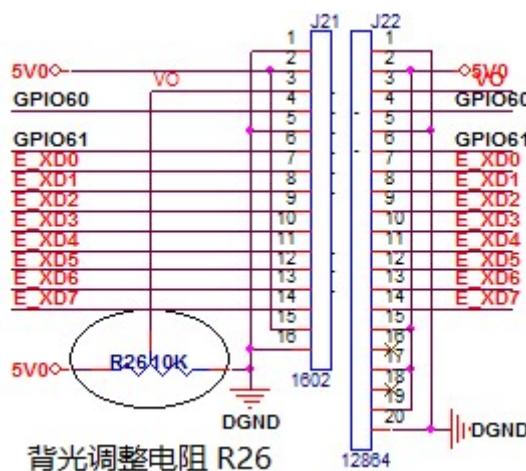
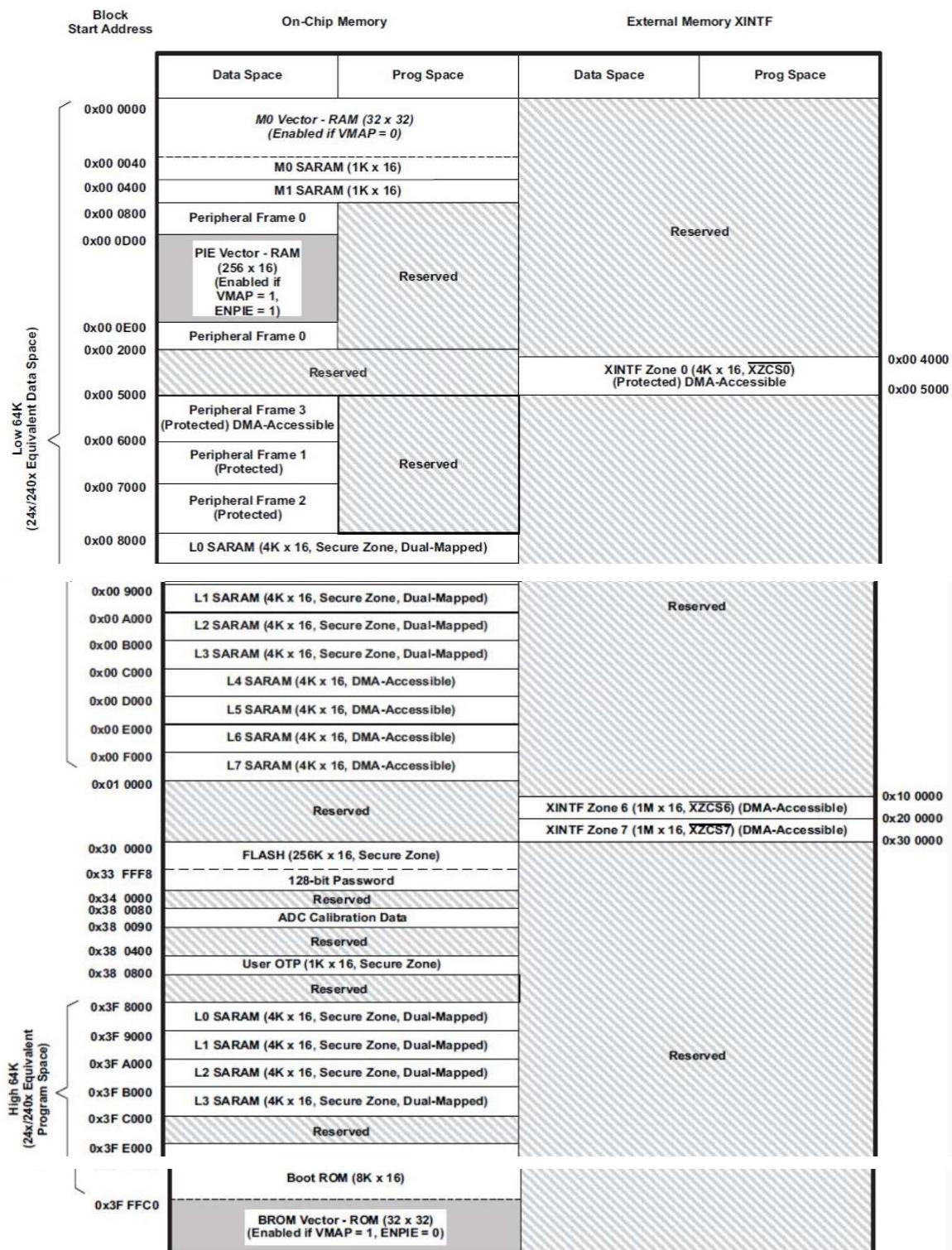


图 4 背光调节

程序解析

XINTF 简介：



TMS320F28335 为哈佛结构的 DSP，哈佛结构是一种将程序指令存储和数据存储分开的存储器结构。哈佛结构是一种并行体系结构，其主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。其存储空间映射如上图所示：

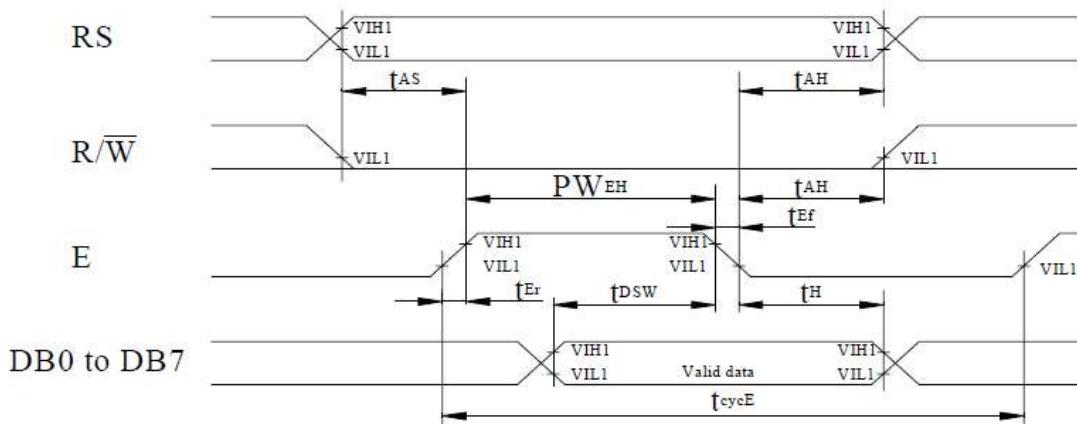
TMS320F28335 片上有 256K×16 位 FLASH 存储器，34K×16 位单周期单次访问随机存储器的 SARAM、8K×16 位的 BOOT ROM、1K×16 位的 OTP ROM。

各个分区的空间分配：

Zone0 存储区域： 0X004000—0X004FFF, 4K×16 位 可编程 最少一个等待周期
 Zone6 存储区域： 0X100000—0X1FFFFF, 1M×16 位 10ns 最少一个等待周期
 Zone7 存储区域： 0X200000—0X2FFFFF, 1M×16 位 70ns 最少一个等待周期

LCD1602 的写时序图：

13.1 Write Operation



首先通过 RS 的高低来控制要发送的数据是命令还是数据。R/W 是用来控制是写还是读。在 E 的下降沿时将数据输出到 LCD1602 液晶屏里。

```
//#####
// 释放日期: 2013.1.17
//#####

#include "DSP2833x_Device.h"      // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"     // DSP2833x Examples Include File
#define LCD_DATA (*((volatile Uint16 *)0x2c0000)) //片选信号的地址
#define EN   GpioDataRegs.GPBDAT.bit.GPIO61
#define RW   GpioDataRegs.GPADAT.bit.GPIO27
#define RS   GpioDataRegs.GPBDAT.bit.GPIO60
#define LOW  0
#define HIGH 1
#define CLEAR_SCREEN   0x01      //清屏指令: 清屏且 AC 值为 00H
#define AC_INIT        0x02      //将 AC 设置为 00H. 且游标移到原点位置
#define CURSE_ADD      0x06      //设定游标移到方向及图像整体移动方向 (默认游标右移, 图像整体不动)
#define FUN_MODE        0x30      //工作模式: 8 位基本指令集
#define DISPLAY_ON      0x0c      //显示开, 显示游标, 且游标位置反白
#define DISPLAY_OFF     0x08      //显示关
#define CURSE_DIR       0x14      //游标向右移动:AC=AC+1
#define SET(CG)_AC      0x40      //设置 AC, 范围为: 00H~3FH
#define SET(DD)_AC      0x80
```

```
extern uchar A[]={“ f28335 dsp ”};  
extern uchar B[]={“matlab sysbios”};  
void LcdInit();  
void WriteCmd12864(Uint16 cmd);  
void WriteData12864(Uint16 dat);  
void delay (Uint16 t);  
void configtestled(void);  
void InitXintf(void);  
void DisplayCgrom(uchar *hz);  
Uint16 p, x,y;  
//unsigned Uint16 B[]={0xb3,0x34};  
void main(void)  
{  
    // Step 1. Initialize System Control:  
    // PLL, WatchDog, enable Peripheral Clocks  
    // This example function is found in the DSP2833x_SysCtrl.c file.  
    InitSysCtrl();  
    InitXintf();  
    // Step 2. Initialize GPIO:  
    // This example function is found in the DSP2833x_Gpio.c file and  
    // illustrates how to set the GPIO to it's default state.  
    // InitGpio(); // Skipped for this example  
    InitXintf16Gpio(); //zq  
    // Step 3. Clear all interrupts and initialize PIE vector table:  
    // Disable CPU interrupts  
    DINT;  
    // Initialize the PIE control registers to their default state.  
    // The default state is all PIE interrupts disabled and flags  
    // are cleared.  
    // This function is found in the DSP2833x_PieCtrl.c file.  
    InitPieCtrl();  
    // Disable CPU interrupts and clear all CPU interrupt flags:  
    IER = 0x0000;  
    IFR = 0x0000;  
    // InitPieVectTable();  
    configtestled();  
    RS=LOW;  
    delay(5);  
    RW=LOW;  
    delay(5);  
    EN=LOW;  
    LcdInit();  
    RS=LOW;  
    delay(5);
```

```
RW=LOW;
delay(5);
EN=LOW;
LcdInit();
delay(5);
    WriteCmd12864(CLEAR_SCREEN);
    delay(50);
    WriteCmd12864(0x80);
    delay(300);
    DisplayCgrom(A);
    WriteCmd12864(0xc0);
    delay(80);
    DisplayCgrom(B);
while(1);
}

void configtestled(void)
{
EALLOW;
GpioCtrlRegs.GPBMUX2.bit.GPIO60 = 0; // GPIO60 = GPIO60
GpioCtrlRegs.GPBDIR.bit.GPIO60 = 1;
GpioCtrlRegs.GPBMUX2.bit.GPIO61 = 0; // GPIO60 = GPIO60
GpioCtrlRegs.GPBDIR.bit.GPIO61 = 1;
GpioCtrlRegs.GPAMUX2.bit.GPIO27= 0;
GpioCtrlRegs.GPADIR.bit.GPIO27 = 1;
// GpioCtrlRegs.GPBMUX2.bit.GPIO54 = 0; // GPIO60 = GPIO60
// GpioCtrlRegs.GPBDIR.bit.GPIO54 = 1;
EDIS;
}

void delay (Uint16 t)
{ Uint16 i;
while(t--)
{
for(i=0;i<125;i++)
;
}
}

void LcdInit()
{
    WriteCmd12864(0x38); //设置 8 位格式, 2 行, 5x8
DELAY_US(2000);
    WriteCmd12864(0x38); //设置 8 位格式, 2 行, 5x8
DELAY_US(2000);
    WriteCmd12864(0x0c); //关显示, 不显示光标, 光标不闪烁;
DELAY_US(2000);
}
```

```
WriteCmd12864(0x01); //清除屏幕显示：数据指针清零，所有显示清零；  
DELAY_US(50000);  
WriteCmd12864(0x06); //设定输入方式，增量不移位  
DELAY_US(2000);  
WriteCmd12864(0x0c); //整体显示，关光标，不闪烁  
DELAY_US(2000);  
#if 0  
    delay(500);  
    WriteCmd12864(FUN_MODE);  
    delay(5);  
    WriteCmd12864(FUN_MODE);  
    delay(5);  
    WriteCmd12864(DISPLAY_ON);  
    delay(5);  
    WriteCmd12864(CLEAR_SCREEN);  
    delay(6);  
    WriteCmd12864(AC_INIT);  
    delay(4);  
#endif  
}  
void WriteCmd12864(Uint16 cmd)  
{  
    // Uint16 i=5;  
    RS=LOW;  
    RW=LOW;  
    EN=HIGH;  
    LCD_DATA=cmd;  
    // while(i--);  
    EN=LOW;  
}  
void WriteData12864(Uint16 dat)  
{ // Uint16 i=5;  
    RS=HIGH;  
    RW=LOW;  
    EN=HIGH;  
    LCD_DATA=dat;  
    // while(i--);  
    EN=LOW;  
}  
void DisplayCgrom(uchar *hz)  
{  
    while(*hz != '\0')  
    {  
        WriteData12864(*hz);  
    }  
}
```

```
    hz++;
    delay(20);
}
}

#ifndef LCD_init
{
    // LCD_write_command(0x38); //设置 8 位格式, 2 行, 5x8
    // DELAY_US(5000); //延迟 5ms
    // LCD_write_command(0x38); //设置 8 位格式, 2 行, 5x8
    // DELAY_US(5000);
    LCD_write_command(0x38); //设置 8 位格式, 2 行, 5x8
    DELAY_US(2000);
    LCD_write_command(0x38); //设置 8 位格式, 2 行, 5x8
    DELAY_US(2000);
    LCD_write_command(0x0c); //关显示, 不显示光标, 光标不闪烁;
    DELAY_US(2000);
    LCD_write_command(0x01); //清除屏幕显示: 数据指针清零, 所有显示清零;
    DELAY_US(50000);
    LCD_write_command(0x06); //设定输入方式, 增量不移位
    DELAY_US(2000);
    LCD_write_command(0x0c); //整体显示, 关光标, 不闪烁
    DELAY_US(2000);
}
#endif
```

实验 10: LCD12864 液晶屏实验 (Example_2833x_Lcd12864)

一 实验目的:

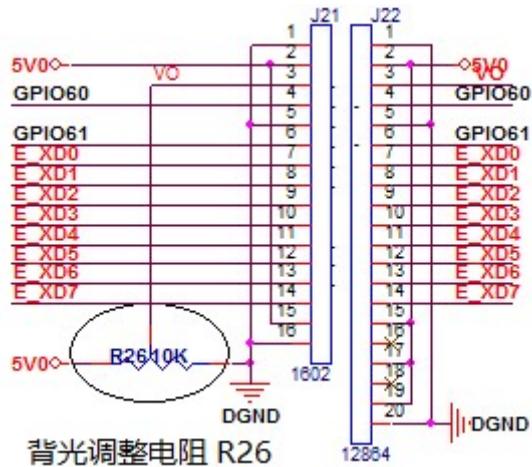
- ✧ 了解 LCD12864 屏总线时序;
- ✧ 了解如何使用 XINTF;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3 (隔离) 仿真器一套;
- ✧ SXD28335 开发板一套, 12864 液晶屏一块;

三 实验步骤:

首先, 我们从图中可以看到要控制 LCD12864 首先要了解 LCD12864 的控制时序图。 (注: GPIO60—>对应的是 LCD 的 R S、GPIO61——>对应的是 Lcd12864 的 E N, LCD12864 的 R W 信号被固定为低电平, 只向 LCD12864 写数据, 而不读数据)。



由于三兄弟选用的是 5V 的 Lcd12864 液晶屏。所以要将 3V3 的 IO 口电压转换为 5V 的电压来驱动 LCD12864。从原理图中可知用电平转换芯片是 SN74LVC4245ADW 芯片。

- ❖ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处。
- ❖ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:
然后单击图中红色的方框处的调试按钮, 进行调试。 (注: 此时认为兄弟你已经会设置配置文件, 并将其设置为默认配置)。

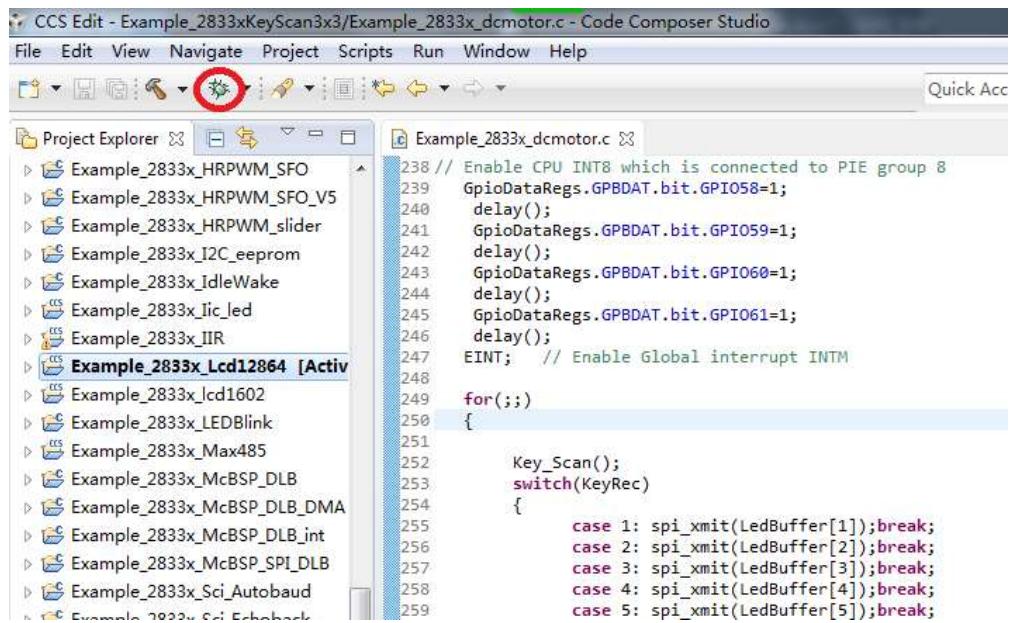


图 2 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

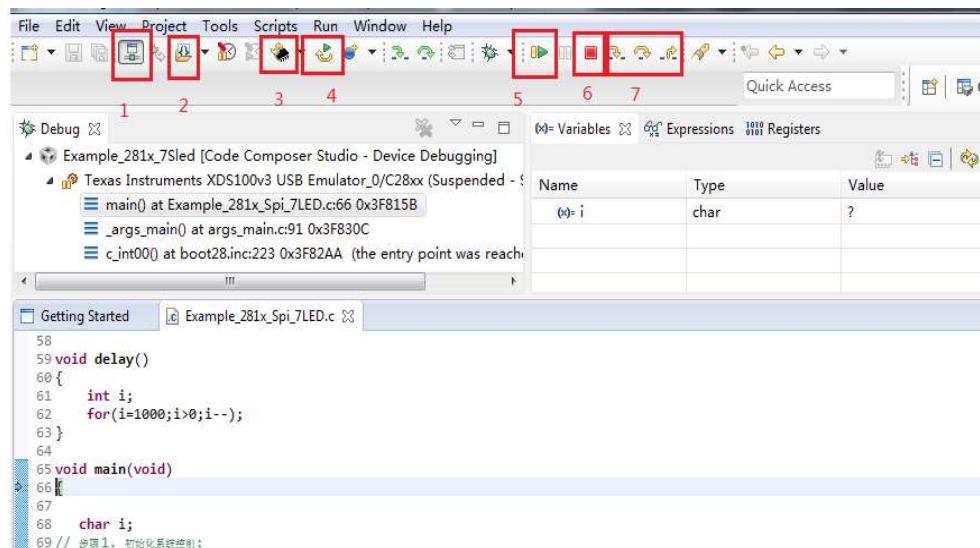


图 3 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 CPU 软 Reset；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行。

试验现象：

✧ LCD12864 液晶屏上会显示四行字母：

如果液晶屏没有显示，可以用螺丝刀旋转 12864 屏后面的滑动变阻器，来调节液晶屏的背光。

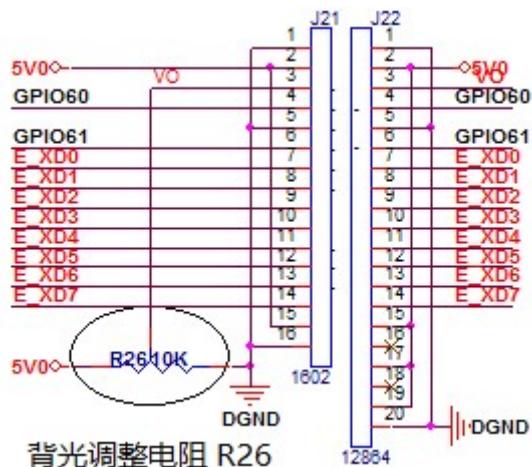
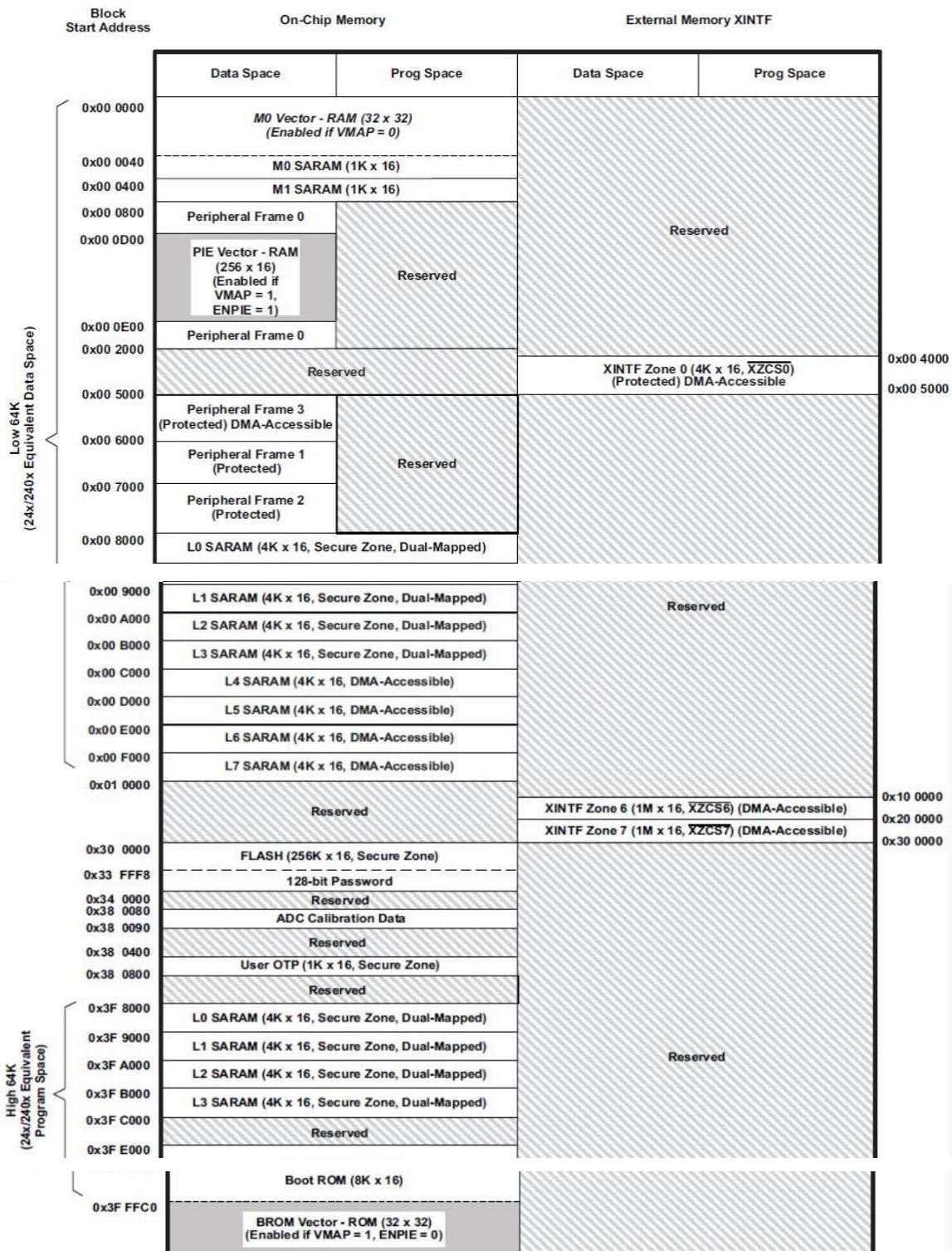


图 4 背光调节

程序解析

XINTF 简介：



TMS320F28335 为哈佛结构的 DSP，哈佛结构是一种将程序指令存储和数据存储分开的存储器结构。哈佛结构是一种并行体系结构，其主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。其存储空间映射如上图所示：

TMS320F28335 片上有 256K×16 位 FLASH 存储器，34K×16 位单周期单次访问随机存储器的 SARAM、8K×16 位的 BOOT ROM、1K×16 位的 OTP ROM。

各个分区的空间分配：

玻尔电子致力于 C2000 全系列开发平台及应用方案的推广

Zone0 存储区域: 0X004000—0X004FFF, 4K×16 位 可编程 最少一个等待周期
Zone6 存储区域: 0X100000—0X1FFFFF, 1M×16 位 10ns 最少一个等待周期
Zone7 存储区域: 0X200000—0X2FFFFF, 1M×16 位 70ns 最少一个等待周期

应用说明

用带中文字库的 128X64 显示模块时应注意以下几点:

- ①欲在某一个位置显示中文字符时, 应先设定显示字符位置, 即先设定显示地址, 再写入中文字符编码。
- ②显示 ASCII 字符过程与显示中文字符过程相同。不过在显示连续字符时, 只须设定一次显示地址, 由模块自动对地址加 1 指向下一个字符位置, 否则, 显示的字符中将会有一个空 ASCII 字符位置。
- ③当字符编码为 2 字节时, 应先写入高位字节, 再写入低位字节。

```
#####
// Original Author: D.F.
//
// $TI Release: 2833x/2823x Header Files and Peripheral Examples V133 $
// $Release Date: June 8, 2012 $
#####

#include "DSP2833x_Device.h"      // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"    // DSP2833x Examples Include File

#define LCD_DATA  (*((volatile Uint16 *)0x2c0000))
#define EN      GpioDataRegs.GPBDAT.bit.GPIO61
#define RW      GpioDataRegs.GPADAT.bit.GPIO27
#define RS      GpioDataRegs.GPBDAT.bit.GPIO60
#define LOW     0
#define HIGH    1

#define CLEAR_SCREEN   0x01      //清屏指令: 清屏且 AC 值为 00H
#define AC_INIT        0x02      //将 AC 设置为 00H。且游标移到原点位置
#define CURSE_ADD      0x06      //设定游标移到方向及图像整体移动方向(默认游标右移, 图像整体不动)

#define FUN_MODE       0x30      //工作模式: 8 位基本指令集
#define DISPLAY_ON     0x0c      //显示开, 显示游标, 且游标位置反白
#define DISPLAY_OFF    0x08      //显示关
#define CURSE_DIR      0x14      //游标向右移动:AC=AC+1
#define SET(CG)_AC     0x40      //设置 AC, 范围为: 00H~3FH
#define SET_DD_AC      0x80

extern uchar A[]={“三兄弟嵌入式”};
extern uchar B[]={“C2000 DSP”};
extern uchar C[]={“MATLAB SYSBIOS”};
extern uchar D[]={“sxdembed.taobao.com”};

void LcdInit();
void WriteCmd12864(Uint16 cmd);
void WriteData12864(Uint16 dat);
```

```
void delay (Uint16 t);
void configtestled(void);
void InitXintf(void);
void DisplayCgrom(uchar *hz);

Uint16 p, x, y;
//unsigned Uint16 B[]={0xb3, 0x34};

void main(void)
{
    // Step 1. Initialize System Control:
    // PLL, WatchDog, enable Peripheral Clocks
    // This example function is found in the DSP2833x_SysCtrl.c file.
    InitSysCtrl();
    InitXintf();

    // Step 2. Initialize GPIO:
    // This example function is found in the DSP2833x_Gpio.c file and
    // illustrates how to set the GPIO to it's default state.
    // InitGpio(); // Skipped for this example
    InitXintf16Gpio(); //zq

    // Step 3. Clear all interrupts and initialize PIE vector table:
    // Disable CPU interrupts
    DINT;

    // Initialize the PIE control registers to their default state.
    // The default state is all PIE interrupts disabled and flags
    // are cleared.
    // This function is found in the DSP2833x_PieCtrl.c file.
    InitPieCtrl();
    // Disable CPU interrupts and clear all CPU interrupt flags:
    IER = 0x0000;
    IFR = 0x0000;
    // InitPieVectTable();
    configtestled();
    RS=LOW;
    delay(5);
    RW=LOW;
    delay(5);
    EN=LOW;
    LcdInit();
    RS=LOW;
    delay(5);
    RW=LOW;
    delay(5);
    EN=LOW;
    LcdInit();
    delay(5);
```

```
WriteCmd12864(CLEAR_SCREEN);
delay(50);
WriteCmd12864(0x82);
delay(300);
DisplayCgrom(A);
delay(80);
WriteCmd12864(0x90);
delay(80);
DisplayCgrom(B);
delay(80);
WriteCmd12864(0x88);
delay(80);
DisplayCgrom(C);
delay(80);
WriteCmd12864(0x98);
delay(80);
DisplayCgrom(D);
while(1);
}

void configtestled(void)
{
    EALLOW;
    GpioCtrlRegs.GPBMUX2.bit.GPIO60 = 0; // GPIO60 = GPIO60
    GpioCtrlRegs.GPBDIR.bit.GPIO60 = 1;
    GpioCtrlRegs.GPBMUX2.bit.GPIO61 = 0; // GPIO60 = GPIO60
    GpioCtrlRegs.GPBDIR.bit.GPIO61 = 1;
    GpioCtrlRegs.GPAMUX2.bit.GPIO27= 0;
    GpioCtrlRegs.GPADIR.bit.GPIO27 = 1;
//    GpioCtrlRegs.GPBMUX2.bit.GPIO54 = 0; // GPIO60 = GPIO60
//    GpioCtrlRegs.GPBDIR.bit.GPIO54 = 1;
    EDIS;
}

void delay (UInt16 t)
{ UInt16 i;
    while(t--)
    {
        for(i=0;i<125;i++)
            ;
    }
}

void LcdInit()
{
    delay(500);
    WriteCmd12864(FUN_MODE);
```

```
delay(5);
WriteCmd12864(FUN_MODE);
delay(5);
WriteCmd12864(DISPLAY_ON);
delay(5);
WriteCmd12864(CLEAR_SCREEN);
delay(6);
WriteCmd12864(AC_INIT);
delay(4);
}

void WriteCmd12864(Uint16 cmd)
{
// Uint16 i=5;
RS=LOW;
RW=LOW;
EN=HIGH;
LCD_DATA=cmd;
// while(i--);
EN=LOW;
}

void WriteData12864(Uint16 dat)
{ // Uint16 i=5;
RS=HIGH;
RW=LOW;
EN=HIGH;
LCD_DATA=dat;
// while(i--);
EN=LOW;
}

void DisplayCgrom(uchar *hz)
{
while(*hz != '\0')
{
    WriteData12864(*hz);
    hz++;
    delay(2);
}
}
```

实验 11:ADC 循序采样模式测试 (Example_2833xadc_seqmode_test)

一、实验目的:

- ✧ 了解如何使用 dsp23885 内部集成的 ad 外设;

二、实验设备

注:模拟量输入范围: 0.0V~3.0V;

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, 螺丝刀一把;

三、实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

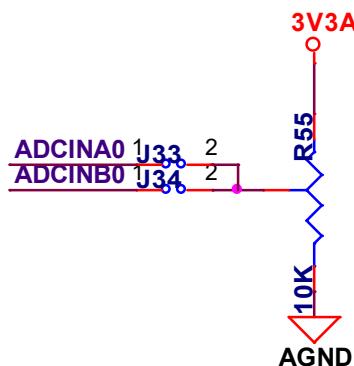


图 1-1 AD 硬件连接图

从上图可知, 需要用跳线帽将 J33 短路。

- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程;
然后单击图中红色的方框处的调试按钮, 进行调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

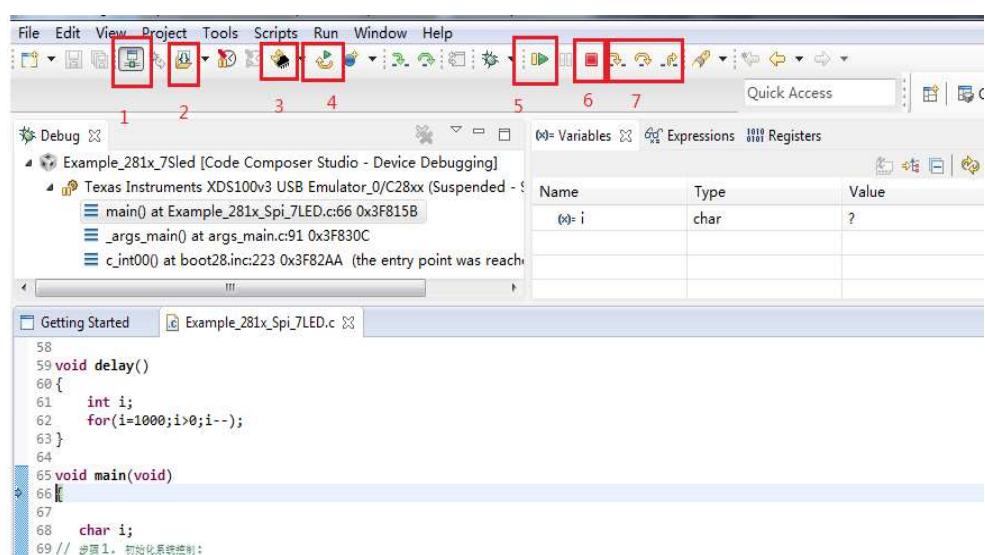


图 1-3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ◆ 图中 3 是 C P U 软 R e s e t ；
- ◆ 图中 4 是调试时恢复到程序的开始处。
- ◆ 图中 5 是全速运行；
- ◆ 图中 6 是停止调试；
- ◆ 图中 7 是用于单步调试的；

四、试验现象：

首先将 SampleTable 添加到观察窗口中，如下图所示：

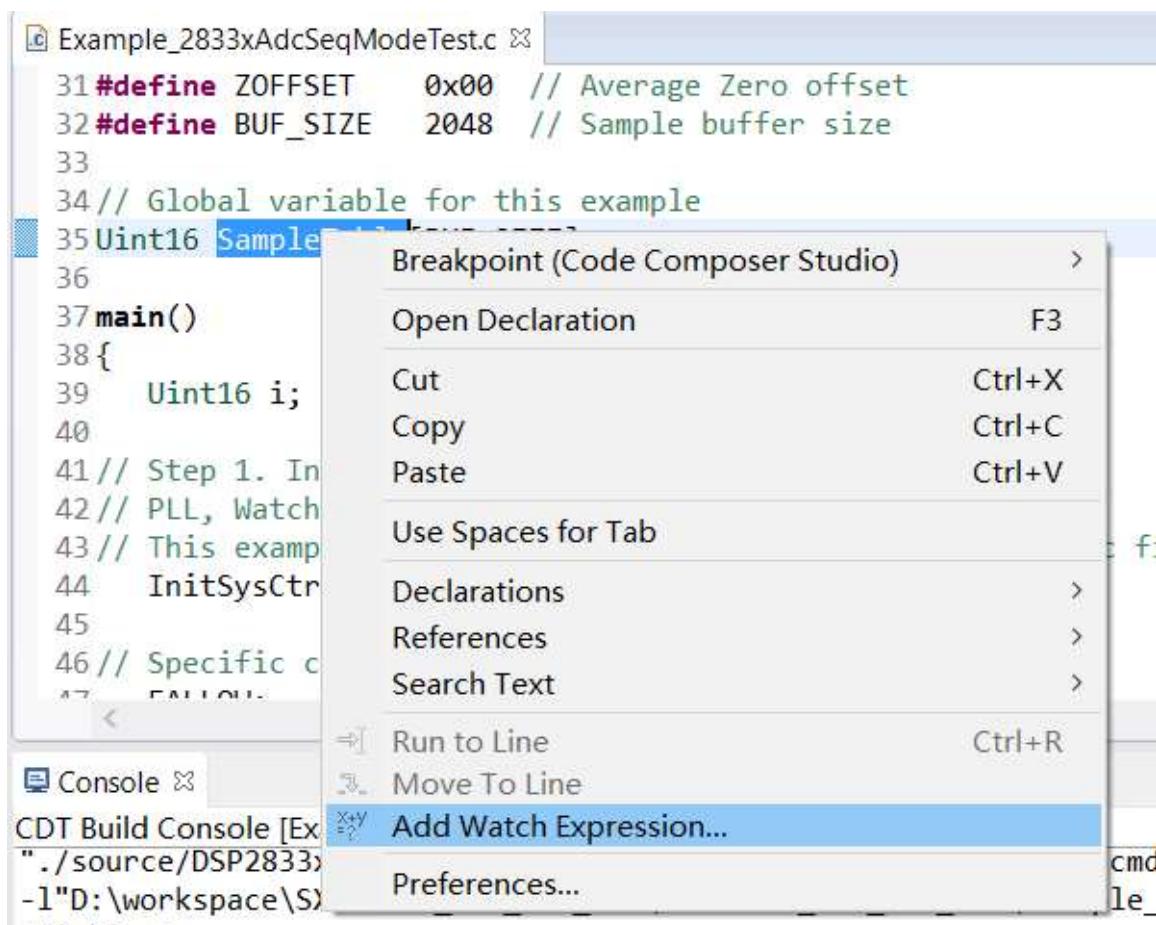


图 1-4 添加变量到观察窗口中

直接用鼠标左键单击全速运行，这时程序会全速运行。调节滑动变阻器 R98。

Expression	Type	Value	Address
SampleTable	unsigned int[2048]	0x0000C000@Data	0x0000C000@Data
[0 ... 99]			
[0]	unsigned int	1312	0x0000C000@Data
[1]	unsigned int	1313	0x0000C001@Data
[2]	unsigned int	1313	0x0000C002@Data
[3]	unsigned int	1314	0x0000C003@Data
[4]	unsigned int	1311	0x0000C004@Data
[5]	unsigned int	1307	0x0000C005@Data
[6]	unsigned int	1307	0x0000C006@Data
[7]	unsigned int	1308	0x0000C007@Data
[8]	unsigned int	1309	0x0000C008@Data
[9]	unsigned int	1305	0x0000C009@Data

Figure 4.8.3 Phenomenon

实验 12: AD 转换实验 (Example_2833x_AdSeq_ovdTest)

一、实验目的:

✧ 了解如何使用 dsp23885 内部集成的 ad 外设;

二、实验设备

注:模拟量输入范围: 0.0V~3.0V;

✧ PC 机一台;

✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;

✧ SXD28335 开发板一套, 螺丝刀一把;

三、实验步骤:

✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

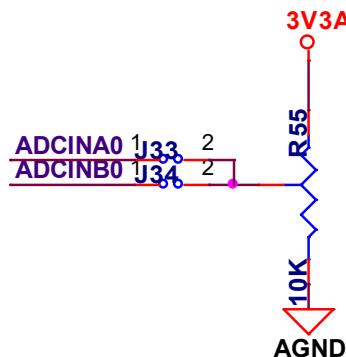


图 1-1 AD 硬件连接图

从上图可知, 需要用跳线帽将 J33 短路。

接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;

✧ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:

然后单击图中红色的方框处的调试按钮，进行调试。

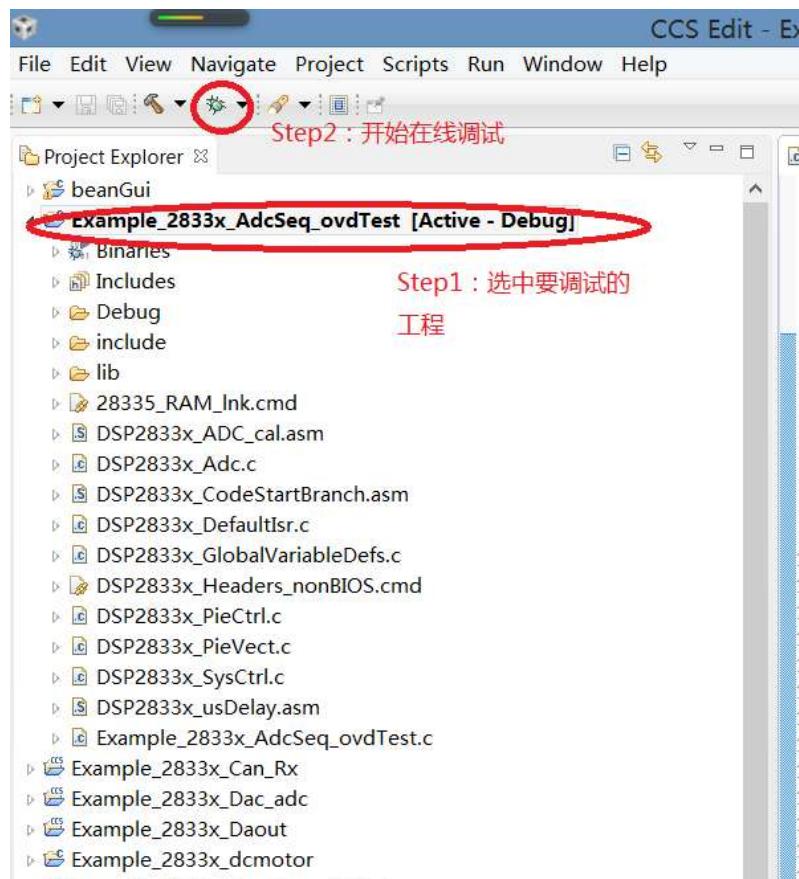


图 1-2 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

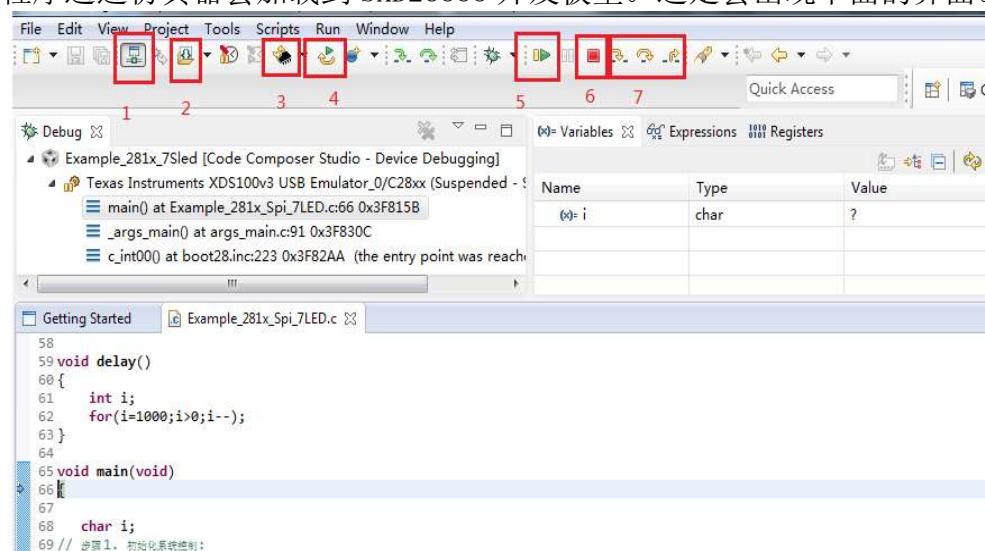


图 1-3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ◆ 图中 3 是 C P U 软 R e s e t ；
- ◆ 图中 4 是调试时恢复到程序的开始处。
- ◆ 图中 5 是全速运行；

- ◆ 图中 6 是停止调试;
- ◆ 图中 7 是用于单步调试的;

四、试验现象:

首先将 **SampleTable** 添加到观察窗口中, 如下图所示:

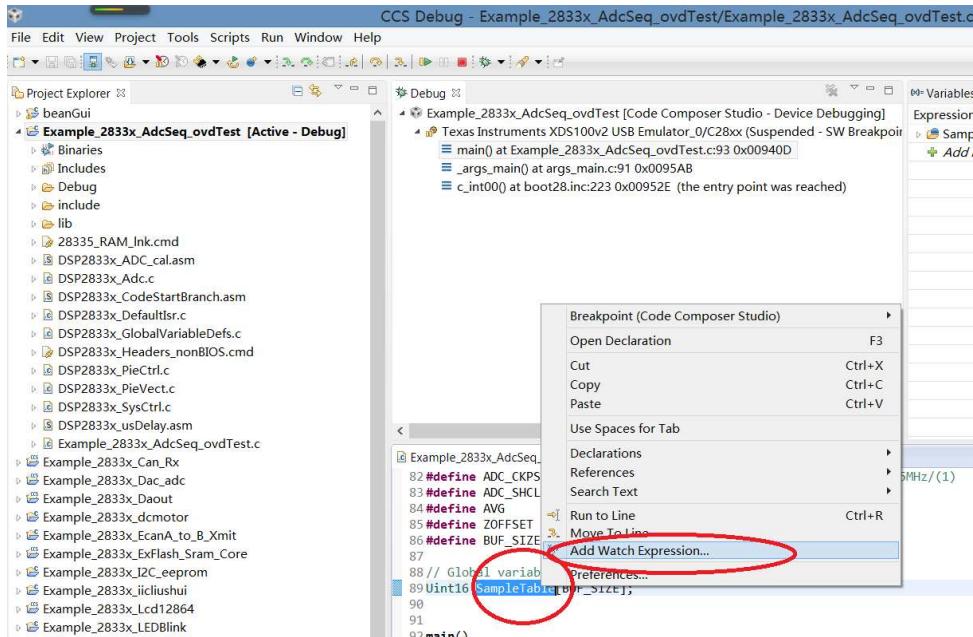


图 1-4 添加变量到观察窗口中

直接用鼠标左键单击全速运行, 这时程序会全速运行。调节滑动变阻器 R98, R99。

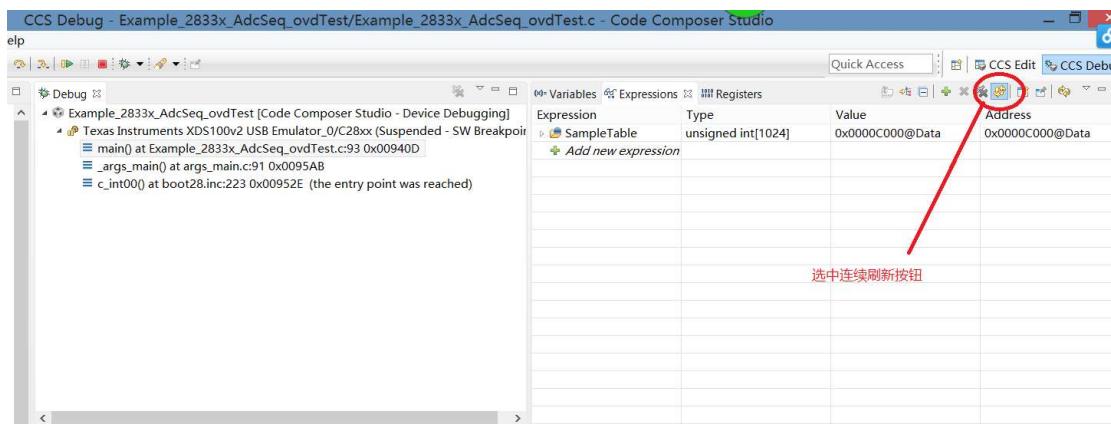


图 1-5 设置连续刷新按钮

现象:

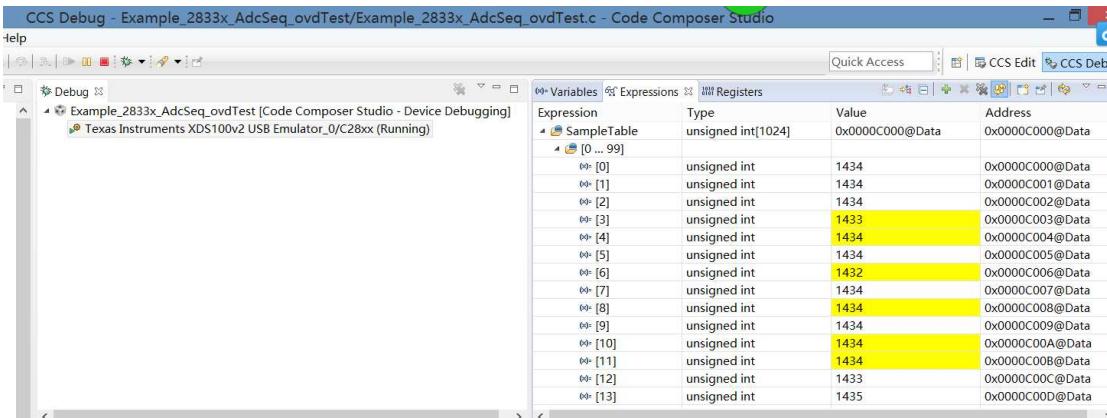


图 1-6 黄色表示采样的值在变化

五、AD 模块简介：

TMS320F28335 片上有 1 个 12 位 A/D 转换器，其前端为 2 个 8 选 1 多路切换器和 2 路同时采样 / 保持器，构成 16 个模拟输入通道，模拟通道的切换由硬件自动控制，并将各模拟通道的转换结果顺序存入 16 个结果寄存器中。

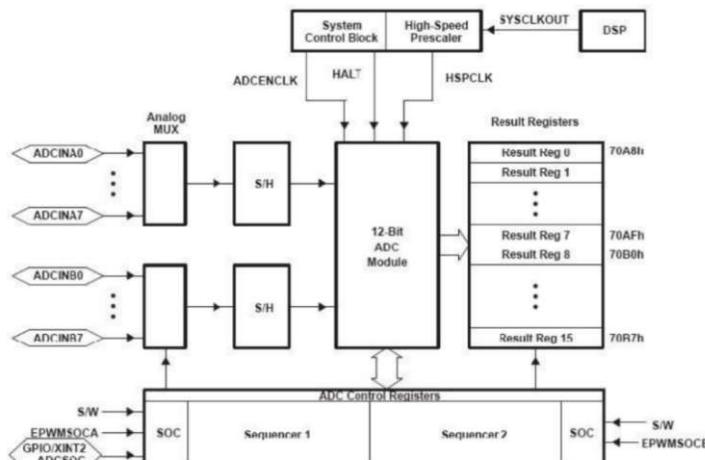


图 1-7 黄色表示采样的值在变化

ADC 特点：

带 2 个 8 选 1 多路切换器和双采样/保持器的 12-位的 ADC，共有 16 个模拟输入通道； 模拟量输入范围：0.0V~3.0V； 转换率：在 25MHz 的 ADC 时钟下为 80ns； 自动排序功能：可以提供一次触发顺序转换 16 通道模拟输入，每次转换能够编程选择 16 通道的任何 1 个，排序可以选择 2 个独立的 8 通道排序或者是 1 个大的 16 通道排序；

转换结果存储在 16 个结果寄存器中； 转换结果=4095×（输入的模拟信号-ADCLO）÷3；

多种 A/D 触发方式：软件启动、PWM 模块和外部中断 2 引脚；

中断方式：可以在每次转换结束或每隔一次转换结束触发中断；

28335 的 ADC 和 2812 的差不多，12 位的 AD，一个 ADC 转换器，16 个模拟开关构成 16 通道输入，这 16 通道可以分为两个 8 通道的（独立）和一个 16 通道的（级联模式）； ADC 的时钟 12.5M，采样频率 6.25M。输入范围 0~3V，低于 0 采样值为 0，高于 3 采样值为 4095，计算

可以由软件触发、GPIO XINT2、ePWM1~6 触发（工作在双通道模式，一个 PWM 模块触发一个）。ADC 模块主要有两个 8 通道的模拟开关、两个采样保持器和一个 12 位的 ADC 转换器构成，这两个采样保持器 A、B 分别对应着 DSP 引脚上的 INA0~INA7 和 INB0~INB7。两个采样保持器可以单独工作和级联成一个采样保持器，这就是 ADC 的两种操作模式：级联模式和双排序模式。每一种模式下还有两种操作方式，顺序采样和同步采样（同步采样就是两个采样保持器对应的输入引脚同时采样保持）

最大转换通道寄存器 ADCMAXCONV，这个寄存器决定有多少采样通道。在这种模式下寄存器的低四位有效，采样通道数= (ADCMAXCONVO^4) +1。

ADC 输入通道选择排序控制寄存器 ADCCHSELSEQn。总共有 4 个寄存器，ADCCHSELSEQ1, ADCCHSELSEQ2, ADCCHSELSEQ3, ADCCHSELSEQ4，每个寄存器都是 16 位的，顺序的 4 位决定一个输入通道，转换顺序是从 ADCCHSELSEQ1 最低 4 位到 ADCCHSELSEQ4 的最高 4 位，最多 16 个。
配合着 ADCMAXCONV 使用，ADCMAXCONV 决定对多少个通道采样，ADCCHSELSEQn 决定采样顺序。

ADC 功能包括：

1、ADC 时钟

外部高速时钟 HSPCLK 经过变换后作为 AD 的时钟。

HSPCLK 先要被控制寄存器 3 ADCTRL3 寄存器中的 ADCCLKPS[3:0]位除，随后经过 2 分频（当 ADCTRL1 寄存器中的位 CPS=1）或不分频（当 ADCTRL1 寄存器中的位 CPS=0）就得到 ADC 的时钟基准，但是最大为 25M。

ADCCLKPS=0 时，ADCCLK=HSPCLK/ (ADCTRL1[7]+1)

ADCCLKPS!=0 时，ADCCLK=HSPCLK/[2x(ADCCLKPS) x (ADCTRL1[7]+1)] 例如：外部晶振时钟 30M，PLL 倍频后为 150M，

HSPCLK=25M, ADCCLKPS=5, ADCTRL1[7]=1, 则 ADCCLK 可以有上式算出。

2、采样频率

ADC 转换包括采样、保持、量化、编码四个阶段，他把连续的模拟量量化为开关数字量，就相当于在模拟量和 AD 引脚中间有一个开关，开关闭合，模拟量就输入到 AD 采样引脚，开关断开，ADC 引脚上的模拟量就没有了，采样频率就是采样和保持的时间，就是这个开关闭合的时间。采样时间长短不影响其他操作，这个采样时间控制 SOC 脉冲宽度。由 ADC 时钟和 ADC 控制寄存器 1ADCTRL1 中的 ACQ_PS[11:8]位决定。

Fsoc=ADCCLK/ (ACQ_PS[11:8]+1)。

3、ADC 采样模式

ADC 采样有顺序采样和同时采样两种模式。

顺序采样就是按照自动排序器的设置一个通道一个通道采样，而同时采样是按照顺序排序器的设置一对一对的采样，但是这一对的编号要一样，即 ADCINA0 与 ADCINB0, ADCINA1 和 ADCINB1,,,同时采样。

2812 的 ADC 是 12 位 16 通道的，可以分两个 8 通道的也可以级联为一个 16 通道的，这样的话就有 4 种工作模式，即： a、双通道顺序采样

- b、双通道同步采样
- c、级联模式顺序采样
- d、级联模式同步采样

就每种工作模式进行介绍，不对 C 代码进行详解，在讲这些之前，先说一下涉及到的比较重要而且难理解的寄存器 第一个：最大转换通道寄存器

ADCMAXCONV

ADCMAXCONV 中能用的位是最后七位，在双通道采样模式下，自动排序寄存器 SEQ1 (A 通道) 用到的是 MAXCONV1_2-0, 就是低三位，采样的通道数 =MAXCONV1_2-0+1; 自动排序寄存器 SEQ2 (B 通道) 用到的是 MAXCONV2_2-0, 就是高三位，采样的通道数=MAXCONV2_2-0+1;

在级联模式下，自动排序器 SEQ 用到的是 MAXCONV1_3-0, 采样的通道数 =MAXCONV1_3-0+1。 第二个：自动排序器 SEQ1 SEQ2 SEQ

自动排序器就是管理在什么时间 A、B 通道的哪一个通道进行采样，就是把这 16 个通道排列顺序。

```
InitAdc();           // 初始化 ad 工作模式
// Specific ADC setup for this example:
AdcRegs.ADCTRL1.bit.ACQ_PS = ADC_SHCLK; //ad 时钟的配置
AdcRegs.ADCTRL3.bit.ADCCLKPS = ADC_CKPS;
AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;          // 级联模式
// AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x1;
AdcRegs.ADCTRL1.bit.CONT_RUN = 1; // 连续运行
AdcRegs.ADCTRL1.bit.SEQ_OVRD = 1; // Enable Sequencer override feature
AdcRegs.ADCCHSELSEQ1.all = 0x1111;         // 选择通道
AdcRegs.ADCCHSELSEQ2.all = 0x9999;
AdcRegs.ADCCHSELSEQ3.all = 0x0000;
AdcRegs.ADCCHSELSEQ4.all = 0x0000;
AdcRegs.ADCMAXCONV.bit.MAX_CONV1 = 0x7; // 最大转换通道数
// Step 5. User specific code, enable interrupts:
// 清除采样结果寄存器
for (i=0; i<BUF_SIZE; i++)
{
    SampleTable[i] = 0;
}
// 启动 ad 转换
AdcRegs.ADCTRL2.all = 0x2000;
for(;;)
{
    array_index = 0;
    for (i=0; i<(BUF_SIZE/16); i++)
    {
        // 等待转换结束
        while (AdcRegs.ADCST.bit.INT_SEQ1== 0) {}
        GpioDataRegs.GPBSET.bit.GPIO34 = 1; // 标志，用于检测 ad 转换结束，由于太
```

快，GPIO34 的现象看不出来

```
AdcRegs. ADCST.bit.INT_SEQ1_CLR = 1;  
// 读取转换的 8 通道结果  
#if INLINE_SHIFT  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT0) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT1) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT2) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT3) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT4) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT5) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT6) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT7) >> 4);  
#endif //-- INLINE_SHIFT  
#if NO_SHIFT || POST_SHIFT  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT0));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT1));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT2));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT3));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT4));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT5));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT6));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT7));  
#endif //-- NO_SHIFT || POST_SHIFT  
while (AdcRegs. ADCST.bit.INT_SEQ1 == 0) {}  
    GpioDataRegs.GPBCLEAR.bit.GPIO34 = 1; // Clear GPIO34 for monitoring -  
optional  
    AdcRegs. ADCST.bit.INT_SEQ1_CLR = 1;  
#if INLINE_SHIFT  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT8) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT9) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT10) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT11) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT12) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT13) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT14) >> 4);  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT15) >> 4);  
#endif //-- INLINE_SHIFT  
#if NO_SHIFT || POST_SHIFT  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT8));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT9));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT10));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT11));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT12));  
    SampleTable[array_index++] = ( (AdcRegs. ADCRESULT13));
```

```
SampleTable[array_index++] = ( (AdcRegs. ADCRESULT14));
SampleTable[array_index++] = ( (AdcRegs. ADCRESULT15));
#endif // -- NO_SHIFT || POST_SHIFT
}
```

实验 13：DMA 方式存取 ADC 转换结果（Example_2833xadc_dma）

一、实验目的：

- ✧ 了解如何使用 dsp23885 内部集成的 ad 外设；
- ✧ 了解如何使用 dsp23885 内部集成的 DMA 外设；

二、实验设备

注：模拟量输入范围：0.0V~3.0V；

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板一套，螺丝刀一把；

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

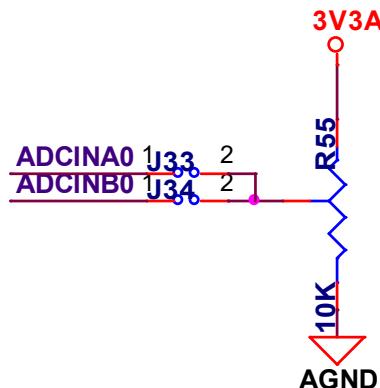


图 1-1 AD 硬件连接图

从上图可知，需要用跳线帽将 J33 短路。

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

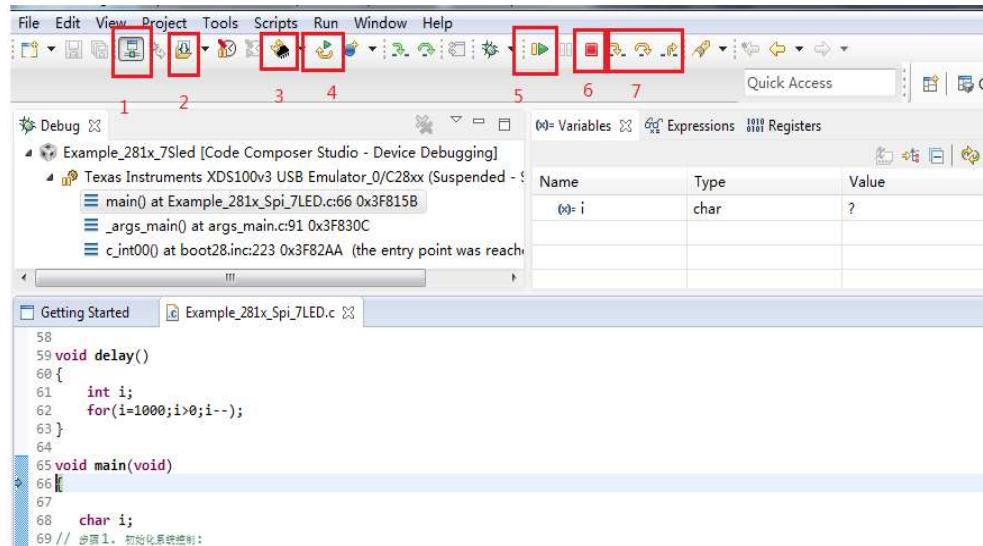


图 1-3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮;
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t ;
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行;
- ✧ 图中 6 是停止调试;
- ✧ 图中 7 是用于单步调试的;

四、试验现象：

首先将 DMABuf1 添加到观察窗口中，如下图所示：

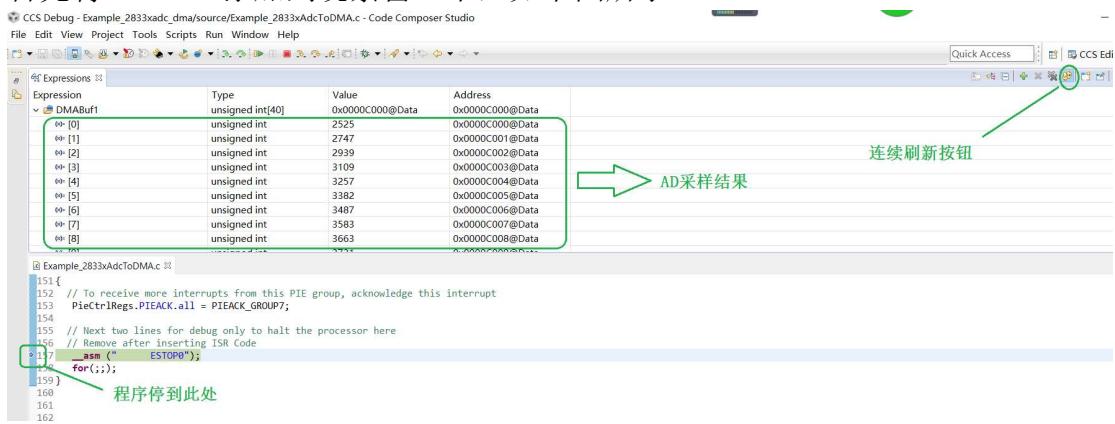


图 1-4 添加变量到观察窗口中

实验 14: PWM 启动 ad 转换试验 (Example_2833x_AdcSoc)

一、实验目的:

- ✧ 了解如何使用 dsp23885 内部集成的 ad 外设;
- ✧ 了解如何使用 dsp23885 内部集成的 pwm 外设启动 AD 转换;
- ✧ 原理: Pwm 周期性的产生触发 ad 转换信号触发 ad 模块采样;

二、实验设备

注:模拟量输入范围: 0.0V~3.0V;

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, 螺丝刀一把;

三、实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

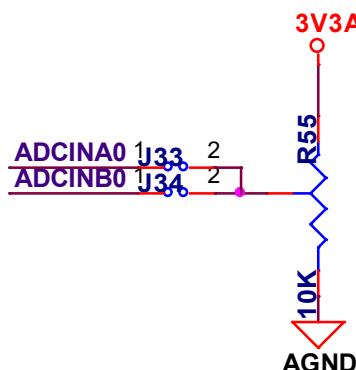


图 1 AD 硬件连接图

从上图可知, 需要用一个跳线冒将 J33 或 J34 短路。

- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示: 然后单击图中红色的方框处的调试按钮, 进行调试。

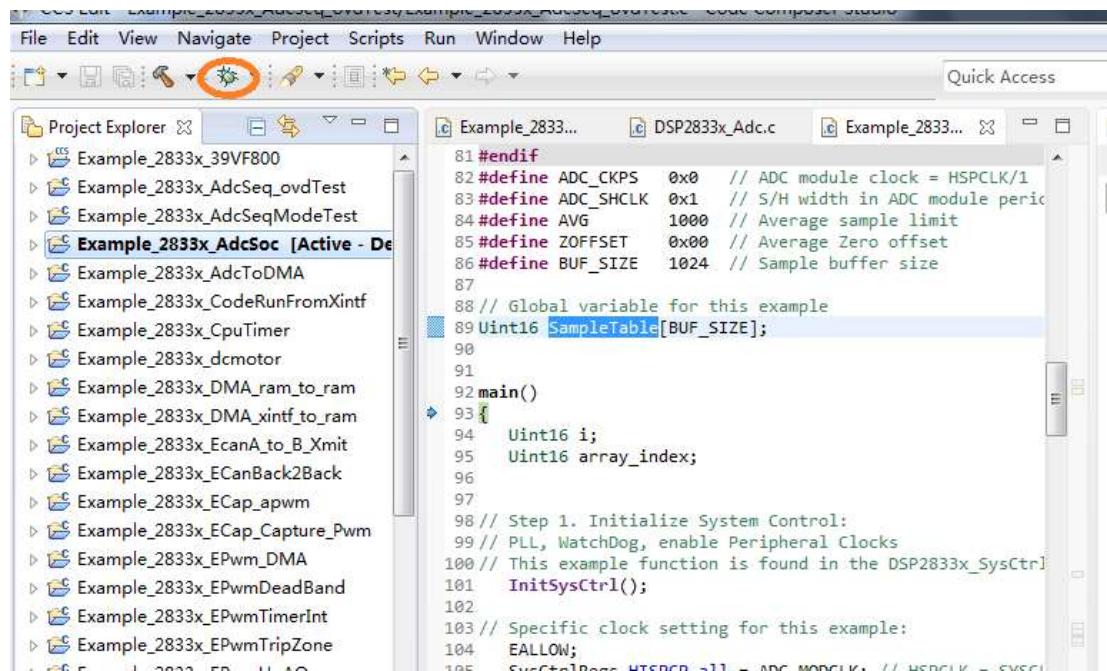


图 2 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

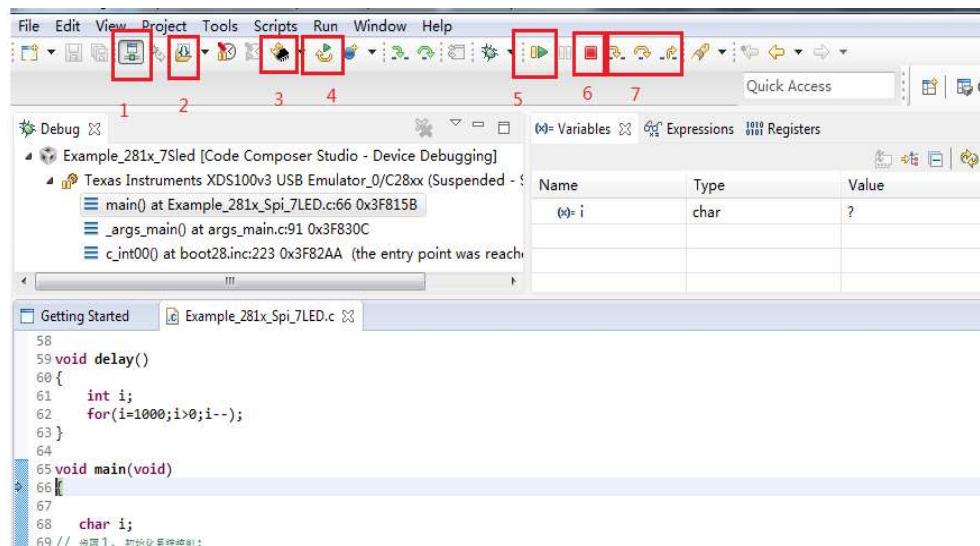


图 3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ◆ 图中 3 是 C P U 软 R e s e t；
- ◆ 图中 4 是调试时恢复到程序的开始处。
- ◆ 图中 5 是全速运行；
- ◆ 图中 6 是停止调试；
- ◆ 图中 7 是用于单步调试的；

四、试验现象：

首先将 Voltage1 和 Voltage2 添加到观察窗口中，如下图所示：

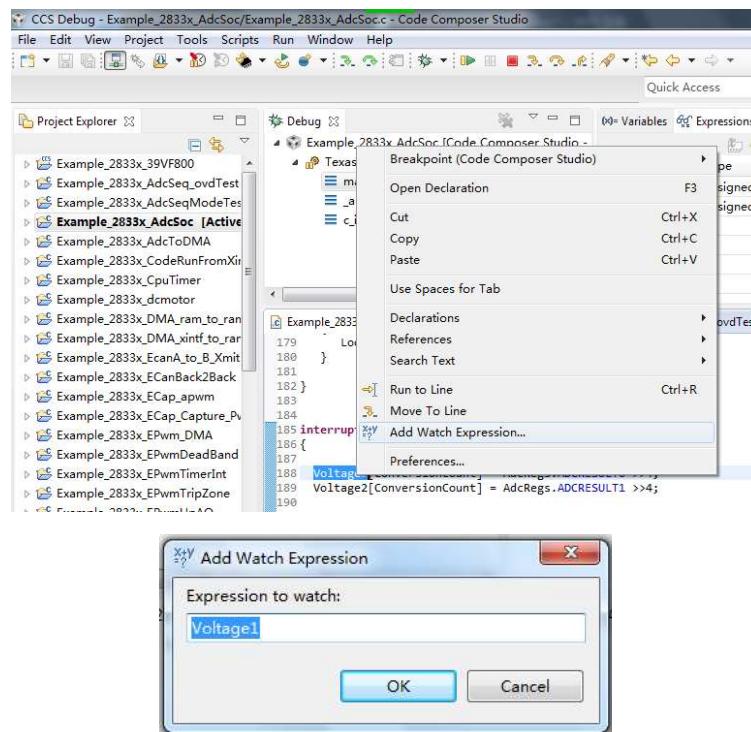


图 4 添加变量到观察窗口中

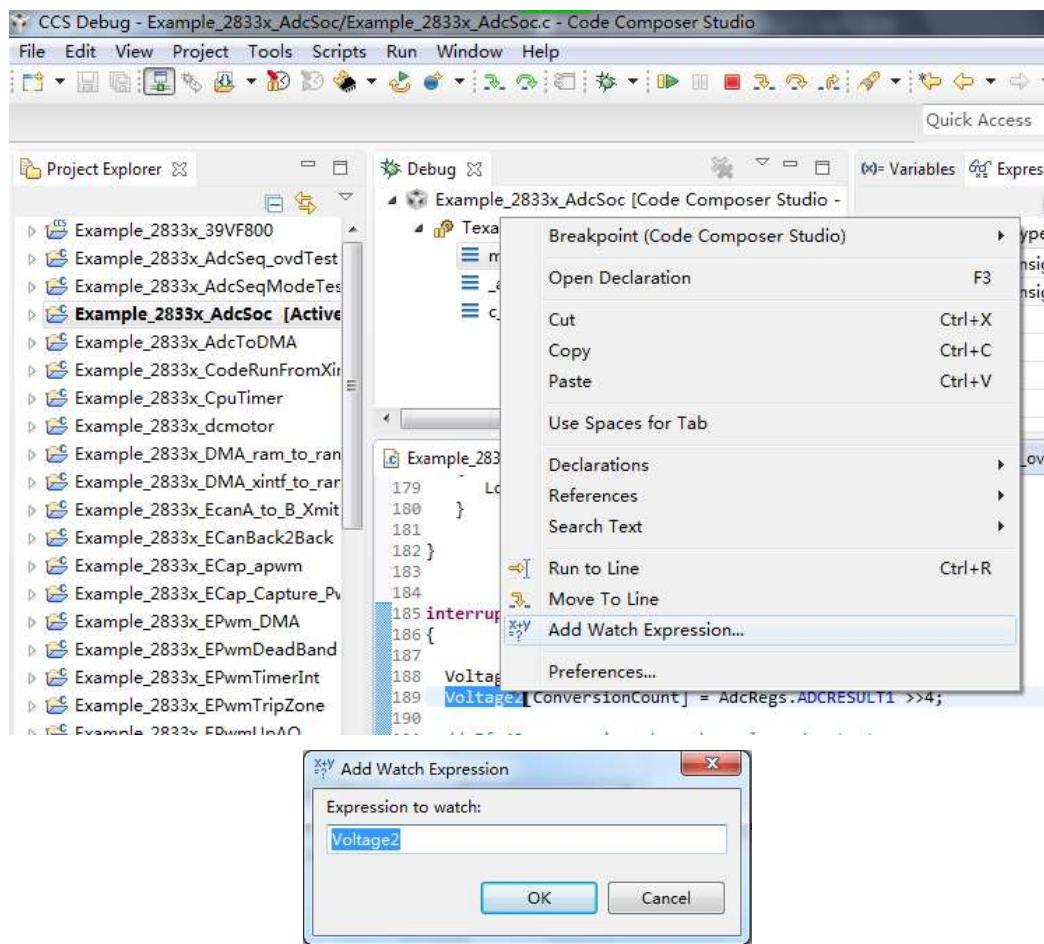


图 5 添加变量到观察窗口中

直接用鼠标左键单击全速运行，这时程序会全速运行。调节其中的两个滑动

变阻器 R98, R99。

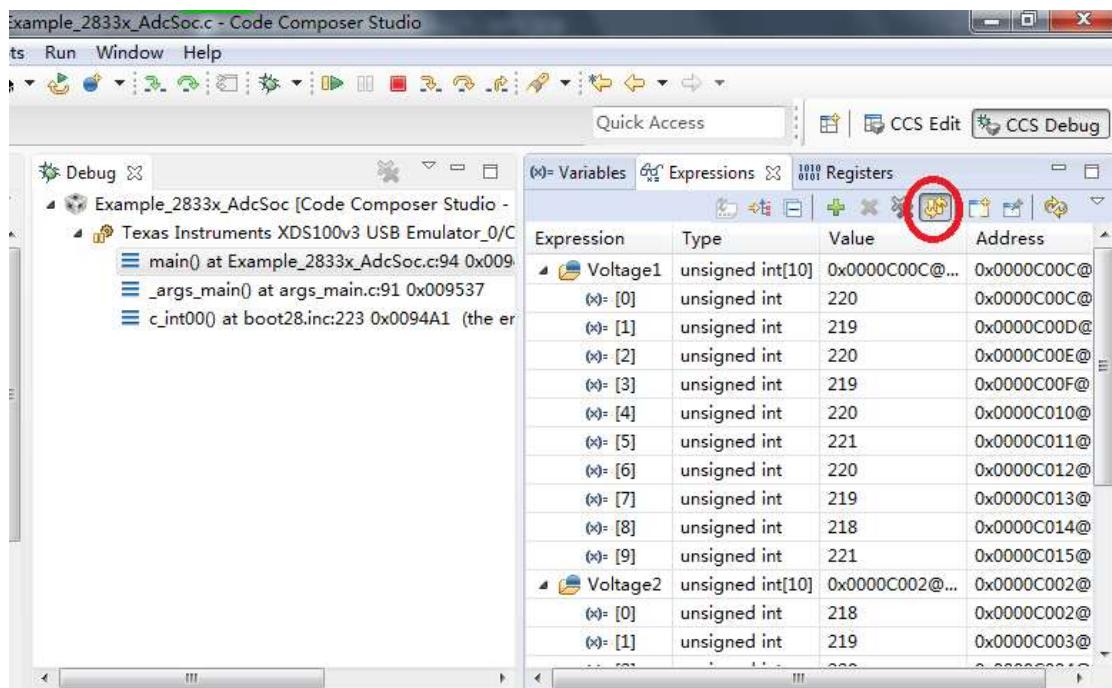


图 6 设置连续刷新按钮

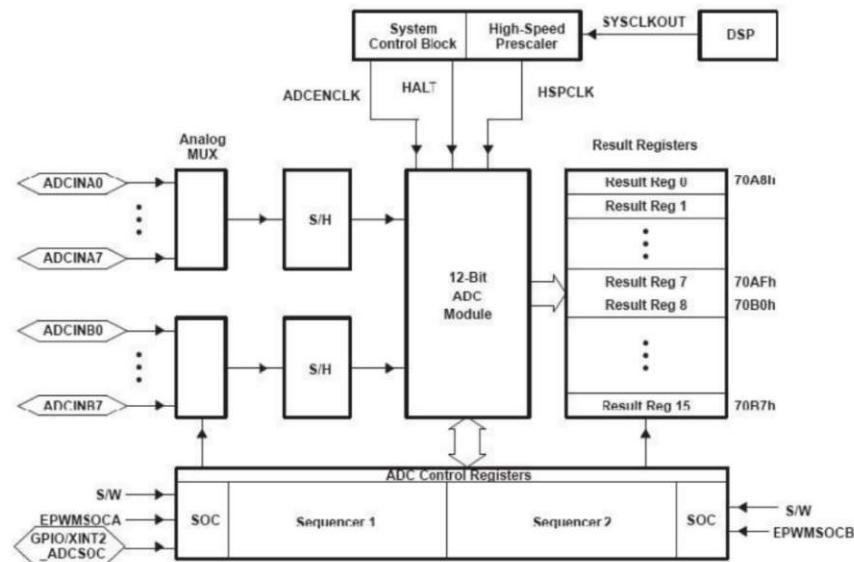
现象：

Expression	Type	Value	Address
↳ Voltage1	unsigned int[10]	0x0000C00C@...	0x0000C00C@...
↳ [0]	unsigned int	236	0x0000C00C@...
↳ [1]	unsigned int	235	0x0000C00D@...
↳ [2]	unsigned int	237	0x0000C00E@...
↳ [3]	unsigned int	235	0x0000C00F@...
↳ [4]	unsigned int	236	0x0000C010@...
↳ [5]	unsigned int	237	0x0000C011@...
↳ [6]	unsigned int	235	0x0000C012@...
↳ [7]	unsigned int	235	0x0000C013@...
↳ [8]	unsigned int	237	0x0000C014@...
↳ [9]	unsigned int	235	0x0000C015@...
↳ Voltage2	unsigned int[10]	0x0000C002@...	0x0000C002@...
↳ [0]	unsigned int	2811	0x0000C002@...
↳ [1]	unsigned int	2814	0x0000C003@...

图 7 黄色表示采样的值在变化

五、AD 模块简介：

TMS320F28335 片上有 1 个 12 位 A/D 转换器，其前端为 2 个 8 选 1 多路切换器和 2 路同时采样 / 保持器，构成 16 个模拟输入通道，模拟通道的切换由硬件自动控制，并将各模拟通道的转换结果顺序存入 16 个结果寄存器中。



ADC 特点：

带 2 个 8 选 1 多路切换器和双采样/保持器的 12-位的 ADC，共有 16 个模拟输入通道； 模拟量输入范围：0.0V~3.0V； 转换率：在 25MHz 的 ADC 时钟下为 80ns； 自动排序功能：可以提供一次触发顺序转换 16 通道模拟输入，每次转换能够编程选择 16 通道的任何 1 个，排序可以选择 2 个独立的 8 通道排序或者是 1 个大的 16 通道排序； 转换结果存储在 16 个结果寄存器中； 转换结果=4095×(输入的模拟信号-ADCL0)÷3； 多种 A/D 触发方式：软件启动、PWM 模块和外部中断 2 引脚； 中断方式：可以在每次转换结束或每隔一次转换结束触发中断；

```
#include "DSP28x_Project.h"      // Device Headerfile and Examples Include File
// 中断函数：由 pwm 周期触发
_interrupt void adc_isr(void);
Uint16 LoopCount;
Uint16 ConversionCount;
// 采到的电压值
Uint16 Voltage1[10];
Uint16 Voltage2[10];
main()
{
// 初始化系统控制，PLL 配置，看门狗禁止
// 外设时钟使能
    InitSysCtrl();
    EALLOW;
    #if (CPU_FRQ_150MHZ)      // 默认的系统时钟
        #define ADC_MODCLK 0x3 // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 150/(2*3) = 25.0 MHz
    #endif
}
```

```
#if (CPU_FRQ_100MHZ)
    #define ADC_MODCLK 0x2 // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 100/(2*2) = 25.0 MHz
#endif

EDIS;
// Define ADCCLK clock frequency ( less than or equal to 25 MHz )
EALLOW;
SysCtrlRegs.HISPCP.all = ADC_MODCLK;
EDIS;

//禁止 cpu 中断
DINT;

//初始化 PIE 模块
InitPieCtrl();
// 禁止 cpu 中断， 并清除中断标志
IER = 0x0000;
IFR = 0x0000;
// 初始化 PIE 向量表
InitPieVectTable();
// 重新映射 AD 中断函数到 PIE 向量表中
EALLOW; // This is needed to write to EALLOW protected register
PieVectTable.ADCINT = &adc_isr;
EDIS; // This is needed to disable write to EALLOW protected registers

// 配置 AD 模块 s
InitAdc(); // For this example, init the ADC
// Step 5. User specific code, enable interrupts:
// 在 PIE 级使能 AD 中断
PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
IER |= M_INT1; // cpu 级使能 ad 中断
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM
LoopCount = 0;
ConversionCount = 0;
// 配置 ad 外设
AdcRegs.ADCMAXCONV.all = 0x0001; // 最大转换通道数
AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x1; // 通道选择 ADCINA1
AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x9; // 通道选择 ADCINB1
AdcRegs.ADCTRL2.bit.EPWM_SOC_A_SEQ1 = 1; // 使能 pwm 启动 ad 转换
AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1; // 使能 ad 中断
// 配置 pwm 外设，使其能触发 ad 转换
EPwm1Regs.ETSEL.bit.SOCAREN = 1; // 使能 a 组的 SOC
EPwm1Regs.ETSEL.bit.SOCASEL = 4; // soc 产生条件
EPwm1Regs.ETPS.bit.SOCAPRD = 1; // 产生脉冲时间
EPwm1Regs.CMPA.half.CMPA = 0x0080; // 设置 pwm 比较值
EPwm1Regs.TBPRD = 0xFFFF; // 设置 pwm 周期值
```

```
EPwm1Regs.TBCTL.bit.CTRMODE = 0;           // up 计数, 启动 pwm
//大循环, 如果 ad 中断条件满足则处理 ad 中断, 处理完毕后再次回到此循环
for(;;)
{
    LoopCount++;
}
//ad 中断处理函数
__interrupt void adc_isr(void)
{
//读取两个通道的结果值
Voltage1[ConversionCount] = AdcRegs.ADCRESULT0 >>4;
Voltage2[ConversionCount] = AdcRegs.ADCRESULT1 >>4;
// If 40 conversions have been logged, start over
if(ConversionCount == 9)
{
    ConversionCount = 0;
}
else
{
    ConversionCount++;
}
// Reinitialize for next ADC sequence
AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;           //复位排序器
AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;          // 清楚排序器的中断标志
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;      // Acknowledge interrupt to PIE
return;
}
```

实验 15: Flash 版点灯程序实验(Example_2833x_Flash)

一 实验目的:

- ✧ 了解 DSP28335 的 Flash 启动方式;
- ✧ 了解 DSP28335 的 Flash 烧写方法;

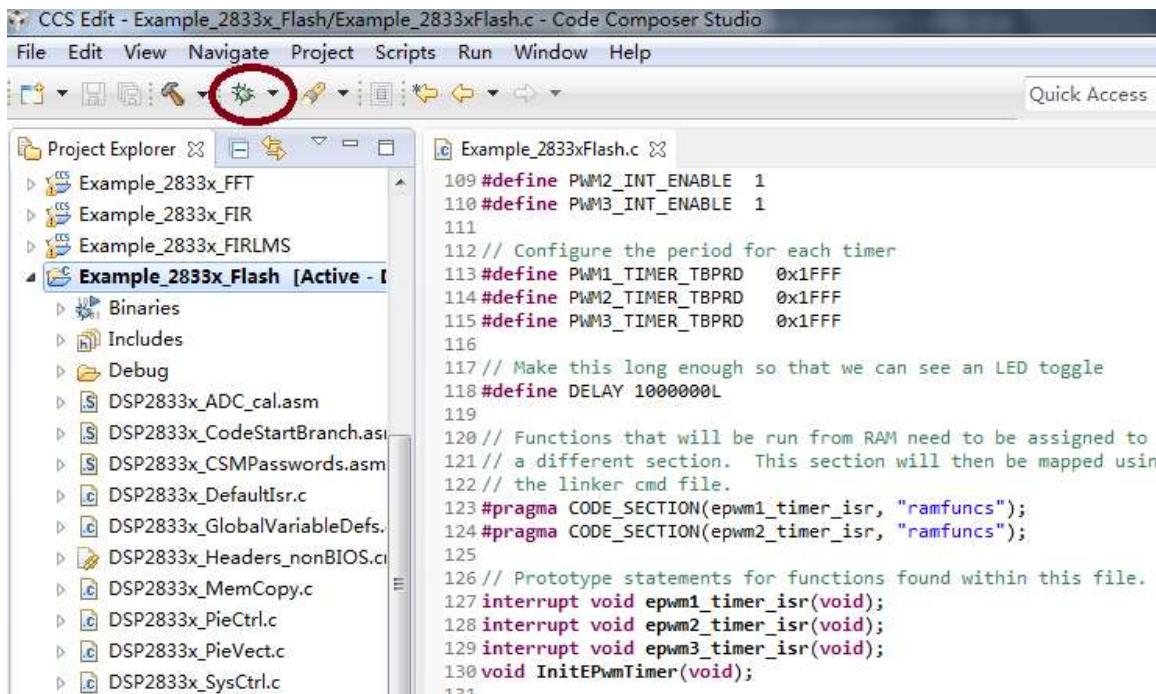
二 实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

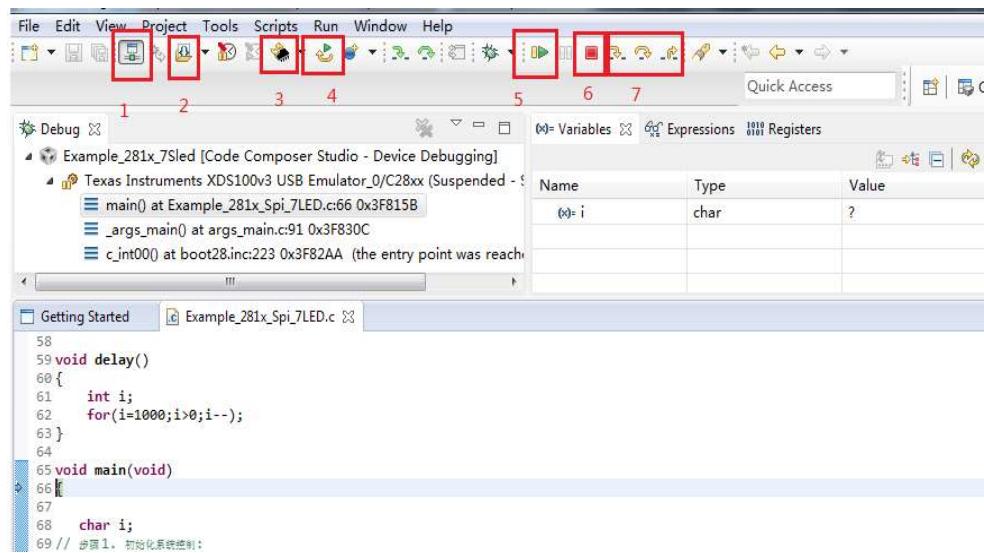
三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开;

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：
然后单击图中红色的方框处的调试按钮，进行调试。



- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。



- ① 图中 1 图标是用来进行与开发板进行连接的按钮；
- ② 图中 2 是用来下载 Debug 文件下的.out 文件的
- ③ 图中 3 是 C P U 软 R e s e t；
- ④ 图中 4 是调试时恢复到程序的开始处。
- ⑤ 图中 5 是全速运行；
- ⑥ 图中 6 是停止调试；
- ⑦ 图中 7 是用于单步调试的；

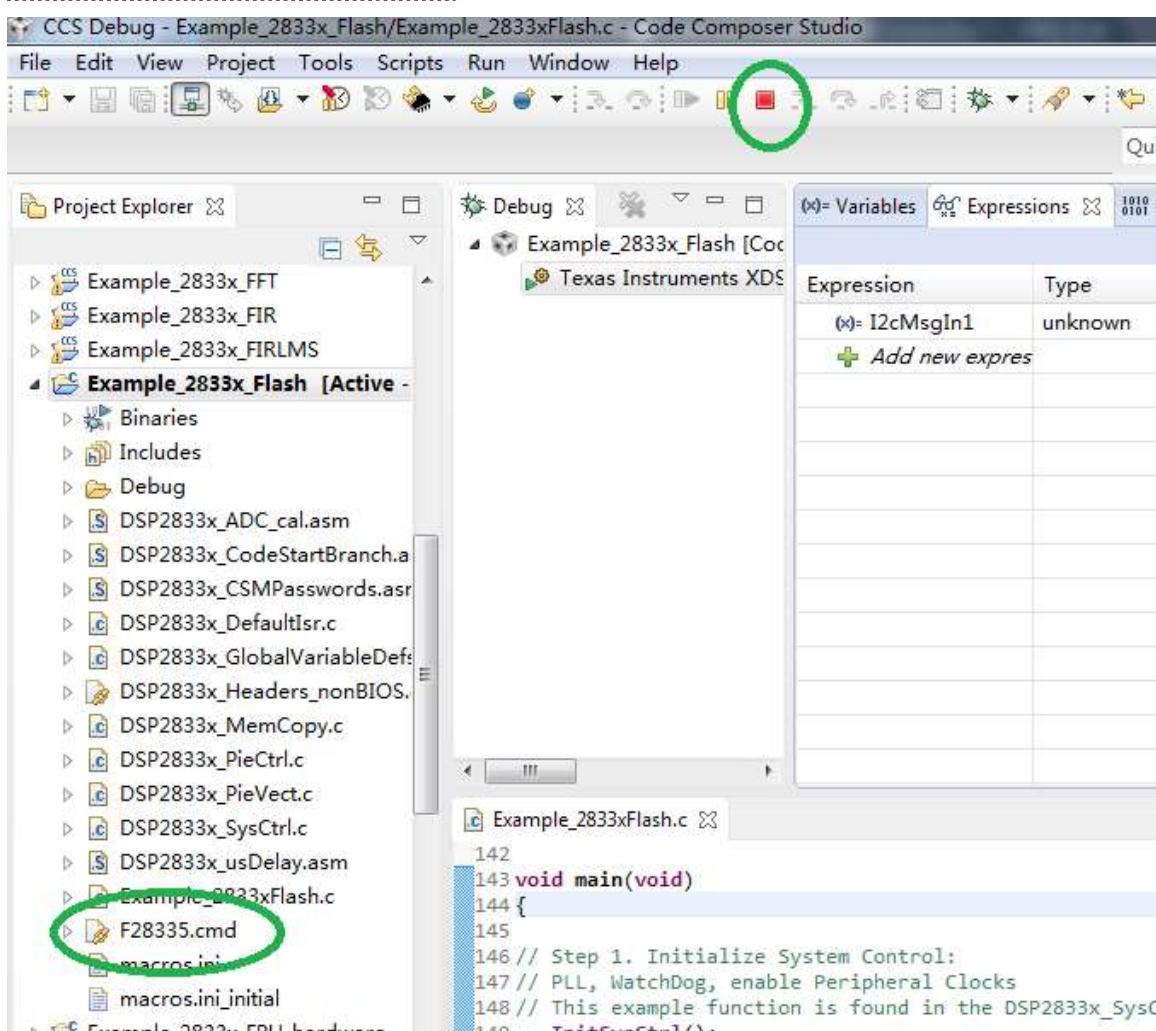
◆ 这时我们点击上图中第 5 个图标。全速运行

四 实验现象

我们可以观察到 LED 灯在不断的闪烁。

我们点击停止调试的红色小方块。停止在线调试。给开发板断电。我们可以从下面的工程中看到 我们用的 CMD 是 F28335.cmd，这个 cmd 是将程序分配到 flash 里了。所以它掉电后不会把数据丢失，这时我们如果想上电后还能运行刚才的那个程序那么就需要把四位的拨码开关 SW2（在开发板电源按键的旁边）全拨到 ON 的状态（flash 启动模式）。给开发板上电，我们会看到 Led 灯又开始闪烁了，这正是我们刚才的那个程序。

如果把上面的 cmd 替换成 28335_RAM_Lnk.cmd，则程序是 ram 版的，由于掉电丢失，所以不能掉电后再上电还执行前面的程序。



五 程序解析

◆ 下面是 F28335.cmd 的内容

MEMORY

{

PAGE 0: /* 程序空间 */

ZONE0 : origin = 0x004000, length = 0x001000 /* XINTF zone 0 */

玻尔电子致力于 C2000 全系列开发平台及应用方案的推广

```
RAML0      : origin = 0x008000, length = 0x001000      /* on-chip RAM block L0 */
RAML1      : origin = 0x009000, length = 0x001000      /* on-chip RAM block L1 */
RAML2      : origin = 0x00A000, length = 0x001000      /* on-chip RAM block L2
*/
RAML3      : origin = 0x00B000, length = 0x001000      /* on-chip RAM block L3
*/
ZONE6      : origin = 0x0100000, length = 0x100000      /* XINTF zone 6 */
ZONE7A     : origin = 0x0200000, length = 0x00FC00      /* XINTF zone 7 - program
space */
FLASHH     : origin = 0x300000, length = 0x008000      /* on-chip FLASH */
FLASHG     : origin = 0x308000, length = 0x008000      /* on-chip FLASH */
FLASHF     : origin = 0x310000, length = 0x008000      /* on-chip FLASH */
FLASHE     : origin = 0x318000, length = 0x008000      /* on-chip FLASH */
FLASHD     : origin = 0x320000, length = 0x008000      /* on-chip FLASH */
FLASHC     : origin = 0x328000, length = 0x008000      /* on-chip FLASH */
FLASHA     : origin = 0x338000, length = 0x007F80      /* on-chip FLASH */
CSM_RSVD   : origin = 0x33FF80, length = 0x000076      /* Part of FLASHA.
Program with all 0x0000 when CSM is in use. */
BEGIN      : origin = 0x33FFF6, length = 0x000002      /* Part of FLASHA. Used
for "boot to Flash" bootloader mode. */
CSM_PWL    : origin = 0x33FFF8, length = 0x000008      /* Part of FLASHA. CSM
PAGE 1 : /* 数据空间 */
}
SECTIONS
{
/* 这里是非常重要的，在这里可以将程序放到片内的 ram 中运行，也可以放到片内的
Flash 中运行。还有二者联合的方法，就是 load 的时候加载到片内的 Flash 中，而运行的时候
将呈现搬到 Ram 中去运行 */
.cinit       : > FLASHA      PAGE = 0 /* 将 cinit 段放到片内 Flash 中 */
.pinit       : > FLASHA,      PAGE = 0 /* 将 pinit 段放到片内 Flash 中 */
.text        : > FLASHA      PAGE = 0 /* 将代码段放到片内 Flash 中 */

codestart    : > BEGIN      PAGE = 0
ramfuncs     : LOAD = FLASHD, /* 这是二者联合的方法，就是 load 的时候加
载到片内的 Flash 中，而运行的时候将呈现搬到 Ram 中去运行 */
               RUN = RAML0,
               LOAD_START(_RamfuncsLoadStart),
               LOAD_END(_RamfuncsLoadEnd),
               RUN_START(_RamfuncsRunStart),
               PAGE = 0
}

```

✧ 28335_RAM_1nk.cmd

MEMORY

```
{  
/*物理空间划分*/  
PAGE 0 :/*程序段*/  
  
    BEGIN      : origin = 0x000000, length = 0x000002      /* Boot to M0 will go here  
*/  
    RAMM0      : origin = 0x000050, length = 0x0003B0  
    RAML0      : origin = 0x008000, length = 0x001000  
    RAML1      : origin = 0x009000, length = 0x001000  
    RAML2      : origin = 0x00A000, length = 0x001000  
    RAML3      : origin = 0x00B000, length = 0x001000  
    ZONE7A     : origin = 0x200000, length = 0x00FC00      /* XINTF zone 7 - program  
space */  
    CSM_RSVD   : origin = 0x33FF80, length = 0x000076      /* Part of FLASHA.  
Program with all 0x0000 when CSM is in use. */  
    CSM_PWL    : origin = 0x33FFF8, length = 0x000008      /* Part of FLASHA. CSM  
password locations in FLASHA */  
    ADC_CAL    : origin = 0x380080, length = 0x000009  
    RESET      : origin = 0x3FFFC0, length = 0x000002  
    IQTABLES   : origin = 0x3FE000, length = 0x000b50  
    IQTABLES2  : origin = 0x3FEB50, length = 0x00008c  
    FPUTABLES  : origin = 0x3FEBDC, length = 0x0006A0  
    BOOTROM    : origin = 0x3FF27C, length = 0x000D44  
  
PAGE 1 :/*数据段*/  
    /* BOOT_RSVD is used by the boot ROM for stack. */  
    /* This section is only reserved to keep the BOOT ROM from */  
    /* corrupting this area during the debug process */  
    BOOT_RSVD  : origin = 0x000002, length = 0x00004E      /* Part of M0, BOOT rom  
will use this for stack */  
    RAMM1      : origin = 0x000400, length = 0x000400      /* on-chip RAM block M1 */  
    RAML4      : origin = 0x00C000, length = 0x001000  
    RAML5      : origin = 0x00D000, length = 0x001000  
    RAML6      : origin = 0x00E000, length = 0x001000  
    RAML7      : origin = 0x00F000, length = 0x001000  
    ZONE7B     : origin = 0x20FC00, length = 0x000400      /* XINTF zone 7 - data  
space */  
}  
SECTIONS  
{  
/*下面是把所有程序和数据都放到了片内的 Ram 空间*/  
    codestart    : > BEGIN,      PAGE = 0  
    ramfuncs    : > RAML0,      PAGE = 0  
    .text        : > RAML1,      PAGE = 0 /*将代码段放到片内的 ram 区域中 */  
    .InitBoot   : > RAML1,      PAGE = 0
```

```

.cinit          : > RAML0,      PAGE = 0 /*将.cinit 段放到片内的 ram 区域中 */
.pinit          : > RAML0,      PAGE = 0
.switch         : > RAML0,      PAGE = 0
.stack          : > RAMM1,      PAGE = 1
.ebss           : > RAML4,      PAGE = 1
.econst          : > RAML5,      PAGE = 1
.esysmem        : > RAMM1,      PAGE = 1
.IQmath          : > RAML1,      PAGE = 0
.IQmathTables   : > IQTABLES,   PAGE = 0, TYPE = NOLOAD
/* Uncomment the section below if calling the IQNexp() or IQexp()
functions from the IQMath.lib library in order to utilize the
relevant IQ Math table in Boot ROM (This saves space and Boot ROM
is 1 wait-state). If this section is not uncommented, IQmathTables2
will be loaded into other memory (SARAM, Flash, etc.) and will take
up space, but 0 wait-state is possible.
*/
/*
IQmathTables2    : > IQTABLES2, PAGE = 0, TYPE = NOLOAD
{
    IQmath.lib<IQNexpTable.obj> (IQmathTablesRam)
}
*/
FPUmathTables   : > FPUTABLES,   PAGE = 0, TYPE = NOLOAD
DMARAML4         : > RAML4,      PAGE = 1
DMARAML5         : > RAML5,      PAGE = 1
DMARAML6         : > RAML6,      PAGE = 1
DMARAML7         : > RAML7,      PAGE = 1
ZONE7DATA        : > ZONE7B,     PAGE = 1
.reset           : > RESET,      PAGE = 0, TYPE = DSECT /* not used
*/
csm_rsvd         : > CSM_RSVD   PAGE = 0, TYPE = DSECT /* not used for SARAM
examples */
csmpasswds       : > CSM_PWL    PAGE = 0, TYPE = DSECT /* not used for SARAM
examples */
/* Allocate ADC_cal function (pre-programmed by factory into TI reserved memory)
*/
.adc_cal         : load = ADC_CAL, PAGE = 0, TYPE = NOLOAD
}

```

实验 16: SCI 串口实验(Example_2833x_Sci_Echoback)

一 实验目的:

- ✧ 熟悉 TMS320F28335 的 SCI 配置过程;
- ✧ 学会用 SCI 进行通信;

二 实验设备

- ◆ 计算机（已安装 CCSv6.0 开发环境）
 - ◆ SXD28335 或 SXD28335B 开发板
 - ◆ 5V 2A（或 3A）DC 电源
 - ◆ USB 转串口线一条
 - ◆ 仿真器（本手册都是以三兄弟嵌入式生产的 XDS100V3 仿真器为例，其余仿真器类似）

三 实验步骤

(1) 首先将 CCS6.0 开发环境打开；给开发板上电。看一下如下原理图：

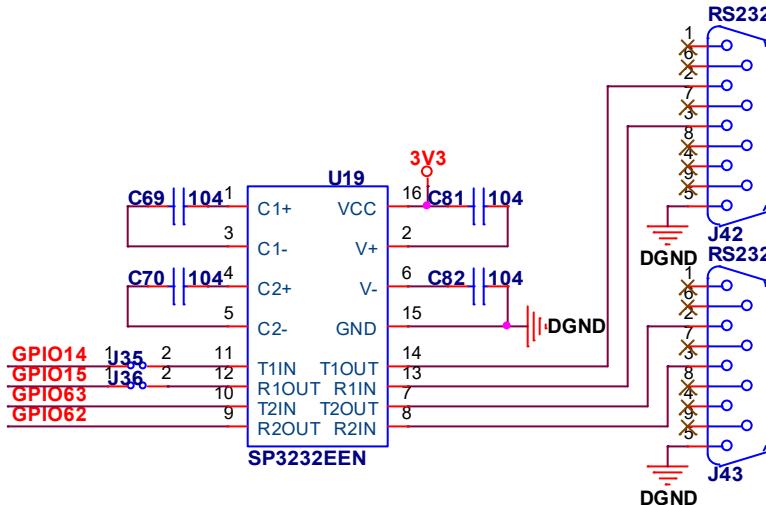


图 1 串口的硬件连接图

GPIO63	114	GPIO64/XD15	ID	GPIO12
GPIO62	113	GPIO63/SCITXD/CXD16		GPIO13
GPIO61	112	GPIO62/SCIRXD/CXD17		GPIO14
GPIO60	111	GPIO67/MPSRB/XD18		GPIO15
GPIO59	110	GPIO60/MCLKR/XD19		GPIO16
GPIO58	100	GPIO59/MFSRA/XD20		GPIO17
GPIO57	99	GPIO58/MCLKA/XD21		GPIO18
GPIO56	98	GPIO57/SPISTEAr/XD22		GPIO19
GPIO55	97	GPIO56/SPICLKA/XD23		GPIO20
GPIO54	96	GPIO55/SPISIMOA/XD24		GPIO21
GPIO53	95	GPIO54/SPISIMOA/XD25		GPIO22
GPIO52	94	GPIO53/EQEPI/XD26		GPIO23
GPIO51	93	GPIO52/FOEPI/XD27		GPIO24
GPIO49	92	GPIO51/SCITXDB/CANRXA		GPIO25
GPIO48	91	GPIO50/SCIRXD/CANTXA		GPIO26
GPIO47	90	GPIO49/SCITXDB/CANRXB		GPIO27
GPIO46	89	GPIO48/SCIRXD/CANTXB		GPIO28
GPIO45	88	GPIO47/SCITXDB/MDXB		GPIO29
GPIO44	87	GPIO46/SCIRXD/MDXB		GPIO30
GPIO43	86	GPIO45/TZn1/CANTXB/MDXB		GPIO31
GPIO42	85	GPIO44/TZn2/CANTXB/MDXB		GPIO32
GPIO41	84	GPIO43/TZn3/XHOLDn/SCITXDB/MCLKXB		GPIO33
GPIO40	83	GPIO42/TZn4/XHOLDn/SCIRXD/B/MFSXn		GPIO34
GPIO39	82	GPIO41/TZn5/XHOLDn/SCIRXD/B/MFSXn		GPIO35
GPIO38	81	GPIO40/TZn6/XHOLDn/SCITXDB/MCLKXB		GPIO36
GPIO37	80	GPIO39/TZn7/XHOLDn/SCIRXD/B/MFSXn		GPIO37
GPIO36	79	GPIO38/TZn8/XHOLDn/SCITXDB/MCLKXB		GPIO38
GPIO35	78	GPIO37/TZn9/XHOLDn/SCIRXD/B/MFSXn		GPIO39
GPIO34	77	GPIO36/TZn10/XHOLDn/SCITXDB/MCLKXB		GPIO40
GPIO33	76	GPIO35/TZn11/XHOLDn/SCIRXD/B/MFSXn		GPIO41
GPIO32	75	GPIO34/TZn12/XHOLDn/SCITXDB/MCLKXB		GPIO42
GPIO31	74	GPIO33/TZn13/XHOLDn/SCIRXD/B/MFSXn		GPIO43
GPIO30	73	GPIO32/TZn14/XHOLDn/SCITXDB/MCLKXB		GPIO44
GPIO29	72	GPIO31/TZn15/XHOLDn/SCIRXD/B/MFSXn		GPIO45
GPIO28	71	GPIO30/TZn16/XHOLDn/SCITXDB/MCLKXB		GPIO46
GPIO27	70	GPIO29/TZn17/XHOLDn/SCIRXD/B/MFSXn		GPIO47
GPIO26	69	GPIO28/TZn18/XHOLDn/SCITXDB/MCLKXB		GPIO48
GPIO25	68	GPIO27/TZn19/XHOLDn/SCIRXD/B/MFSXn		GPIO49
GPIO24	67	GPIO26/TZn20/XHOLDn/SCITXDB/MCLKXB		GPIO50
GPIO23	66	GPIO25/TZn21/XHOLDn/SCIRXD/B/MFSXn		GPIO51
GPIO22	65	GPIO24/TZn22/XHOLDn/SCITXDB/MCLKXB		GPIO52
GPIO21	64	GPIO23/TZn23/XHOLDn/SCIRXD/B/MFSXn		GPIO53
GPIO20	63	GPIO22/TZn24/XHOLDn/SCITXDB/MCLKXB		GPIO54
GPIO19	62	GPIO21/TZn25/XHOLDn/SCIRXD/B/MFSXn		GPIO55
GPIO18	61	GPIO20/TZn26/XHOLDn/SCITXDB/MCLKXB		GPIO56
GPIO17	60	GPIO19/TZn27/XHOLDn/SCIRXD/B/MFSXn		GPIO57
GPIO16	59	GPIO18/TZn28/XHOLDn/SCITXDB/MCLKXB		GPIO58
GPIO15	58	GPIO17/TZn29/XHOLDn/SCIRXD/B/MFSXn		GPIO59
GPIO14	57	GPIO16/TZn30/XHOLDn/SCITXDB/MCLKXB		GPIO60
GPIO13	56	GPIO15/TZn31/XHOLDn/SCIRXD/B/MFSXn		GPIO61
GPIO12	55	GPIO14/TZn32/XHOLDn/SCITXDB/MCLKXB		GPIO62
GPIO11	54	GPIO13/TZn33/XHOLDn/SCIRXD/B/MFSXn		GPIO63
GPIO10	53	GPIO12/TZn34/XHOLDn/SCITXDB/MCLKXB		GPIO64
GPIO9	52	GPIO11/TZn35/XHOLDn/SCIRXD/B/MFSXn		GPIO65
GPIO8	51	GPIO10/TZn36/XHOLDn/SCITXDB/MCLKXB		GPIO66
GPIO7	50	GPIO9/TZn37/XHOLDn/SCIRXD/B/MFSXn		GPIO67
GPIO6	49	GPIO8/TZn38/XHOLDn/SCITXDB/MCLKXB		GPIO68
GPIO5	48	GPIO7/TZn39/XHOLDn/SCIRXD/B/MFSXn		GPIO69
GPIO4	47	GPIO6/TZn40/XHOLDn/SCITXDB/MCLKXB		GPIO70
GPIO3	46	GPIO5/TZn41/XHOLDn/SCIRXD/B/MFSXn		GPIO71
GPIO2	45	GPIO4/TZn42/XHOLDn/SCITXDB/MCLKXB		GPIO72
GPIO1	44	GPIO3/TZn43/XHOLDn/SCIRXD/B/MFSXn		GPIO73
GPIO0	43	GPIO2/TZn44/XHOLDn/SCITXDB/MCLKXB		GPIO74
GPIO19	42	GPIO1/TZn45/XHOLDn/SCIRXD/B/MFSXn		GPIO75
GPIO18	41	GPIO0/TZn46/XHOLDn/SCITXDB/MCLKXB		GPIO76
GPIO17	40	GPIO19/TZn47/XHOLDn/SCIRXD/B/MFSXn		GPIO77
GPIO16	39	GPIO18/TZn48/XHOLDn/SCITXDB/MCLKXB		GPIO78
GPIO15	38	GPIO17/TZn49/XHOLDn/SCIRXD/B/MFSXn		GPIO79
GPIO14	37	GPIO16/TZn50/XHOLDn/SCITXDB/MCLKXB		GPIO80
GPIO13	36	GPIO15/TZn51/XHOLDn/SCIRXD/B/MFSXn		GPIO81
GPIO12	35	GPIO14/TZn52/XHOLDn/SCITXDB/MCLKXB		GPIO82
GPIO11	34	GPIO13/TZn53/XHOLDn/SCIRXD/B/MFSXn		GPIO83
GPIO10	33	GPIO12/TZn54/XHOLDn/SCITXDB/MCLKXB		GPIO84
GPIO9	32	GPIO11/TZn55/XHOLDn/SCIRXD/B/MFSXn		GPIO85
GPIO8	31	GPIO10/TZn56/XHOLDn/SCITXDB/MCLKXB		GPIO86
GPIO7	30	GPIO9/TZn57/XHOLDn/SCIRXD/B/MFSXn		GPIO87
GPIO6	29	GPIO8/TZn58/XHOLDn/SCITXDB/MCLKXB		GPIO88
GPIO5	28	GPIO7/TZn59/XHOLDn/SCIRXD/B/MFSXn		GPIO89
GPIO4	27	GPIO6/TZn60/XHOLDn/SCITXDB/MCLKXB		GPIO90
GPIO3	26	GPIO5/TZn61/XHOLDn/SCIRXD/B/MFSXn		GPIO91
GPIO2	25	GPIO4/TZn62/XHOLDn/SCITXDB/MCLKXB		GPIO92
GPIO1	24	GPIO3/TZn63/XHOLDn/SCIRXD/B/MFSXn		GPIO93
GPIO0	23	GPIO2/TZn64/XHOLDn/SCITXDB/MCLKXB		GPIO94
GPIO19	22	GPIO1/TZn65/XHOLDn/SCIRXD/B/MFSXn		GPIO95
GPIO18	21	GPIO0/TZn66/XHOLDn/SCITXDB/MCLKXB		GPIO96
GPIO17	20	GPIO19/TZn67/XHOLDn/SCIRXD/B/MFSXn		GPIO97
GPIO16	19	GPIO18/TZn68/XHOLDn/SCITXDB/MCLKXB		GPIO98
GPIO15	18	GPIO17/TZn69/XHOLDn/SCIRXD/B/MFSXn		GPIO99
GPIO14	17	GPIO16/TZn70/XHOLDn/SCITXDB/MCLKXB		GPIO100
GPIO13	16	GPIO15/TZn71/XHOLDn/SCIRXD/B/MFSXn		GPIO101
GPIO12	15	GPIO14/TZn72/XHOLDn/SCITXDB/MCLKXB		GPIO102
GPIO11	14	GPIO13/TZn73/XHOLDn/SCIRXD/B/MFSXn		GPIO103
GPIO10	13	GPIO12/TZn74/XHOLDn/SCITXDB/MCLKXB		GPIO104
GPIO9	12	GPIO11/TZn75/XHOLDn/SCIRXD/B/MFSXn		GPIO105
GPIO8	11	GPIO10/TZn76/XHOLDn/SCITXDB/MCLKXB		GPIO106
GPIO7	10	GPIO9/TZn77/XHOLDn/SCIRXD/B/MFSXn		GPIO107
GPIO6	9	GPIO8/TZn78/XHOLDn/SCITXDB/MCLKXB		GPIO108
GPIO5	8	GPIO7/TZn79/XHOLDn/SCIRXD/B/MFSXn		GPIO109
GPIO4	7	GPIO6/TZn80/XHOLDn/SCITXDB/MCLKXB		GPIO110
GPIO3	6	GPIO5/TZn81/XHOLDn/SCIRXD/B/MFSXn		GPIO111
GPIO2	5	GPIO4/TZn82/XHOLDn/SCITXDB/MCLKXB		GPIO112
GPIO1	4	GPIO3/TZn83/XHOLDn/SCIRXD/B/MFSXn		GPIO113
GPIO0	3	GPIO2/TZn84/XHOLDn/SCITXDB/MCLKXB		GPIO114
GPIO19	2	GPIO1/TZn85/XHOLDn/SCIRXD/B/MFSXn		GPIO115
GPIO18	1	GPIO0/TZn86/XHOLDn/SCITXDB/MCLKXB		GPIO116

图 2 对应的 DSP 引脚

从图中得知 GPIO62 和 GPIO63 对应的是串口 C, 其余两个 IO 对应的是串口 B, 我们在程序里通过两个宏定义来决定是用哪个串口:下面配置为 SCIC 串口

```
#define _SCI_C 1  
#define SCI_B 0
```

接下来看一下位选我们在光盘或是网上找到 WIN7 超级终端；并进行软件的安装。这个过程非常简单，这里就不描述，如果不会，请百度之；

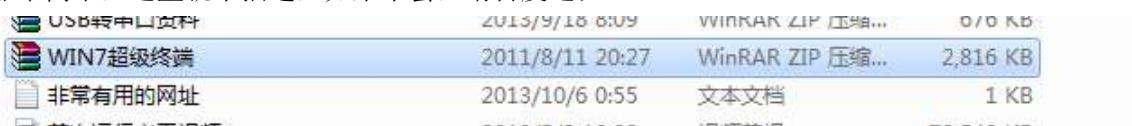


图 3 超级终端的安装

装完串口终端后，会出现图 4 图标；



图 4 超级终端在桌面上的图标

双击打开这个图标；出现如下界面；

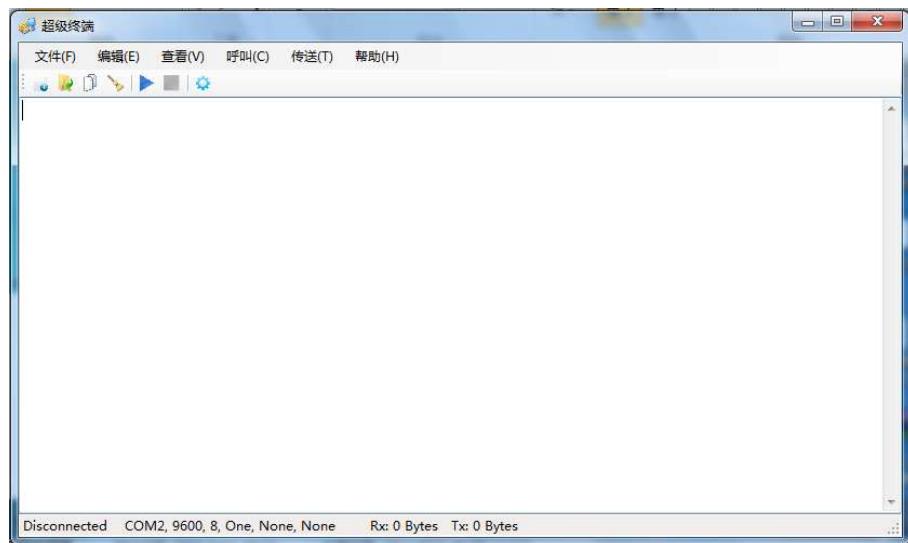


图 5 串口终端的界面



请先保持克制，请先不要激动的去按下下图的红色框中的 ；因为我们要设置串口的工作方式；

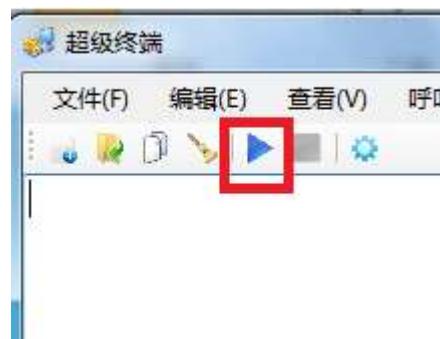


图 6 串口终端启动按钮

首先，选择文件→ 属性；如下图

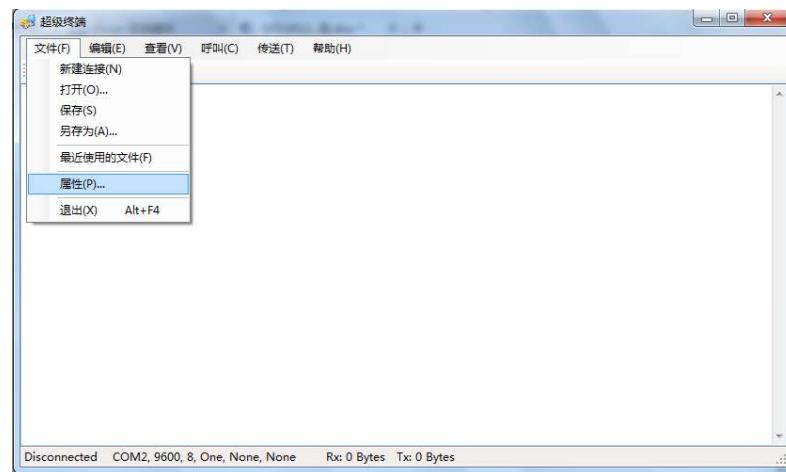


图 7 属性配置

出现如下界面；



图 8 中断配置界面

在这里我们就按上图中的配置进行设置，除了端口需要客户您根据自己的电脑进行选择：
方法是在“设备管理器”里去看，如下图；

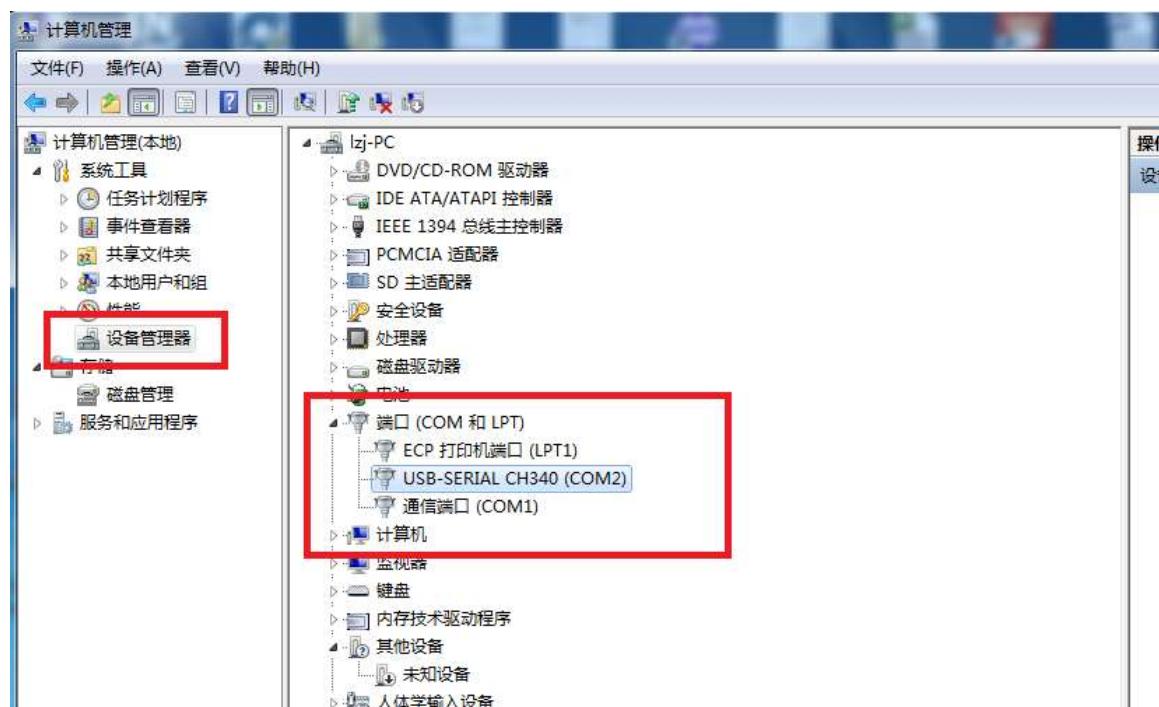


图 9 设备管理器

这里我的是 COM2 ,



图 10 配置界面

单击 确定按钮，连接建立成功

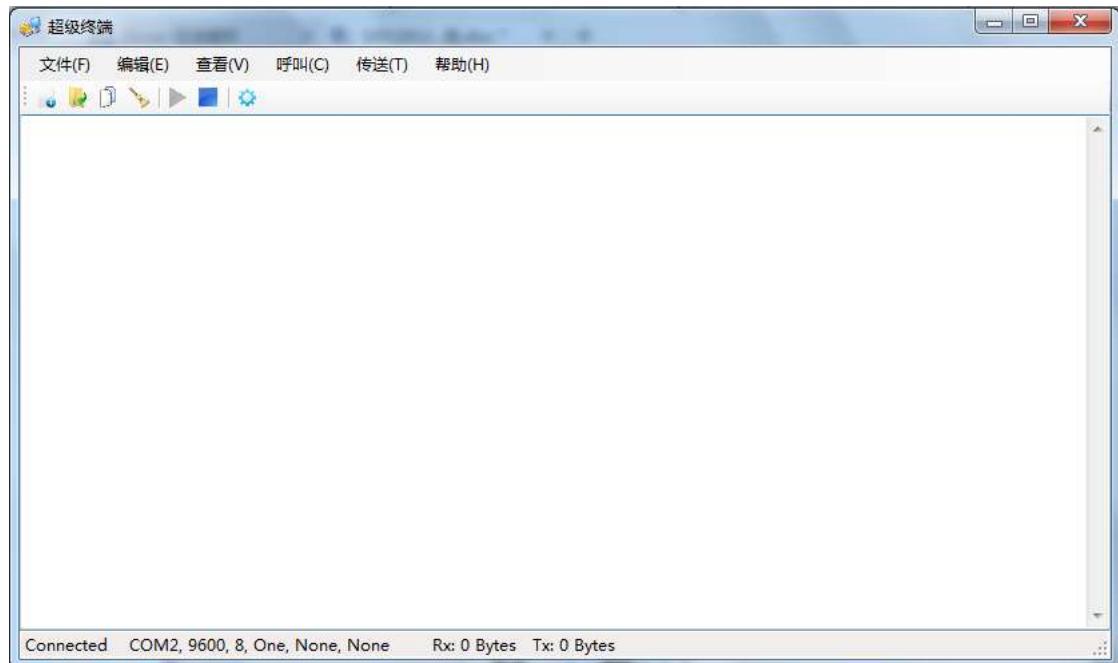


图 11 开启串口终端

- (2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- (3) 单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。（注：此时认为兄弟你已经会设置配置文件，并将其设置为默认配置）。

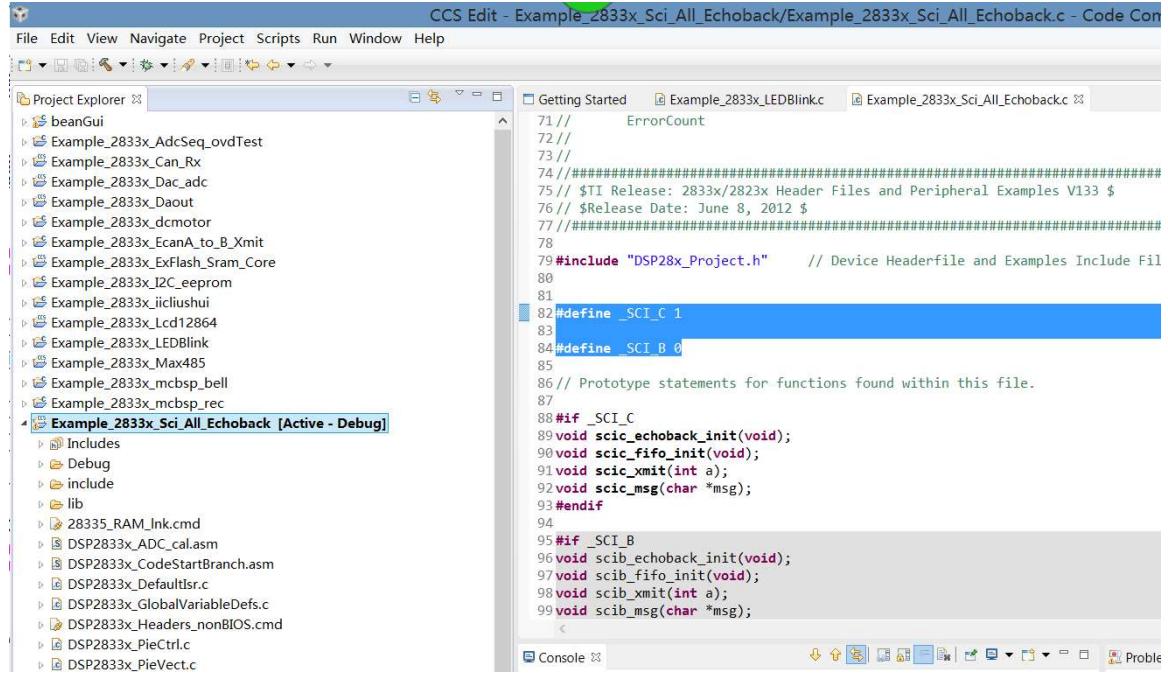


图 12 在线调试程序

- (4) 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

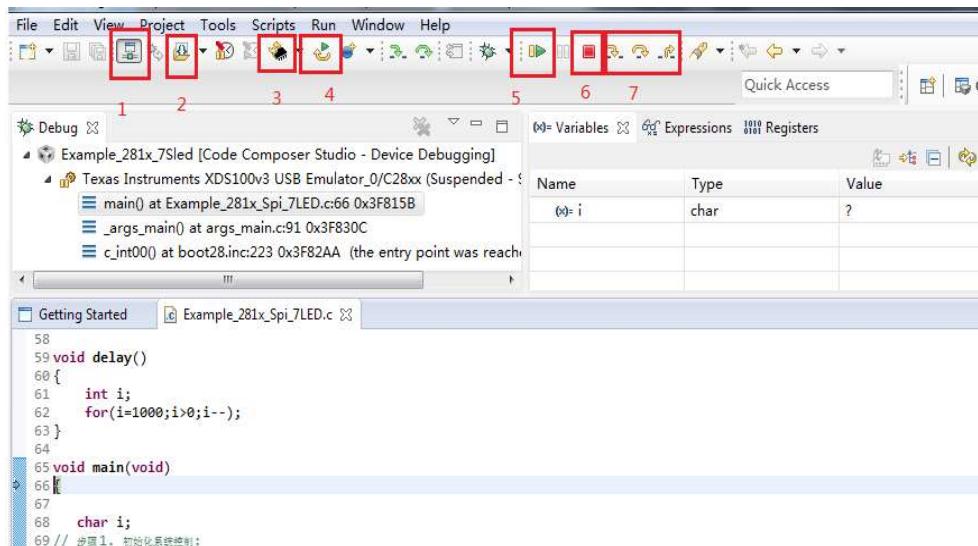


图 13 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t ；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行。

四 试验现象：在这里我们一次输入一个字母；

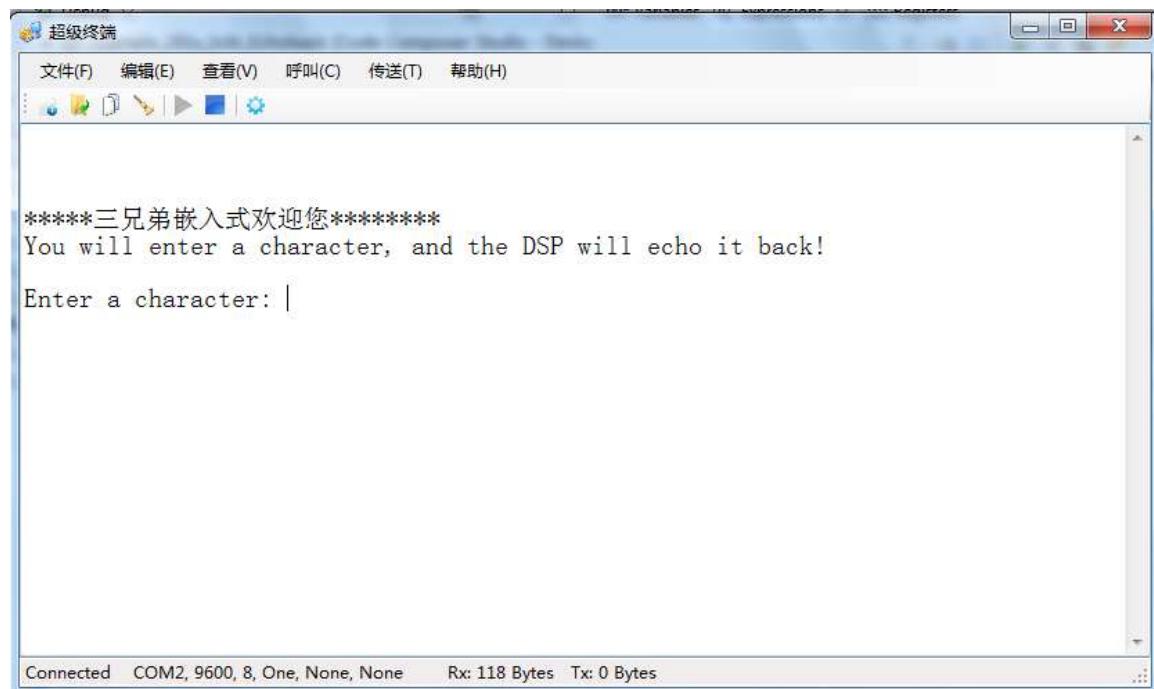
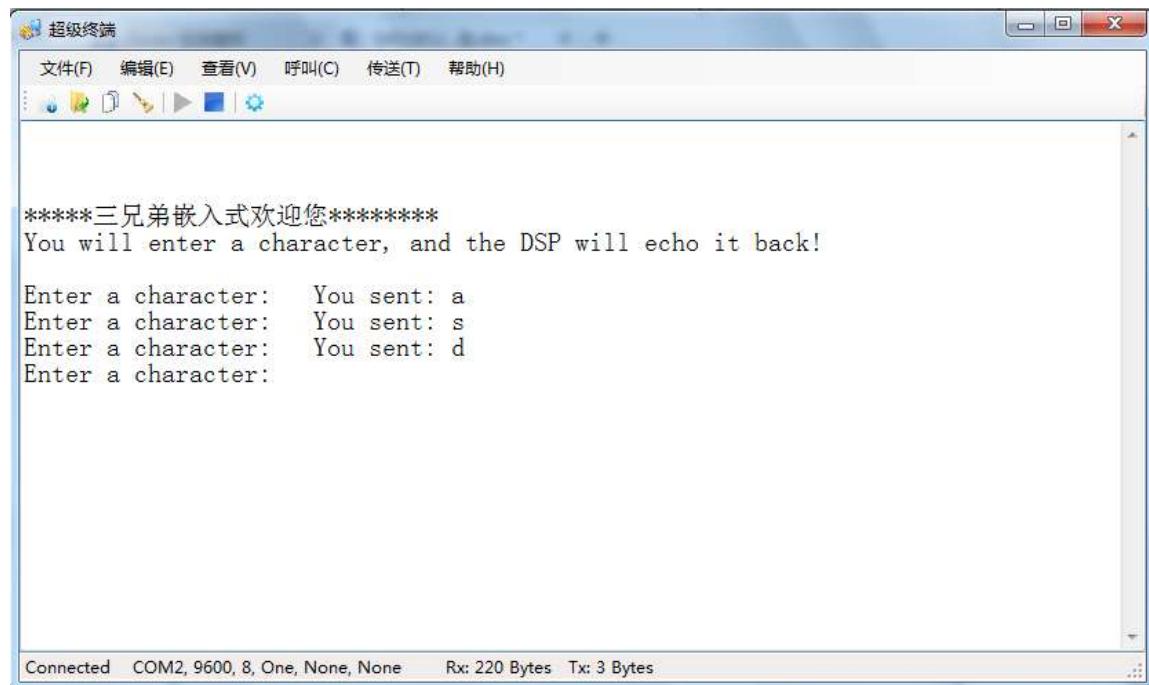


图 14 调试结果

可以看到输入的字母被返回；



程序解析

串行通信接口 (SCI) 模块 (SCI-A, SCI-B, SCI-C)

该器件包括三个串行通信接口 (SCI) 模块。SCI 模块支持 CPU 与其它异步外设之间的使用标准非归零码 (NRZ) 格式的数字通信。SCI 接收器和发射器是双缓冲的，并且它们中的每一个有其自身独立的使能和中断位。两个器件都可独立或者同时地运行在全双工模式。为了确保数据完整性，SCI 在中断检测、奇偶校验、超载、和组帧错误方面对接收到的数据进行检查。通过一个 16 位波特率选择寄存器，可将比特率设定为超过 65000 个不同的速度。

每个 SCI 模块的特性包括：

- 两个外部引脚：
 - SCITXD: SCI 发送-输出引脚
 - SCIRXD: SCI 接收-输入引脚
 - 注释：两个引脚如果不被用于 SCI 的话，可被用作 GPIO。
 - 波特率被设定为 64K 个不同速率：

$$\text{Baudrate} = \frac{\text{LSPCLK}}{(\text{BRR} + 1) * 8} \quad \text{when } \text{BRR} \neq 0$$

$$\text{Baudrate} = \frac{\text{LSPCLK}}{16} \quad \text{when } \text{BRR} = 0$$

- 数据-字格式
 - 一个开始位
 - 数据-字长度可被设定为 1 至 8 位
 - 可选偶/奇/无奇偶校验位
 - 一个或者两个停止位
- 四个错误检测标志：奇偶、超载、组帧、和中断检测
- 两个唤醒多处理器模式：空闲线路和地址位
- 半双工或者全双工运行
- 双缓冲接收和发送功能

玻尔电子致力于 C2000 全系列开发平台及应用方案的推广

- 可通过带有状态标志的中断驱动或者轮询算法来完成发射器和接收器操作。
- 发射器: TXRDY 标志 (发射器缓冲寄存器已经准备好接收另外字符) 和 TX EMPTY (TX 空) 标志 (发射器移位寄存器已空)
- 接收器: RXRDY 标志 (接收器缓冲寄存器已经准备好接收另外的字符), BRKDT 标志 (发生了中断条件), 和 RX ERROR 错误标志 (监控四个中断条件)
- 用于发射器和接收器中断的独立使能位 (除了 BRKDT)
- NRZ (非归零码) 格式

增强型特性:

- 自动波特率检测硬件逻辑电路
- 16 级发送/接收 FIFO

SCI 端口运行由下面 3 个表中列出的寄存器配置和控制。

表 4-10. SCI-A 寄存器⁽¹⁾

名称	地址	大小 (x 16)	说明
SCICCRA	0x7050	1	SCI-A 通信控制寄存器
SCICCTL1A	0x7051	1	SCI-A 控制寄存器 1
SCIHBAUDA	0x7052	1	SCI-A 波特率寄存器, 高位
SCILBAUDA	0x7053	1	SCI-A 波特率寄存器, 低位
SCICCTL2A	0x7054	1	SCI-A 控制寄存器 2
SCIRXSTA	0x7055	1	SCI-A 接收状态寄存器
SCIRXEMUA	0x7056	1	SCI-A 接收仿真数据缓冲寄存器
SCIRXBUFA	0x7057	1	SCI-A 接收数据缓冲寄存器
SCITXBUFA	0x7059	1	SCI-A 发送数据缓冲寄存器
SCIFFTXA ⁽²⁾	0x705A	1	SCI-A FIFO 发送寄存器
SCIFFRXA ⁽²⁾	0x705B	1	SCI-A FIFO 接收寄存器
SCIFFCTA ⁽²⁾	0x705C	1	SCI-A FIFO 控制寄存器
SCIPRIA	0x705F	1	SCI-A 优先级控制寄存器

(1) 这个表中的寄存器被映射到外设帧 2 空间。这空间只允许 16 位访问。32 位访问会产生未定义的后果。

(2) 这些寄存器是用于 FIFO 模式的全新寄存器。

表 4-11. SCI-B 寄存器⁽¹⁾⁽²⁾

名称	地址	大小 (x 16)	说明
SCICCRB	0x7750	1	SCI-B 通信控制寄存器
SCICCTL1B	0x7751	1	SCI-B 控制寄存器 1
SCIHBAUDB	0x7752	1	SCI-B 波特率寄存器, 高位
SCILBAUDB	0x7753	1	SCI-B 波特率寄存器, 低位
SCICCTL2B	0x7754	1	SCI-B 控制寄存器 2
SCIRXSTB	0x7755	1	SCI-B 接收状态寄存器
SCIRXEMUB	0x7756	1	SCI-B 接收仿真数据缓冲寄存器
SCIRXBUFB	0x7757	1	SCI-B 接收数据缓冲寄存器
SCITXBUFB	0x7759	1	SCI-B 发送数据缓冲寄存器
SCIFFTXB ⁽²⁾	0x775A	1	SCI-B FIFO 发送寄存器
SCIFFRXB ⁽²⁾	0x775B	1	SCI-B FIFO 接收寄存器
SCIFFCTB ⁽²⁾	0x775C	1	SCI-B FIFO 控制寄存器
SCIPRB	0x775F	1	SCI-B 优先级控制寄存器

(1) 这个表中的寄存器被映射到外设帧 2 空间。这空间只允许 16 位访问。32 位访问会产生未定义的后果。

(2) 这些寄存器是用于 FIFO 模式的全新寄存器。

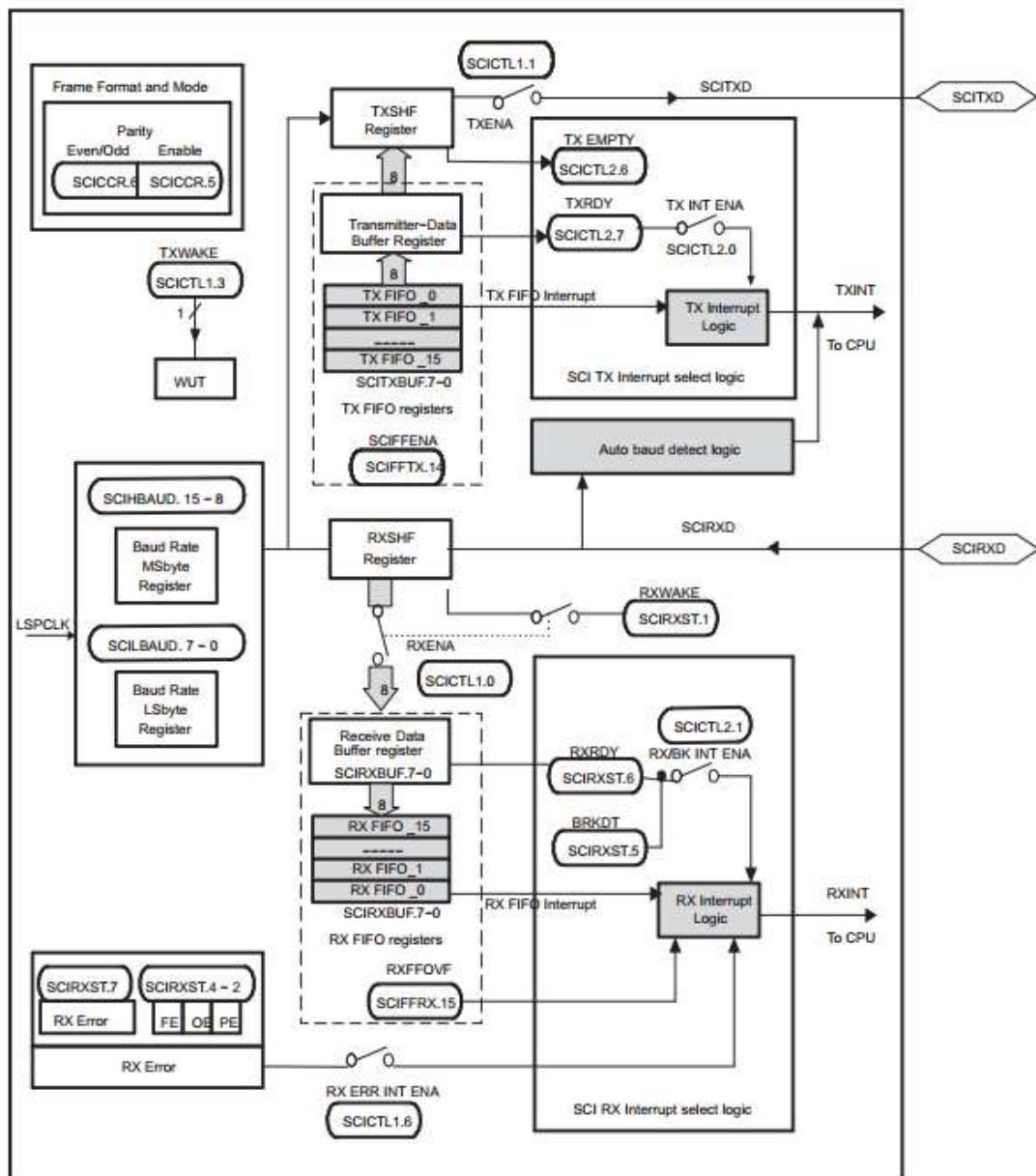
表 4-12. SCI-C 寄存器⁽¹⁾⁽²⁾

名称	地址	大小 (x 16)	说明
SCICCRC	0x7770	1	SCI-C 通信控制寄存器
SCICTL1C	0x7771	1	SCI-C 控制寄存器 1
SCIHBAUDC	0x7772	1	SCI-B 波特率寄存器, 高位
SCILBAUDC	0x7773	1	SCI-C 波特率寄存器, 低位
SCICTL2C	0x7774	1	SCI-C 控制寄存器 2
SCIRXSTC	0x7775	1	SCI-C 接收状态寄存器
SCIRXEMUC	0x7776	1	SCI-C 接收仿真数据缓冲寄存器
SCIRXBUC	0x7777	1	SCI-C 接收数据缓冲寄存器
SCITXBUC	0x7779	1	SCI-C 传输数据缓冲寄存器
SCIFFTXC ⁽²⁾	0x777A	1	SPI-C FIFO 发送寄存器
SCIFFRXC ⁽²⁾	0x777B	1	SPI-C FIFO 接收寄存器
SCIFFCTC ⁽²⁾	0x777C	1	SPI-C FIFO 控制寄存器
SCIPRC	0x777F	1	SPI-C 优先级控制寄存器

(1) 这个表中的寄存器被映射到外设帧 2 空间。这空间只允许 16 位访问。32 位访问会产生未定义的后果。

(2) 这些寄存器是用于 FIFO 模式的全新寄存器。

Figure 1-2. Serial Communications Interface (SCI) Module Block Diagram



```
#include "DSP28x_Project.h"      // Device Headerfile and Examples Include File
#define _SCI_A 1      //宏定义，用来决定哪个 SCI 口将被使用，1 有效（正在使用）
                     //如果 SCI_A 口被使用，则 USB 转串口就要接到 J3（最右边的座子
                     上）
#define _SCI_C 0 //宏定义，用来决定哪个 SCI 口将被使用，1 有效（正在使用）
                     //如果 SCI_C 口被使用，则 USB 转串口就要接到 J4（中间的座子上）

#if _SCI_A
    InitSciaGpio(); //配置哪两个 IO 口为串口
#endif

msg = "\r\n\nHello World!\0";
scia_msg(msg); //串口 A 将 msg 字符串发送出去。
for(;;)
{
    msg = "\r\nEnter a character: \0";
    scia_msg(msg); //发送提示符 Enter a character:
    // Wait for inc character
    while(SciaRegs.SCIFFRX.bit.RXFFST !=1) { } // 等待接收标志
    // Get character
    ReceivedChar = SciaRegs.SCIRXBUF.all; //接收字符
    // Echo character back
    msg = " You sent: \0";
    scia_msg(msg);
    scia_xmit(ReceivedChar); //将接收的字符发送出去
    LoopCount++;
}
```

实验 17：FIFO 模式下串口实验(Example_2833xScia_FFDLB)

一 实验目的：

- ◆ 熟悉 TMS320F28335 的 SCI 配置过程；
- ◆ 学会在 SCI 的 FIFO 模式下实现通信；

二 实验设备

- ◆ 计算机（已安装 CCSv6.0 开发环境）
- ◆ SXD28335 或 SXD28335B 开发板
- ◆ 5V 2A (或 3A)DC 电源
- ◆ 仿真器（本手册都是以三兄弟嵌入式生产的 XDS100V3 仿真器为例，其余仿真器类似）

三 实验步骤

(1) 首先将 CCS6.0 开发环境打开；

由于此程序是自测试程序，所以不需要使用外部的硬件，SCI 的外设已经在芯片内部构成了一个回路。

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 单击鼠标左键选择要调试的工程：

(5) 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

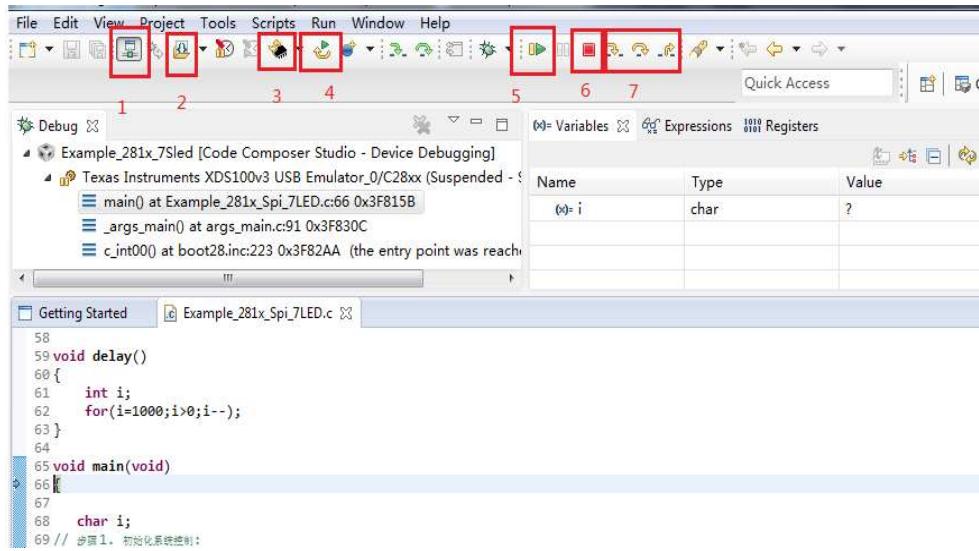


图 1 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t ；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 试验现象：

在试验前先将下面四个变量添加到观察窗口中。

LoopCount - 发送字节的数目

ErrorCount - 对接收数据进行检查，如果不发送的数据和接收的数据不一致则 ErrorCount 将增加

SendChar - 发送字节

ReceivedChar - 接收字节

方法：

1、选中要添加到观察窗口的变量，如 ReceivedChar；

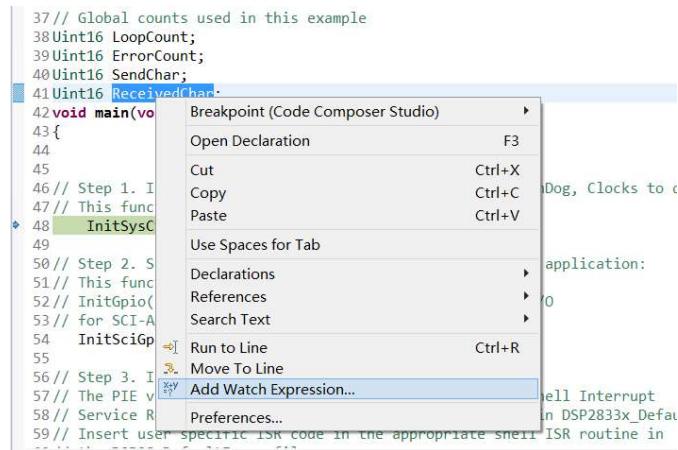


图 2 添加变量到观察窗口

首先双击 ReceivedChar->右键->Add Watch Expression->ok;

我们直接单击全速运行图标，这时程序会全速运行。

Expression	Type	Value	Address
LoopCount	unsigned int	3050	0x0000C001@Data
ErrorCount	unsigned int	0	0x0000C000@Data
SendChar	unsigned int	0x0016 (Hex)	0x0000C003@Data
ReceivedChar	unsigned int	0x0029 (Hex)	0x0000C002@Data
Add new expression			

图 3 实验现象

程序解析：

```

scia_fifo_init(); // 初始化 SCI FIFO 寄存器，使能 FIFO
scia_loopback_init(); // 初始化 SCI 功能，使能自循环模式
SendChar = 0;
// 此程序功能是循环发送数据 0x00 到 0xff。每发送一次就检测一下接收的数据是否正确。
for(;;)
{
    scia_xmit(SendChar); // 发送字节
    while(SciaRegs.SCIRXBUF.all != 0) { } // 检测是否接收到数据，如果没有接
    ReceivedChar = SciaRegs.SCIRXBUF.all; // 读取接收到的数据
    if(ReceivedChar != SendChar) error(); // 判断发送的字节和接收的字节是否一致，如果不
    一致则跳到 error() 函数

    SendChar++; // 发送字节自动加一
    SendChar &= 0x00FF;
    LoopCount++; // 发送计数器累加
}

```

实验 18：FIFO 中断模式下串口实验 (Example_2833xSci_FFDLB_int)

一 实验目的：

- 熟悉 TMS320F28335 的 SCI 配置过程；

◆ 学会基于 FIFO 中断的 SCI 程序；

二 实验设备

◆ 计算机（已安装 CCSv6.0 开发环境）

◆ SXD28335 或 SXD28335B 开发板

◆ 5V 2A (或 3A)DC 电源

◆ 仿真器（本手册都是以三兄弟嵌入式生产的 XDS100V3 仿真器为例，其余仿真器类似）

三 实验步骤

(1) 首先将 CCS6.0 开发环境打开；

由于此程序是自测试程序，所以不需要使用外部的硬件，SCI 的外设已经在芯片内部构成了一个回路。

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处；

(3) 单击鼠标左键选择要调试的工程；

(6) 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

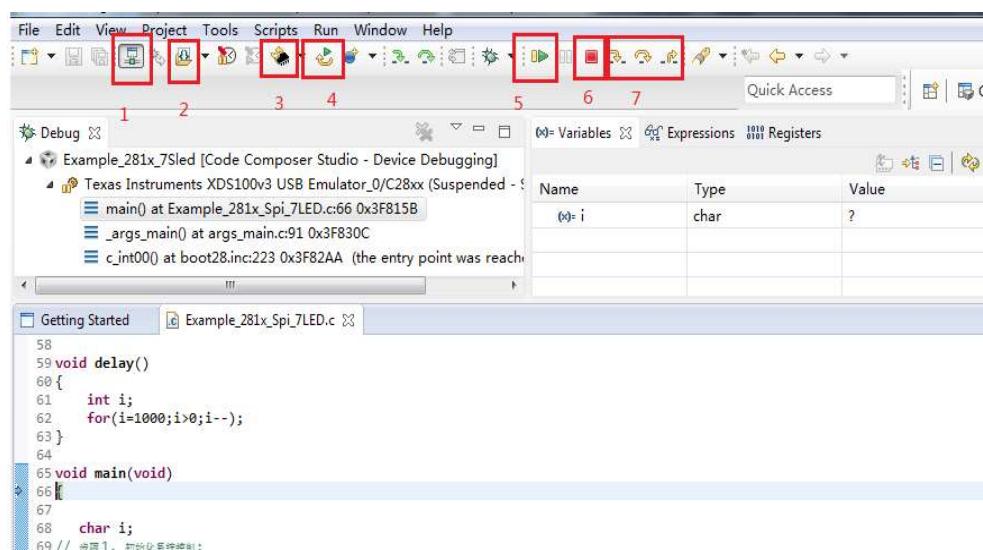


图 1 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 CPU 软 Reset；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 试验现象：

在运行前需要将下面四个变量添加到观察窗口；

```
sdataA[8]; // SCI-A 的发送数据  
sdataB[8]; // SCI-B 的发送数据  
rdataA[8]; // SCI-A 的接收数据  
rdataB[8]; // SCI-B 的接收数据
```

方法如下，选中变量->Add Watch Expression->ok；

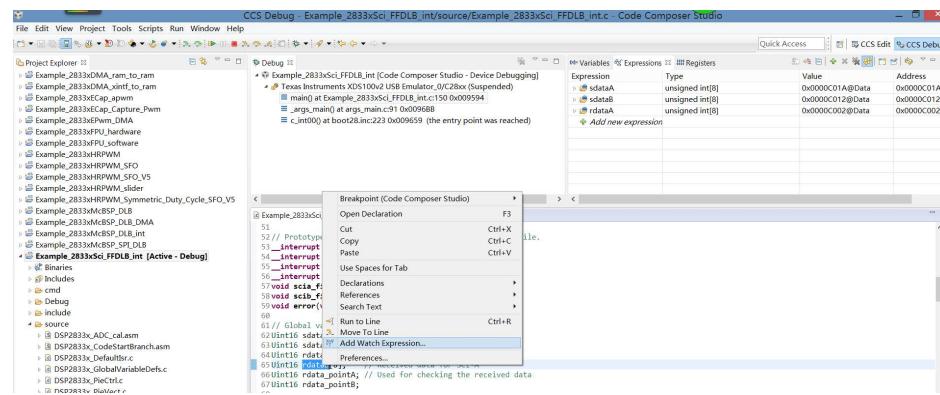


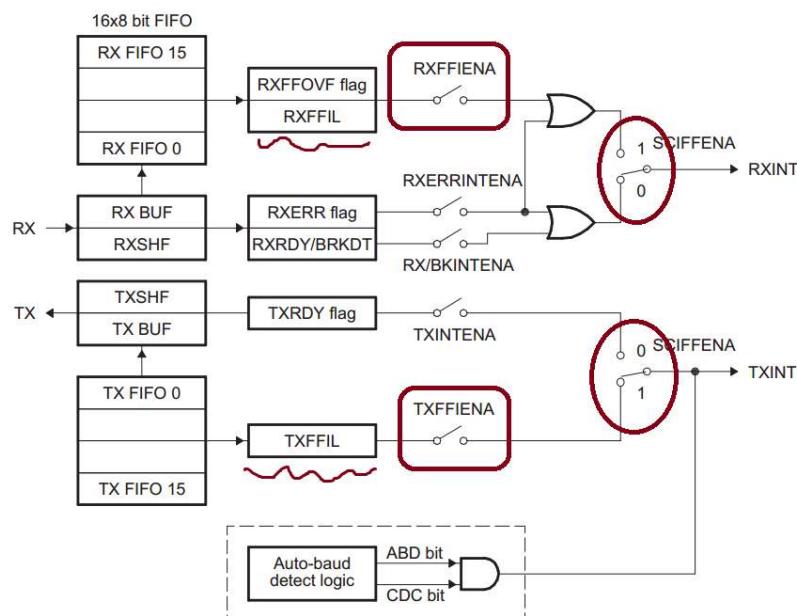
图 2 添加变量到观察窗口中

点击全速运行图标；

Expression	Type	连续刷新按钮	Value	Address
sdataA	unsigned int[8]		0x0000C01A@Data	0x0000C01A@
[0]	unsigned int		207	0x0000C01A@
[1]	unsigned int		209	0x0000C01B@
[2]	unsigned int		211	0x0000C01C@
[3]	unsigned int		213	0x0000C01D@
[4]	unsigned int		215	0x0000C01E@
[5]	unsigned int		216	0x0000C01F@
[6]	unsigned int		218	0x0000C020@
[7]	unsigned int		220	0x0000C021@
ddataA	unsigned int[8]		0x0000C012@Data	0x0000C012@
[0]	unsigned int		72	0x0000C002@
[1]	unsigned int		75	0x0000C003@
[2]	unsigned int		77	0x0000C004@
[3]	unsigned int		79	0x0000C005@
[4]	unsigned int		80	0x0000C006@
[5]	unsigned int		82	0x0000C007@
[6]	unsigned int		87	0x0000C008@
[7]	unsigned int		89	0x0000C009@
rdataA	unsigned int[8]		0x0000C00A@Data	0x0000C00A@

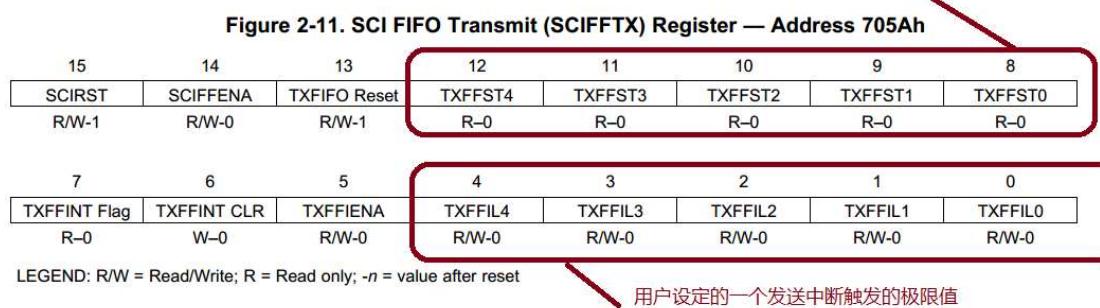
图 3 实验现象

FIFO 功能简介；



FIFO 中断功能要分两种。一种是接收 FIFO 中断，另一种是发送 FIFO 中断；首先介绍一下发送 FIFO 中断：

2.9 SCI FIFO Registers (SCIFFTX, SCIFFRX, SCIFFRCT)



发送 FIFO 中断的条件是：

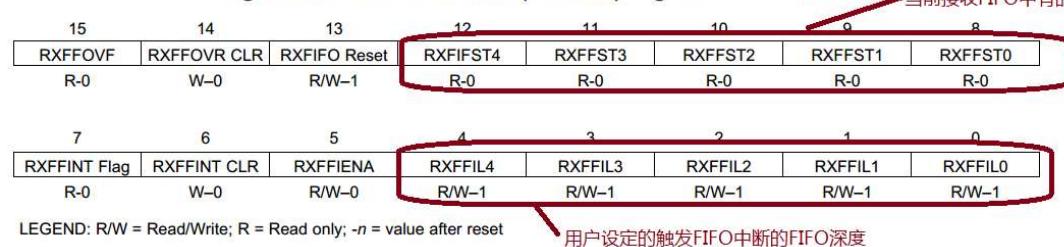
- 1、SCIFFTX 寄存器中表示当前发送 FIFO 中还没发送的字节数目 (TXFFST) 和设置的字节数相等或小于设置的值 (TXFFIL) 时。
- 2、TXFFIENA 被使能。
- 3、SCIFFENA 被使能。

那么发生中断后，就要往 FIFO 中放数据，怎么放数据呢，下面是以 FIFO 的深度为 8 的发送中断为例：

```
_interrupt void scibTx_fifoIsr(void)
{
    Uint16 i;
    //由于 FIFO 的深度为 8，所以我们向 FIFO 中放入 8 个数据
    for(i=0; i< 8; i++)
    {
        ScibRegs. SCITXBUF=sdataB[i];
    }
    //将发送的数据进行减 1 操作，那么数据会随着发送次数的增加数值逐渐减少
    for(i=0; i< 8; i++)
    {
        sdataB[i] = (sdataB[i]-1) & 0x00FF;
    }
    ScibRegs. SCIFFTX.bit.TXFFINTCLR=1; // 清除中断标志
    PieCtrlRegs. PIEACK.all |=0x100; // Issue PIE ACK
}
```

接收 FIFO 中断的条件是：

Figure 2-12. SCI FIFO Receive (SCIFFRX) Register — Address 705Bh



- 4、SCIFFRX 寄存器中表示当前接收 FIFO 中接收字节数目 (RXFFST) 和设置的

字节数相等或大于设置的值(RXFFIL)时。

5、RXFFIENA 被使能。

6、SCIFFENA 被使能。

下面介绍一下接收中断的实现过程;

```
_interrupt void sciaRx_fifoIsr(void)
{
    Uint16 i;
    for(i=0;i<8;i++)
    {
        rdataA[i]=SciaRegs.SCIRXBUT.all; //一次性读取 8 个字节，因为 FIFO 的深度为 8
    }
    for(i=0;i<8;i++) // 检查接收的数据是否与发送的数据一致
    {
        if(rdataA[i] != (rdata_pointA+i) & 0x00FF) error();
    }
    rdata_pointA = (rdata_pointA+1) & 0x00FF;
    SciaRegs.SCIFFRX.bit.RXFFOVRCRLR=1; //清除溢出标志
    SciaRegs.SCIFFRX.bit.RXFFINTCLR=1; // 清除中断标志
    PieCtrlRegs.PIEACK.all|=0x100; // Issue PIE ack
}
```

程序解释:

```
// 将中断函数映射到中断向量表中
EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.SCIRXINTA = &sciaRx_fifoIsr;
PieVectTable.SCITXINTA = &sciaTx_fifoIsr;
PieVectTable.SCIRXINTB = &scibRx_fifoIsr;
PieVectTable.SCITXINTB = &scibTx_fifoIsr;
EDIS; // This is needed to disable write to EALLOW protected registers
// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP2833x_InitPeripherals.c
// InitPeripherals(); // Not required for this example
    scia_fifo_init(); // Init SCI-A
    scib_fifo_init(); // Init SCI-B
```

实验 19：MAX485 总线实验(Example_2833x_Max485)

一 实验目的：

✧ 了解如何使用 SCI 总线实现 485 总线通信；

✧ 了解如何使用 SP3485EN 芯片；

二 实验设备：

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, 485 总线收发器一个;
- ✧ 用户要有 485 收发器, (可以跟我们联系进行购买)

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

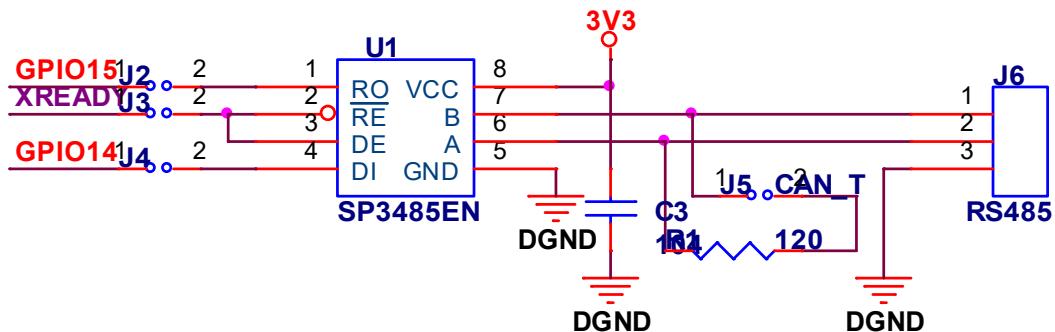


图 1 MAX485 原理图

与 485 总线收发器的连接方法如下图(示意图, 实际接插件位号如上图):

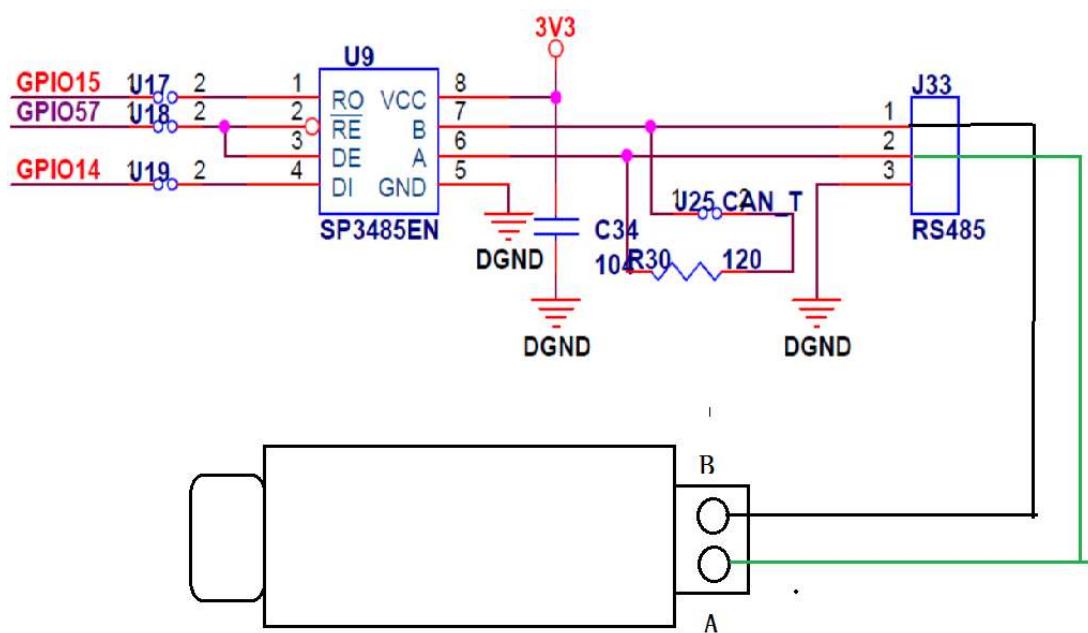


图 2 与 485 总线收发器的连接方法

用跳线帽将 J2、J3、J4 和 J5 链接上。

◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG

端插到 SXD28335 开发板的 JATG 针处；

◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。（注：此时

认为兄弟你已经会设置配置文件，并将其设置为默认配置）。

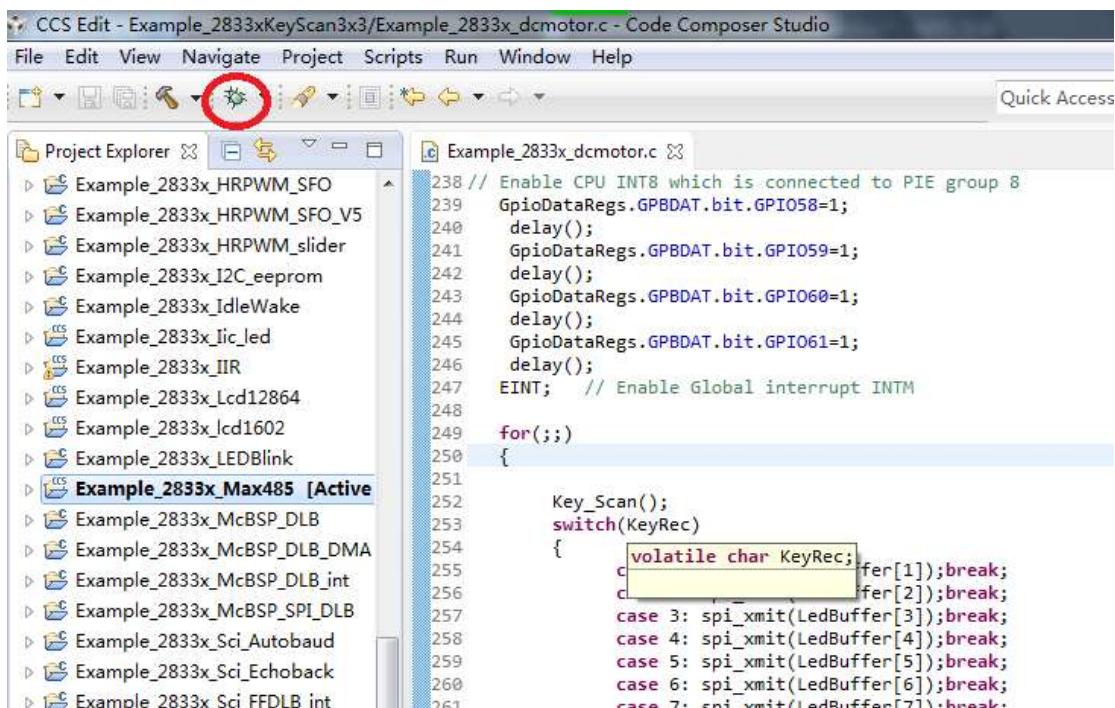


图 3 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的

界面。

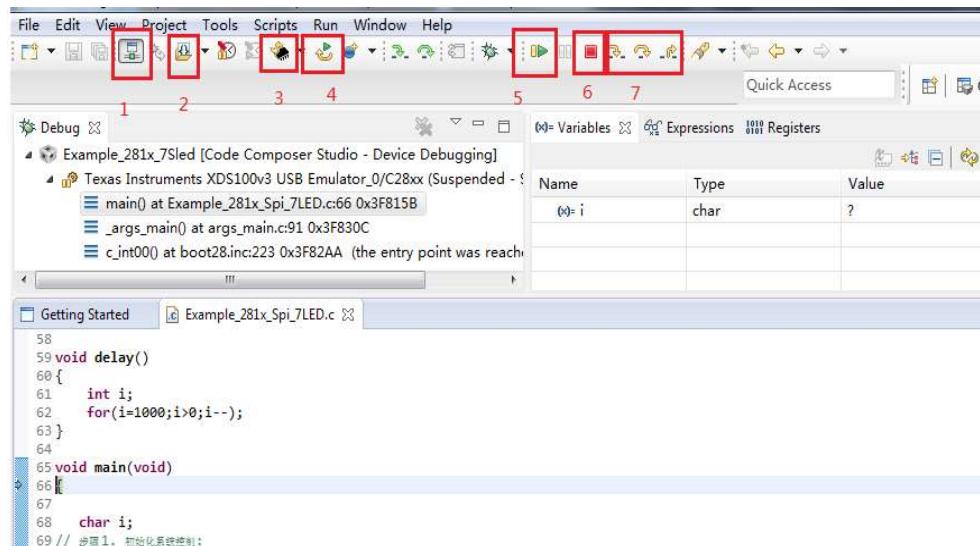


图 4 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

插上我们的 485 模块到电脑上，并且这个模块的 B 接开发板 MAX485 接口的 B，A 接开发板 MAX485 接口的 A 信号线；打开串口终端，进行如下配置（注：客户要根据自己电脑）；



图 6 串口配置图

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行

四 试验现象：

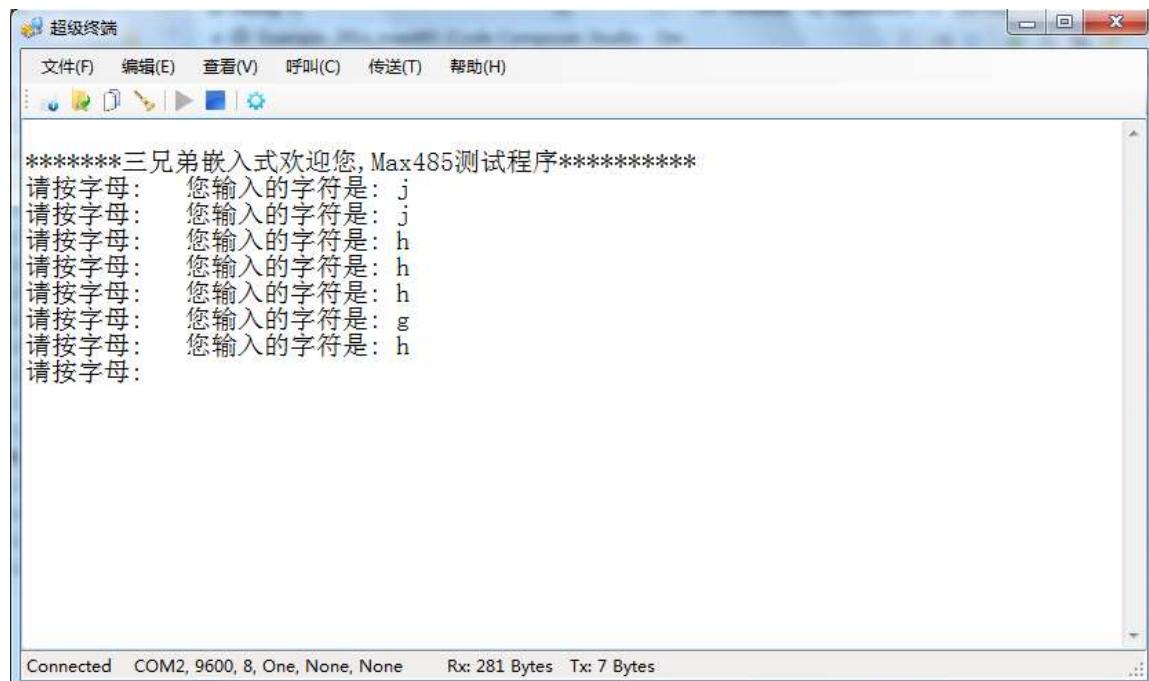


图 7 实验现象

程序解析

```
//收发使能控制接口的配置  
EALLOW;  
GpioCtrlRegs.GPBMUX1.bit.GPIO34 =0; //配置为普通 IO 口  
GpioCtrlRegs.GPBDIR.bit.GPIO34=1; //配置为输出
```

```
GpioCtrlRegs.GPBPUD.bit.GPIO34=0; //使能上拉  
GpioDataRegs.GPBSET.bit.GPIO34=1; //置 1  
EDIS;  
scib_echoback_init(); // 初始化 SCI 为循环模式  
Transmit_EN; //在发送数据前要先使能发送  
DELAY_US(5);  
msg = "\r\n*****三兄弟嵌入式欢迎您, Max485 测试程序*****\0";  
scib_msg(msg); //将 msg 发送出去
```

实验 20: SPI_DA 程序实验 (Example_2833x_Daout)

一 实验目的:

- ✧ 熟悉 TMS320F28335 的 SPI 工作原理;
- ✧ 学会使用 DA 芯片;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, 万用表一个;

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

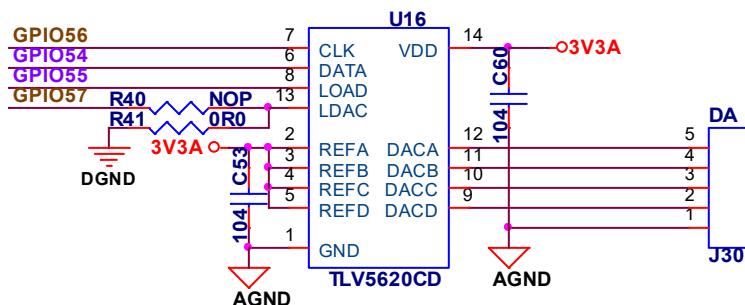


图 1 硬件连接原理图

- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;

◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。

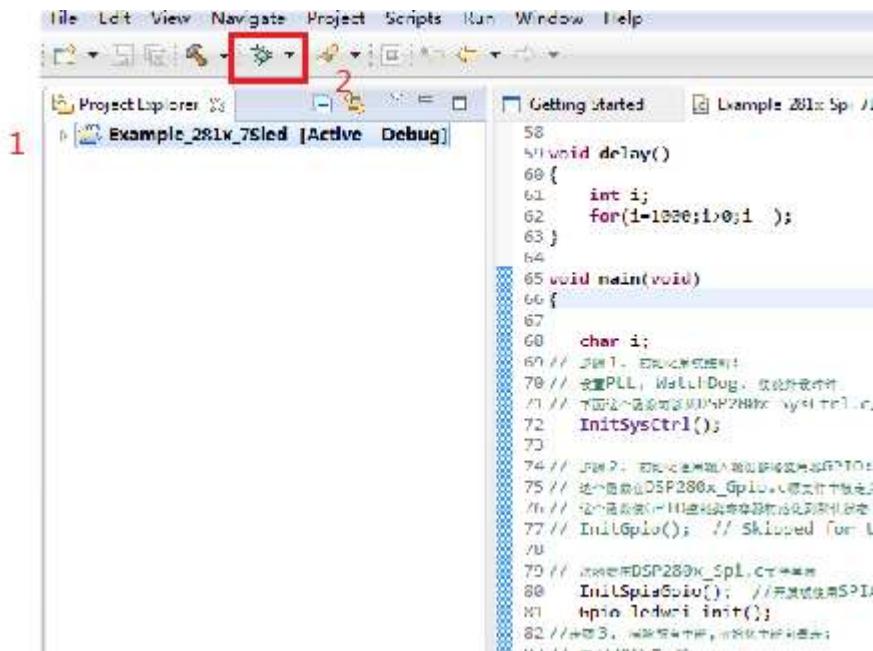


图 2 程序调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

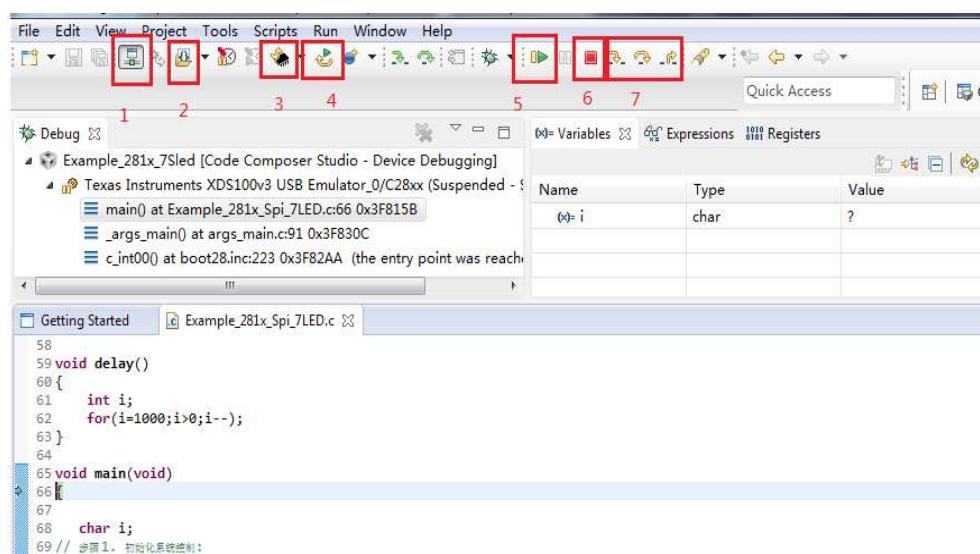


图 3 程序调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t ;

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行。

四 实验现象：

利用万用表测量，万用表的地接到 J30 的 1 脚。红色表笔测量 J30 的脚 2、3、4 和 5，其数值分别对应着 0.4v、0.8v、1.2v 和 1.6v 左右。

SPI 模块的设置

SPI 模块的波特率可由如下两种情况计算得出：

①SPIBRR=3~127， 波特率的计算公式为：

$$\text{SPI 波特率} = \text{LSPCLK} / (\text{SPIBRR} + 1)$$

②SPIBRR=0~2， 波特率的计算公式为：

$$\text{SPI 波特率} = \text{LSPCLK} / 4$$

LSPCLK 是 TMS320F28335 的低速外设时钟频率； SPIBRR 是 SPIBRR 寄存器的值。将 LSPCLK 设置成 37.5 MHz。

通信中将 SPISIMO、SPISOMI 和 SPICLK 设置为基本功能 SPI 口， SPISTE 设置为一般 I/O 口。当主 / 从控制器进行数据交换

时，SPISTE 配置成低电平，数据传输结束后再配置成高电平。

TMS320F28335 的数据寄存器都是 16 位的，且接收和发送都是双缓冲的，发送缓冲寄存器 SPITXBUF 中的数据以左对齐的方式发送，先发送数据的最高位，因此在发送数据前必须将等待发送的数据放在 SPITXBUF 的高 8 位。TMS320F28335 是以右对齐的方式来接收数据的，8 位的数据被放在 SPITXBUF 的低 8 位上。

SPI 模块有 4 种时钟模式：上升沿无延时模式、上升沿有延时模式、下降沿无延时模式和下降沿有延时模式。

接收数据流程

通过 SPI 读取数据比较简单，只需要依次传送读指令和待读数据的低位地址，就可以在 SPISOMI 引脚上接收到数据。因为 TMS320F28335 为主控制器，所以必须先发送一个无意义的数，才能够启动时钟。在 SPI 状态寄存器(SPISTS)中有一个 SPI 中断标志位(SPIINT FLAG)，该位是一个只读标志位，由硬件设置。当 SPI 已经完成数据发送或者接收，正在等待下一步的操作时，SPI 中断标志位被置 1，若使能 SPI 中断，将产生一个 SPI 中断请求。可以通过查询 SPI 中断标志位来判断数据是否完成接收。若该标志为 1，已接收的数据将被放入接收缓冲寄存器 SPIRXBUF 中，通过读 SPIRXBUF 寄存器即可得到需要的数据。

发送数据流程

向 SPITXBUF 寄存器中写入待发的数据，SPI 时钟就会自动启

动，数据会由输出引脚顺次传出；数据传送完后，SPI 时钟自动停止。也可以通过查询 SPI 中断标志位来判断数据是否完成发送，若该标志位为 1，则可接着发送下一个数据。SPI 设置成主模式时，发送完一个数据，必须要空读一下 SPIRXBUF 寄存器，以清除 SPI 中断标志位。由于在读取数据的过程中已经包含读 SPIRXBUF 寄存器，因此在读取数据

六 SPI 程序解析

```
#include "DSP2833x_Device.h"      // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"    // DSP2833x Examples Include File
/*这里使用了宏定义来控制更新锁存信号的功能,重点就是在时序上*/
#define SetLOAD GpioDataRegs.GPBDAT.bit.GPIO55=1; //将 LOAD 置高
#define ClrLOAD GpioDataRegs.GPBDAT.bit.GPIO55=0; //将 LOAD 置低
void WriteDAC(unsigned char add,unsigned char rng,unsigned char vol);
void delay(unsigned int t);
void spi_xmit(Uint16 a);
void spi_fifo_init(void);
void spi_init(void);
void main(void)
{
    int temp;
    /*初始化系统*/
    InitSysCtrl();
    /*初始化 GPIO;*/
    InitSpiaGpio();
    ///初始化 IO 口
    EALLOW;
    GpioCtrlRegs.GPBMUX2.bit.GPIO55 = 0; // 配置 GPIO17 为 GPIO 口
    GpioCtrlRegs.GPBDIR.bit.GPIO55 = 1;      // 定义 GPIO17 输出引脚
    GpioCtrlRegs.GPBPUD.bit.GPIO55 = 0;      // 禁止上拉 GPIO17 引脚
    EDIS;
    /* 关中断 */
    DINT;
    IER = 0x0000;
    IFR = 0x0000;
    /* 初始化 PIE 控制寄存器 */
    InitPieCtrl();
    /* 初始化 PIE 参数表 */
```

```
InitPieVectTable();
//初始化 SPI
spi_init(); // 初始化 SPI
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM
SetLOAD; //把刷新锁存控制信号拉高
temp=47;//REF=2.2V;VO(DACA|B|C|D) =REF* CODE/256
while(1)
{
    WriteDAC(0, 0, temp); //0.4V
    WriteDAC(1, 0, temp*2); //0.8V
    WriteDAC(2, 0, temp*3); //1.2V
    WriteDAC(3, 0, temp*4); //1.6V
    delay(1500); //在此设断点, 观察变量 DAC0 和 DAC1 的值, 另外加三用表直接测量四路 DA 的输出电压值
}
void WriteDAC(unsigned char add, unsigned char rng, unsigned char vol)
{
    unsigned short int data;
    data=0x0000;
    ///大家要知道这里所定义的各个变量的含义, add 是 4 个通道的地址 (00, 01, 10, 11)
    ///                                              RNG 是输出范围的倍数, 可以是 0 或 1。
    ///                                              VOL 是 0~256 数据
    data = ((add<<14) | (rng<<13) | (vol<<5));
    //注意这里的有效数据是 11 位, SPI 初始化中也进行了定义
    while(SpiRegs.SPISTS.bit.BUFFULL_FLAG ==1); //判断 SPI 的发送缓冲区是否是空的, 等于 0 可写数据
    SpiRegs.SPITXBUF = data; //把发送的数据写入 SPI 发送缓冲区
    while( SpiRegs.SPISTS.bit.BUFFULL_FLAG==1); //当发送缓冲区出现满标志位时, 开始锁存数据
    delay(1500); //同通过一负跳变锁存要发送的数据, 看 TLV5620 数据手册即可得知
    ClrLOAD;
    delay(150);
    SetLOAD;
    delay(1500);
}
void delay(unsigned int t)
{
    while(t>0)
    t--;
}
//初始化 SPI 函数
void spi_init()
```

```
{  
    SpiRegs.SPICCR.all =0x0a; //进入初始状态，数据在上升沿输出，自测禁止，11位数据模式  
    SpiRegs.SPICL.all =0x0006; //使能主机模式，正常相位，使能主机发送，禁止接收  
    //溢出中断，禁止 SPI 中断；  
  
    SpiRegs.SPIBRR =0x0031; //SPI 波特率=37.5M/50=0.75MHZ;  
    SpiRegs.SPICCR.all =0x8a; //退出初始状态；  
    SpiRegs.SPIPRI.bit.FREE = 1; //自由运行  
}
```

实验 21：DA 与 AD 联合试验（Example_2833x_Dac_adc）

一、实验目的：

- ✧ 了解如何使用 dsp23885 内部集成的 ad 外设；
- ✧ 了解 TLV5620CD 这款由 SPI 控制的 DA 芯片；

二、实验设备

注：模拟量输入范围：0.0V~3.0V；

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板一套；
- ✧ 飞线一根，用于将 DA 的某一输出通道连接到 AD 的 A0 口。

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

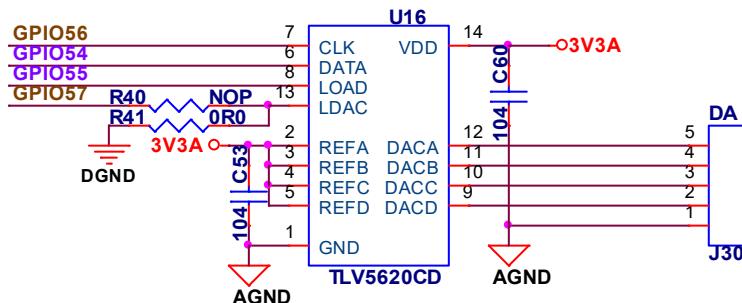


图 1 4 通道输出的 DA 硬件连接图

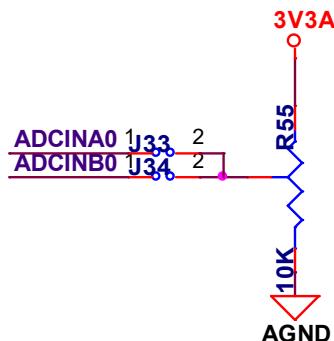


图 2 AD 输入的硬件连接图

从上图可知,为了不影响测量 DA 输出的值,我们要将 J33、J34 的跳线帽去掉。
用杜邦线将 J30 的 5 脚链接到 J33 的 1 脚(方形焊盘)。这样 DA 输出和 AD 输入构成一个回路。

- ◆ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:
然后单击图中红色的方框处的调试按钮, 进行调试。

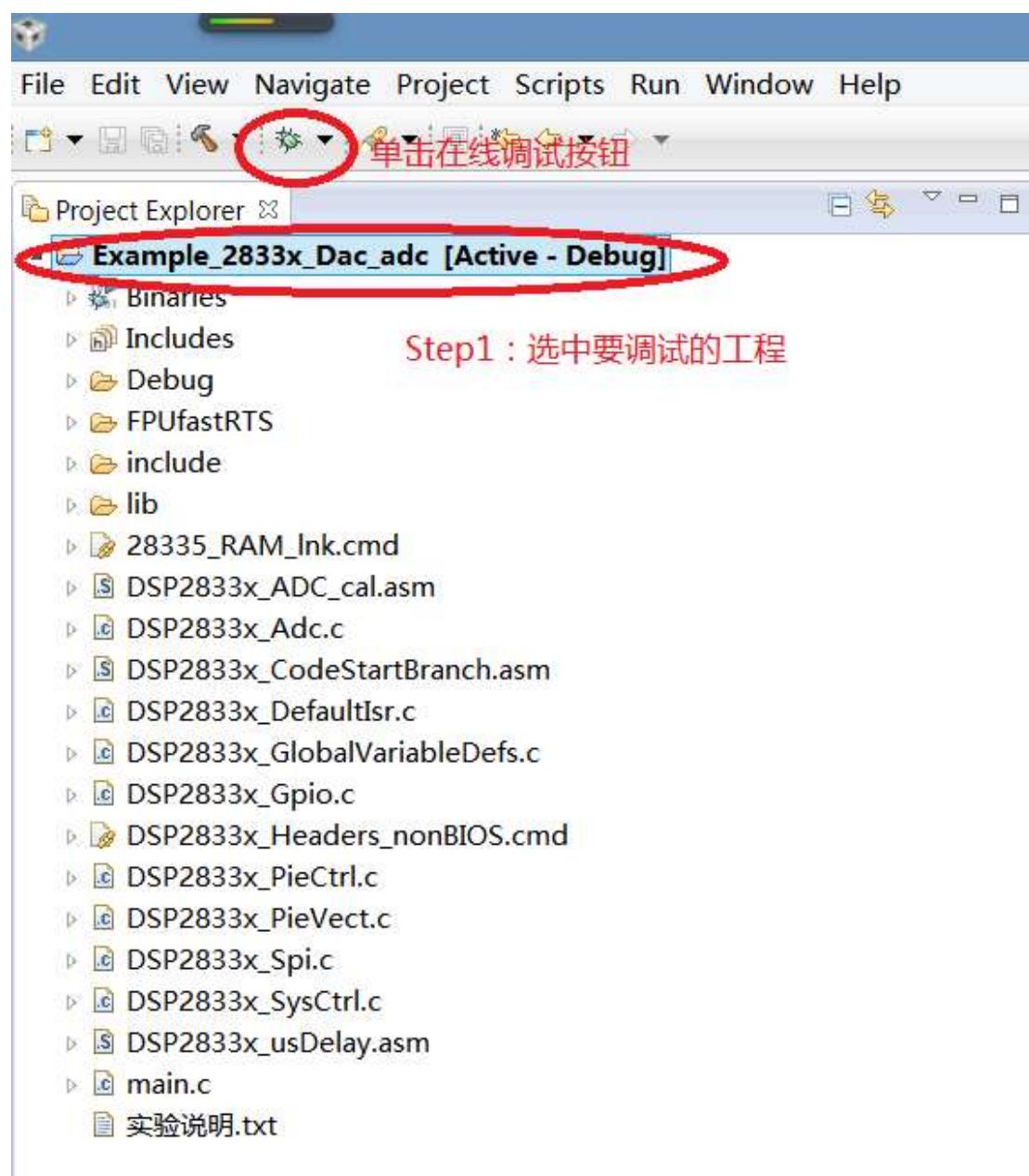


图 3 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

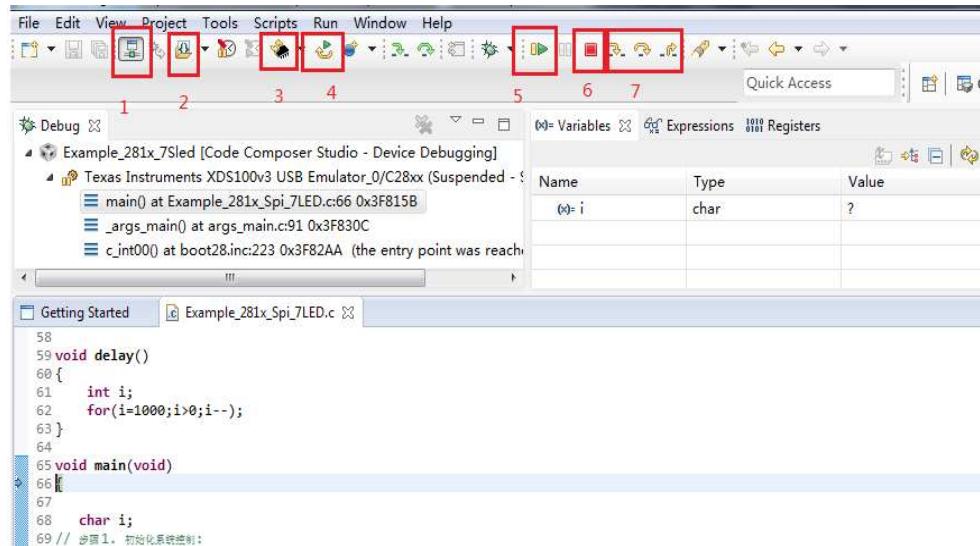


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮;
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t ;
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行;
- ✧ 图中 6 是停止调试;
- ✧ 图中 7 是用于单步调试的;

四、试验现象：

首先将 Vin (测到的 DA 输出值) 添加到观察窗口中，如下图所示：

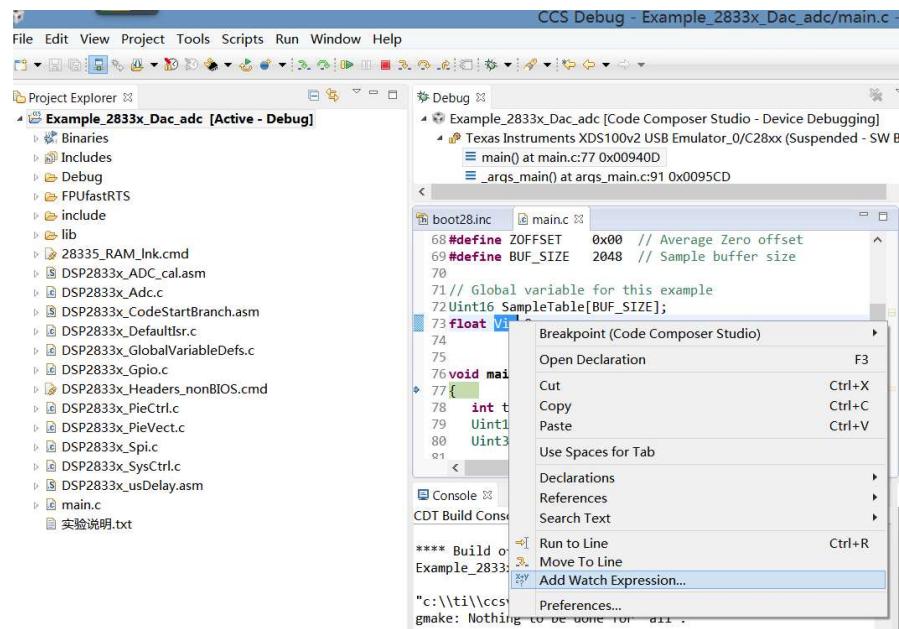


图 4 添加变量到观察窗口中

现象：

Vin 的值会根据你链接的通道不同，显示的值不同。四种可能的值：0.4V 0.8V 1.2V
1.6V。这些都是理想值，实际值会和这些值有一些偏差。

```
#include "DSP2833x_Device.h"      // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"    // DSP2833x Examples Include File
/*这里使用了宏定义来控制更新锁存信号的功能, 重点就是在时序上*/
#define SetLOAD GpioDataRegs.GPBDAT.bit.GPIO55=1; //将 LOAD 置高
#define ClrLOAD GpioDataRegs.GPBDAT.bit.GPIO55=0; //将 LOAD 置低
void WriteDAC(unsigned char add,unsigned char rng,unsigned char vol);
void delay(unsigned int t);
void spi_xmit(Uint16 a);
void spi_fifo_init(void);
void spi_init(void);
// AD 模块的时钟配置参数
#define ADC_MODCLK 0x5    // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 150/(2*4)
#define ADC_CKPS   0x1    // ADC module clock = HSPCLK/2*ADC_CKPS    = 15MHz/(1*2)
#define ADC_SHCLK  0xf    // S/H width in ADC module periods
#define AVG        100    // Average sample limit
#define ZOFFSET    0x00    // Average Zero offset
#define BUF_SIZE   2048   // Sample buffer size
// Global variable for this example
Uint16 SampleTable[BUF_SIZE];
// AD 测量到的输入值
float Vin=0;
void main(void)
{
    int temp;
    Uint16 i;
    Uint32 Sum=0;
    /*初始化系统*/
    InitSysCtrl();
    // Specific clock setting for this example:
    EALLOW;
    SysCtrlRegs.HISPCP.all = ADC_MODCLK; // HSPCLK = SYSCLKOUT/ (2*ADC_MODCLK)
    =15MHZ
    EDIS;
    /*初始化 GPIO;*/
    InitSpiaGpio();
    ///初始化 IO 口
    EALLOW;
    GpioCtrlRegs.GPBMUX2.bit.GPIO55= 0; // 配置 GPIO55 为 GPIO 口
    GpioCtrlRegs.GPBDIR.bit.GPIO55 = 1;    // 定义 GPIO55 输出引脚
    GpioCtrlRegs.GPBPU.D.bit.GPIO55 = 0;    // 禁止上拉 GPIO55 引脚
    EDIS;
    /* 关中断 */
}
```

```
DINT;
IER = 0x0000;
IFR = 0x0000;
/* 初始化 PIE 控制寄存器 */
InitPieCtrl();
/* 初始化 PIE 参数表 */
InitPieVectTable();
// 步骤 4. 初始化片内外设:
InitAdc(); // For this example, init the ADC
///初始化 SPI
spi_init(); // 初始化 SPI
EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM
// Specific ADC setup for this example:
AdcRegs.ADCTRL1.bit.ACQ_PS = ADC_SHCLK;//设置采样窗口时间: (15+1) *ADCCLK
AdcRegs.ADCTRL3.bit.ADCCLKPS = ADC_CKPS;//ADC 内核时钟分频: HSPCLK/2=6.25MHZ
AdcRegs.ADCTRL1.bit.SEQ_CASC = 1; // 1选择级联模式
AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0; //通道选择 ADCAINO
AdcRegs.ADCTRL1.bit.CONT_RUN = 1; // 设置为连续运行
// 采样表清 0
for (i=0; i<BUF_SIZE; i++)
{
    SampleTable[i] = 0;
}
SetLOAD; //把刷新锁存控制信号拉高
temp=47;//REF=2.2V;VO(DACA|B|C|D) =REF* CODE/256
        WriteDAC(0, 0, temp); //0.4V
        WriteDAC(1, 0, temp*2); //0.8V
        WriteDAC(2, 0, temp*3); //1.2V
        WriteDAC(3, 0, temp*4); //1.6V
// 软件启动 SEQ1
AdcRegs.ADCTRL2.all = 0x2000;
for (i=0; i<AVG; i++)
{
    while (AdcRegs.ADCST.bit.INT_SEQ1== 0) {} // 等待中断
    AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;
    SampleTable[i] =((AdcRegs.ADCRESULT0>>4) );
}
for (i=0;i<AVG;i++)
{
    Sum+=SampleTable[i];
    Sum=Sum/2;
}
//输入电压和 AD 值之间的关系 Vin/Sum=3/4096;
```

```
Vin=(float)(Sum*3)/4096; //采样的电压值
delay(1500); //在此设断点, 观察变量 DAC0 和 DAC1 的值, 另外加三用表直接测量四路 DA 的输出电压值
}
void WriteDAC(unsigned char add,unsigned char rng,unsigned char vol)
{
    unsigned short int data;
    data=0x0000;
    ///大家要知道这里所定义的各个变量的含义, add 是 4 个通道的地址 (00, 01, 10, 11)
    ///                                              RNG 是输出范围的倍数, 可以是 0 或 1。
    ///                                              VOL 是 0~256 数据
    data = ((add<<14) | (rng<<13) | (vol<<5));
    //注意这里的有效数据是 11 位, SPI 初始化中也进行了定义
    while(SpiRegs.SPISTS.bit.BUFFULL_FLAG ==1); //判断 SPI 的发送缓冲区是否是空的, 等于 0 可写数据
        SpiRegs.SPITXBUF = data; //把发送的数据写如 SPI 发送缓冲区
    while( SpiRegs.SPISTS.bit.BUFFULL_FLAG==1); //当发送缓冲区出现满标志位时, 开始琐存数据
        delay(1500); //同通过一负跳变琐存要发送的数据, 看 TLV5620 数据手册即可得知
        ClrLOAD;
        delay(150);
        SetLOAD;
        delay(1500);
    }
void delay(unsigned int t)
{
    while(t>0)
        t--;
}
//初始化 SPI 函数
void spi_init()
{
    SpiRegs.SPICCR.all =0x0a;//进入初始状态, 数据在上升沿输出, 自测禁止, 11 位数据模式
    SpiRegs.SPICCTL.all =0x0006; // 使能主机模式, 正常相位, 使能主机发送, 禁止接收
                                //溢出中断, 禁止 SPI 中断;
    SpiRegs.SPIBRR =0x0031; //SPI 波特率=37.5M/50=0.75MHZ;
    SpiRegs.SPICCR.all =0x8a; //退出初始状态;
    SpiRegs.SPIPRI.bit.FREE = 1; // 自由运行
}
```

实验 22：SPI 内循环程序实验（Example_2833xspi_loopback）

一 实验目的：

◆ 熟悉 TMS320F28335 的 SPI 工作原理；

二 实验设备：

◆ PC 机一台；

◆ XDS100v2 或 XDS100V3(隔离)仿真器一套；

◆ SXD28335 开发板一套；

三 实验步骤：

◆ 首先将 CCS6.0 开发环境打开；

◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处；给开发板上电。进行调试。

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

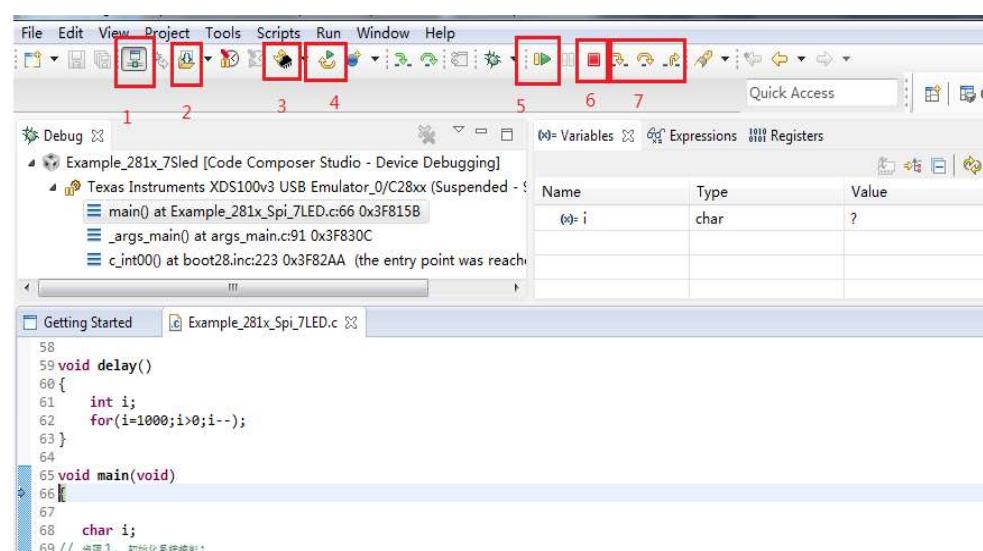


图 1 程序调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t ；

图中 4 是调试时恢复到程序的开始处。

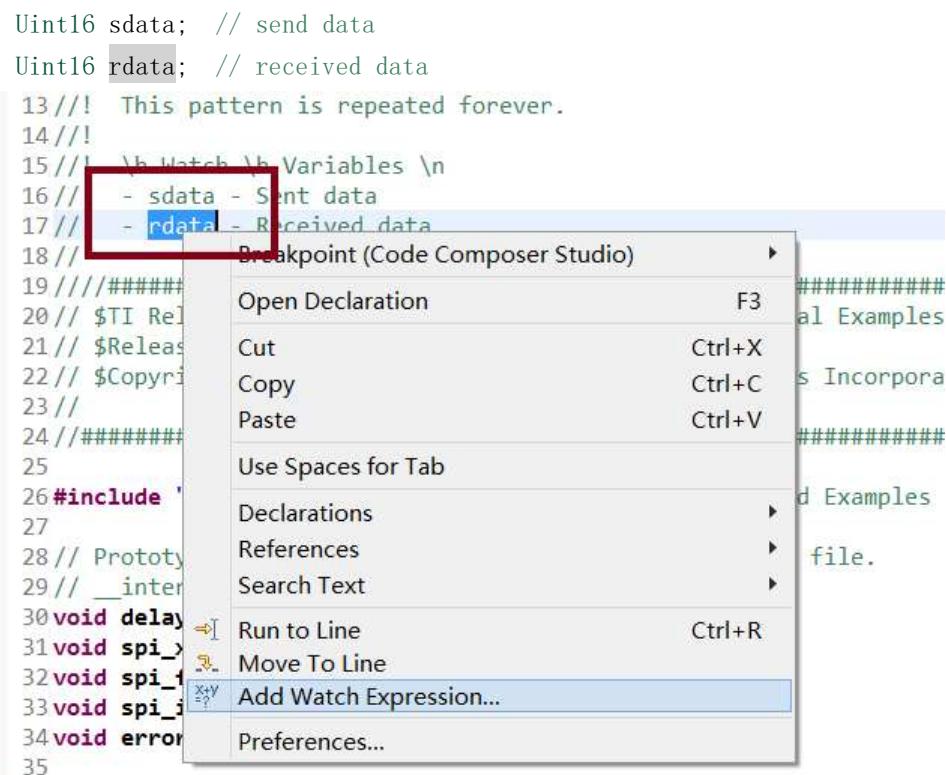
图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 实验现象：

◆ 在全速运行前需要将下面两个局部变量添加到观察窗口中。



```
Uint16 sdata; // send data
Uint16 rdata; // received data
13//! This pattern is repeated forever.
14//!
15//! \b Watch \b Variables \n
16//! - sdata - Sent data
17//! - rdata - Received data
18//!
19//#####
20// $TI Rel
21// $Release
22// $Copyri
23///
24//#####
25
26#include '
27
28// Prototy
29// _inter
30void delay
31void spi_x
32void spi_t
33void spi_i
34void error
35
```

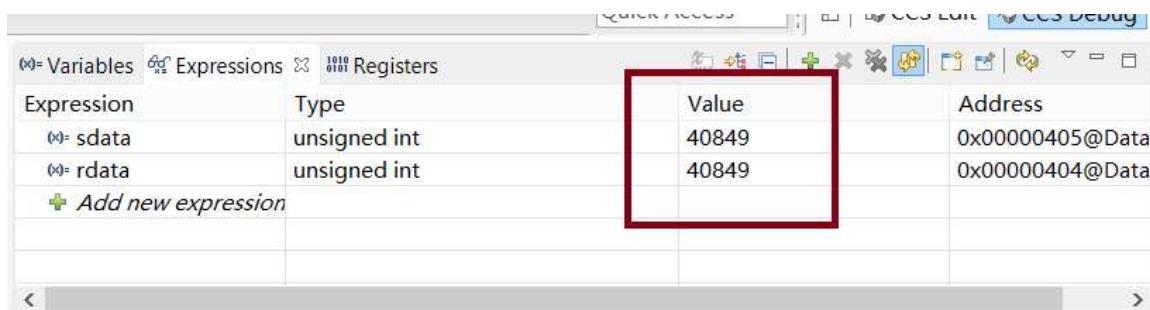
The screenshot shows a code editor with several lines of C code. A context menu is open over the variable declarations 'sdata' and 'rdata'. The menu has the following items:

- Breakpoint (Code Composer Studio)
- Open Declaration F3
- Cut Ctrl+X
- Copy Ctrl+C
- Paste Ctrl+V
- Use Spaces for Tab
- Declarations
- References
- Search Text
- Run to Line Ctrl+R
- Move To Line
- Add Watch Expression... (highlighted)
- Preferences...

图 2 变量的添加

◆ 接下来开始全速运行；

实验现象如下图所示：发送和接收的内容一致；



Expression	Type	Value	Address
sdata	unsigned int	40849	0x00000405@Data
rdata	unsigned int	40849	0x00000404@Data
<i>Add new expression</i>			

图 5 实验现象

程序解析

```

for(;;)
{
    // 发送数据
    spi_xmit(sdata);
    // 等待接收数据
    while(SpiaRegs.SPIFFRX.bit.RXFFST !=1) { }
    // 核对发送与接收是否一致
    rdata = SpiaRegs.SPIRXBUF;
    if(rdata != sdata) error();
    sdata++;
}

```

实验 23: FIFO 模式下 SPI 程序实验 (Example_2833xSpi_FFDLB)

一 实验目的:

- ✧ 熟悉 TMS320F28335 的 SPI 工作原理;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开;

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处；给开发板上电。进行调试。
- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

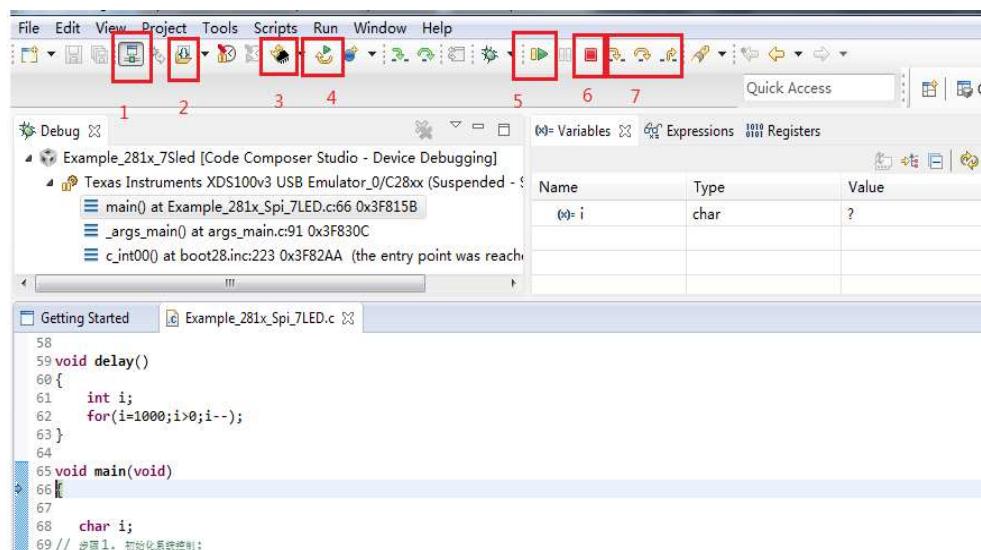


图 1 程序调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 实验现象：

- ◆ 在全速运行前需要将下面两个局部变量添加到观察窗口中。

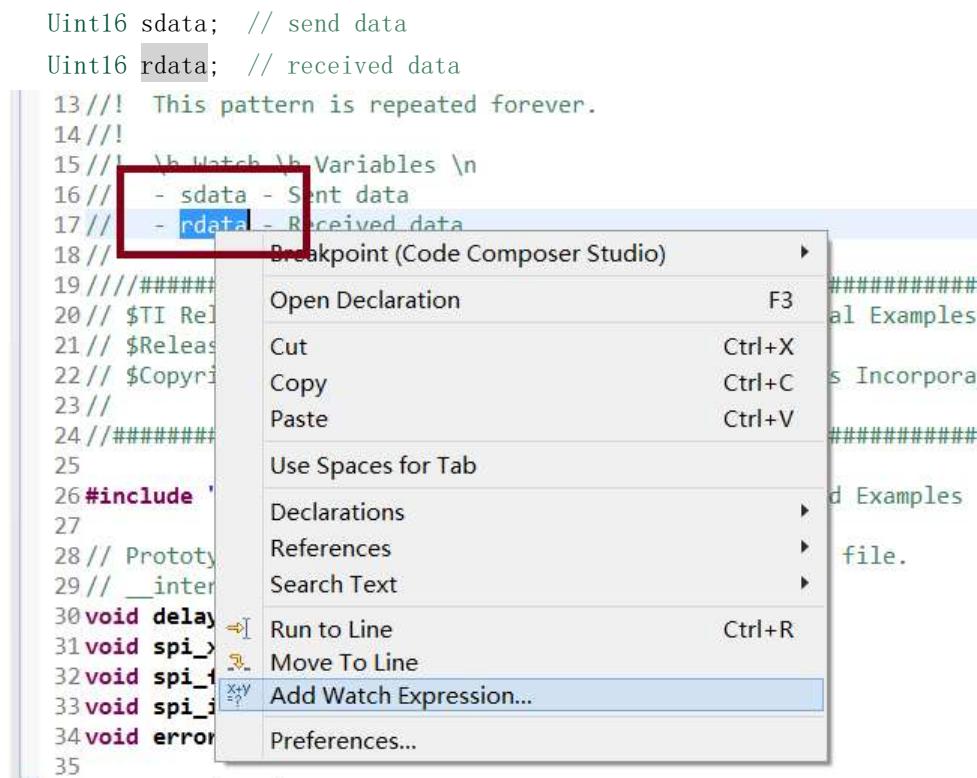


图 2 变量的添加



图 3 局部变量添加到观察窗口中的现象

- ◆ 从图中可以看到两个变量为“未知状态”，这是为什么呢？原因很简单，因为程序刚开始还没进入主函数之前局部变量还没有生效，必须等到程序执行到局部变量声明的函数中时才会有效。
- ◆ 下面我们设置一个断点，如下图所示

```

83 // Interrupts are not used in this example.
84 sdata = 0x0000;
85 for(;;)
86 {
87     // Transmit data
88     spi_xmit(sdata);
89     // Wait until data is received
90     while(SpiRegs.SPIFFRX.bit.RXFFST !=1) { }
91     // Check against sent data
92     rdata = SpiRegs.SPIRXBUF;
93     if(rdata != sdata) error();
94     sdata++;
95 }
96
97

```

在93行处双击添加一个断点

图 4 断点的设置

接下来开始全速运行，程序会停到设置的断点处，然后我们再次单击全速运行按钮；循环以上操作；

实验现象如下图所示：发送和接收的内容一致；

Expression	Type	Value	Address
sdata	unsigned int	40849	0x00000405@Data
rdata	unsigned int	40849	0x00000404@Data
<i>Add new expression</i>			

图 5 实验现象

程序解析

```

for(;;)
{
    // 发送数据，这是向 FIFO 中发如 1 个数据
    spi_xmit(sdata);
    // 等待 FIFO 接收到数据，如果没有接收到就一直等待
    while(SpiRegs.SPIFFRX.bit.RXFFST !=1) { }
    // 接收到数据后将数据读出
    rdata = SpiRegs.SPIRXBUF;
    // 检测接收到的数据是否正确
    if(rdata != sdata) error();
}

```

```
sdata++;
}
```

实验 24: SPI 工作于 FIFO 中断方式下实验 (Example_2833xSpi_FFDLB_int)

一 实验目的:

- ✧ 熟悉 TMS320F28335 的 SPI 工作原理;
- ✧ 熟悉 TMS320F28335 的 SPI 的 FIFO 发送中断实现过程;
- ✧ 熟悉 TMS320F28335 的 SPI 的 FIFO 接收中断实现过程;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开;
- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处; 给开发板上电。进行调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

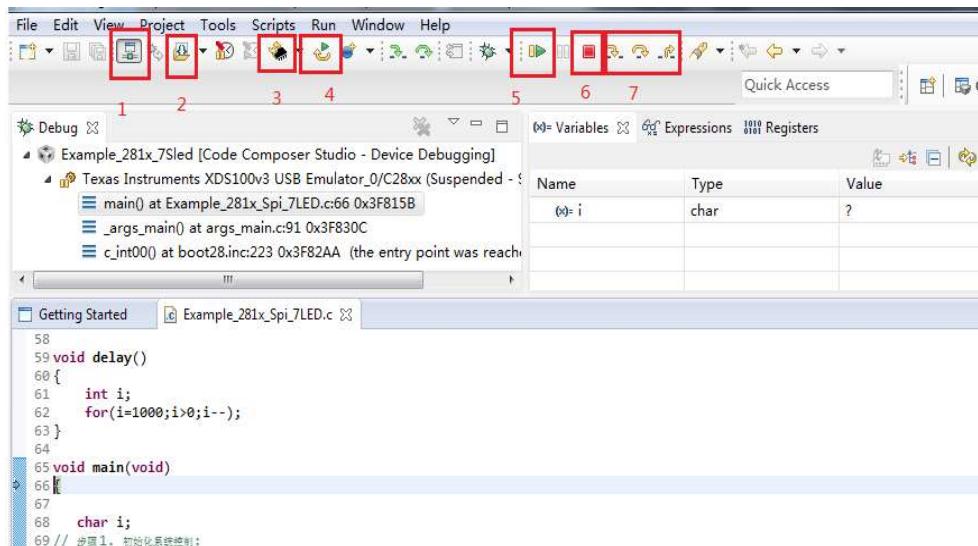


图 1 程序调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 CPU 软 Reset；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 实验现象：

◆ 在全速运行前需要将下面两个局部变量添加到观察窗口中。

```
Uint16 sdata[8];      // 发送数据的数组  
Uint16 rdata[8];      // 接收数据的数组
```

◆ 下面我们设置一个断点，如下图所示

```

166
167 __interrupt void spiRxFifoIsr(void)
168 {
169     Uint16 i;
170     for(i=0;i<8;i++)
171     {
172         rdata[i]=SpiaRegs.SPIRXBUF;      // Read data
173     }
174     for(i=0;i<8;i++)                  // Check received data
175     {
176         if(rdata[i] != rdata_point+i) error();
177     }
178     rdata_point++;
179     SpiaRegs.SPIFFRX.bit.RXFFOVFCLR=1; // Clear Overflow flag
180     SpiaRegs.SPIFFRX.bit.RXFFINTCLR=1; // Clear Interrupt flag
181     PieCtrlRegs.PIEACK.all|=0x20;      // Issue PIE ack
182 }
183
184

```

在此处设置一个断点

图 2 断点的设置

- ◆ 接下来开始全速运行，程序会停到设置的断点处，然后我们再次单击全速运行按钮；循环以上操作；

实验现象如下图所示：

Expression	Type	设置连续刷新按钮	Value	Address
sdata	unsigned int[8]		0x0000C001@Data	0x0000C001@D
[0]	unsigned int		3	0x0000C001@D
[1]	unsigned int		4	0x0000C002@D
[2]	unsigned int	发送数据	5	0x0000C003@D
[3]	unsigned int		6	0x0000C004@D
[4]	unsigned int		7	0x0000C005@D
[5]	unsigned int		8	0x0000C006@D
[6]	unsigned int		9	0x0000C007@D
[7]	unsigned int		10	0x0000C008@D
rdata	unsigned int[8]		0x0000C009@Data	0x0000C009@D
[0]	unsigned int		0	0x0000C009@D
[1]	unsigned int		1	0x0000C00A@D
[2]	unsigned int	接收数据	2	0x0000C00B@D
[3]	unsigned int		3	0x0000C00C@D
[4]	unsigned int		4	0x0000C00D@D
[5]	unsigned int		5	0x0000C00E@D
[6]	unsigned int		6	0x0000C00F@D
[7]	unsigned int		7	0x0000C010@D

图 3 实验现象

SPI 模块的 FIFO 功能解析

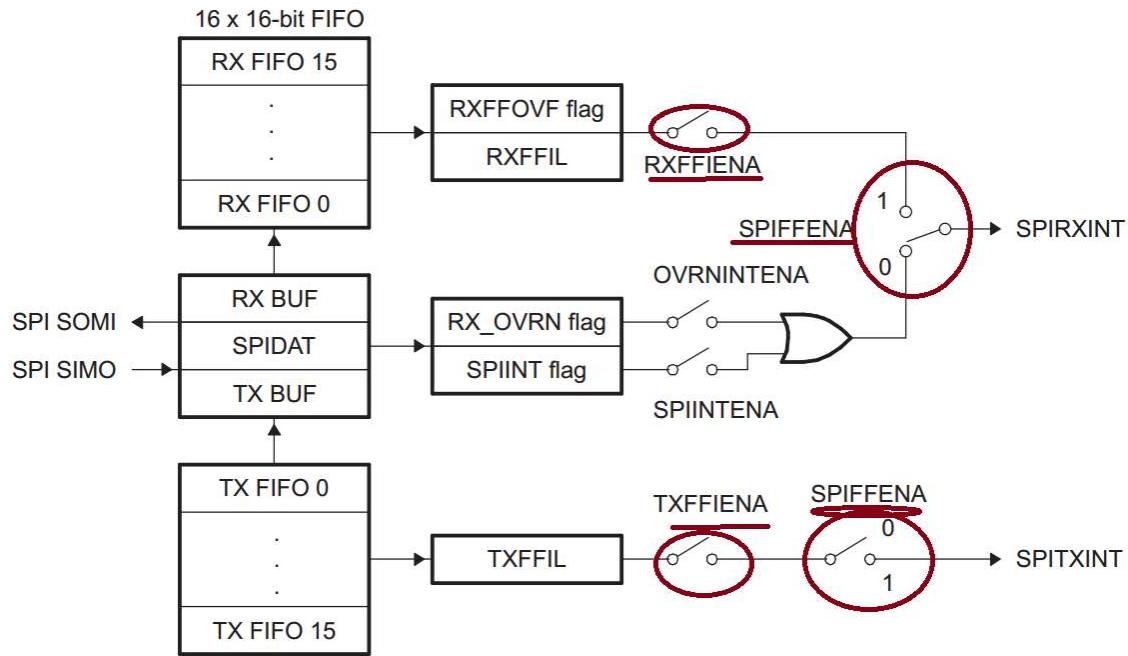
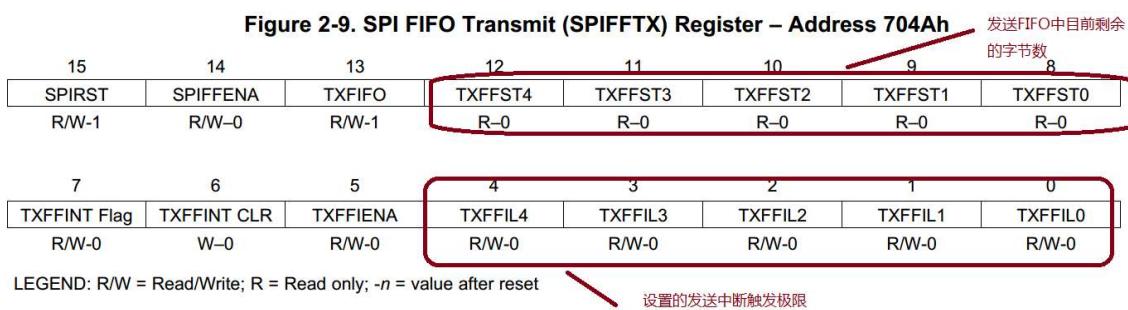


图 4 SPI 的 FIFO 功能模块

SPI 的 FIFO 中断功能一般分为两个常用的中断：发送 FIFO 中断和接收 FIFO 中断；那么下面分别对两个中断方式的实现进行一下介绍：

发送中断的介绍：



要想触发中断，首先要使能 TXFFIENA 和 SPIFFENA。并设置 TXFFIL（中断触发极限）。如果 TXFFST (FIFO 目前还有多少没有发送的字节数) 少于或等于 TXFFIL 则触发发送 FIFO 中断，在中断函数里我们可以向 FIFO 里放数据；FIFO 的深度为 8 的发送 FIFO 中断函数的实现如下：

```
_interrupt void spiTx_fifo_isr(void)
```

```

{
    Uint16 i;
    for(i=0;i<8;i++)
    {
        SpiaRegs.SPITXBUF=sdata[i];      // 向发送 FIFO 中存入 8 个字节
    }

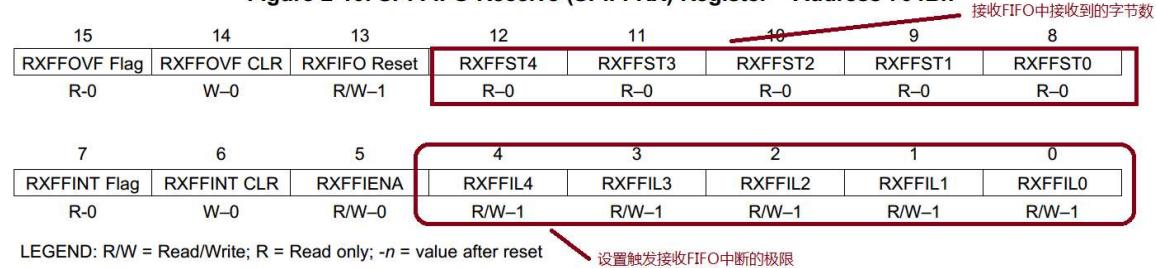
    for(i=0;i<8;i++)                  // 更新下次要发送的数据
    {
        sdata[i] = sdata[i]+ 1;
    }

    SpiaRegs.SPIFFTX.bit.TXFFINTCLR=1; // 清除中断标志
    PieCtrlRegs.PIEACK.all|=0x20;     // Issue PIE ACK
}

```

接收 FIFO 中断的介绍：

Figure 2-10. SPI FIFO Receive (SPIFFRX) Register – Address 704Bh



要想触发中断，首先要使能 RXFFIENA 和 SPIFFENA。并设置 RXFFIL（中断触发极限）。如果 RXFFST（FIFO 目前接收到的字节数）大于或等于 RXFFIL 则触发接收 FIFO 中断，在中断函数里我们可以读取 FIFO 里的数据；FIFO 的深度为 8 的接收 FIFO 中断函数的实现如下：

```

__interrupt void spiRx_fifoIsr(void)
{
    Uint16 i;
    for(i=0;i<8;i++)
    {
        rdata[i]=SpiaRegs.SPIRXBUF;      //连续读取 8 个字节
    }

    for(i=0;i<8;i++)                  // 检查接收的数据是否与发送的数据一致
    {
        if(rdata[i] != rdata_point+i) error();
    }
    rdata_point++;
}

```

```
SpiaRegs.SPIFFRX.bit.RXFFOVFCLR=1; //清除溢出标志
SpiaRegs.SPIFFRX.bit.RXFFINTCLR=1; // 清除中断标志
PieCtrlRegs.PIEACK.all|=0x20; // Issue PIE ack
}
// 将 FIFO 的中断函数映射到中断向量表中
EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.SPIRXINTA = &spiRx_fifoIsr;
PieVectTable.SPITXINTA = &spiTx_fifoIsr;
EDIS; // This is needed to disable write to EALLOW protected registers
```

实验 25：IIC_EEPROM 实验 (Example_2833x_I2C_eeprom)

一、实验目的：

- ✧ 了解 IIC 总线时序；方法是向 EEPROM 写入数据，然后再从 EEPROM 中将写入的数据读出。
- ✧ 了解如何使用 IIC 外设模块控制 AT24C512BW；

二、实验设备

- ✧ 计算机（已安装 CCSv6.0 开发环境）
- ✧ SXD28335 或 SXD28335B 开发板
- ✧ 5V 2A (或 3A)DC 电源
- ✧ USB 转串口线一条

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

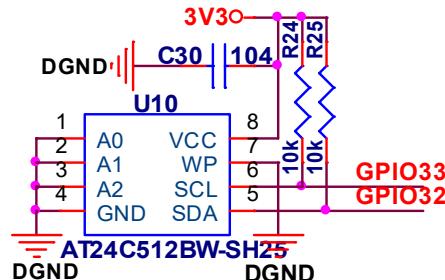
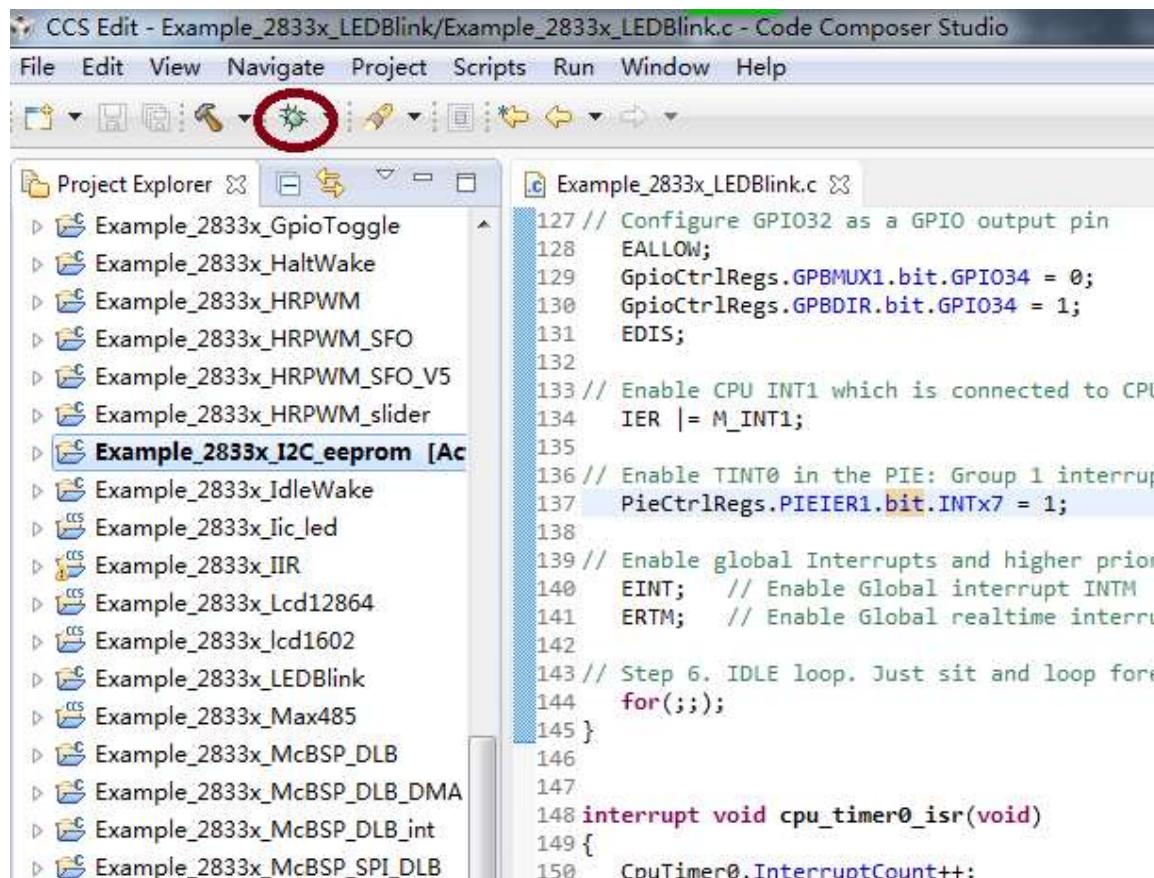


图 1 硬件连接图

这里我们需要配置 GPIO32 和 GPIO33 为 IIC 外设模式

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：
然后单击图中红色的方框处的调试按钮，进行调试。



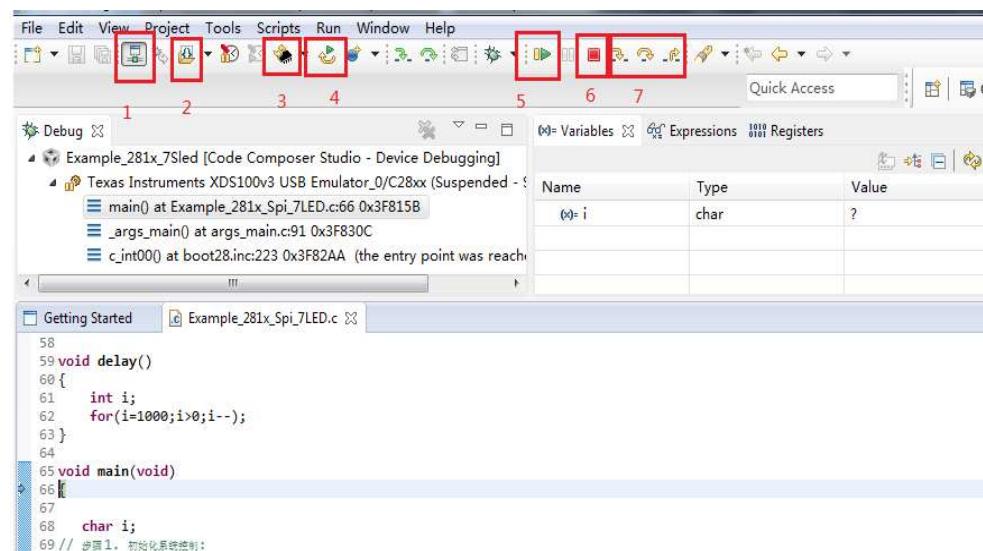
```

CCS Edit - Example_2833x_LED Blink/Example_2833x_LED Blink.c - Code Composer Studio
File Edit View Navigate Project Scripts Run Window Help
Project Explorer Example_2833x_LED Blink.c
Example_2833x_GpioToggle
Example_2833x_HaltWake
Example_2833x_HRPWM
Example_2833x_HRPWM_SFO
Example_2833x_HRPWM_V5
Example_2833x_HRPWM_slider
Example_2833x_I2C_eeprom [Ac]
Example_2833x_IdleWake
Example_2833x_Iic_led
Example_2833x_IIR
Example_2833x_Lcd12864
Example_2833x_lcd1602
Example_2833x_LED Blink
Example_2833x_Max485
Example_2833x_McBSP_DL8
Example_2833x_McBSP_DL8_DMA
Example_2833x_McBSP_DL8_int
Example_2833x_McBSP_SPI_DL8

127 // Configure GPIO32 as a GPIO output pin
128 EALLOW;
129 GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;
130 GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;
131 EDIS;
132
133 // Enable CPU INT1 which is connected to CPU
134 IER |= M_INT1;
135
136 // Enable TINT0 in the PIE: Group 1 interrupt
137 PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
138
139 // Enable global Interrupts and higher prior
140 EINT; // Enable Global interrupt INTM
141 ERTM; // Enable Global realtime interrupt
142
143 // Step 6. IDLE loop. Just sit and loop forever
144 for(;;);
145
146
147
148 interrupt void cpu_timer0_isr(void)
149 {
150     CpuTimer0.InterruptCount++;

```

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。



图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的；

图中 3 是 CPU 软 Reset；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

四 试验现象:

✧ 本试验的过程是首先我们将 I2cMsgOut1 结构体里的

```
0x12,          // Msg Byte 1
0x34,          // Msg Byte 2
0x56,          // Msg Byte 3
0x78,          // Msg Byte 4
0x9A,          // Msg Byte 5
0xBC,          // Msg Byte 6
0xDE,          // Msg Byte 7
0xF0,          // Msg Byte 8
0x11,          // Msg Byte 9
0x10,          // Msg Byte 10
0x11,          // Msg Byte 11
0x12,          // Msg Byte 12
0x13,          // Msg Byte 13
0x12
```

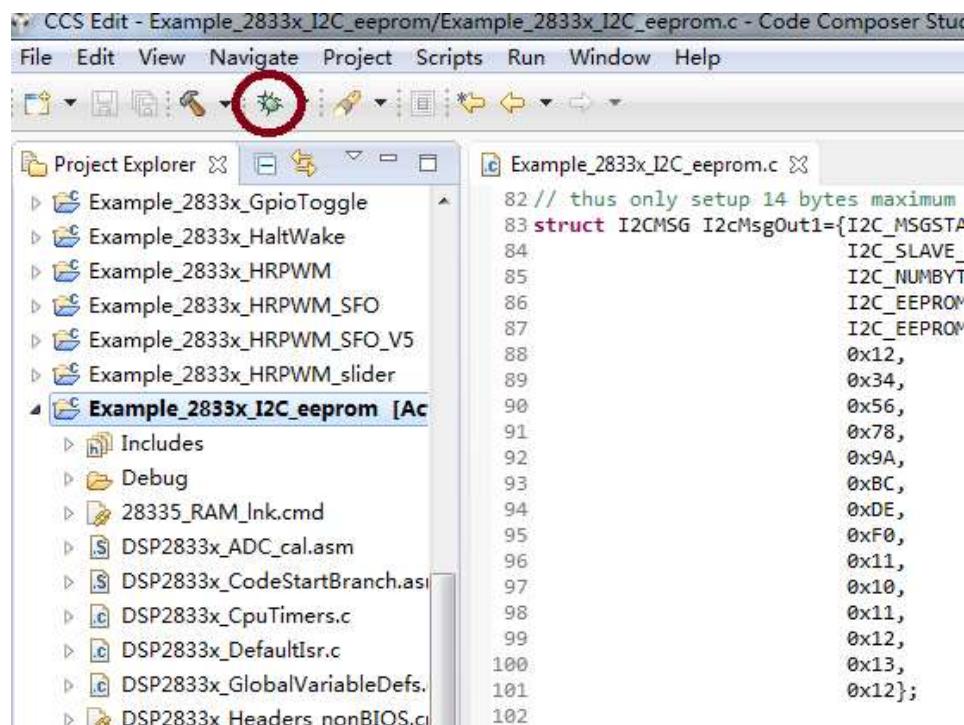
发送到 EEPROM 里. 然后我们从 EEPROM 中读取这些变量中的前 10 个放到接收结构体 I2cMsgIn1 里.

这样的话我们只需要观察 I2cMsgIn1 中的数据是否和 I2cMsgOut1 中的一样即可.

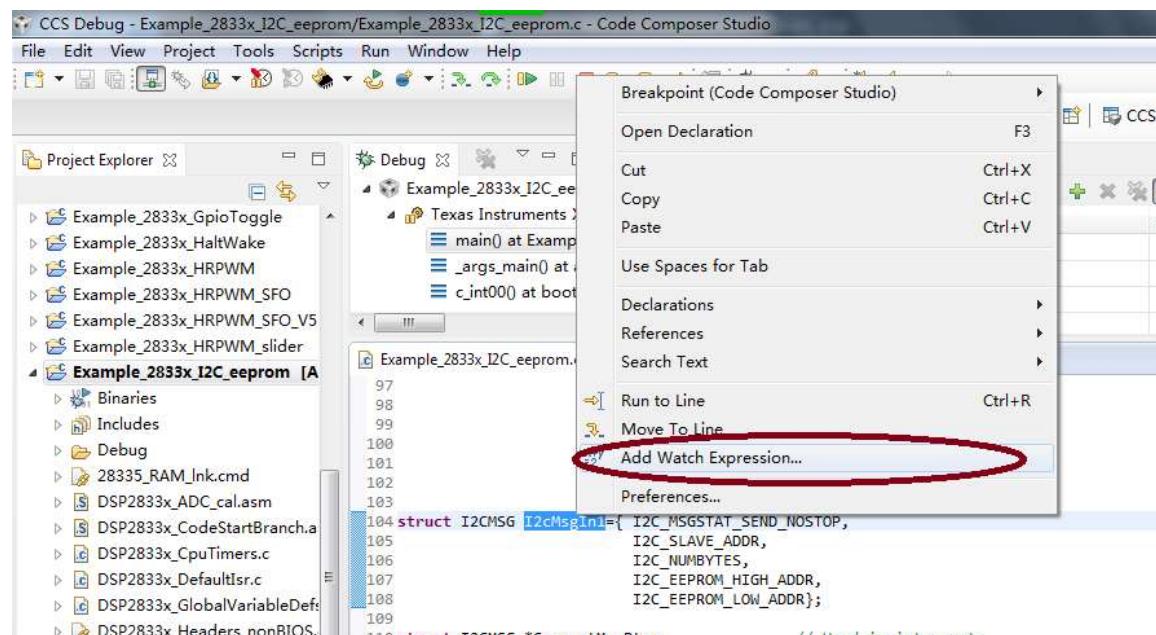
I2cMsgOut1 的定义如下图所示:

```
struct I2CMSG I2cMsgOut1={I2C_MSGSTAT_SEND_WITHSTOP,
    I2C_SLAVE_ADDR,
    I2C_NUMBYTES,
    I2C_EEPROM_HIGH_ADDR,
    I2C_EEPROM_LOW_ADDR,
    0x12,          // Msg Byte 1
    0x34,          // Msg Byte 2
    0x56,          // Msg Byte 3
    0x78,          // Msg Byte 4
    0x9A,          // Msg Byte 5
    0xBC,          // Msg Byte 6
    0xDE,          // Msg Byte 7
    0xF0,          // Msg Byte 8
    0x11,          // Msg Byte 9
    0x10,          // Msg Byte 10
    0x11,          // Msg Byte 11
    0x12,          // Msg Byte 12
    0x13,          // Msg Byte 13
    0x12};         // Msg Byte 14
```

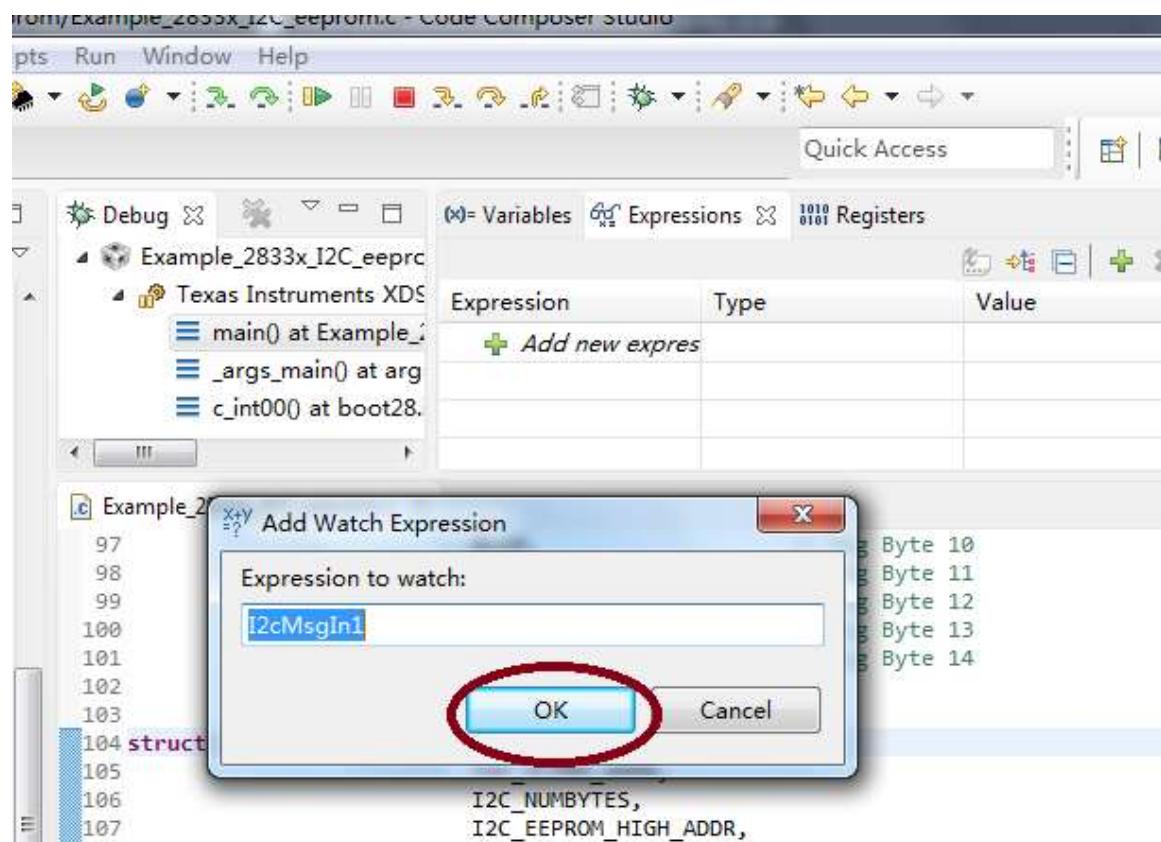
Step1: 用鼠标左键单击工程. 然后点击红色圈中的图标. 进行在线调试。



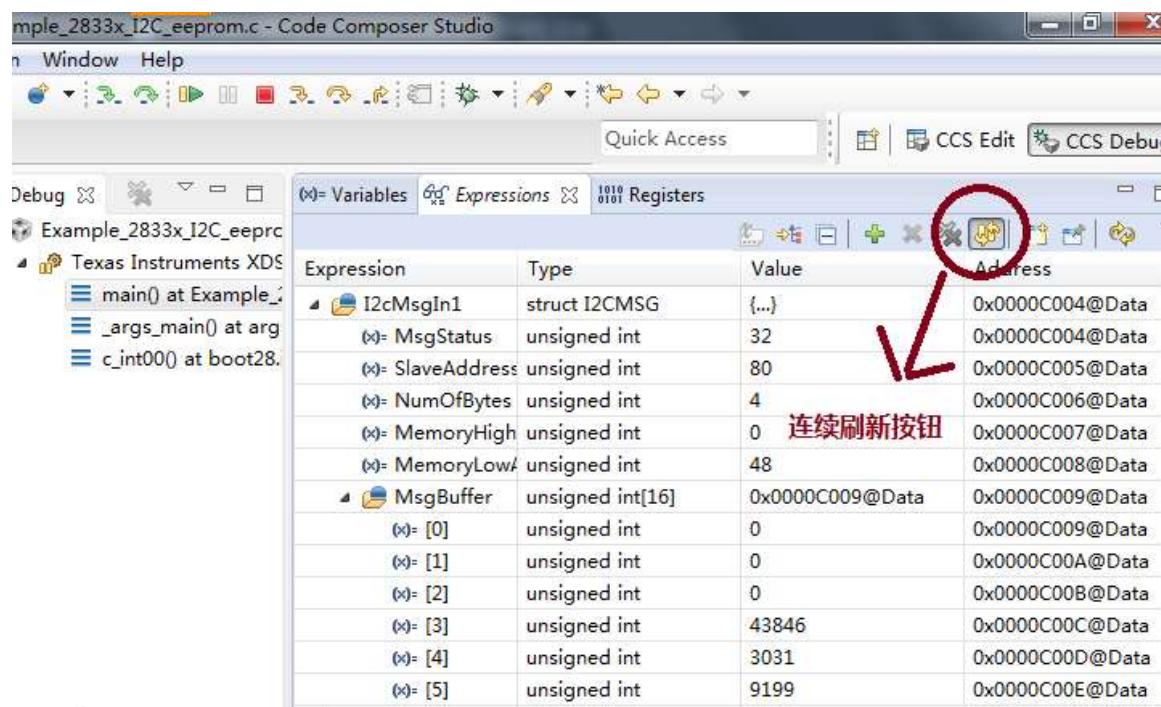
Step2: 选中 I2cMsgIn1 右键，选择 add watch expression，这样我们就可以通过 SXD 的仿真器来观察这个接收数组的变化。



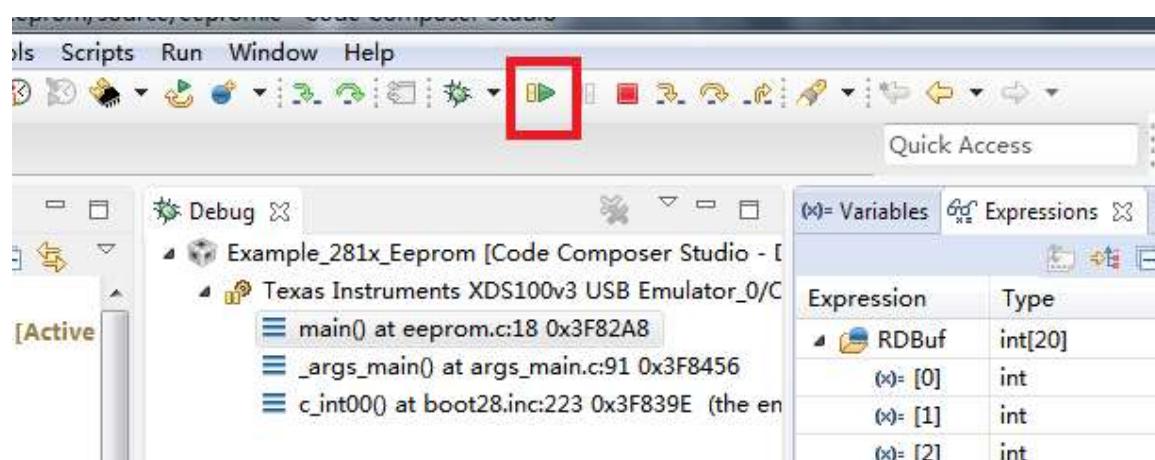
Step3: 选择 O K



Step4: 接收数组在下图的方框中显示。



Step5: 全速运行

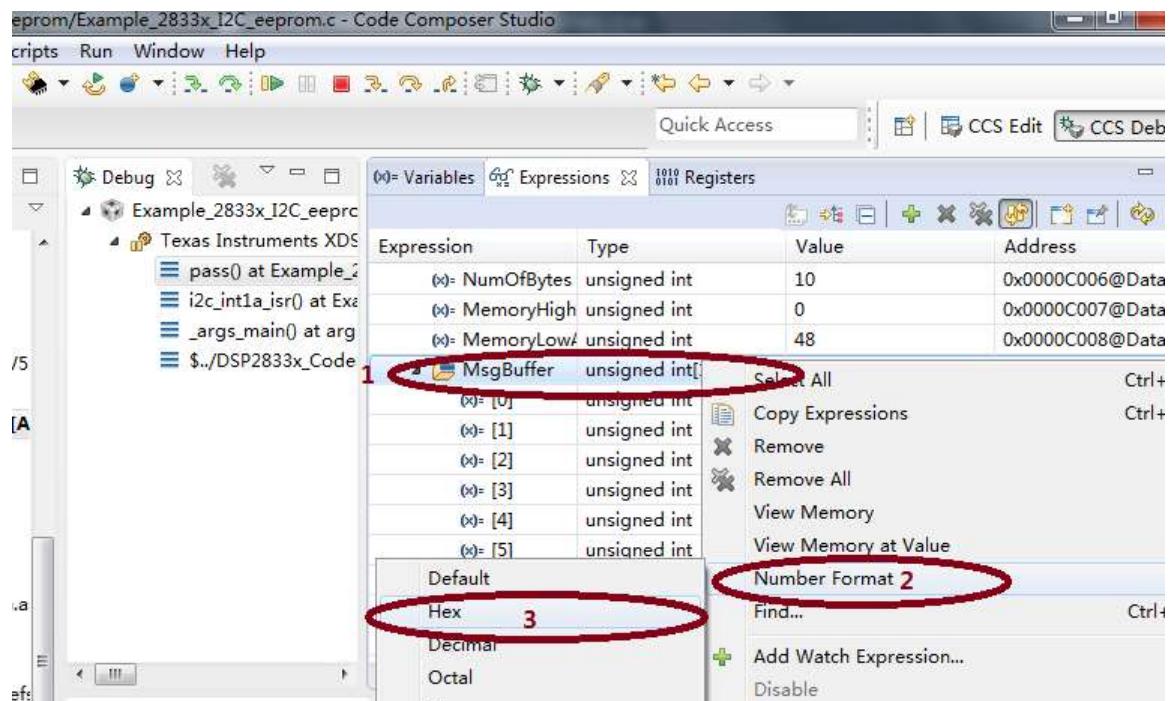


现象：

The screenshot shows the 'Variables' tab in the debugger. It displays a table with columns 'Expression', 'Type', 'Value', and 'Address'. The table shows a sequence of 16 unsigned integers starting at address 0x0000C009. The values are: 18, 52, 86, 120, 154, 188, 222, 240, and 17. The 'Registers' tab is also visible at the bottom.

Expression	Type	Value	Address
(x)= [0]	unsigned int	18	0x0000C009@Data
(x)= [1]	unsigned int	52	0x0000C00A@Data
(x)= [2]	unsigned int	86	0x0000C00B@Data
(x)= [3]	unsigned int	120	0x0000C00C@Data
(x)= [4]	unsigned int	154	0x0000C00D@Data
(x)= [5]	unsigned int	188	0x0000C00E@Data
(x)= [6]	unsigned int	222	0x0000C00F@Data
(x)= [7]	unsigned int	240	0x0000C010@Data
(x)= [8]	unsigned int	17	0x0000C011@Data

结果如上图所示，这是以 10 进制显示的，我们可以把它转化为 16 进制格式，这样就可以直接跟发送结构体进行比较了。方法如下



程序解析

```
#define I2C_SLAVE_ADDR          0x50 //EEPROM 的从地址
#define I2C_NUMBYTES            10   //要读出的字节数
InitI2CGpio(); //将相应的 IO 口配置为 IIC 口功能;
EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.I2CINT1A = &i2c_int1a_isr;//将 IIC 中断函数影射到 PIE 向量表中
EDIS; // This is needed to disable write to EALLOW protected registers
I2CA_Init(); //配置 IIC 工作方式
```

实验 26: IIC 流水灯实验 (Example_2833x_iicliushui)

1、实验目的:

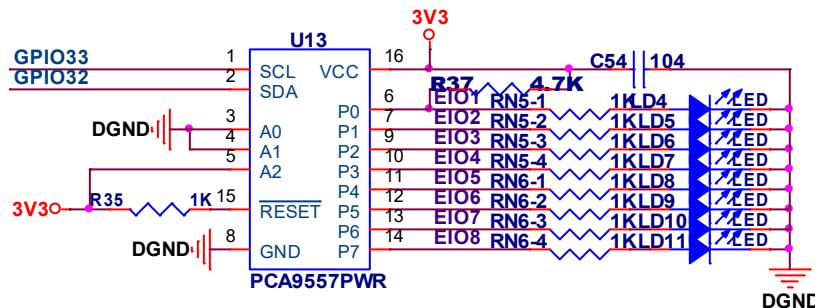
- (1) 了解 I I C 总线时序;
- (2) 了解如何使用 I I C 总线控制 P C A 9 5 5 7 芯片来扩展 I O 口;

2、实验设备:

- (1) PC 机一台;
- (2) XDS100v2 或 XDS100V3(隔离)仿真器一套;
- (3) SXD28335 开发板一套;

3、实验步骤:

- (1) 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

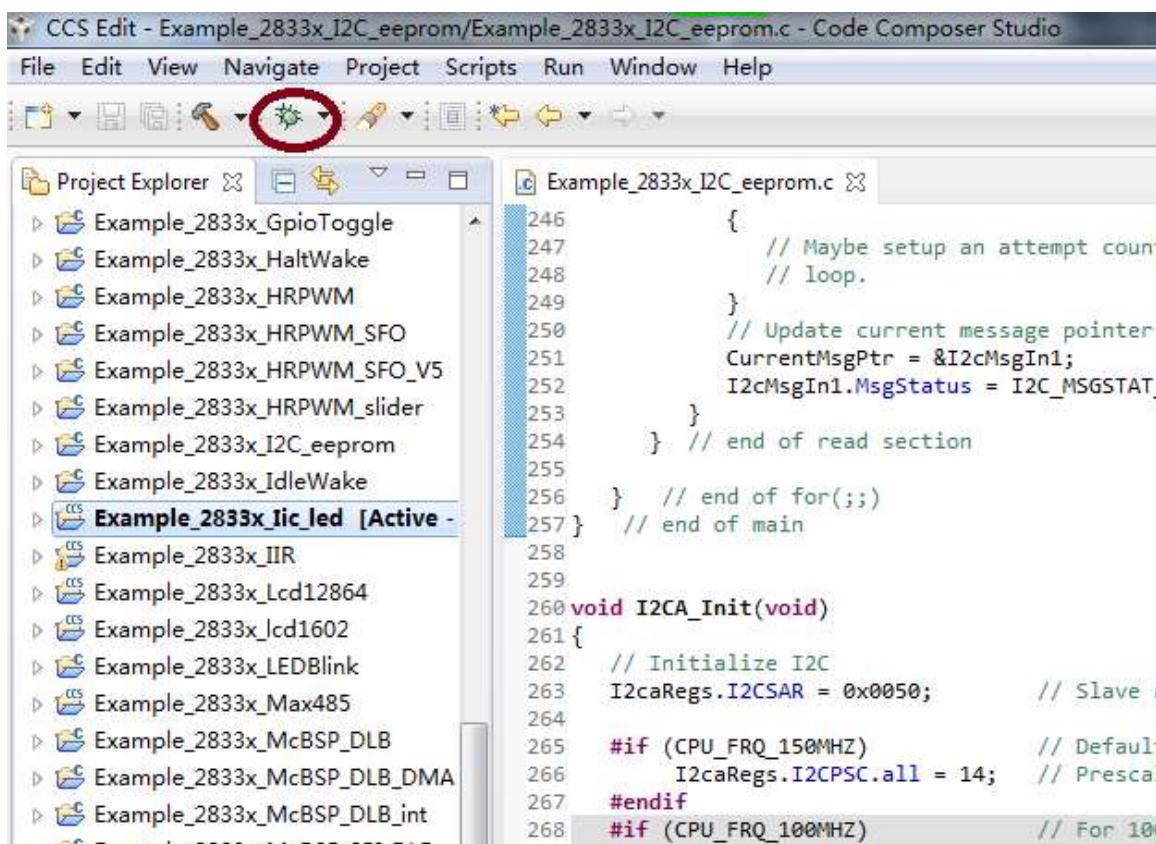


从图中可以看到要控制 IIC 流水灯首先要了解 PCA9557 这个芯片。

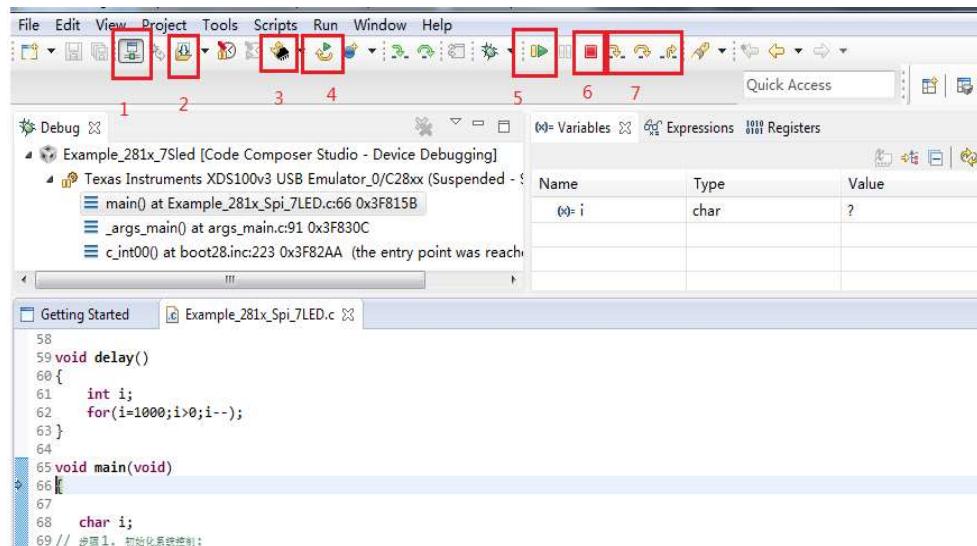
(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。（注：此时认为兄弟你已经会设置配置文件，并将其设置为默认配置）。



(7) 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。



图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

图中 3 是 C P U 软 R e s e t ；

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；

这里。我们直接用鼠标左键单击图标 5 即可，这时程序会全速运行。

(8) 试验现象：8 位 Led 灯会从左到右依次点亮。

程序解析

```

while(1)
{
    Led_Test();
}

Step1: 程序进入主函数进行一些常规的配置，如配置系统时钟、禁止看门狗和初始化中断向量等。
Step2: 进入 while(1) 循环，在这个循环里执行 Led_Test() ;
void Led_Test()
{
    Uint16 led, i;
    InitI2CGpio(); //配置 GPIO32 和 GPIO33 为 I I C 总线.
    I2CA_Init(); //配置 I I C 工作方式
    // I2CA_WriteConfig(&I2cMsgConfigOut1);
    // 配置 P C A 9 5 5 7 芯片
    I2cWriteRegister(0x1C, I2C_CONFIG_REG, I2C_OUTPUT_ALL);
    DSP28x_usDelay(100000);
    for(i=0; i<8; i++)
    {
        //地址 =LED_I2CADDR; 1 为配置 px 为输出; 输出的数据是:
        1<<((led%7)+1) 1<<((led%7)+1)
    }
}

```

```
//向PCA9557写数据
I2cWriteRegister(0x1c, 1, 1<<((led%8))); //light one led
DSP28x_usDelay(2000000);
DSP28x_usDelay(2000000);
led++;
#if 0
    if(led==0x07)
    {
        led=0;
        flag=0;
    }
    else
        led++;
#endif
}
```

具体的配置信息用户要参考 PCA9557 数据手册。

实验 27：CH452 按键实验（Example_F28335_Ch452）

一 实验目的：

◆ 了解如何 CH452 的使用；

二 实验设备：

◆ PC 机一台；

◆ XDS100v2 或 XDS100V3(隔离)仿真器一套；

◆ SXD28335 开发板一套；

三 实验步骤：

首先将 CCS6.0 开发环境打开；

◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

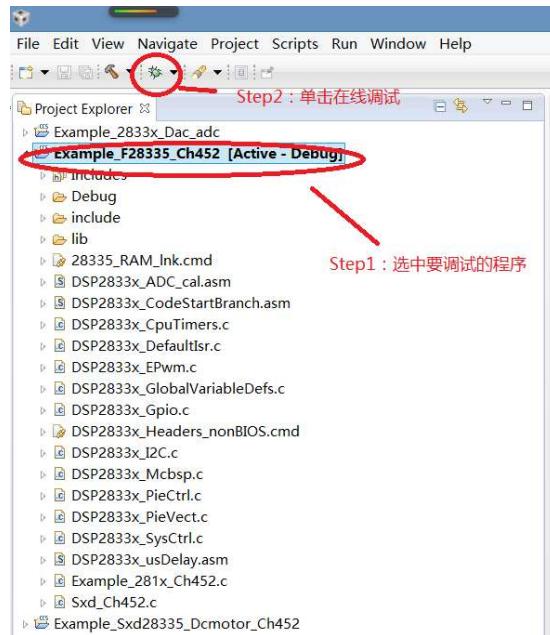


图 3 调试过程

然后单击图中红色的方框处的调试按钮，进行调试。

- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

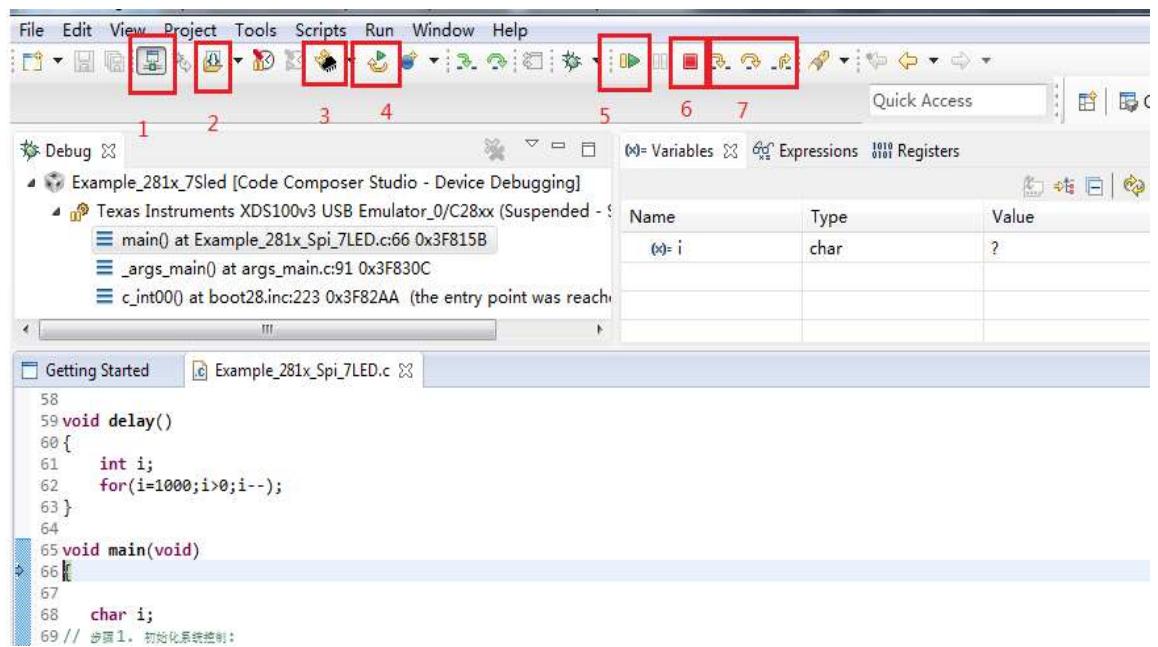


图 4 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的

- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；



试验现象：

通过按下开发板上的按键，模块的数码管会显示你所按的按键。具体资料请看一下我们提供的 CH452 数据手册；

实验 28：音频实验之警报实验 (Example_2833x_mcbsp_be11)

一 实验目的：

- ✧ 了解 DSP28335 的 MCBSP 外设；
- ✧ 了解 TLV320AIC23IPW 这款新品；

二 实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板一套，耳机一个；

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开；Mcbsp 是多通道串行接口。可以当作 SPI 使用。也可以当作非 SPI 功能。MCBSP 其实就是一个串行总线协议。只不过有两个时钟：接收时钟和发送时钟。这导致接收和发送独立使用自己的时钟；还有就是多了两个状态标志信号：一个是发送同步信号一个是接收同步信号。

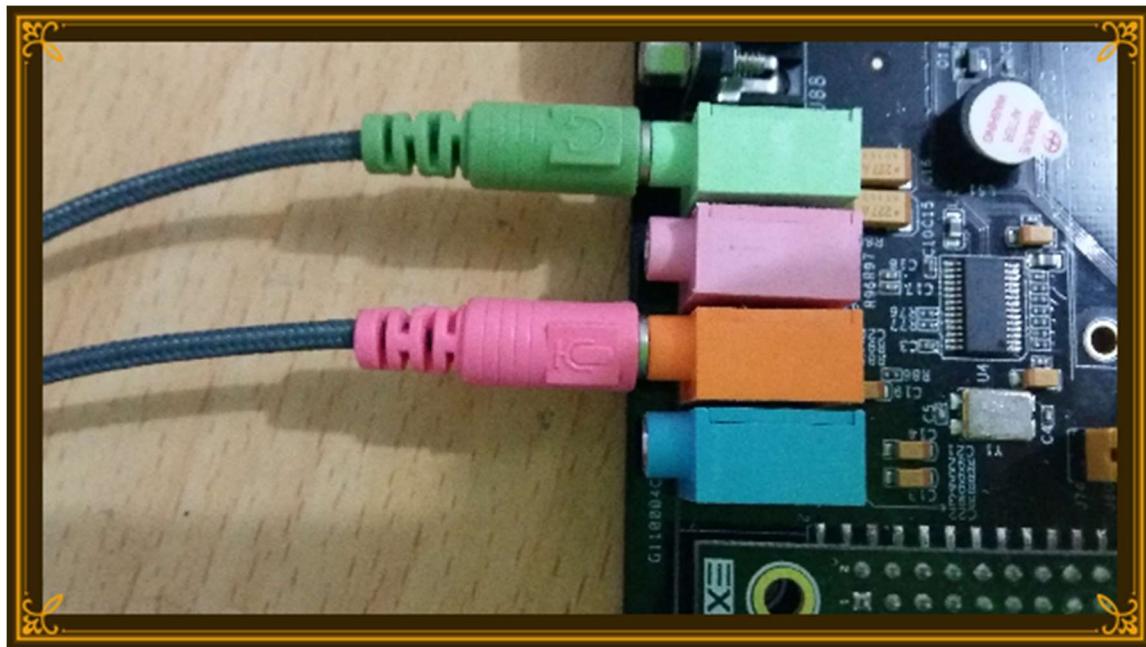
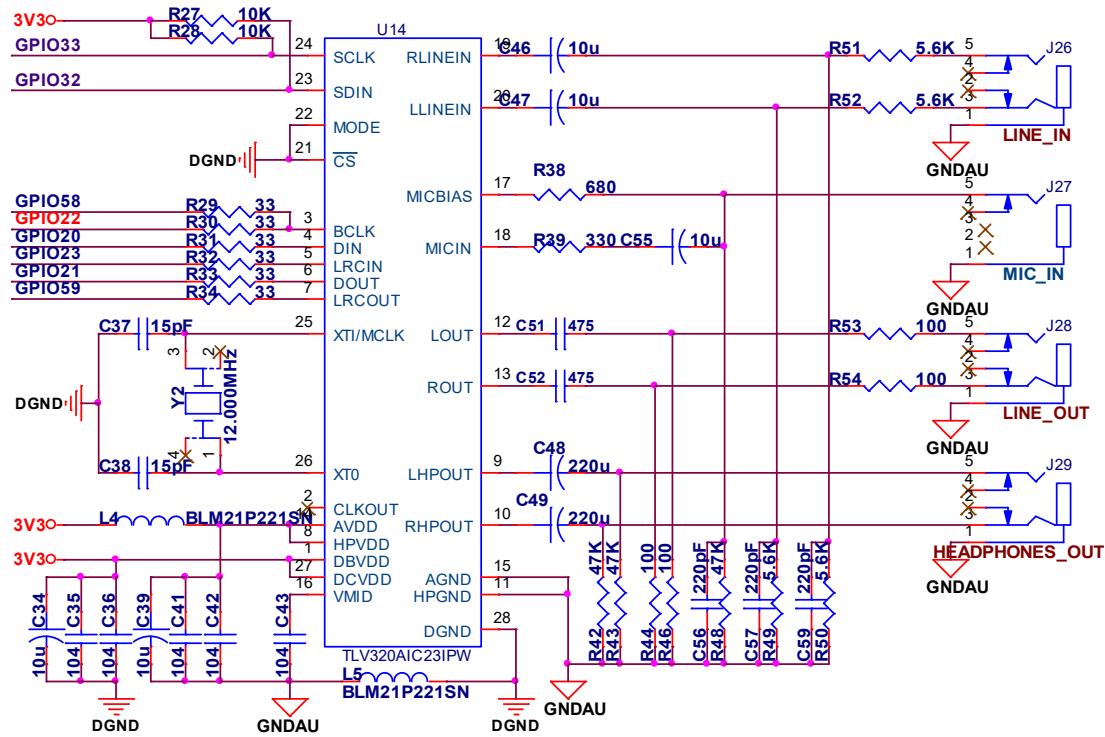


图 1 音频硬件原理图

从图中可知 GPIO32、GPIO33 为 IIC 功能，而 GPIO58 为 MCLKRA（接收时钟：数据从接收管脚向接收移位寄存器传输数据时要以此时钟信号为时间基准）；GPIO59 为 MFSRA（接收帧同步信号：用于检测接收数据的）。GPIO20 是串行发送数据接口 MDXA；GPIO21 是串行数据接收接口 MDRA；GPIO22 则是发送时钟信号 MCLKXA；GPIO23 为帧接收同步信号 MFSXA；

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

然后单击图中红色的方框处的调试按钮，进行调试。

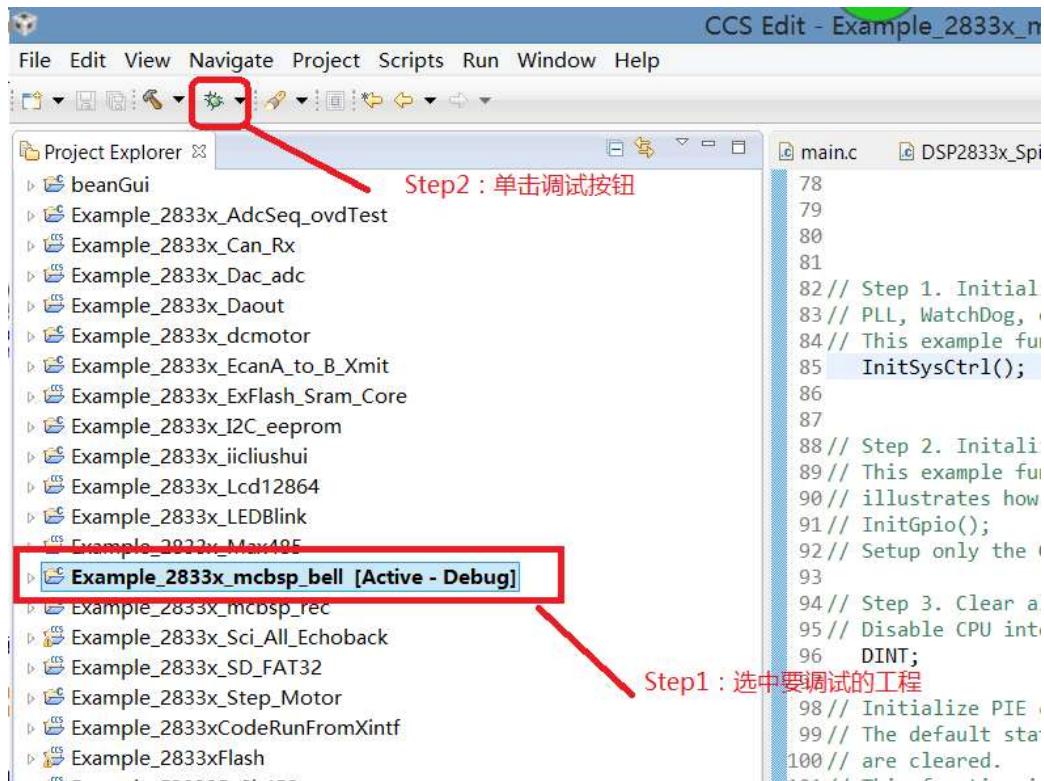


图 3 调试方法

◇ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

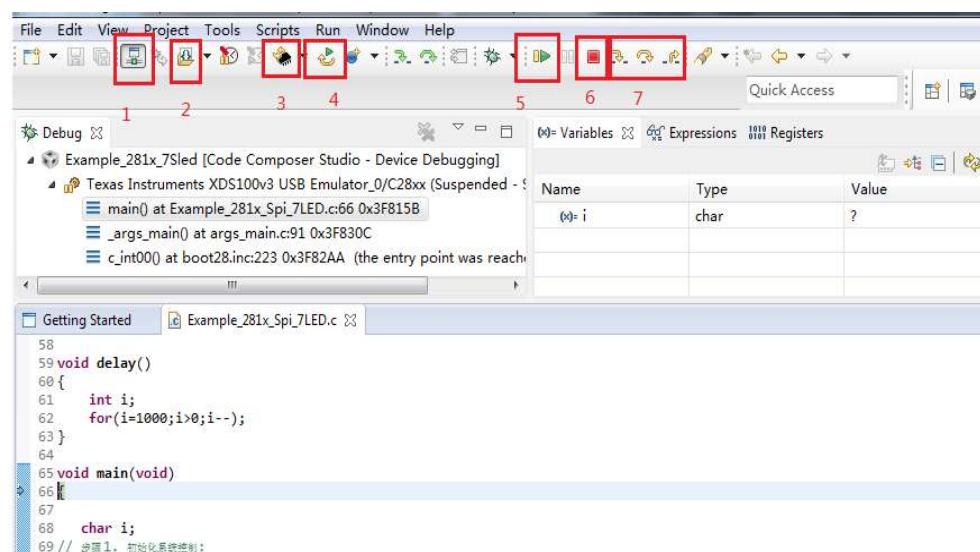


图 4 调试界面

- ① 图中 1 图标是用来进行与开发板进行连接的按钮；
- ② 图中 2 是用来下载 Debug 文件下的.out 文件的；
- ③ 图中 3 是 C P U 软 R e s e t；
- ④ 图中 4 是调试时恢复到程序的开始处；
- ⑤ 图中 5 是全速运行；
- ⑥ 图中 6 是停止调试；
- ⑦ 图中 7 是用于单步调试的；

四 实验现象

耳机会发出警报声；

程序解析

```
void mcbsp_InitMcbspaGpio(void)
{
    EALLOW;
    /* 配置为 MCBSPI 的功能口
        GpioCtrlRegs.GPAMUX2.bit.GPIO20 = 2; // GPIO20 is MDXA pin
        GpioCtrlRegs.GPAMUX2.bit.GPIO21 = 2; // GPIO21 is MDRA pin
        GpioCtrlRegs.GPAMUX2.bit.GPIO22 = 2; // GPIO22 is MCLKXA pin
        //GpioCtrlRegs.GPAMUX1.bit.GPIO7 = 2; // GPIO7 is MCLKRA pin (Comment as
        //needed)
        GpioCtrlRegs.GPBMUX2.bit.GPIO58 = 1; // GPIO58 is MCLKRA pin (Comment as
        //needed)
        GpioCtrlRegs.GPAMUX2.bit.GPIO23 = 2; // GPIO23 is MFSXA pin
        //GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 2; // GPIO5 is MFSRA pin (Comment as
        //needed)
        GpioCtrlRegs.GPBMUX2.bit.GPIO59 = 1; // GPIO59 is MFSRA pin (Comment as
        //needed)
        /* Enable internal pull-up for the selected pins */
        // Pull-ups can be enabled or disabled by the user.
        // This will enable the pullups for the specified pins.
        // Comment out other unwanted lines.
        GpioCtrlRegs.GPAPUD.bit.GPIO20 = 0; // Enable pull-up on GPIO20 (MDXA)
        GpioCtrlRegs.GPAPUD.bit.GPIO21 = 0; // Enable pull-up on GPIO21 (MDRA)
        GpioCtrlRegs.GPAPUD.bit.GPIO22 = 0; // Enable pull-up on GPIO22 (MCLKXA)
        //GpioCtrlRegs.GPAPUD.bit.GPIO7 = 0; // Enable pull-up on GPIO7 (MCLKRA)
        (Comment as needed)
        GpioCtrlRegs.GPBPU.D.bit.GPIO58 = 0; // Enable pull-up on GPIO58 (MCLKRA)
        (Comment as needed)
        GpioCtrlRegs.GPAPUD.bit.GPIO23 = 0; // Enable pull-up on GPIO23 (MFSXA)
        //GpioCtrlRegs.GPAPUD.bit.GPIO5 = 0; // Enable pull-up on GPIO5 (MFSRA)
        (Comment as needed)
        GpioCtrlRegs.GPBPU.D.bit.GPIO59 = 0; // Enable pull-up on GPIO59 (MFSRA)
        (Comment as needed)
        /* Set qualification for selected pins to asynch only */
        // This will select asynch (no qualification) for the selected pins.
        // Comment out other unwanted lines.
        GpioCtrlRegs.GPAQSEL2.bit.GPIO20 = 3; // Asynch input GPIO20 (MDXA)
        GpioCtrlRegs.GPAQSEL2.bit.GPIO21 = 3; // Asynch input GPIO21 (MDRA)
        GpioCtrlRegs.GPAQSEL2.bit.GPIO22 = 3; // Asynch input GPIO22 (MCLKXA)
        //GpioCtrlRegs.GPAQSEL1.bit.GPIO7 = 3; // Asynch input GPIO7 (MCLKRA)
        (Comment as needed)
```

```
GpioCtrlRegs.GPBQSEL2.bit.GPIO58 = 3; // Asynch input GPIO58 (MCLKRA) (Comment  
as needed)  
GpioCtrlRegs.GPAQSEL2.bit.GPIO23 = 3; // Asynch input GPIO23 (MFSXA)  
//GpioCtrlRegs.GPAQSEL1.bit.GPIO5 = 3; // Asynch input GPIO5 (MFSRA)  
(Comment as needed)  
GpioCtrlRegs.GPBQSEL2.bit.GPIO59 = 3; // Asynch input GPIO59 (MFSRA) (Comment  
as needed)  
EDIS;  
}  
void mcbsp_test(void)  
{  
    Uint16 temp, i, mn;  
    mcbsp_InitMcbspaGpio(); //配置 MCBSP 的 IO 口  
    mcbsp_InitI2CGpio(); //配置为 IIC 的 IO 口  
    I2CA_Init(); //通过 IIC 就行初始化  
    // Clear Counters  
    // PassCount = 0;  
    // FailCount = 0;  
    //配置 AIC23  
    AIC23Write(0x00, 0x00);  
    Delay(100);  
    AIC23Write(0x02, 0x00);  
    Delay(100);  
    AIC23Write(0x04, 0x7f);  
    Delay(100);  
    AIC23Write(0x06, 0x7f);  
    Delay(100);  
    AIC23Write(0x08, 0x14);  
    Delay(100);  
    AIC23Write(0x0A, 0x00);  
    Delay(100);  
    AIC23Write(0x0C, 0x00);  
    Delay(100);  
    AIC23Write(0x0E, 0x43);  
    Delay(100);  
    AIC23Write(0x10, 0x23);  
    Delay(100);  
    AIC23Write(0x12, 0x01);  
    Delay(100); //AIC23Init  
    //配置 mcbsp a  
    InitMcbspaUser();  
    //发出报警声  
    for(mn=10;mn>0;mn--)  
    {
```

```
for (temp=30000; temp>0; temp-=100)
{
    for (i=0; i<2; i++)
    {
        y=5000;
        delay(temp);
        McbspaRegs.DXR1.all = y;// 输出左声道数据
        McbspaRegs.DXR2.all = y;// 输出右声道数据
        y=-5000;
        delay(temp);
        McbspaRegs.DXR1.all = y;// 输出左声道数据
        McbspaRegs.DXR2.all = y;// 输出右声道数据
    }
}
}
```

实验 29：音频实验之录音实验（Example_2833x_mcbsp_rec）

一 实验目的：

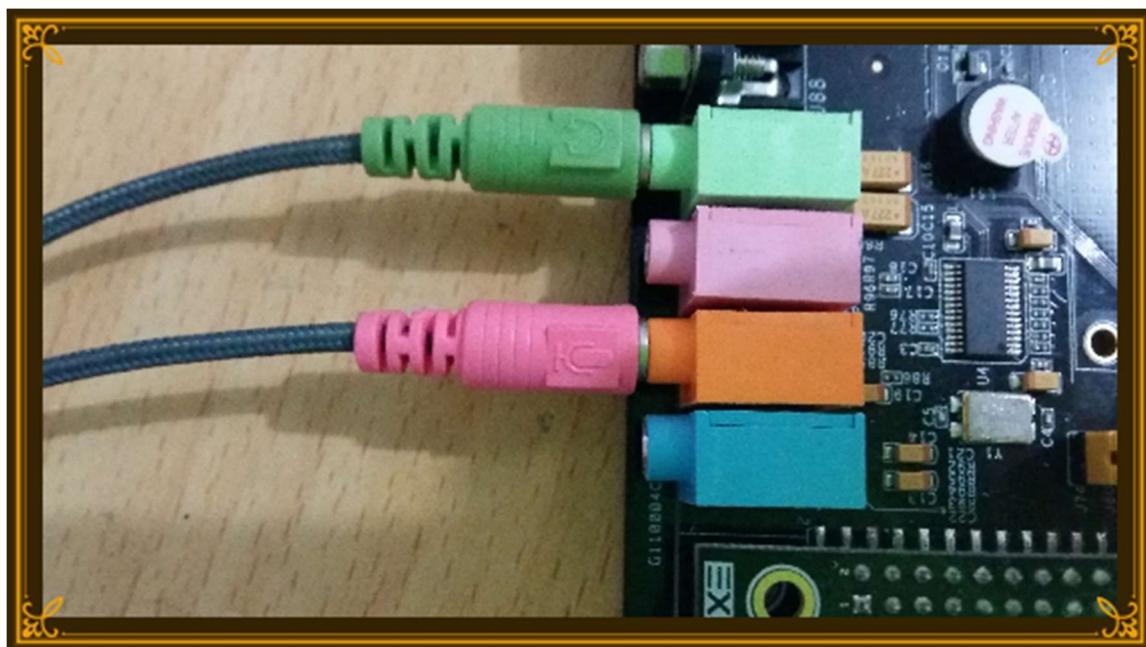
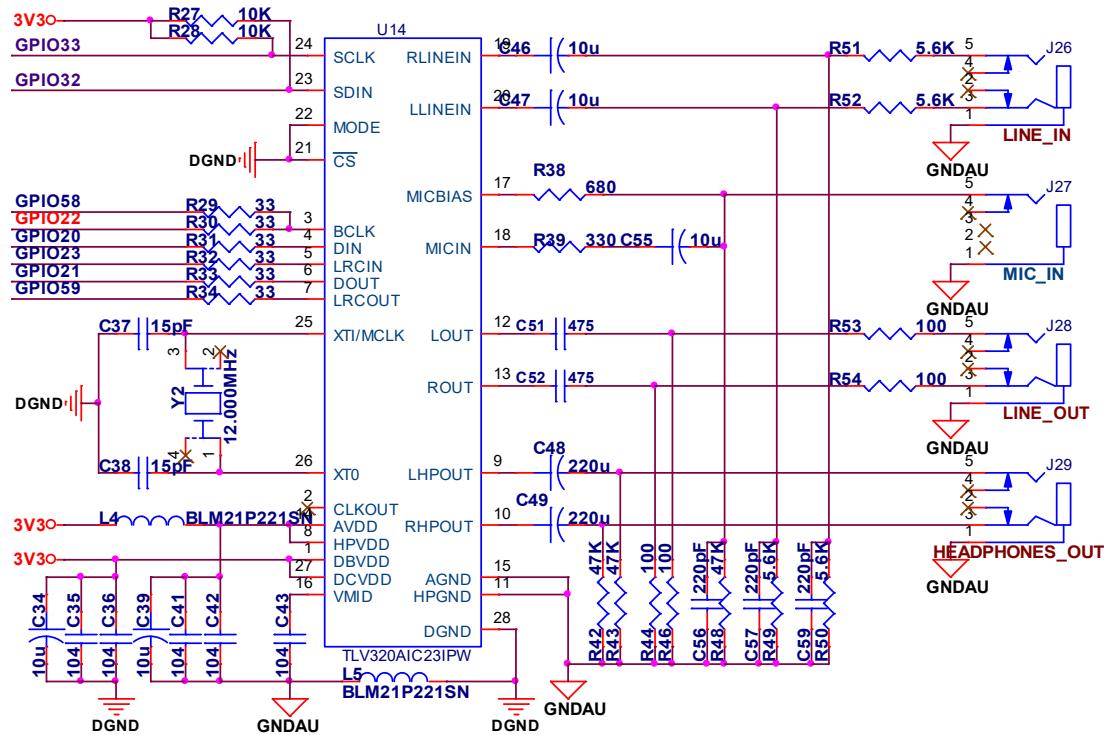
- ✧ 了解 DSP28335 的 MCBSP 外设；
- ✧ 了解 TLV320AIC23IPW 这款新品；

二 实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板一套，耳机一个；

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开；Mcbsp 是多通道串行接口。可以当作 SPI 使用。也可以当作非 SPI 功能。MCBSP 其实就是一个串行总线协议。只不过有两个时钟：接收时钟和发送时钟。这导致接收和发送独立使用自己的时钟；还有就是多了两个状态标志信号：一个是发送同步信号一个是接收同步信号。



- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：然后单击图中红色的方框处的调试按钮，进行调试。

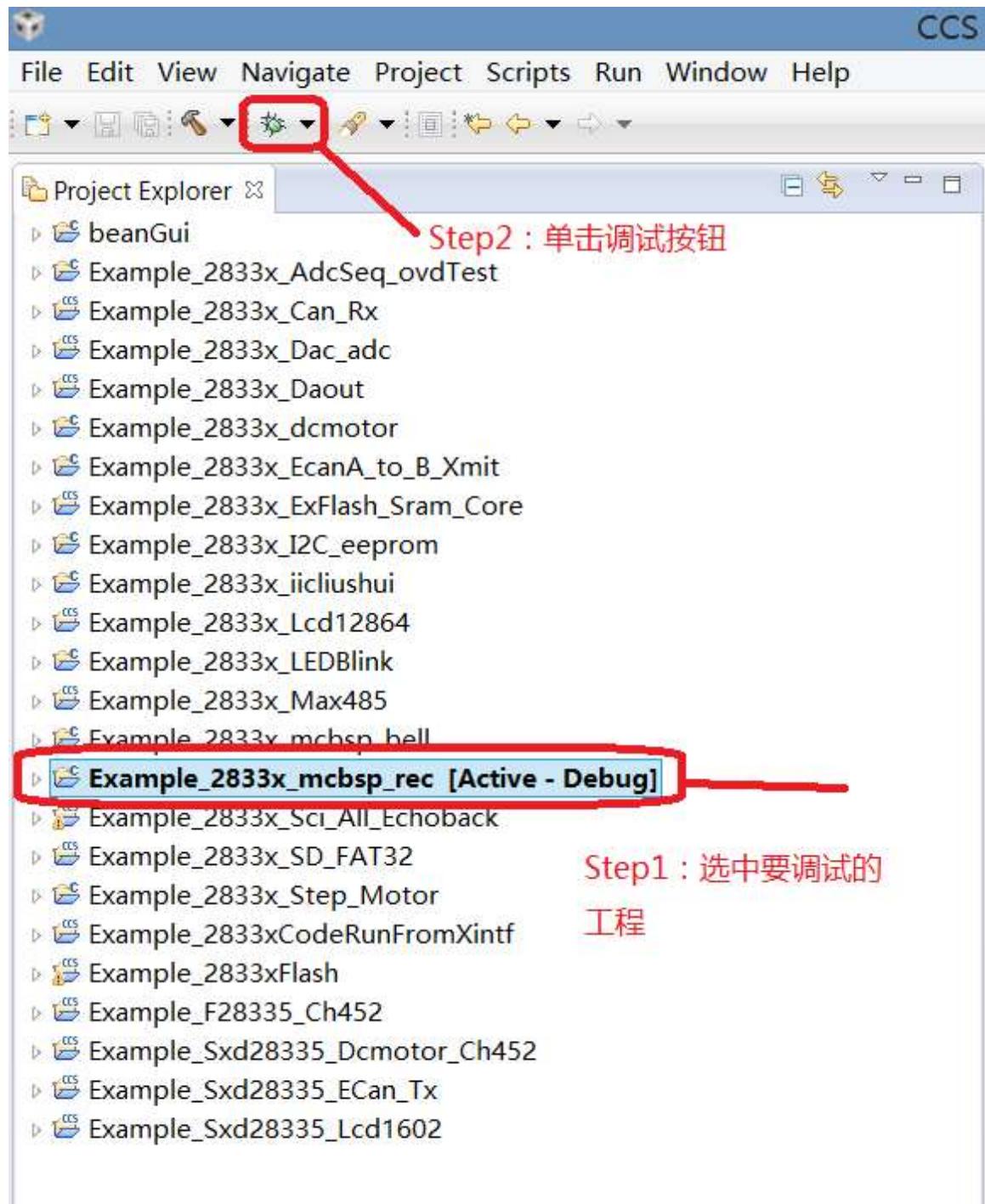


图 3 调试方法

✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

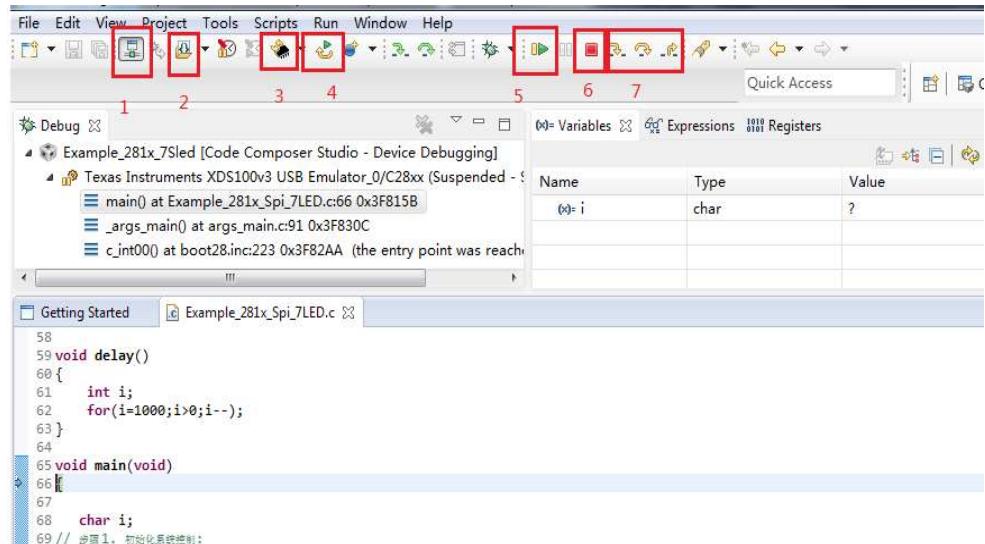


图 4 调试界面

- ⑧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ⑨ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ⑩ 图中 3 是 C P U 软 R e s e t ；
- 11 图中 5 是全速运行；
- 12 图中 6 是停止调试；
- 13 图中 7 是用于单步调试的；
- 14 图中 8 是用来恢复到程序的开始处。

四 实验现象

通过话筒将声音输入到 DSP28335 内部。28335 再通过耳麦将接收到的声音发送出去；

程序解析

```
//通过中断将接收的数据发送出去
interrupt void ISRMcbspSend(void)
{
    int temp1, temp2;
    //接收到的数据
    temp1=McbspaRegs.DRR1.all;
    temp2=McbspaRegs.DRR2.all;
    //发送出去
    McbspaRegs.DXR1.all = temp1;           //放音
    McbspaRegs.DXR2.all = temp2;
    PieCtrlRegs.PIEACK.all = 0x0020;
    // PieCtrlRegs.PIEIFR6.bit.INTx5 = 0;
    // ERTM;
}
```

实验 30: Mcbsp 模拟 SPI 总线实验 (Example_2833xMcBSP_SPI_DLDB)

一 实验目的:

- ✧ 了解 DSP28335 的 MCBSP 外设;
- ✧ 了解如何将 Mcbsp 配置为 SPI 模式;

二 实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开; 由于工作在自循环模式下, 所以不需要额外的设备;
- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程,
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

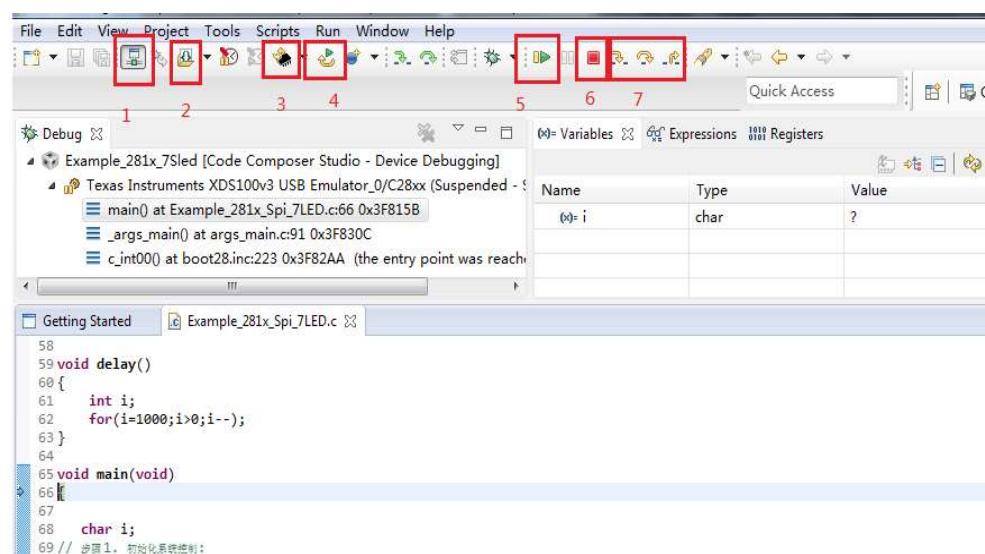


图 1 调试界面

- 15 图中 1 图标是用来进行与开发板进行连接的按钮;
- 16 图中 2 是用来下载 Debug 文件下的.out 文件的
- 17 图中 3 是 C P U 软 R e s e t;
- 18 图中 4 是调试时恢复到程序的开始处。
- 19 图中 5 是全速运行;
- 20 图中 6 是停止调试;
- 21 图中 7 是用于单步调试的;

四 实验现象

首先将下面的四个变量添加到观察窗口中。并设置一个断点。

```

82 // InitPeripherals();      // Not required for this example
83
84 // Step 5. User specific code,
85     init_mcbsp_spi();
86     sdata1 = 0x55aa;
87     sdata2 = 0xaa55;
88
89 // Main loop to transfer 32-bit words through MCBSP in SPI mode periodically
90     for(;;)
91     {
92         mcbsp_xmit(sdata1,sdata2);
93         while( McbspaRegs.SPCR1.bit.RRDY == 0 ) {}           // Master waits until
94         rdata2 = McbspaRegs.DRR2.all;                          // Read DRR2 first.
95         rdata1 = McbspaRegs.DRR1.all;                          // Then read DRR1 to
96         if((rdata2 != sdata2)&&(rdata1 != sdata1)) error(); // Check that cor
97         delay_loop();
98         sdata1^=0xFFFF;
99         sdata2^=0xFFFF;
100        __asm("    nop");                                // Good place for a
101    }
102 }
103

```

图 2 断点的设置

全速运行。



图 3 连续刷新按钮的设置

程序解析：

```

for(;;)
{
    mcbsp_xmit(sdata1,sdata2); // 发送数据
    while( McbspaRegs.SPCR1.bit.RRDY == 0 ) {}           // 等待接收数据
    rdata2 = McbspaRegs.DRR2.all;                          // 读取第一个数据
    rdata1 = McbspaRegs.DRR1.all;                          // 读取第二个数据
    if((rdata2 != sdata2)&&(rdata1 != sdata1)) error(); // 检查接收的数据是否正
    确
    delay_loop();
    sdata1^=0xFFFF;
    sdata2^=0xFFFF;
    __asm("    nop");                                // Good place for a
breakpoint
}

```

实验 31: Mcbsp 自测试模式实验 (Example_2833xMcBSP_DL)

一 实验目的:

- ✧ 了解 DSP28335 的 MCBSP 外设;
- ✧ 了解如何配置 MCBSP 使其能自测试;

二 实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开;
- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处;
- 给开发板上电。进行在线调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

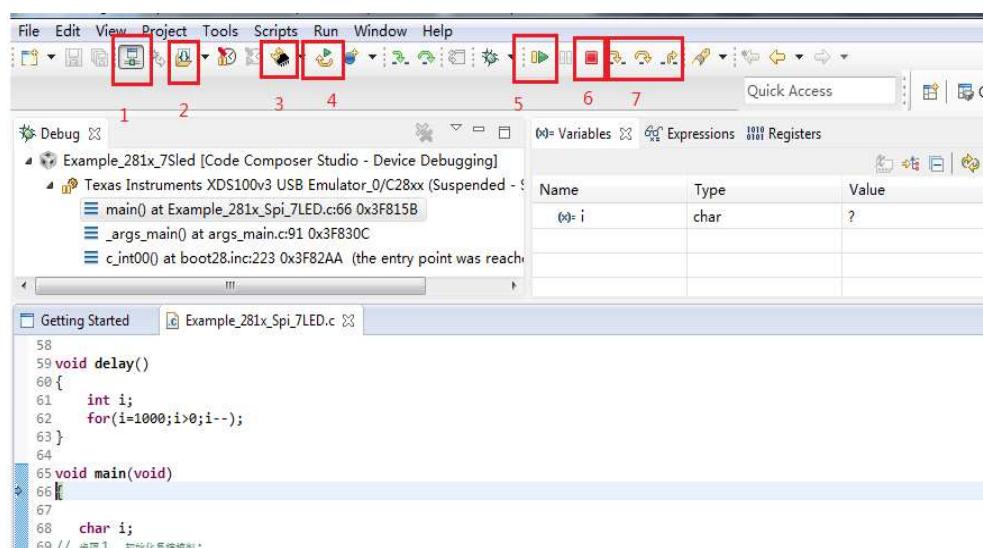


图 1 调试界面

- 22 图中 1 图标是用来进行与开发板进行连接的按钮;
23 图中 2 是用来下载 Debug 文件下的.out 文件的
24 图中 3 是 C P U 软 R e s e t;
25 图中 4 是调试时恢复到程序的开始处。
26 图中 5 是全速运行;
27 图中 6 是停止调试;
28 图中 7 是用于单步调试的;

四 实验现象

条件 1:

```
#include "DSP28x_Project.h"      // Device Headerfile and Examples Include File
// Choose a word size. Uncomment one of the following lines
#define WORD_SIZE     8          // Run a loopback test in 8-bit mode
```

```
//#define WORD_SIZE 16      // Run a loopback test in 16-bit mode
//#define WORD_SIZE 32      // Run a loopback test in 32-bit mode
如果配置字的大小为 8 bit 模式或 16 bit 模式，此时只会用到低 16 bit
(DXR1 和 DRR1)，不会用到高 16 bit (DXR2 和 DRR2)。
```

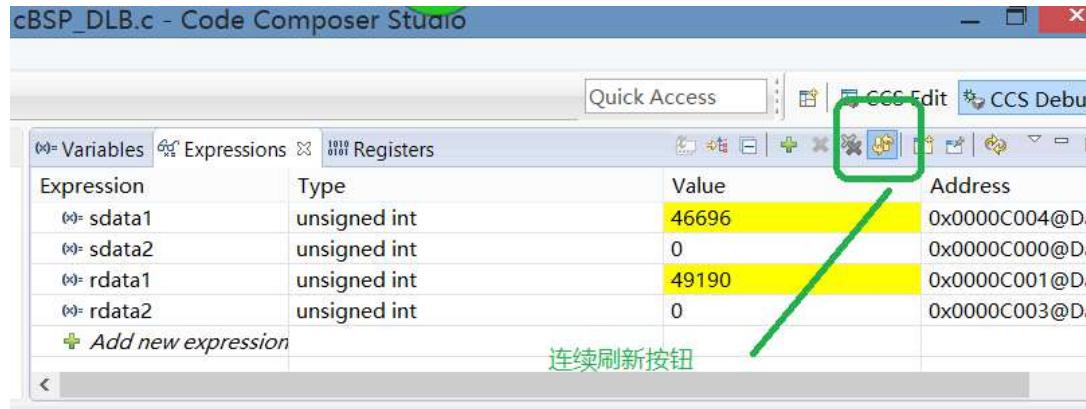


图 2 实验结果

条件 2：

```
#include "DSP28x_Project.h"      // Device Headerfile and Examples Include File
// Choose a word size. Uncomment one of the following lines
//#define WORD_SIZE 8      // Run a loopback test in 8-bit mode
//#define WORD_SIZE 16     // Run a loopback test in 16-bit mode
#define WORD_SIZE 32      // Run a loopback test in 32-bit mode
如果配置字的大小为 32 bit 模式，此时将用到低 16 bit (DXR1 和 DRR1) 和高
16 bit (DXR2 和 DRR2)。
```

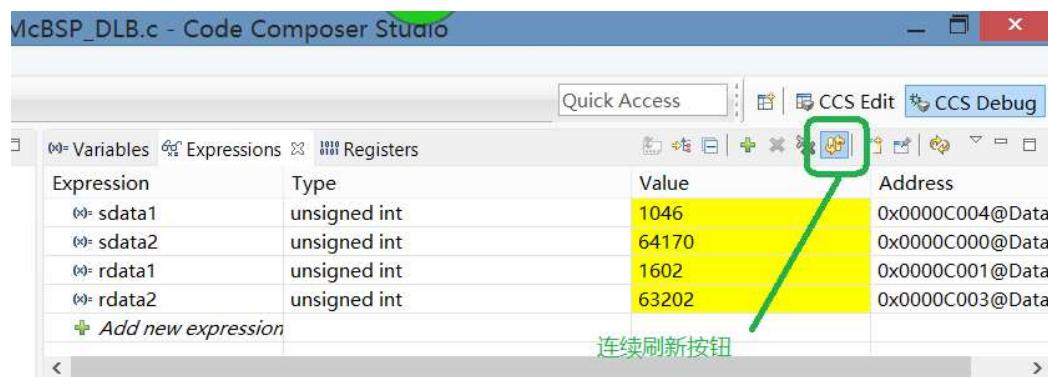


图 3 实验结果

程序解析

```
if(data_size == 8)          //如果是 8 bit 模式
{
    sdata2 = 0x0000;        //高位不用，所以不用关心这个变量
    sdata1 = 0x0000;        // 这个是将被发送的变量
    rdata2_point = 0x0000;   // value is a don't care for 8-bit mode
    rdata1_point = sdata1;
    for(;;)
```

```

{
    mcbsp_xmit(sdata1, sdata2);      // 将 8bit 发送出去
    sdata1++;
    sdata1 = sdata1 & 0x00FF;          //去掉高 8bit 的影响
    while(McbspaRegs.SPCR1.bit.RRDY == 0) { } // 等待接收
    rdata1 = McbspaRegs.DRR1.all;        // 读取接收到的结果
    if(rdata1 != rdata1_point) error(); // 检查接收是否有误，有误则
停止运行
    rdata1_point++;
    rdata1_point = rdata1_point & 0x00FF; // Keep it to 8-bits
    __asm("nop"); //延时一个机器周期
}
}

```

实验 32： McBSP 自测试模式中断实验 (Example_2833xMcBSP_DLDB_int)

一 实验目的：

✧ 了解 DSP28335 的 McBSP 外设；

二 实验设备

✧ PC 机一台；

✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；

✧ SXD28335 开发板一套；

三 实验步骤

✧ 首先将 CCS6.0 开发环境打开；

✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处；

✧ 给开发板上电。进行在线调试。

✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

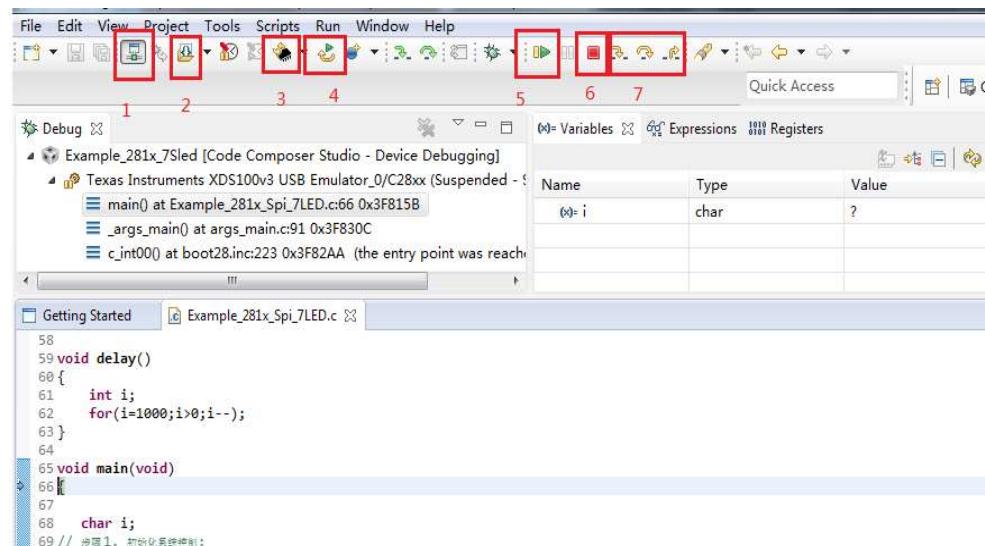


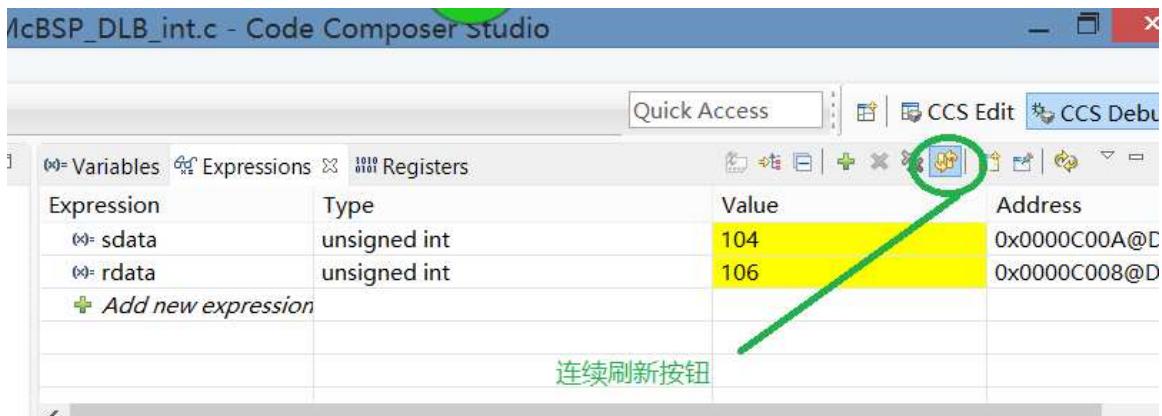
图 1 调试界面

29 图中 1 图标是用来进行与开发板进行连接的按钮；

- 30 图中 2 是用来下载 Debug 文件下的.out 文件的
 31 图中 3 是 C P U 软 R e s e t ;
 32 图中 4 是调试时恢复到程序的开始处。
 33 图中 5 是全速运行；
 34 图中 6 是停止调试；
 35 图中 7 是用于单步调试的；

四 实验现象

本实验是使能了 Mcbsp 的发送中断和接收中断。在发送中断函数里发送一个数据，在接收中断里接收数据；实验现象如下图：sdata 是要发送的数据，rdata 是接收数据；



程序解析

```

EALLOW;
PieVectTable.MRINTA= &Mcbsp_RxINTA_ISR; //将接收中断映射到中断向量表中
PieVectTable.MXINTA= &Mcbsp_TxINTA_ISR; //将发送中断映射到中断向量表中
EDIS;

//发送中断函数
_interrupt void Mcbsp_TxINTA_ISR(void)
{
    McbspaRegs.DXR1.all= sdata; //将数据发送出去
    sdata = (sdata+1)& 0x00FF ;
    // To receive more interrupts from this PIE group, acknowledge this interrupt
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;
}

//接收中断函数
_interrupt void Mcbsp_RxINTA_ISR(void)
{
    rdata=McbspaRegs.DRR1.all; //读取接收到的数据
    if (rdata != (rdata_point) & 0x00FF) ) error(); //检查接收到的数据是否和发送数据一致
    rdata_point = (rdata_point+1) & 0x00FF;
    // To receive more interrupts from this PIE group, acknowledge this interrupt
}

```

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;  
}
```

实验 33: Mcbsp 联合 DMA 实验 (Example_2833xMcBSP_DLDB_DMA)

一 实验目的:

- ✧ 了解 DSP28335 的 MCBSP 外设;
- ✧ 了解 DSP28335 的 DMA 外设;

二 实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤

- ✧ 首先将 CCS6.0 开发环境打开;
- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处;
- ✧ 给开发板上电。进行在线调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

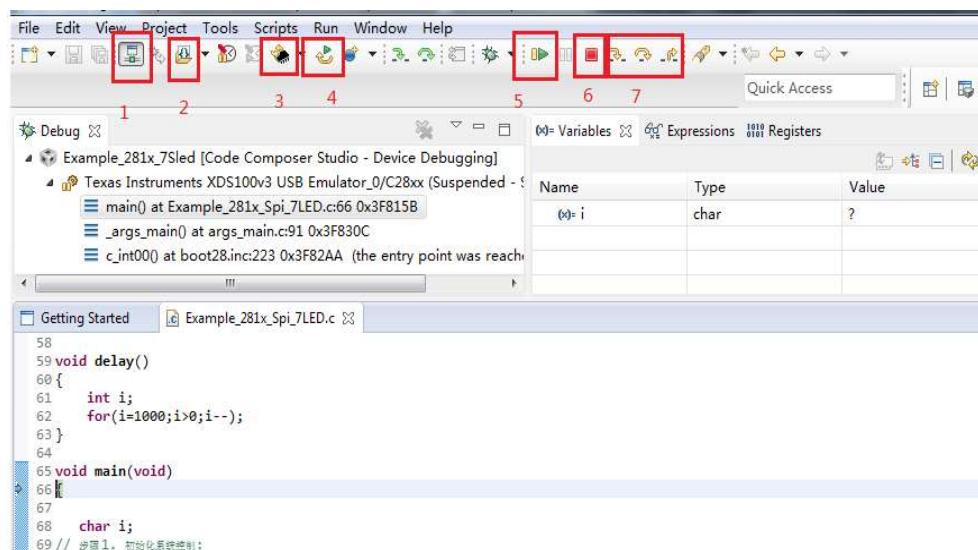


图 1 调试界面

- 36 图中 1 图标是用来进行与开发板进行连接的按钮;
- 37 图中 2 是用来下载 Debug 文件下的.out 文件的
- 38 图中 3 是 C P U 软 R e s e t;
- 39 图中 4 是调试时恢复到程序的开始处。
- 40 图中 5 是全速运行;
- 41 图中 6 是停止调试;
- 42 图中 7 是用于单步调试的;

四 实验现象

本实验使能了 Mcbsp 的自循环模式; 同时用 DMA 将要发送的数据放到 Mcbsp 的发送缓冲区。在 MCbps 的接收过程中, 是通过将接收的数据通过 DMA 放到接收

数组中。通过观察接收和发送数组可以发现他们内容是一样的。

The screenshot shows the 'Variables' tab in CCS. It lists two arrays: 'sdata' and 'rdata'. The 'sdata' array has elements [0 ... 99] and [100 ... 127]. The 'rdata' array has elements [0 ... 99]. The 'Value' column shows the memory address for each element. The 'Address' column shows the starting address of each array. The 'refresh' button in the toolbar is circled in green.

Expression	Type	Value	Address
sdata	unsigned int[128]	0x0000C000@Data	0x0000C000@Data
[0 ... 99]			
[100 ... 127]			
rdata	unsigned int[128]	0x0000C080@Data	0x0000C080@Data
[0 ... 99]			
[0]	unsigned int	0	0x0000C080@Data
[1]	unsigned int	1	0x0000C081@Data
[2]	unsigned int	2	0x0000C082@Data
[3]	unsigned int	3	0x0000C083@Data
...			

图 2 实验现象

程序解析

```
#pragma DATA_SECTION(sdata, "DMARAML4") // 将发送数组 sdata 放到 DMARAML4 中
#pragma DATA_SECTION(rdata, "DMARAML4") // 将接收数组 rdata 放到 DMARAML4 中
Uint16 sdata[128]; // Sent Data
Uint16 rdata[128]; // Received Data
```

而 DMARAML4 是在 28335_RAM_1nk.cmd 文件里定义的，具体定义如下面所示：

```
FPUmathTables : > FPUTABLES, PAGE = 0, TYPE = NOLOAD
DMARAML4 : > RAML4, PAGE = 1
DMARAML5 : > RAML5, PAGE = 1
```

注：PAGE = 1 说明是在数据区

DMA——MCBSP 发送过程：

```
DmaRegs.CH1.TRANSFER_SIZE = 127; // 发送的长度
DmaRegs.CH1.SRC_TRANSFER_STEP = 1; // Move to next word in buffer after each
word in a burst
DmaRegs.CH1.DST_TRANSFER_STEP = 0; // Don't move destination address
DmaRegs.CH1.SRC_ADDR_SHADOW = (UInt32) &sdata[0]; // 源地址
DmaRegs.CH1.SRC_BEG_ADDR_SHADOW = (UInt32) &sdata[0]; // 源开始地址, wrap 模式下用
```

DmaRegs.CH1.DST_ADDR_SHADOW = (UInt32) &McbspaRegs.DXR1.all; // 目的地地址

DmaRegs.CH1.DST_BEG_ADDR_SHADOW = (UInt32) &McbspaRegs.DXR1.all; // 目的地地址的起始地址，只有在 wrap 模式下用到

DMA——MCBSP 接收过程：

```
DmaRegs.CH2.TRANSFER_SIZE = 127; // 发送大小
DmaRegs.CH2.SRC_TRANSFER_STEP = 0; // Don't move source address
DmaRegs.CH2.DST_TRANSFER_STEP = 1; // Move to next word in buffer after each
word in a burst
DmaRegs.CH2.SRC_ADDR_SHADOW = (UInt32) &McbspaRegs.DRR1.all; // 源地址
DmaRegs.CH2.SRC_BEG_ADDR_SHADOW = (UInt32) &McbspaRegs.DRR1.all; // 源地址起始地
```

址，只有在 w r a p 模式下用到

```
DmaRegs.CH2.DST_ADDR_SHADOW = (UInt32) &rdata[0]; //目的地址
DmaRegs.CH2.DST_BEG_ADDR_SHADOW = (UInt32) &rdata[0]; //目的地址起始地址，只有
在 w r a p 模式下才用到
```

实验 34：片内 DMA 数据搬移试验（Example_2833xDMA_ram_to_ram）

一、实验目的：

- ✧ 了解如何使用 dsp23885 内部集成的 DMA 外设；

二、实验设备

注：模拟量输入范围：0.0V~3.0V；

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套；

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；本实验是通过 DMA 将片内 L5 的 1024 个字搬到片内的 L4 空间。
- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，进行调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

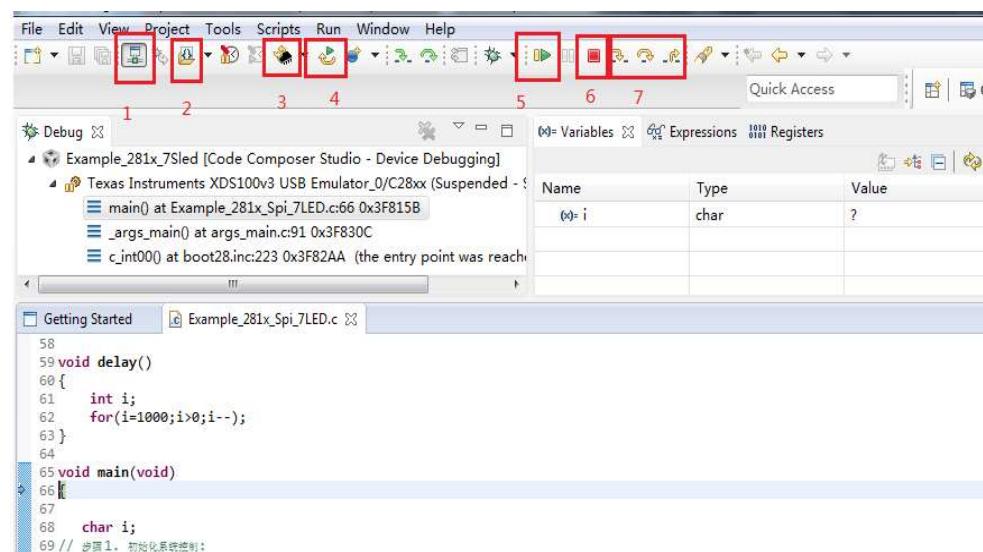


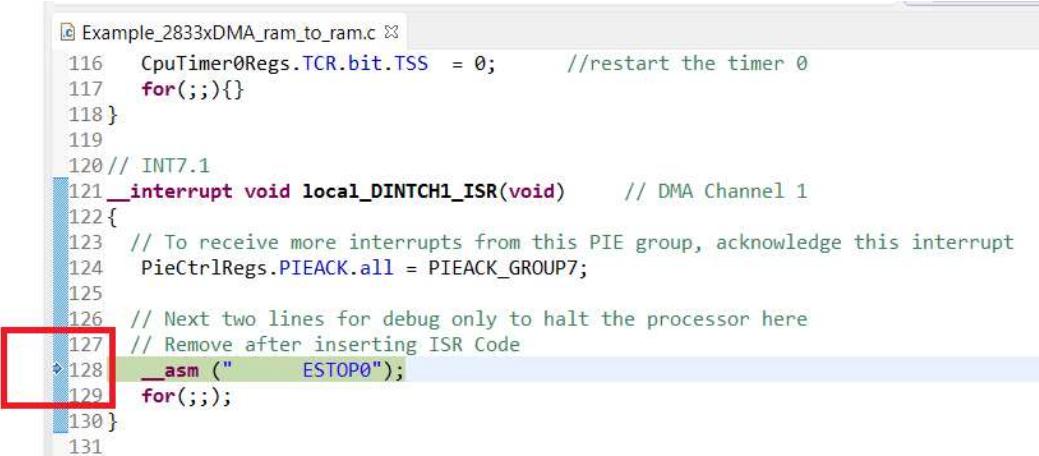
图 1 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；

- ◆ 图中 6 是停止调试;
- ◆ 图中 7 是用于单步调试的;

四、试验现象:

首先将 `DMABuf1` 和 `DMABuf2` 添加到观察窗口中, 直接用鼠标左键单击全速运行。程序会停到这里:



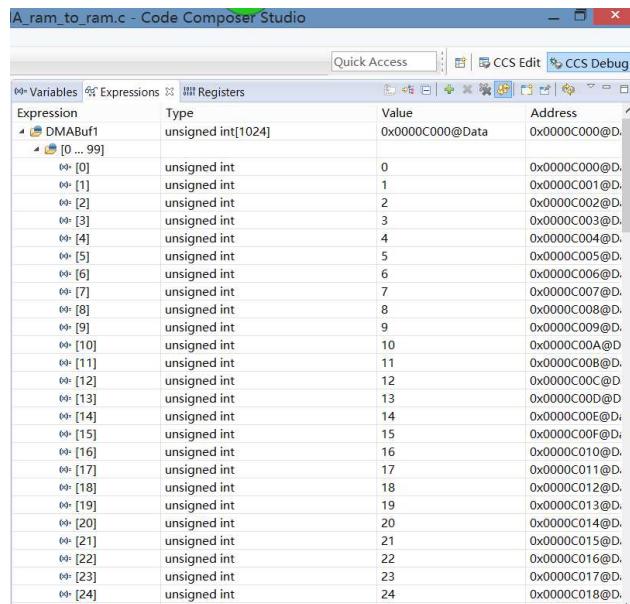
```

116     CpuTimer0Regs.TCR.bit.TSS = 0;      //restart the timer 0
117     for(;;){}
118 }
119
120 // INT7.1
121 __interrupt void local_DINTCH1_ISR(void)      // DMA Channel 1
122 {
123     // To receive more interrupts from this PIE group, acknowledge this interrupt
124     PieCtrlRegs.PIEACK.all = PIEACK_GROUP7;
125
126     // Next two lines for debug only to halt the processor here
127     // Remove after inserting ISR Code
128     _asm ("ESTOP0");
129     for(;;);
130 }
131

```

图 2 运行后现象

现象: `DMABuf1` 的值和 `DMABuf2` 的值一致。



Expression	Type	Value	Address
DMABuf1 [0 ... 99]	unsigned int[1024]	0x0000C000@Data	0x0000C000@D
↳ [0]	unsigned int	0	0x0000C000@D
↳ [1]	unsigned int	1	0x0000C001@D
↳ [2]	unsigned int	2	0x0000C002@D
↳ [3]	unsigned int	3	0x0000C003@D
↳ [4]	unsigned int	4	0x0000C004@D
↳ [5]	unsigned int	5	0x0000C005@D
↳ [6]	unsigned int	6	0x0000C006@D
↳ [7]	unsigned int	7	0x0000C007@D
↳ [8]	unsigned int	8	0x0000C008@D
↳ [9]	unsigned int	9	0x0000C009@D
↳ [10]	unsigned int	10	0x0000C00A@D
↳ [11]	unsigned int	11	0x0000C00B@D
↳ [12]	unsigned int	12	0x0000C00C@D
↳ [13]	unsigned int	13	0x0000C00D@D
↳ [14]	unsigned int	14	0x0000C00E@D
↳ [15]	unsigned int	15	0x0000C00F@D
↳ [16]	unsigned int	16	0x0000C010@D
↳ [17]	unsigned int	17	0x0000C011@D
↳ [18]	unsigned int	18	0x0000C012@D
↳ [19]	unsigned int	19	0x0000C013@D
↳ [20]	unsigned int	20	0x0000C014@D
↳ [21]	unsigned int	21	0x0000C015@D
↳ [22]	unsigned int	22	0x0000C016@D
↳ [23]	unsigned int	23	0x0000C017@D
↳ [24]	unsigned int	24	0x0000C018@D

图 3 实验结果

程序解析:

// 首先将两个数组放到 `DMARAML4` 和 `DMARAML5` 这两个区域中;

```

#pragma DATA_SECTION(DMABuf1, "DMARAML4");
#pragma DATA_SECTION(DMABuf2, "DMARAML5");

```

```

volatile Uint16 DMABuf1[1024];
volatile Uint16 DMABuf2[1024];

```

而 DMARAML4 和 DMARAML5 这两个区域在 CMD 中是按照如下定义的： PAGE=1 说明是放到了数据区域：

```
DMARAML4      : > RAML4,      PAGE = 1
DMARAML5      : > RAML5,      PAGE = 1
// DMA 通道的源和目的地址配置方式：
void DMACH1AddrConfig(volatile UInt16 *DMA_Dest, volatile UInt16 *DMA_Source)
{
    EALLOW;
    // Set up SOURCE address:
    DmaRegs.CH1.SRC_BEG_ADDR_SHADOW = (UInt32)DMA_Source; //源地址的开始
    DmaRegs.CH1.SRC_ADDR_SHADOW =      (UInt32)DMA_Source;
    // Set up DESTINATION address:
    DmaRegs.CH1.DST_BEG_ADDR_SHADOW = (UInt32)DMA_Dest;      //目的地址的开始
    DmaRegs.CH1.DST_ADDR_SHADOW =      (UInt32)DMA_Dest;
    EDIS;
}
```

实验 35: 片内外 RAM 通过 DMA 进行数据搬移(Example_2833xDMA_xintf_to_ram)

一 实验目的:

- ✧ 了解如何使用 dsp23885 内部集成的 DMA 外设；
- ✧ 了解 XINTF 外设的使用方法；

二 实验设备:

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套；

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开；

本实验是利用 DMA 将片外 SRAM 内的 1024 字节搬移到片内的 1024 字节的 Ram 中。也就是将 DMABuf2 (片外) 的内容搬到 DMABuf1 (片内) 中；

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JTAG 端插到 SXD28335 开发板的 JTAG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调试：
- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

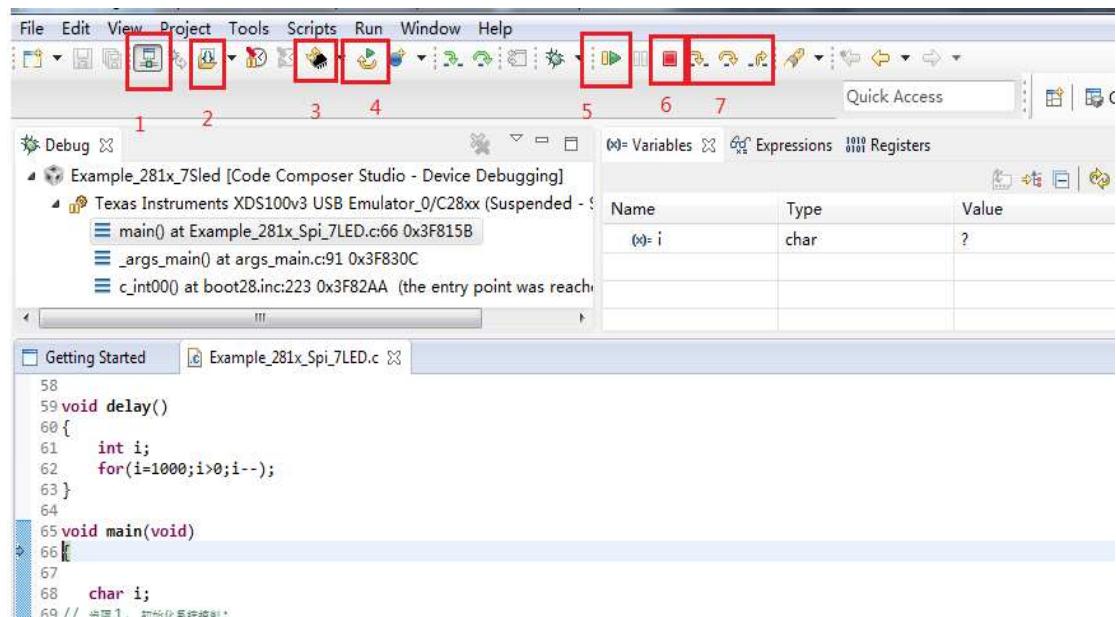
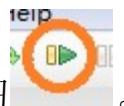


图 3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ◆ 图中 3 是 C P U 软 R e s e t；
- ◆ 图中 4 是调试时恢复到程序的开始处。
- ◆ 图中 5 是全速运行；
- ◆ 图中 6 是停止调试；
- ◆ 图中 7 是用于单步调试的；



单击全速运行按钮。

试验现象：

首先将 `DMABuf1` 和 `DMABuf2` 添加到观察窗口中，直接用鼠标左键单击全速运行。程序会停到这里：

```

116     CpuTimer0Regs.TCR.bit.TSS = 0;      //restart the timer 0
117     for(;;){}
118 }
119
120 // INT7.1
121 __interrupt void local_DINTCH1_ISR(void)      // DMA Channel 1
122 {
123     // To receive more interrupts from this PIE group, acknowledge this interrupt
124     PieCtrlRegs.PIEACK.all = PIEACK_GROUP7;
125
126     // Next two lines for debug only to halt the processor here
127     // Remove after inserting ISR Code
128     __asm ("    ESTOP0");
129     for(;;)
130 }
131

```

图 2 运行后现象

现象：`DMABuf1` 的值和 `DMABuf2` 的值一致。

Expression	Type	Value	Address
DMABuf1	unsigned int[1024]	0x0000C000@Data	0x0000C000@D
[0 ... 99]			
[0]	unsigned int	0	0x0000C000@D
[1]	unsigned int	1	0x0000C001@D
[2]	unsigned int	2	0x0000C002@D
[3]	unsigned int	3	0x0000C003@D
[4]	unsigned int	4	0x0000C004@D
[5]	unsigned int	5	0x0000C005@D
[6]	unsigned int	6	0x0000C006@D
[7]	unsigned int	7	0x0000C007@D
[8]	unsigned int	8	0x0000C008@D
[9]	unsigned int	9	0x0000C009@D
[10]	unsigned int	10	0x0000C00A@D
[11]	unsigned int	11	0x0000C00B@D
[12]	unsigned int	12	0x0000C00C@D
[13]	unsigned int	13	0x0000C00D@D
[14]	unsigned int	14	0x0000C00E@D
[15]	unsigned int	15	0x0000C00F@D
[16]	unsigned int	16	0x0000C010@D
[17]	unsigned int	17	0x0000C011@D
[18]	unsigned int	18	0x0000C012@D
[19]	unsigned int	19	0x0000C013@D
[20]	unsigned int	20	0x0000C014@D
[21]	unsigned int	21	0x0000C015@D
[22]	unsigned int	22	0x0000C016@D
[23]	unsigned int	23	0x0000C017@D
[24]	unsigned int	24	0x0000C018@D
...			

图 3 实验结果

程序解读：

```

// 通过这两个语句将数组 DMABuf1 放到 DMARAML4 空间。将数组 DMABuf2 放到片外 ZONE6DATA 空间
#pragma DATA_SECTION(DMABuf1, "DMARAML4");
#pragma DATA_SECTION(DMABuf2, "ZONE6DATA");
volatile Uint16 DMABuf1[BUF_SIZE];

```

```
volatile Uint16 DMABuf2[BUF_SIZE];  
// 通过下面这三个语句指定了 DMA 的源地址和目的地址  
DMADest = &DMABuf1[0];  
DMASource = &DMABuf2[0];  
DMACH1AddrConfig(DMADest, DMASource);
```

实验 36：Can 发送试验（Example_Sxd28335_ECan_Tx）

一、实验目的：

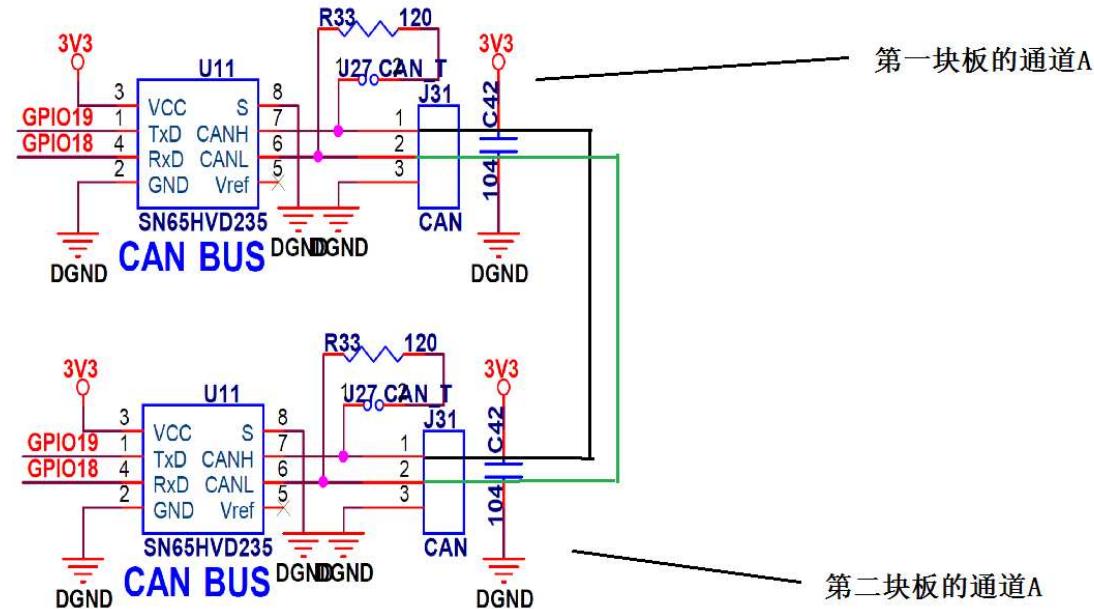
- ✧ 了解如何使用 dsp23885 内部集成 eCAN 外设；
- ✧ 通过两块板子间 CAN 通信的实验学习通信机制；

二、实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套；
- ✧ SXD28335 开发板两套, CAN 总线一条；

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：



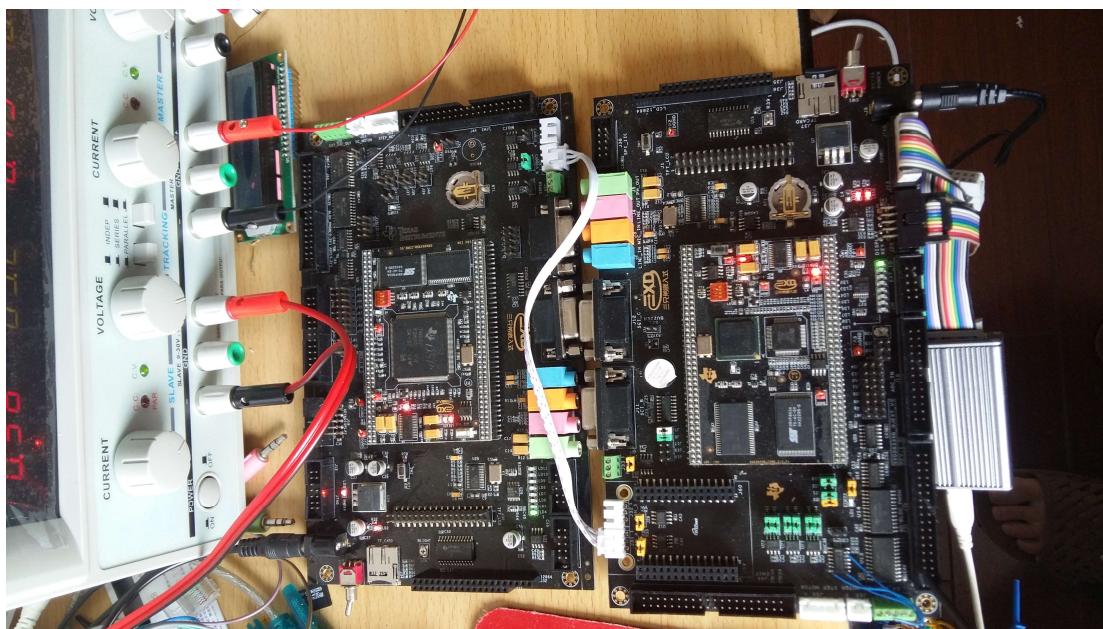


图 1 硬件连接方式

本程序是用第一块开发板的通道 CAN_A 作为发送, 第二块开发板的通道 CAN_A 作为接收, 将发送程序加载到第一块开发板里。将接收程序加载到第二块开发板里。接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;

◆ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:
然后单击图中方框处的调试按钮, 进行调试。

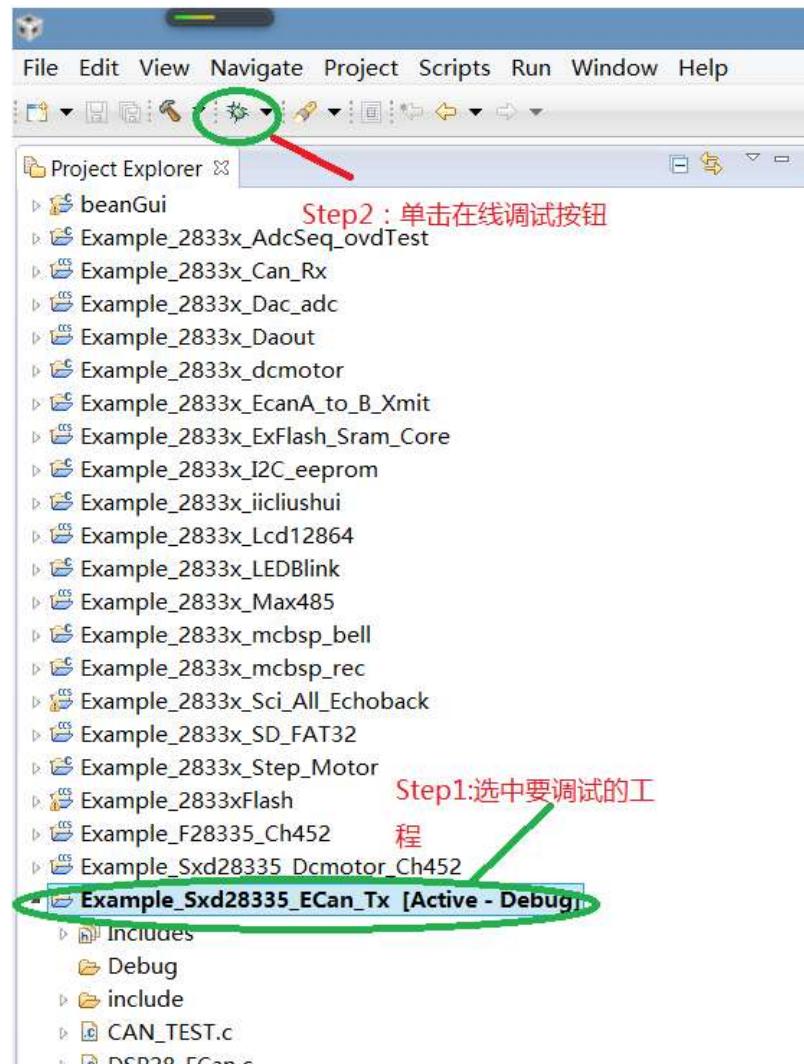


图 2 调试方法

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

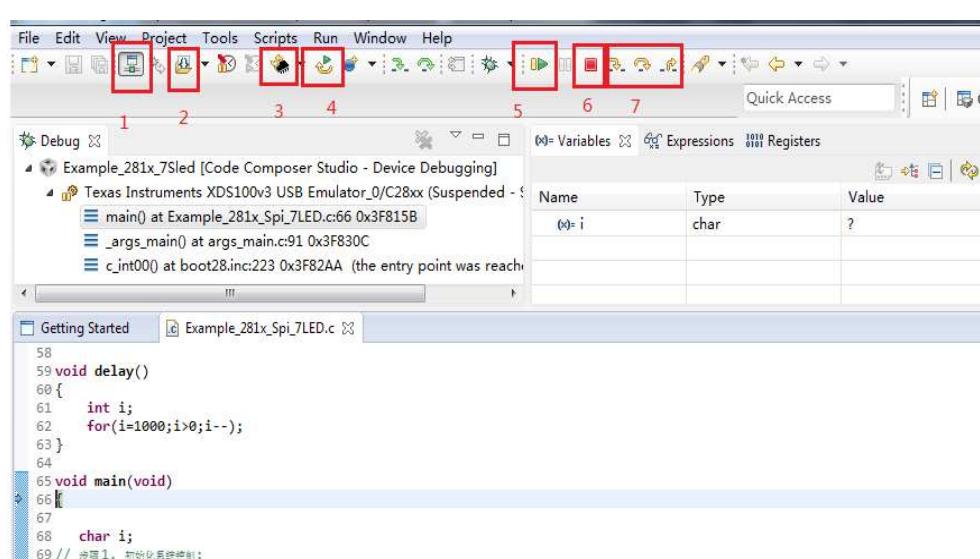


图 3 调试界面

- ◆ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ◆ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ◆ 图中 3 是 C P U 软 R e s e t ；
- ◆ 图中 4 是调试时恢复到程序的开始处。
- ◆ 图中 5 是全速运行；
- ◆ 图中 6 是停止调试；
- ◆ 图中 7 是用于单步调试的；

全速运行；要保证开发板 A 正常工作，并保持当前运行状态；
接下来我们给开发板 B 下载接收程序。

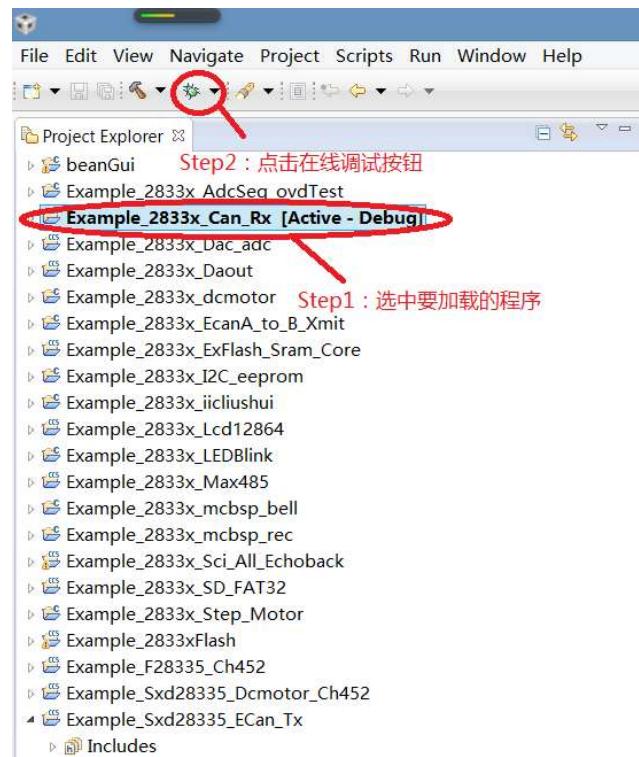


图 4 加载程序方法

加载完程序后，开始在线运行。在接收节点 B 的程序中我们找到接收中断函数。
并将中断函数里的接收变量加载到观察窗口中。

四、试验现象：

首先将 ECanAMboxes 添加到观察窗口中，如下图所示：

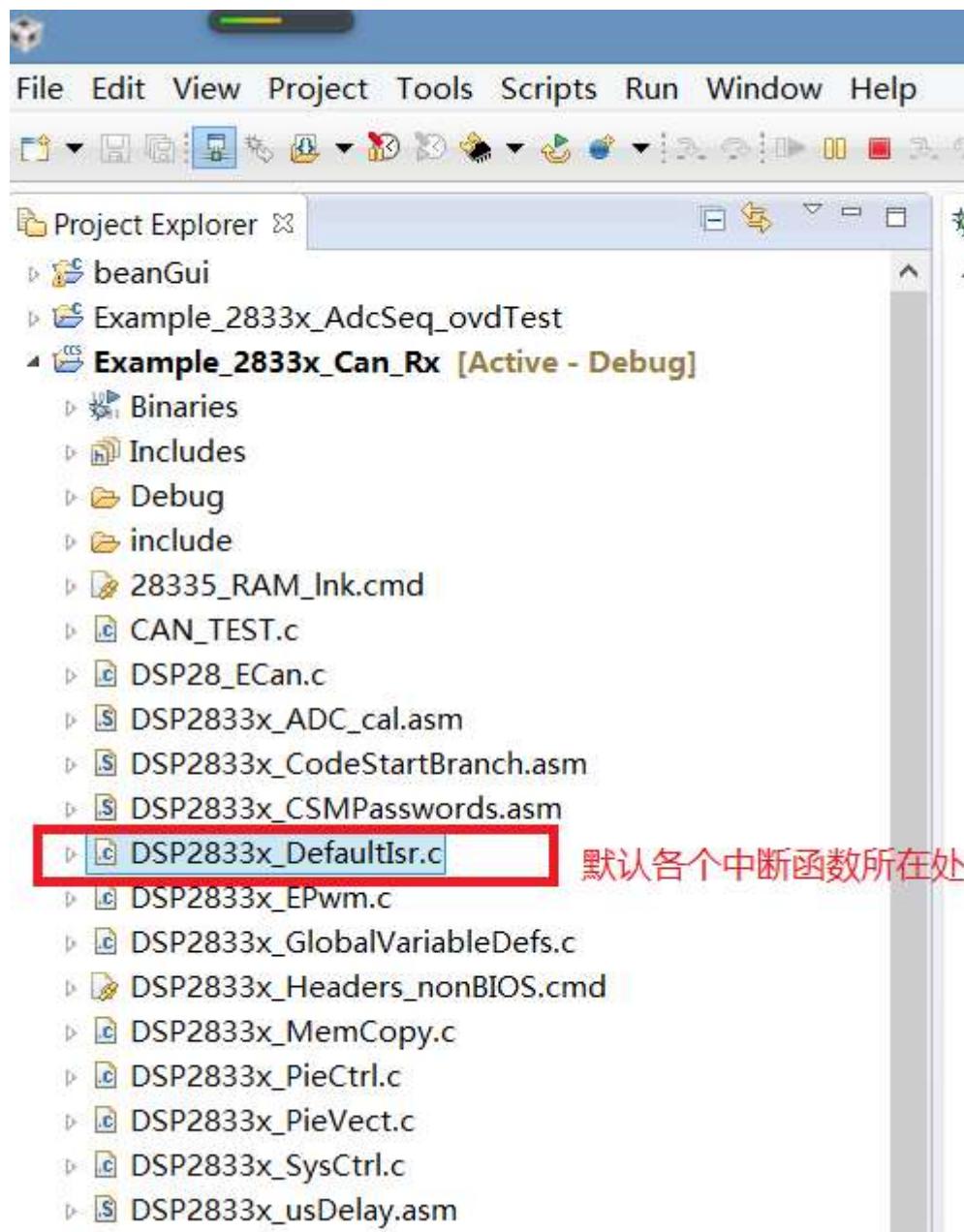


图 4 添加变量到观察窗口中

在 975 行找到接收中断函数：

```
interrupt void ECANOINTA_ISR(void) // eCAN-A
{
    while(ECanaRegs.CANRMP.all != 0x00010000) ;
    ECanaRegs.CANRMP.all = 0x00010000;
    //收到的数据在接收邮箱 Mbox16
    Rec_l = ECanaMboxes.MBOX16.MDL.all;
    Rec_h = ECanaMboxes.MBOX16.MDH.all;
    PieCtrlRegs.PIEACK.bit.ACK9= 1;
    EINT;
}
```

将接收变量 Rec_l 和 Rec_h 添加到观察窗口中。

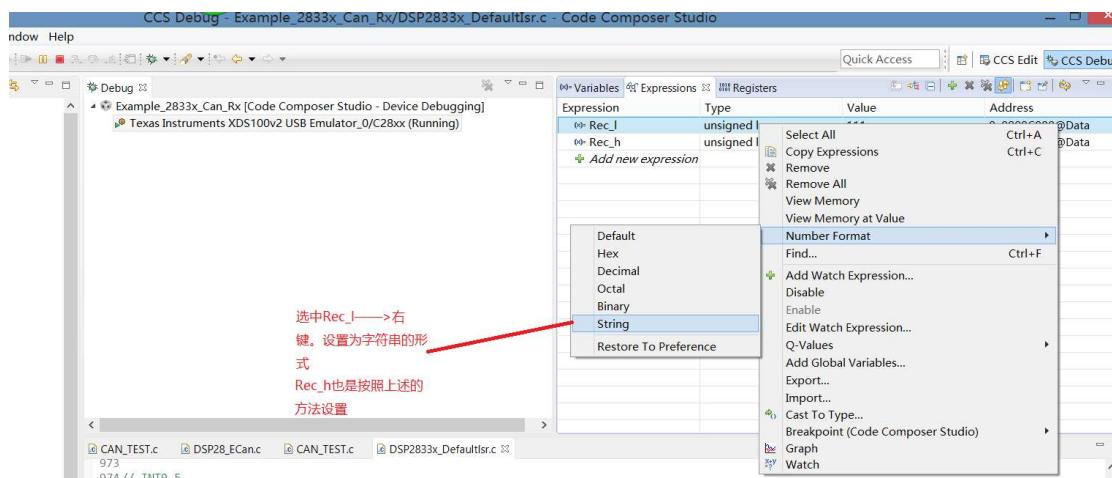


图 5 变量的添加

现象：

会发现这两个字符显示的是“HelloSxD”中的某个字母；
邮箱 ID 相同的 CANA 和 CANB 相互通信。

程序解析 (请先学习一下 CAN 的自循环程序，也就是一个板子的双通道通信程序)

```
void Can_Tx_Test(void)
{
    int temp=0;
    while(1)
    {
        //判断是否发送完毕
        ECanaRegs.CANTRS.all = 0x00000001;
        while(ECanaRegs.CANTA.all == 0);
        ECanaRegs.CANTA.all = 0x00000001;
        //向发送邮箱里装入要发送的数据
        ECanaMboxes.MBOX0.MDL.all = senddata[temp];
        ECanaMboxes.MBOX0.MDH.all = senddata[temp+1];
        temp=temp+2;
        if(temp>8)
            temp=0;
    }
}
```

实验 37：Can 接收试验 (Example_2833x_Can_Rx)

一、实验目的：

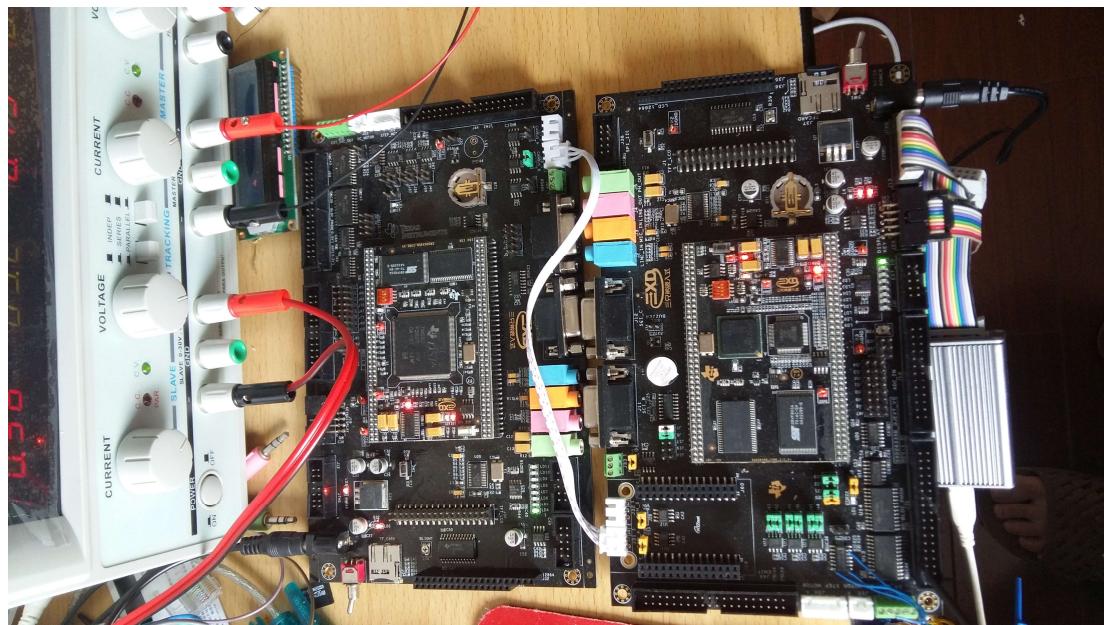
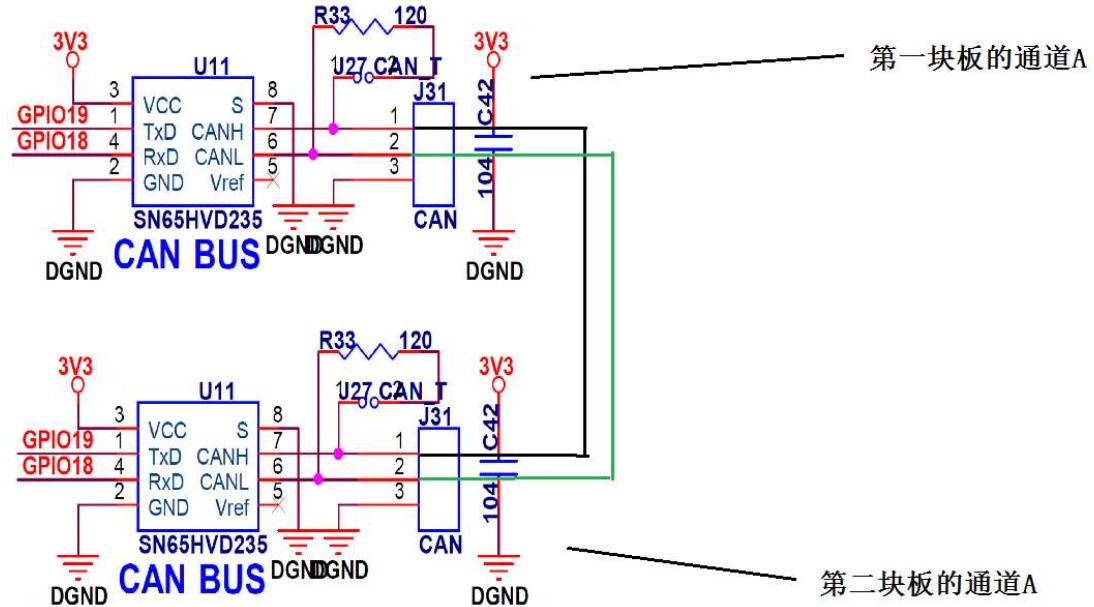
- ✧ 了解如何使用 dsp23885 内部集成 eCAN 外设；
- ✧ 通过两块板子间 CAN 通信的实验学习通信机制；

二、实验设备

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板**两套**, CAN 总线一条；

三、实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：



本程序是用第一块开发板的通道 CAN_A 作为发送, 第二块开发板的通道 CAN_A 作为接收, 将发送程序加载到第一块开发板里。将接收程序加载到第二块开发板里。接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

- ◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：然后单击图中方框处的调试按钮，进行调试。

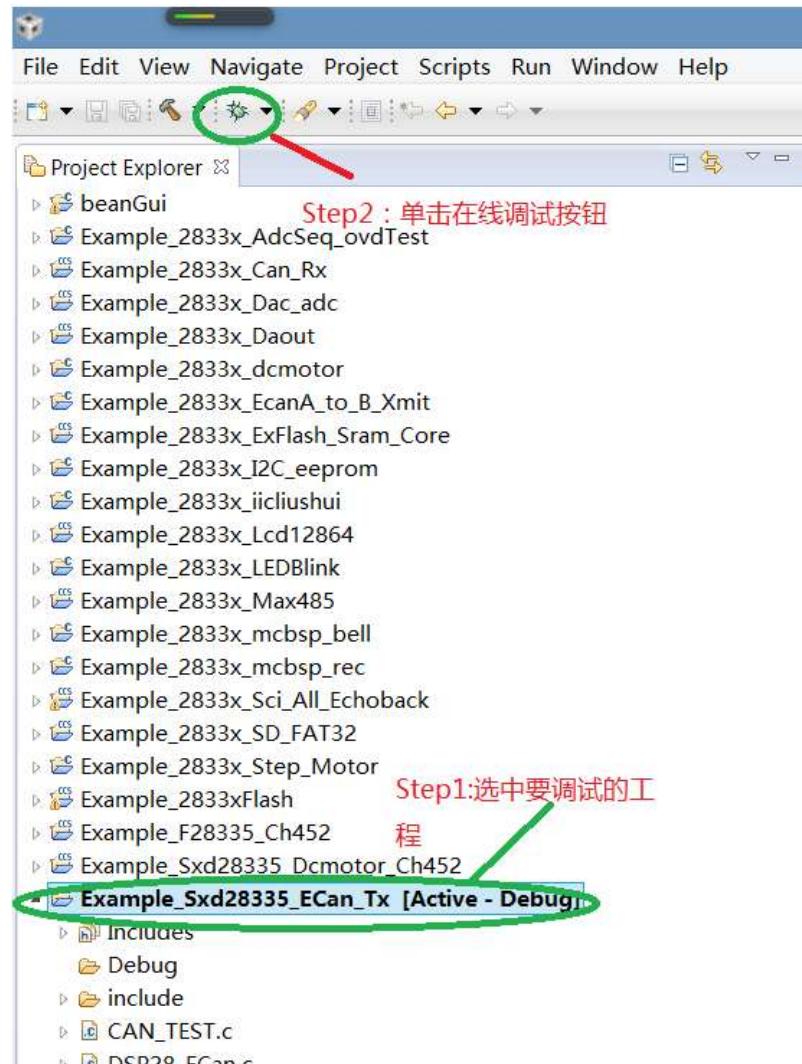


图 2 调试方法

- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

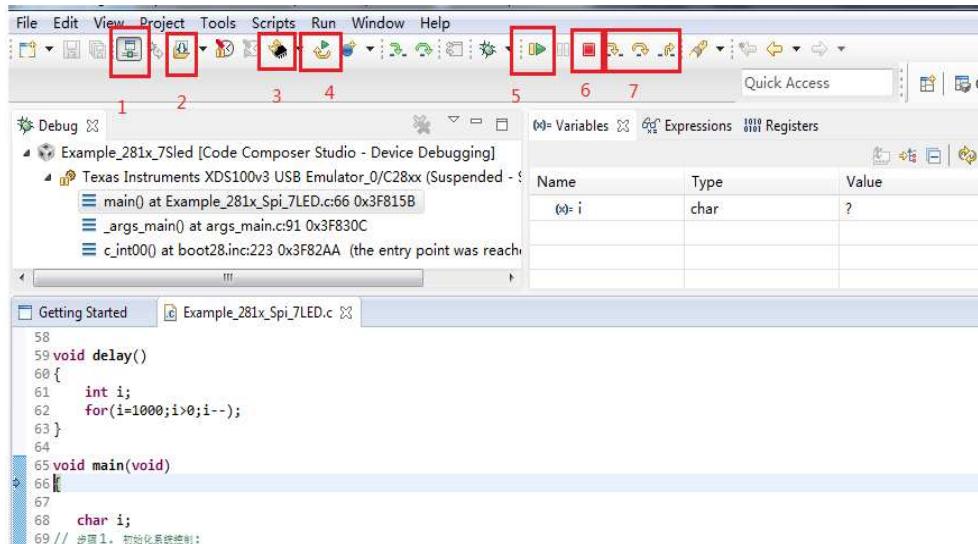


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮;
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t ;
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行;
- ✧ 图中 6 是停止调试;
- ✧ 图中 7 是用于单步调试的;

全速运行；要保证开发板 A 正常工作，并保持当前运行状态；接下来我们给开发板 B 下载接收程序。

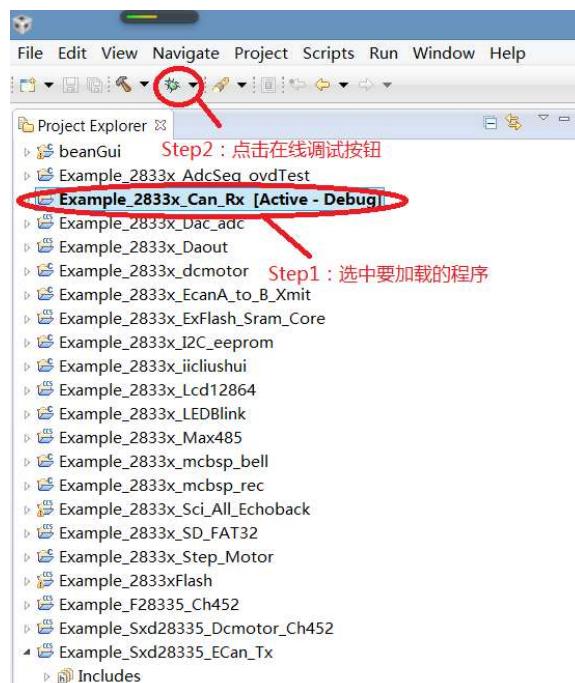


图 4 加载程序方法

加载完程序后，开始在线运行。在接收节点 B 的程序中我们找到接收中断函数。并将中断函数里的接收变量加载到观察窗口中。

四、试验现象：

首先将 ECanAMboxes 添加到观察窗口中，如下图所示：

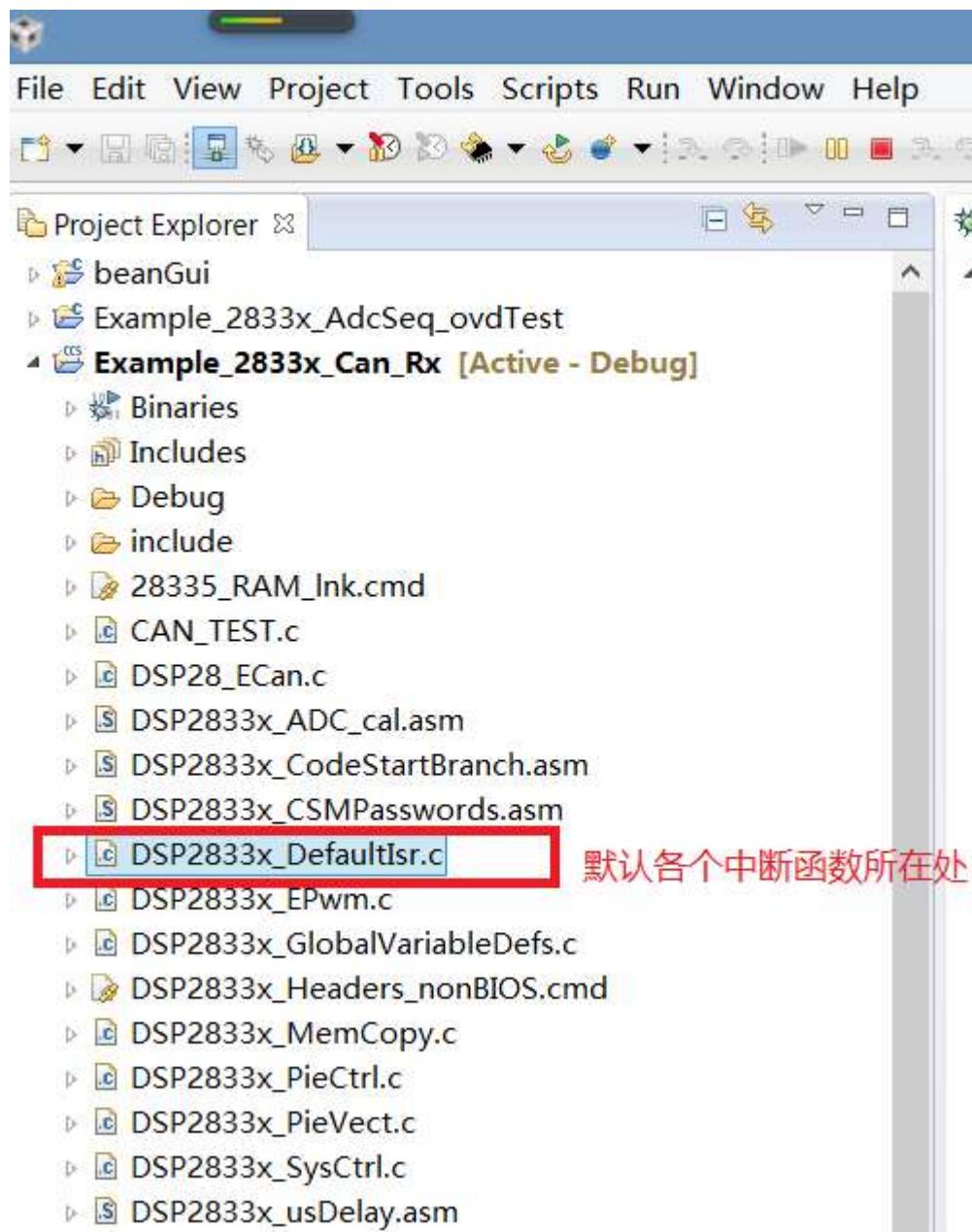


图 4 添加变量到观察窗口中

在 975 行找到接收中断函数：

```
interrupt void ECAN0INTA_ISR(void) // eCAN-A
{
    while(ECanaRegs.CANRMP.all != 0x00010000) ;
    ECanaRegs.CANRMP.all = 0x00010000;
```

```
//收到的数据在接收邮箱 Mbox16
Rec_l = ECanaMboxes.MBOX16.MDL.all;
Rec_h = ECanaMboxes.MBOX16.MDH.all;
PieCtrlRegs.PIEACK.bit.ACK9= 1;
EINT;
}
```

将接收变量 Rec_l 和 Rec_h 添加到观察窗口中。

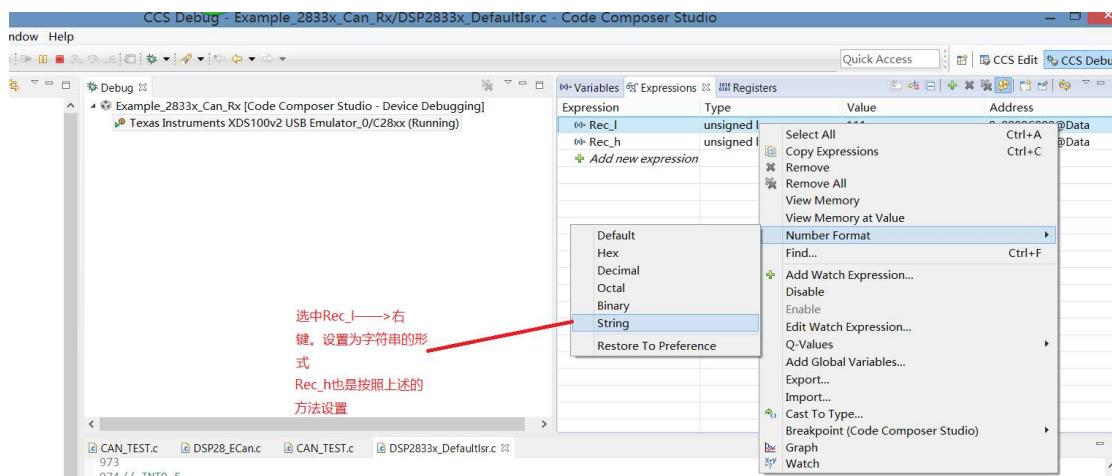


图 5 变量的添加

现象：

会发现这两个字符显示的是“HelloSXD”中的某个字母；
邮箱 ID 相同的 CANA 和 CANB 相互通信。

Expression	Type	Value	Address
0x Rec_l	unsigned long	o (String)	0x0000C000@Data
0x Rec_h	unsigned long	I (String)	0x0000C002@Data
+ Add new expression			

程序解析 (请先学习一下 CAN 的自循环程序，也就是一个板子的双通道通信程序)

```
void Can_Tx_Test(void)
{
    int temp=0;
    while(1)
    {
        //判断是否发送完毕
        ECanaRegs.CANTRS.all = 0x00000001;
        while(ECanaRegs.CANTA.all == 0);
        ECanaRegs.CANTA.all = 0x00000001;
        //向发送邮箱里装入要发送的数据
    }
}
```

```

ECanaMboxes. MBOX0. MDL. all =senddata[temp];
ECanaMboxes. MBOX0. MDH. all =senddata[temp+1];
temp=temp+2;
if(temp>8)
    temp=0;
}
}

```

实验 38: CANA 发送 CANB 接收试验(Example_2833x_EcanA_to_B_Xmit)

一、实验目的:

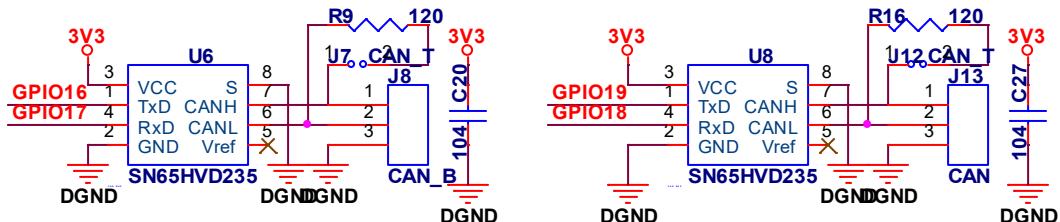
- ✧ 了解如何使用 dsp23885 内部集成两路 eCAN 外设;
- ✧ 通过两路 CAN 通信的实验学习通信机制;

二、实验设备

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, CAN 总线一条;

三、实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:



用我们配套的 CAN 线 (下图中三根并排在一起的线) 将两个通道进行连接;

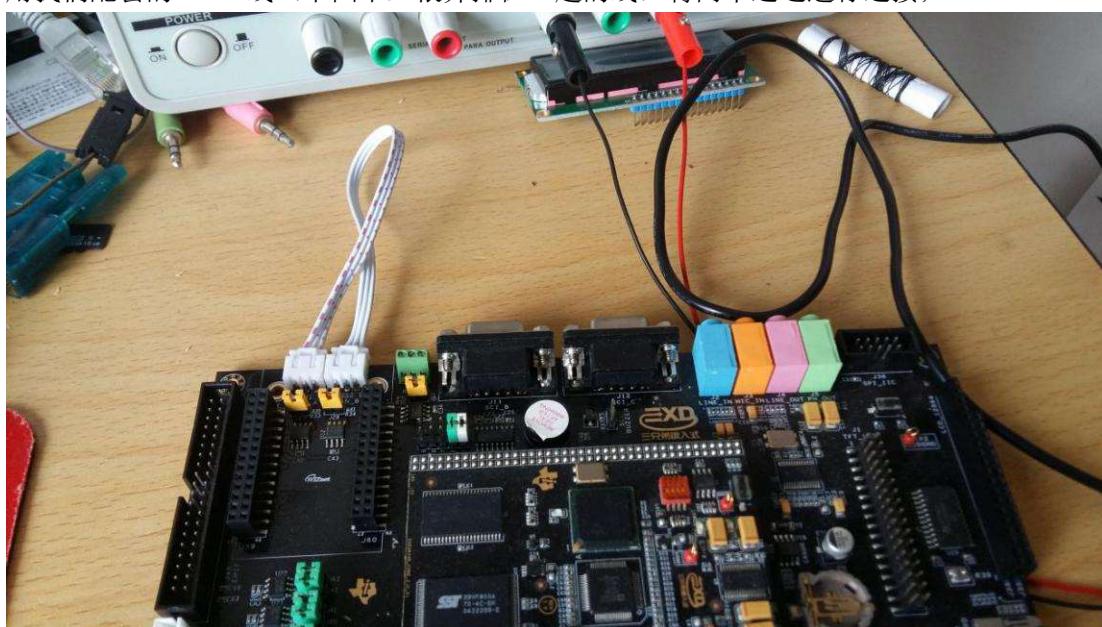


图 1 硬件连接方式

从上图可知, 需要用一个跳线冒将 J7 和 J12 短路。

- ◆ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示: 然后单击图中方框处的调试按钮, 进行调试。

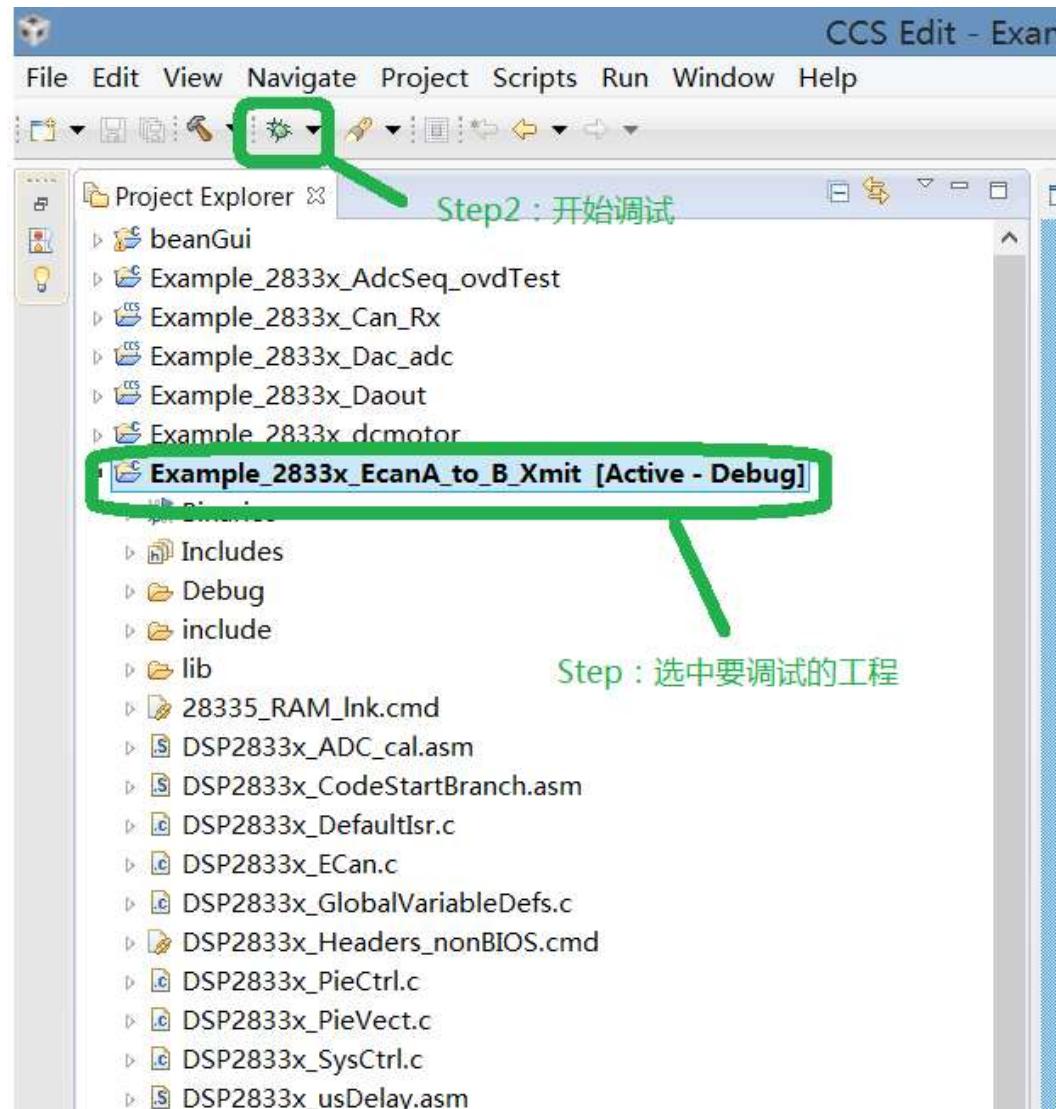


图 2 调试方法

- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

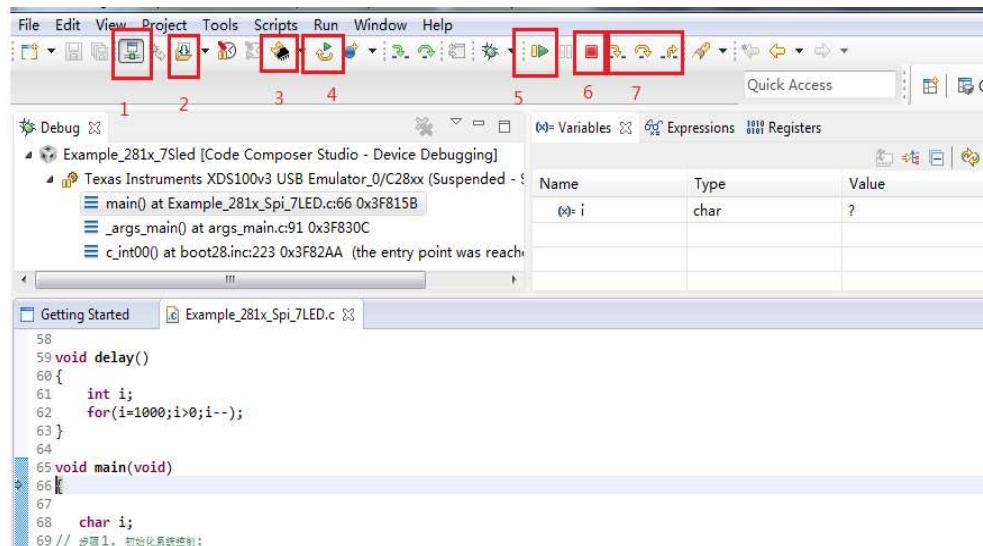


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮;
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t ;
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行;
- ✧ 图中 6 是停止调试;
- ✧ 图中 7 是用于单步调试的;

四、试验现象：

首先将 ECanbMboxes 添加到观察窗口中，如下图所示：

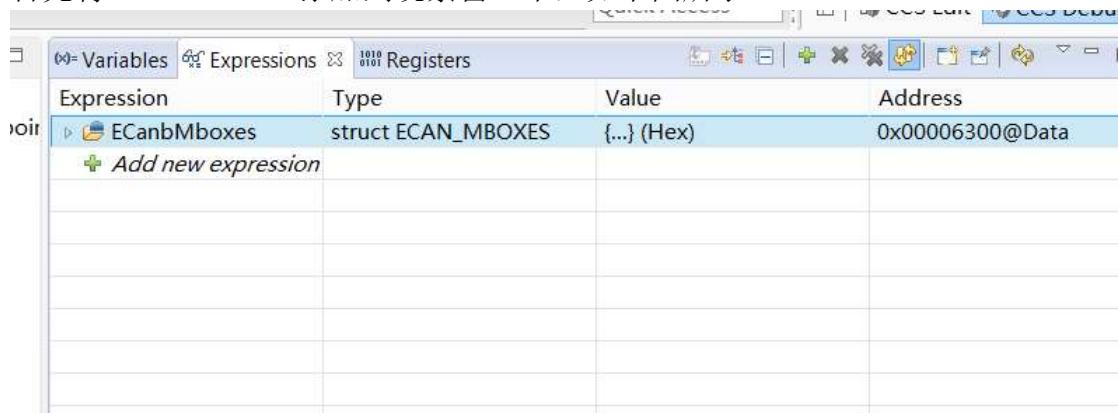


图 4 添加变量到观察窗口中
现象：

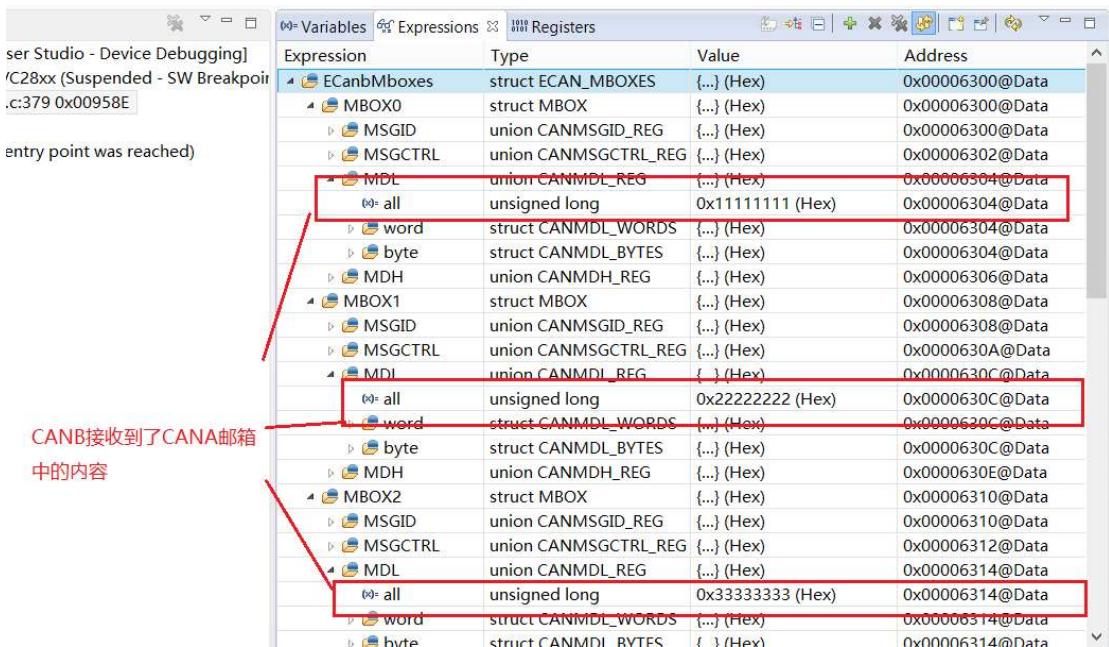


图 5 实验现象

邮箱 ID 相同的 CANA 和 CANB 相互通信。

程序解析

```
void InitECana(void)      // Initialize eCAN-A module
{
/* 创建一个 CAN 控制器的阴影寄存器的结构体 (struct ECAN_REGS ECanaShadow; )，因为这些寄存器只能 32bit 同时获取，如果只获取这 32bit 中的 16bit 可能会返回错误数据。创建一个阴影寄存器后，三兄弟客户们可以通过把要修改的 CAN 控制寄存器的值先复制到阴影寄存器。然后修改阴影寄存器。当修改完成后，可以把阴影寄存器的 32bit 值直接复制给控制寄存器。这就避免了操作部分寄存器导致错误发生的情况*/
struct ECAN_REGS ECanaShadow;
EALLOW;          // EALLOW enables access to protected bits
/* 利用 can 的寄存器将 CAN 的管脚 RX 和 TX 配置为 CAN 的收发功能管脚*/
    ECanaShadow.CANTIOC.all = ECanaRegs.CANTIOC.all;
    ECanaShadow.CANTIOC.bit.TXFUNC = 1;
    ECanaRegs.CANTIOC.all = ECanaShadow.CANTIOC.all;
    ECanaShadow.CANRIOC.all = ECanaRegs.CANRIOC.all;
    ECanaShadow.CANRIOC.bit.RXFUNC = 1;
    ECanaRegs.CANRIOC.all = ECanaShadow.CANRIOC.all;
/* 配置 CAN 为 HECC 模式*/
    ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
    ECanaShadow.CANMC.bit.SCB = 1;
    ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
// 为什么一上来就初始化 MSGCTRL 寄存器全为 0? 很多客户会感觉莫名其妙。其实这也是写程序常用的策略。因为由于 CPU 一上电这些寄存器默认的初值可能比较乱，我们不太好确定具体的值，从而不能知道 CAN 模块具体处于什么状态。为了程序员能够全局掌控 CAN 模块。我们需要配置成我们知道的某一个已知状态后（本程序是全 0）。然后再开始我们自己的配置。
```

```
ECanaMboxes. MBOX0. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX1. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX2. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX3. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX4. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX5. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX6. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX7. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX8. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX9. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX10. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX11. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX12. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX13. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX14. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX15. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX16. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX17. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX18. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX19. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX20. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX21. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX22. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX23. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX24. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX25. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX26. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX27. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX28. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX29. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX30. MSGCTRL. all = 0x00000000;
ECanaMboxes. MBOX31. MSGCTRL. all = 0x00000000;

// TAn, RMPn, GIFn bits are all zero upon reset and are cleared again
// as a matter of precaution.
ECanaRegs. CANTA. all = 0xFFFFFFFF; /* 如果为 1 则表明数据发送成功，我们还没有进入正式通信。所以我们要对其进行清除操作。操作的方式就是对这个寄存器写 1。写 1 到某位则某位将被清除 */
ECanaRegs. CANRMP. all = 0xFFFFFFFF; /* 如果为 1 则表明有数据接收。由于我们刚开始初始化，如果为一则不是我们要的，因为我们还处于初始化状态，还没正式接收数据。所以，不管是否为 1，我们都要把它清除（写 1 到这个寄存器则清除了这个寄存器）。使其为 0。表示没有接收到数据 */
ECanaRegs. CANGIFO. all = 0xFFFFFFFF; /* 清除中断标志位 */
ECanaRegs. CANGIF1. all = 0xFFFFFFFF;
/* 更改配置使能。 该位显示了配置的访问权。 经过一个时钟周期的延迟， 该位被置位。*/
```

```
ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
ECanaShadow.CANMC.bit.CCR = 1; // Set CCR = 1
ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
ECanaShadow.CANES.all = ECanaRegs.CANES.all;
do
{
    ECanaShadow.CANES.all = ECanaRegs.CANES.all;
} while(ECanaShadow.CANES.bit.CCE != 1); // Wait for CCE bit to be
set..
ECanaShadow.CANBTC.all = 0;
#if (CPU_FRQ_150MHZ) // 配置 CAN 的波特率
/* The following block for all 150 MHz SYSCLKOUT (75 MHz CAN clock) -
default. Bit rate = 1 Mbps
See Note at End of File */
ECanaShadow.CANBTC.bit.BRPREG = 4;
ECanaShadow.CANBTC.bit.TSEG2REG = 2;
ECanaShadow.CANBTC.bit.TSEG1REG = 10;
#endif
#if (CPU_FRQ_100MHZ) // CPU_FRQ_100MHz is defined in
DSP2833x_Examples.h
/* The following block is only for 100 MHz SYSCLKOUT (50 MHz CAN clock). Bit
rate = 1 Mbps
See Note at End of File */
ECanaShadow.CANBTC.bit.BRPREG = 4;
ECanaShadow.CANBTC.bit.TSEG2REG = 1;
ECanaShadow.CANBTC.bit.TSEG1REG = 6;
#endif
ECanaShadow.CANBTC.bit.SAM = 1;
ECanaRegs.CANBTC.all = ECanaShadow.CANBTC.all;
ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
ECanaShadow.CANMC.bit.CCR = 0; // Set CCR = 0
ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
ECanaShadow.CANES.all = ECanaRegs.CANES.all;
do
{
    ECanaShadow.CANES.all = ECanaRegs.CANES.all;
} while(ECanaShadow.CANES.bit.CCE != 0); // Wait for CCE bit to be
cleared..
/* Disable all Mailboxes */
ECanaRegs.CANME.all = 0; // Required before writing the MSGIDs
EDIS;
}
```

Main:

```
InitECan();  
// 配置 CAN a 的 ID。在通信过程中，只有 CAN ID 一致的两个邮箱才能相互通信  
ECanaMboxes. MBOX0. MSGID. all = 0x9555AAA0;  
ECanaMboxes. MBOX1. MSGID. all = 0x9555AAA1;  
ECanaMboxes. MBOX2. MSGID. all = 0x9555AAA2;  
ECanaMboxes. MBOX3. MSGID. all = 0x9555AAA3;  
ECanaMboxes. MBOX4. MSGID. all = 0x9555AAA4;  
ECanaMboxes. MBOX5. MSGID. all = 0x9555AAA5;  
ECanaMboxes. MBOX6. MSGID. all = 0x9555AAA6;  
ECanaMboxes. MBOX7. MSGID. all = 0x9555AAA7;  
ECanaMboxes. MBOX8. MSGID. all = 0x9555AAA8;  
ECanaMboxes. MBOX9. MSGID. all = 0x9555AAA9;  
ECanaMboxes. MBOX10. MSGID. all = 0x9555AAAA;  
ECanaMboxes. MBOX11. MSGID. all = 0x9555AAAB;  
ECanaMboxes. MBOX12. MSGID. all = 0x9555AAAC;  
ECanaMboxes. MBOX13. MSGID. all = 0x9555AAAD;  
ECanaMboxes. MBOX14. MSGID. all = 0x9555AAAE;  
ECanaMboxes. MBOX15. MSGID. all = 0x9555AAAF;  
ECanaMboxes. MBOX16. MSGID. all = 0x9555AA10;  
ECanaMboxes. MBOX17. MSGID. all = 0x9555AA11;  
ECanaMboxes. MBOX18. MSGID. all = 0x9555AA12;  
ECanaMboxes. MBOX19. MSGID. all = 0x9555AA13;  
ECanaMboxes. MBOX20. MSGID. all = 0x9555AA14;  
ECanaMboxes. MBOX21. MSGID. all = 0x9555AA15;  
ECanaMboxes. MBOX22. MSGID. all = 0x9555AA16;  
ECanaMboxes. MBOX23. MSGID. all = 0x9555AA17;  
ECanaMboxes. MBOX24. MSGID. all = 0x9555AA18;  
ECanaMboxes. MBOX25. MSGID. all = 0x9555AA19;  
ECanaMboxes. MBOX26. MSGID. all = 0x9555AA1A;  
ECanaMboxes. MBOX27. MSGID. all = 0x9555AA1B;  
ECanaMboxes. MBOX28. MSGID. all = 0x9555AA1C;  
ECanaMboxes. MBOX29. MSGID. all = 0x9555AA1D;  
ECanaMboxes. MBOX30. MSGID. all = 0x9555AA1E;  
ECanaMboxes. MBOX31. MSGID. all = 0x9555AA1F;  
  
// 配置 CAN b 的 ID。在通信过程中，只有 CAN ID 一致的两个邮箱才能相互通信  
ECanbMboxes. MBOX0. MSGID. all = 0x9555AAA0;  
ECanbMboxes. MBOX1. MSGID. all = 0x9555AAA1;  
ECanbMboxes. MBOX2. MSGID. all = 0x9555AAA2;  
ECanbMboxes. MBOX3. MSGID. all = 0x9555AAA3;  
ECanbMboxes. MBOX4. MSGID. all = 0x9555AAA4;  
ECanbMboxes. MBOX5. MSGID. all = 0x9555AAA5;  
ECanbMboxes. MBOX6. MSGID. all = 0x9555AAA6;  
ECanbMboxes. MBOX7. MSGID. all = 0x9555AAA7;  
ECanbMboxes. MBOX8. MSGID. all = 0x9555AAA8;
```

```
ECanbMboxes. MBOX9. MSGID. all = 0x9555AAA9;
ECanbMboxes. MBOX10. MSGID. all = 0x9555AAAA;
ECanbMboxes. MBOX11. MSGID. all = 0x9555AAAB;
ECanbMboxes. MBOX12. MSGID. all = 0x9555AAAC;
ECanbMboxes. MBOX13. MSGID. all = 0x9555AAAD;
ECanbMboxes. MBOX14. MSGID. all = 0x9555AAAE;
ECanbMboxes. MBOX15. MSGID. all = 0x9555AAAF;
ECanbMboxes. MBOX16. MSGID. all = 0x9555AA10;
ECanbMboxes. MBOX17. MSGID. all = 0x9555AA11;
ECanbMboxes. MBOX18. MSGID. all = 0x9555AA12;
ECanbMboxes. MBOX19. MSGID. all = 0x9555AA13;
ECanbMboxes. MBOX20. MSGID. all = 0x9555AA14;
ECanbMboxes. MBOX21. MSGID. all = 0x9555AA15;
ECanbMboxes. MBOX22. MSGID. all = 0x9555AA16;
ECanbMboxes. MBOX23. MSGID. all = 0x9555AA17;
ECanbMboxes. MBOX24. MSGID. all = 0x9555AA18;
ECanbMboxes. MBOX25. MSGID. all = 0x9555AA19;
ECanbMboxes. MBOX26. MSGID. all = 0x9555AA1A;
ECanbMboxes. MBOX27. MSGID. all = 0x9555AA1B;
ECanbMboxes. MBOX28. MSGID. all = 0x9555AA1C;
ECanbMboxes. MBOX29. MSGID. all = 0x9555AA1D;
ECanbMboxes. MBOX30. MSGID. all = 0x9555AA1E;
ECanbMboxes. MBOX31. MSGID. all = 0x9555AA1F;
//CAN B 被配置为接收邮箱
ECanbRegs. CANMD. all = 0xFFFFFFFF;
//CAN a 被配置为接收邮箱
ECanaRegs. CANMD. all = 0x00000000;
//CAN 模块中相应的邮箱被启用
ECanaRegs. CANME. all = 0xFFFFFFFF;
ECanbRegs. CANME. all = 0xFFFFFFFF;
//这些位中的数量决定发送或接收是多少字节的数据
ECanaMboxes. MBOX0. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX1. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX2. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX3. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX4. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX5. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX6. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX7. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX8. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX9. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX10. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX11. MSGCTRL. bit. DLC = 8;
ECanaMboxes. MBOX12. MSGCTRL. bit. DLC = 8;
```

```
ECanaMboxes. MBOX13. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX14. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX15. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX16. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX17. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX18. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX19. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX20. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX21. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX22. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX23. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX24. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX25. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX26. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX27. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX28. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX29. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX30. MSGCTRL.bit.DLC = 8;
ECanaMboxes. MBOX31. MSGCTRL.bit.DLC = 8;

/* Write to DLC field in Master Control reg */

//装入的数据内容，要发送的数据内容

    ECanaMboxes. MBOX0. MDL.all = 0x11111111;
    ECanaMboxes. MBOX0. MDH.all = 0x11111111;
    ECanaMboxes. MBOX1. MDL.all = 0x22222222;
    ECanaMboxes. MBOX1. MDH.all = 0x22222222;
    ECanaMboxes. MBOX2. MDL.all = 0x33333333;
    ECanaMboxes. MBOX2. MDH.all = 0x33333333;
    ECanaMboxes. MBOX3. MDL.all = 0x44444444;
    ECanaMboxes. MBOX3. MDH.all = 0x44444444;
    ECanaMboxes. MBOX4. MDL.all = 0x55555555;
    ECanaMboxes. MBOX4. MDH.all = 0x55555555;
    ECanaMboxes. MBOX5. MDL.all = 0x66666666;
    ECanaMboxes. MBOX5. MDH.all = 0x66666666;
    ECanaMboxes. MBOX6. MDL.all = 0x77777777;
    ECanaMboxes. MBOX6. MDH.all = 0x77777777;
    ECanaMboxes. MBOX7. MDL.all = 0x88888888;
    ECanaMboxes. MBOX7. MDH.all = 0x88888888;
    ECanaMboxes. MBOX8. MDL.all = 0x99999999;
    ECanaMboxes. MBOX8. MDH.all = 0x99999999;
    ECanaMboxes. MBOX9. MDL.all = 0xaaaaaaaaaa;
    ECanaMboxes. MBOX9. MDH.all = 0xaaaaaaaaaa;
    ECanaMboxes. MBOX10. MDL.all = 0xbbbbbbbb;
    ECanaMboxes. MBOX10. MDH.all = 0xbbbbbbbb;
    ECanaMboxes. MBOX11. MDL.all = 0xcccccccc;
```

```
ECanaMboxes. MBOX11. MDH. all = 0xcccccccc;
ECanaMboxes. MBOX12. MDL. all = 0xdddddddd;
ECanaMboxes. MBOX12. MDH. all = 0xdddddddd;
ECanaMboxes. MBOX13. MDL. all = 0xeeeeeeee;
ECanaMboxes. MBOX13. MDH. all = 0xeeeeeeee;
ECanaMboxes. MBOX14. MDL. all = 0xffffffff;
ECanaMboxes. MBOX14. MDH. all = 0xffffffff;
ECanaMboxes. MBOX15. MDL. all = 0x00000000;
ECanaMboxes. MBOX15. MDH. all = 0x00000000;
ECanaMboxes. MBOX16. MDL. all = 0x00000000;
ECanaMboxes. MBOX16. MDH. all = 0x00000000;
ECanaMboxes. MBOX17. MDL. all = 0x11111111;
ECanaMboxes. MBOX17. MDH. all = 0x11111111;
ECanaMboxes. MBOX18. MDL. all = 0x22222222;
ECanaMboxes. MBOX18. MDH. all = 0x22222222;
ECanaMboxes. MBOX19. MDL. all = 0x33333333;
ECanaMboxes. MBOX19. MDH. all = 0x33333333;
ECanaMboxes. MBOX20. MDL. all = 0x44444444;
ECanaMboxes. MBOX20. MDH. all = 0x44444444;
ECanaMboxes. MBOX21. MDL. all = 0x55555555;
ECanaMboxes. MBOX21. MDH. all = 0x55555555;
ECanaMboxes. MBOX22. MDL. all = 0x66666666;
ECanaMboxes. MBOX22. MDH. all = 0x66666666;
ECanaMboxes. MBOX23. MDL. all = 0x77777777;
ECanaMboxes. MBOX23. MDH. all = 0x77777777;
ECanaMboxes. MBOX24. MDL. all = 0x88888888;
ECanaMboxes. MBOX24. MDH. all = 0x88888888;
ECanaMboxes. MBOX25. MDL. all = 0x99999999;
ECanaMboxes. MBOX25. MDH. all = 0x99999999;
ECanaMboxes. MBOX26. MDL. all = 0xaaaaaaaaaa;
ECanaMboxes. MBOX26. MDH. all = 0xaaaaaaaaaa;
ECanaMboxes. MBOX27. MDL. all = 0xbbbbbbbb;
ECanaMboxes. MBOX27. MDH. all = 0xbbbbbbbb;
ECanaMboxes. MBOX28. MDL. all = 0xcccccccc;
ECanaMboxes. MBOX28. MDH. all = 0xcccccccc;
ECanaMboxes. MBOX29. MDL. all = 0xdddddddd;
ECanaMboxes. MBOX29. MDH. all = 0xdddddddd;
ECanaMboxes. MBOX30. MDL. all = 0xeeeeeeee;
ECanaMboxes. MBOX30. MDH. all = 0xeeeeeeee;
ECanaMboxes. MBOX31. MDL. all = 0xffffffff;
ECanaMboxes. MBOX31. MDH. all = 0xffffffff;
/* Write to the mailbox RAM field */
EALLOW;
ECanaRegs. CANMIM. all = 0xFFFFFFFF; //enable interrupt of mailbox
```

```
/* 开始发送 */
// 配置为正常工作模式, 而非自测试(自循环模式)
EALLOW;
ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
ECanaShadow.CANMC.bit.STM = 0;
ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
EDIS;
EALLOW;
ECanbShadow.CANMC.all = ECanbRegs.CANMC.all;
ECanbShadow.CANMC.bit.STM = 0;
ECanbRegs.CANMC.all = ECanbShadow.CANMC.all;
EDIS;
for(i=0; i < TXCOUNT; i++)
{
    ECanaRegs.CANTRS.all = 0xFFFFFFFF; // 发送请求-设置位
while(ECanaRegs.CANTA.all != 0xFFFFFFFF) {} // 所有发送邮箱的消息都被成功发送
    ECanaRegs.CANTA.all = 0xFFFFFFFF; // 清除发送标志位
loopcount++;
}
asm("ESTOP0"); // Stop here
}
```

实验 39: eCAN 自测试模式 (Example_2833x_ecan_back2back)

一 实验目的:

- ✧ 了解如何使用 dsp28335 内部集成的 CAN 外设的使用;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;

三 实验步骤:

ecan_back2back 程序示例 eCAN 模块自测试模式, 发送数据并进行高速回环, 校验接收数据, 并对任何错误进行标识, MBX0 传送到

MBX16, MBX1 传送到 MBX17, 以此类推, 为了测试程序方便, 测试过程需要进行变量 PassCount、ErrorCount、MessageReceivedCount 观察。

- ◆ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ◆ 给开发板上电。单击鼠标左键选择要调试的工程, 如下图所示:

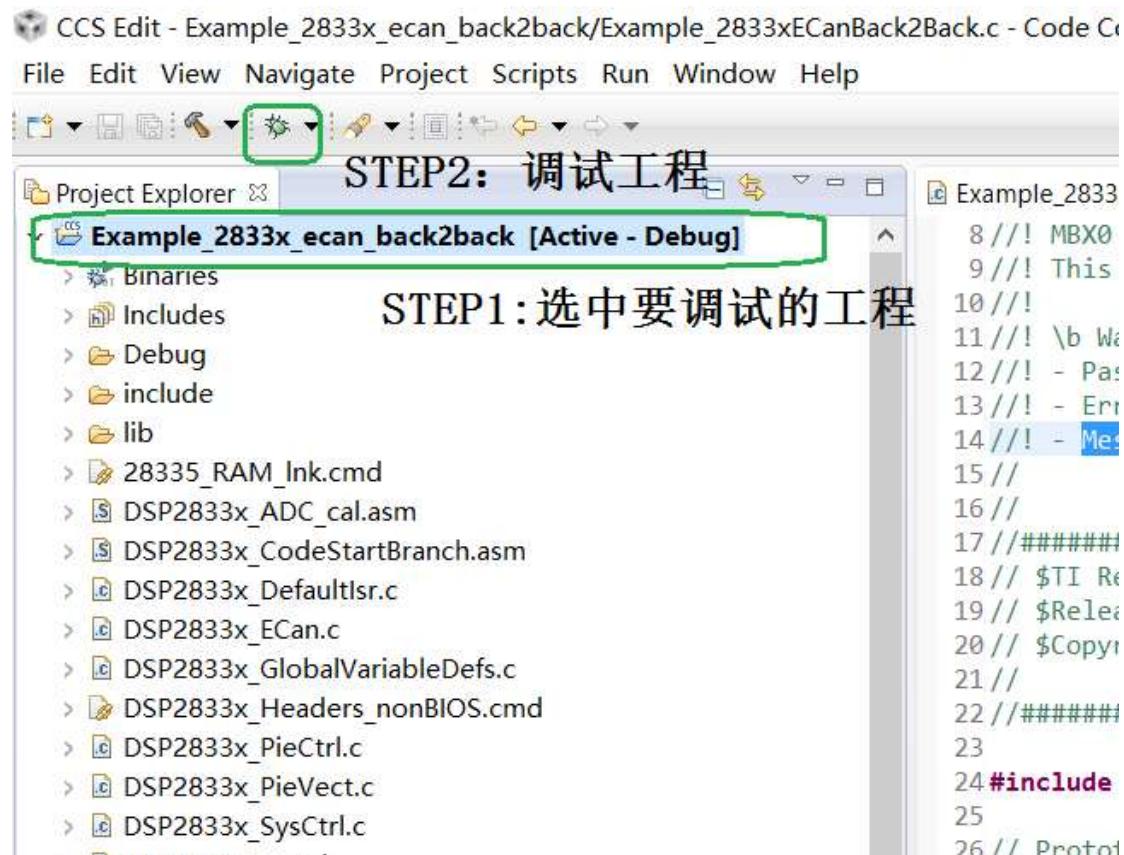


图 3 调试过程

然后单击图中红色的方框处的调试按钮, 进行调试。

- ◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

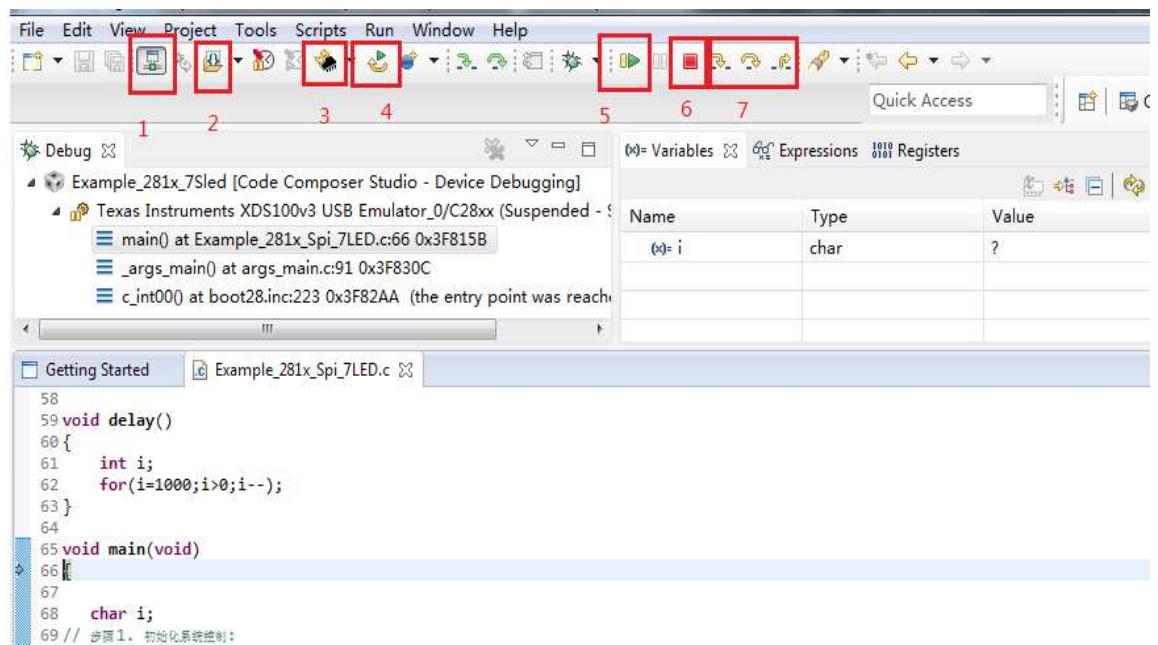
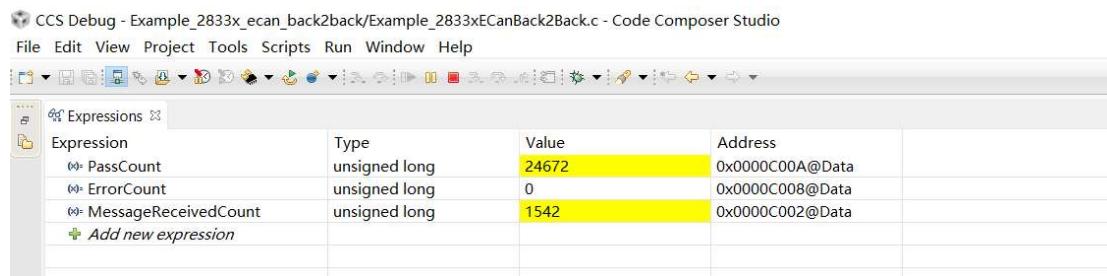


图 4 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；

试验现象：



实验 40：直流电机实验 (Example_Sxd28335_Dcmotor_Ch452)

一 实验目的：

✧ 了解如何使用 dsp28335 内部集成的 PWM 外设的使用；

二 实验设备：

✧ PC 机一台；

✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；

✧ SXD28335 开发板一套，直流电机一个，按键显示模块 CH452 一块；

三 实验步骤：

✧ 首先将 CCS6.0 开发环境打开；看一下如下原理图：

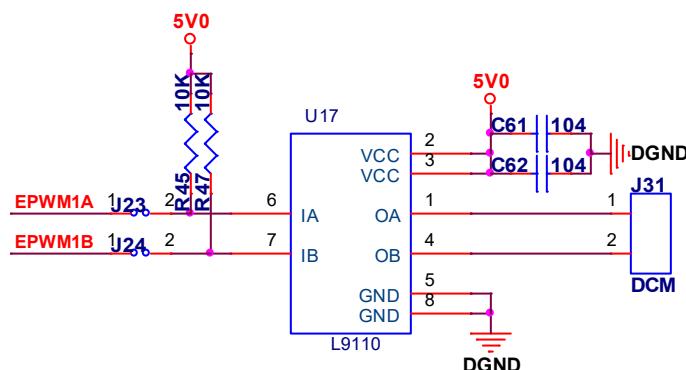


图 1 电机驱动芯片

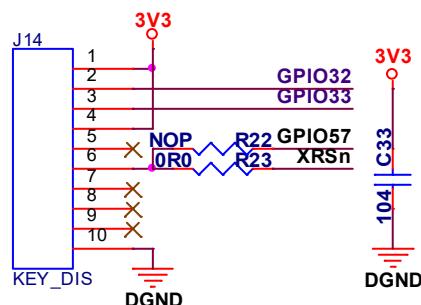


图 2 按键控制电路

本试验是通过开发板上三个按键 SW2、SW6 和 SW10 控制电机加速、减速和换向。按键阵的 SW5(加速)、SW4(减速)、SW3(换向+停止)来控制电机的运行（注意：电机的加减速是分段的。每按一次

会加或减一定的速度。直到设置的最大或最小值) ;

◆ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

◆ 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

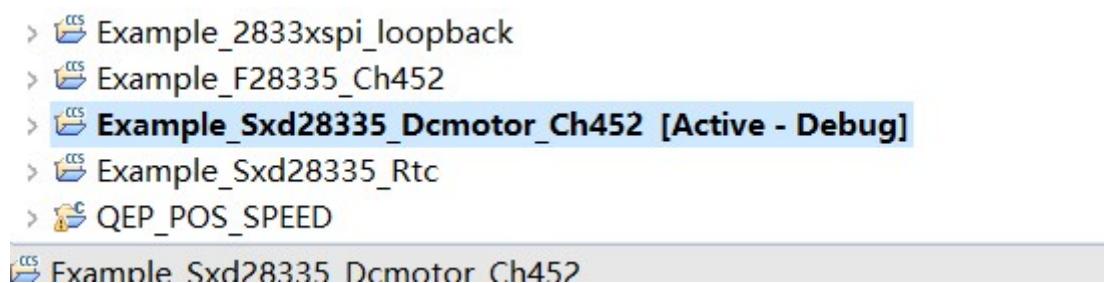


图 3 调试过程

然后单击图中红色的方框处的调试按钮，进行调试。

◆ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

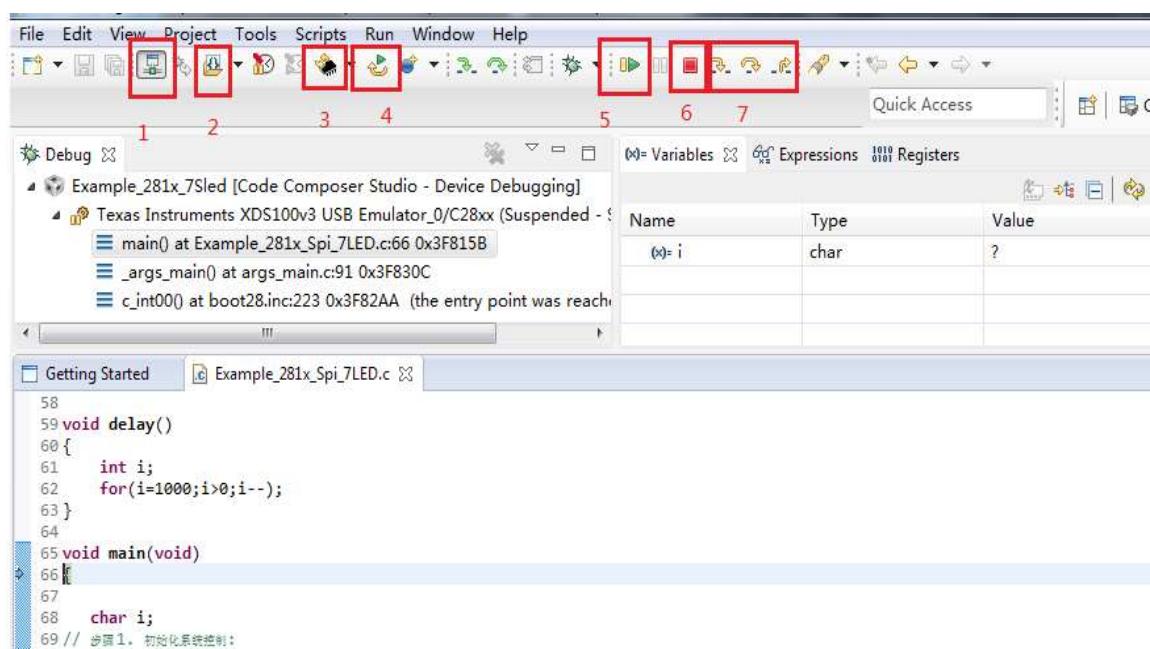


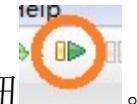
图 4 调试界面

◆ 图中 1 图标是用来进行与开发板进行连接的按钮；

◆ 图中 2 是用来下载 Debug 文件下的.out 文件的

◆ 图中 3 是 CPU 软 Reset；

- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；



我们将电机的插座插到 J31 上，然后单击全速运行按钮。

试验现象：

- ✧ 一开始电机处于静止状态，这时我们通过 SW6 (加速——多按几次) 让电机旋转起来，通过 SW10 (减速-多按几次) 对电机进行减速，SW2 (换向-按一次即可) 电机停止（这时如果按 SW6 电机将向相反的方向进行旋转）。

至于 PWM 的详细工作原理需要用户参考其数据手册和相关实验自己学习和体会。

```
/*
 * Pwm_DC_motor.c
 *
 * Created on: 2014 年 3 月 11 日
 * Author: lzjsxd
 */

#include "DSP2833x_Device.h"      // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"    // DSP2833x Examples Include File

static void InitEPwm1Example(void);
interrupt void epwm1_isr(void);

void Manage_Up(void);
void Manage_Down(void);
static void Motor_Drive(void);

char flags=1;
char Youjian=0;
*****全局变量定义*****
```

```
Uint16 temp=0; //高电平时间
Uint16 Direction=0;//转速方向

// 宏定义每个定时器周期寄存器的周期值;
#define EPWM1_TIMER_TBPRD 3750 // 周期值
#define EPWM1_MAX_CMPA      3700
#define EPWM1_MIN_CMPA      0
#define EPWM1_MAX_CMPB      3700
#define EPWM1_MIN_CMPB      0

/****************端口宏定义*******/
#define Up_Key GpioDataRegs.GPBDAT.bit.GPIO51
#define Down_Key GpioDataRegs.GPBDAT.bit.GPIO52
#define Direction_Key GpioDataRegs.GPBDAT.bit.GPIO53
extern void Read_key(void);
extern void deal_key(void);
extern void Test_CH452();
extern unsigned int BYTE_WR(unsigned int BYTE_ADDR,unsigned int
T_ADDR,unsigned int T_DATA);
extern unsigned int BYTE_RD(unsigned int BYTE_ADDR,unsigned int T_ADDR);
extern void Init_ch452(void);
char flag;

void InitEPwm1Example()
{
    // 设置时间基准的时钟信号 (TBCLK)
    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; // 递增计数模式
    EPwm1Regs.TBPRD = EPWM1_TIMER_TBPRD; // 设置定时器周期
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE; // 禁止相位加载
    EPwm1Regs.TBPHS.half.TBPHS = 0x0000; // 时基相位寄存器的值赋值 0
    EPwm1Regs.TBCTR = 0x0000; // 时基计数器清零
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV2; // 设置时基时钟速率为系统时钟
    SYSCLKOUT/4=37.5MHZ;
    EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV2;//由时基时钟频率和时基周期可知 PWM1 频率=10KHZ;

    // 设置比较寄存器的阴影寄存器加载条件: 时基计数到 0
    EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
```

```
EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;

// 设置比较寄存器的值
EPwm1Regs.CMPA.half.CMPA = EPWM1_MIN_CMPA;           // 设置比较寄存器 A 的值
EPwm1Regs.CMPB = EPWM1_MIN_CMPB;                      // 设置比较寄存器 B 的值

// 设置动作限定；首先默认为转动方向为正转，这时只有 PWM1A 输出占空比；
EPwm1Regs.AQCTLA.bit.ZRO = AQ_SET;                   // 计数到 0 时 PWM1A 输出高电平
EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;                 // 递增计数时，发生比较寄存器 A 匹配时清除 PWM1A 输出

EPwm1Regs.AQCTLB.bit.ZRO = AQ_CLEAR;                 // 计数到 0 时 PWM1B 输出低电平
EPwm1Regs.AQCTLB.bit.CBU = AQ_CLEAR;                 // 递增计数时，发生比较寄存器 A 匹配时清除 PWM1B 输出

// 3 次 0 匹配事件发生时产生一个中断请求；一次匹配是 100us，一共 300us 产生一次中断；
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO;           // 选择 0 匹配事件中断
EPwm1Regs.ETSEL.bit.INTEN = 1;                        // 使能事件触发中断
EPwm1Regs.ETPS.bit.INTPRD = ET_3RD;                  // 3 次事件产生中断请求

}

/*-----
/*形式参数: void                                     */
/*返回值:void                                       */
/*函数描述:按键的处理程序                           */
/*-----*/
void Manage_Up(void) //加
{
    if(temp!=3500)
        temp=temp+500;
}

void Manage_Down(void)//减
{
    if(temp!=0)
        temp=temp-500;
```

```
}

/*-----*/
/*形式参数: void */
/*返回值:void */
/*函数描述:电机驱动程序 */
/*-----*/
void Motor_Drive(void)
{
    EPwm1Regs.CMPA.half.CMPA = temp;//改变脉宽
    EPwm1Regs.CMPB = temp;//改变脉宽
}

void DC_dianji_test(void)
{
    Init_ch452();
    InitEPwm1Gpio();

    EALLOW; // This is needed to write to EALLOW protected registers
    PieVectTable.EPWM1_INT = &epwm1_isr;
    EDIS; // This is needed to disable write to EALLOW protected
registers

    // Step 4. Initialize all the Device Peripherals:
    // This function is found in DSP2833x_InitPeripherals.c
    // InitPeripherals(); // Not required for this example

    // For this example, only initialize the ePWM

    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;
    EDIS;

    InitEPwm1Example();

    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
    EDIS;

    IER |= M_INT3;
    // Enable EPWM INTn in the PIE: Group 3 interrupt 1-3
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    while(flags)
```

```
{  
    if(Youjian==1)  
    {  
        Youjian=0;  
        deal_key();  
        // 更新 CMPA 和 CMPB 比较寄存器的值  
        Motor_Drive();  
    }  
}  
  
IER = 0;  
  
}  
  
interrupt void epwm1_isr(void)  
{  
  
    Read_key();  
  
    // 清除这个定时器的中断标志位  
    EPwm1Regs.ETCLR.bit.INT = 1;  
  
    // 清除 PIE 应答寄存器的第三位, 以响应组 3 内的其他中断请求;  
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  
}
```

实验 41：PWM 计数方式之 UpDown 模式实验（Example_2833xEPwmUpDownAQ）

一 实验目的：

- ✧ 了解 28335 的 pwm 外设模块的使用方法；
- ✧ 了解 PWM 定时器几种计数方式；

二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套，示波器一台，或观察 LD1 这个 Led 灯的亮

度变化；

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；



图 1 PWM1A 上链接有 LD1

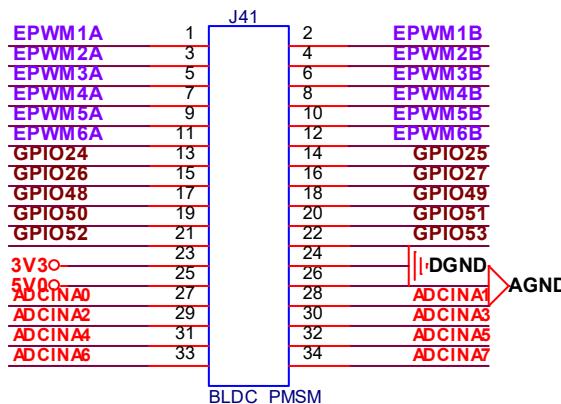


图 2 PWM 外扩接口

本实验是 PWM 通用定时器以上/下计数的方式计数。触发 PWM 中断。在中断函数中改变比较值的大小，从而改变占空比。您可以通过示波器看波形，占空比从设定的最小值增到最大值。然后再从最大值减小到最小值。也可以通过观察 LD1 这个灯。会发现这个灯由暗逐渐变亮。然后又从亮逐渐变暗的过程。

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 在上电前先链接好示波器与开发板。最好不要带电插拔示波器探针。如果是隔离的示波器可以带电插拔示波器探针。给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调试；
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

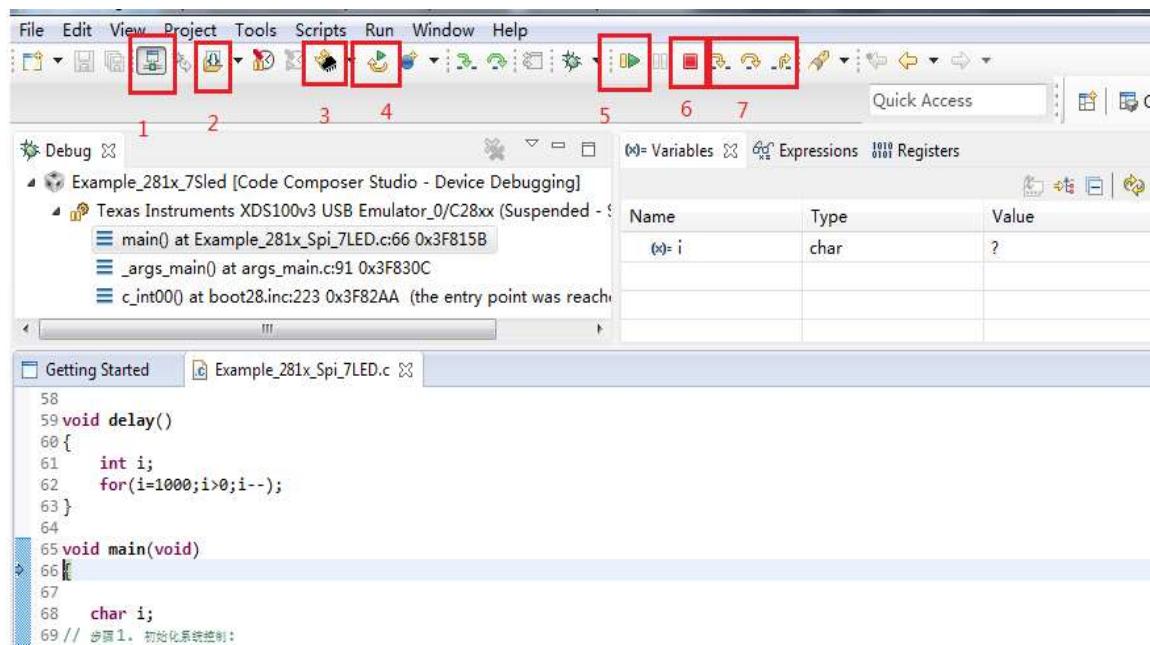


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；

单击全速运行按钮 。

试验现象：

用示波器测量 PWM1A\PWM1B、PWM2A\PWM2B 或 PWM3A\PWM3B。可以发现波形的占空比在设定的最大值和最小值之间来回变换。

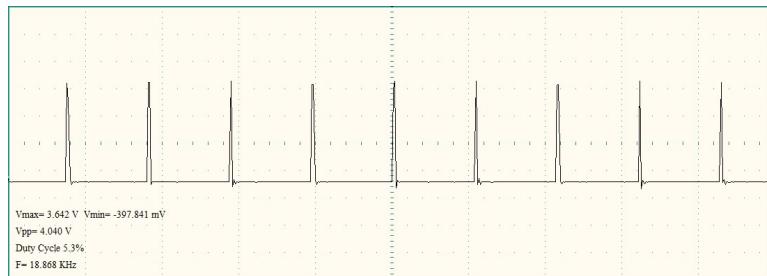


图 4 占空比最小时

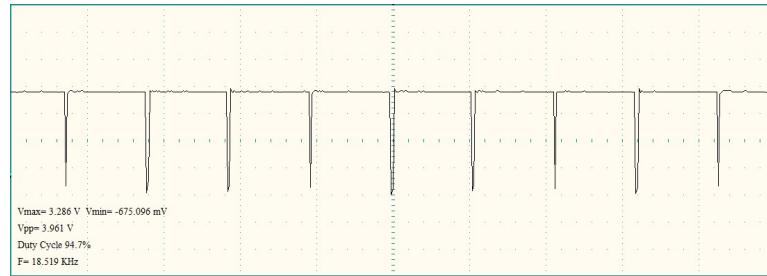
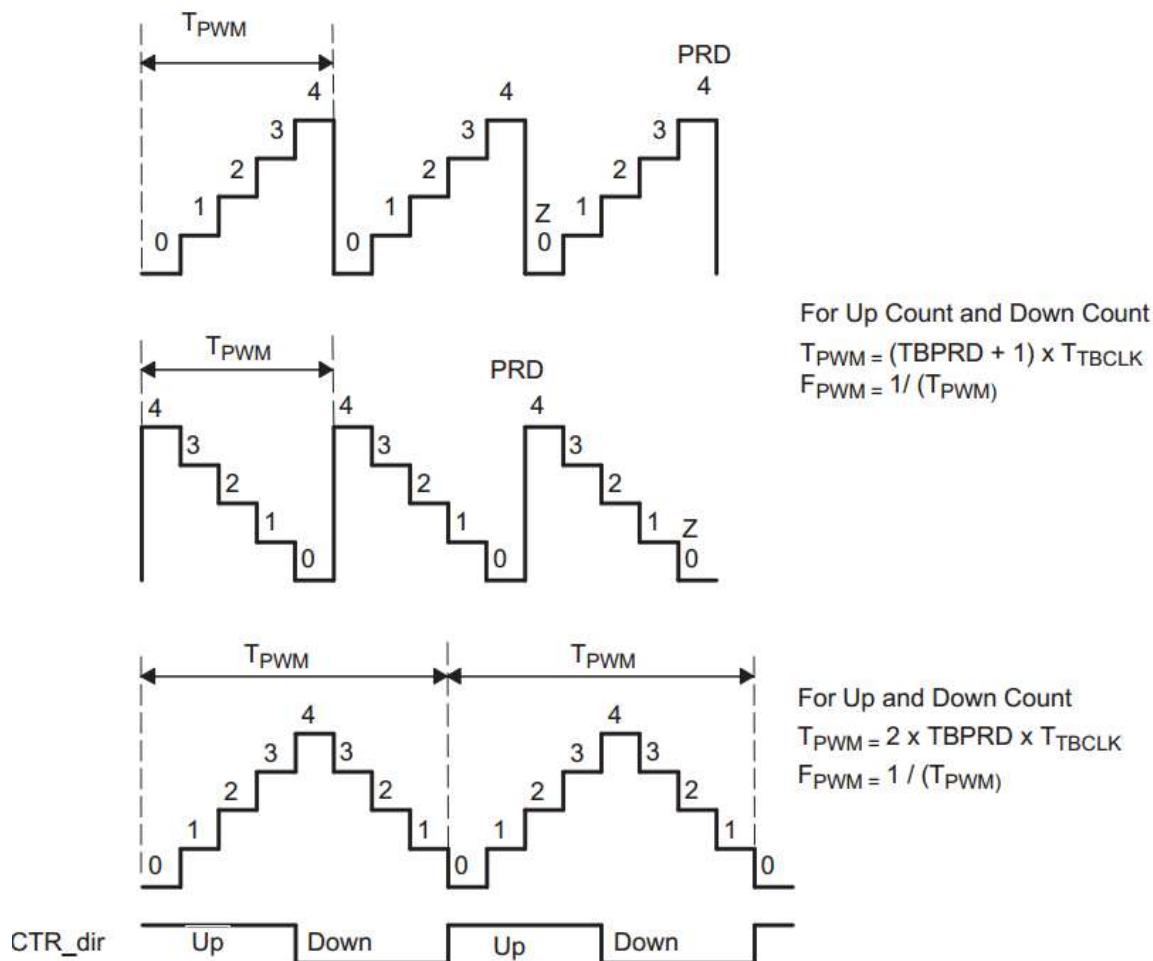


图 4 占空比最大时

也可以观察 LD1 的亮暗程度变化。

原理解读：

下图是 PWM 的几种计数方式，分别为向上计数、向下计数和上下计数。本程序就是使用第三种计数方式。



请参考 Example_2833xEpwmUpAQ 的实验手册。只是计数方式不同而已。

实验 42: PWM 计数方式之 Up 计数实验 (Example_2833xEpwmUpAQ)

一 实验目的:

- ✧ 了解 28335 的 pwm 外设模块的使用方法;
- ✧ 了解 PWM 定时器几种计数方式;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套;

- ✧ SXD28335 开发板一套，示波器一台，或观察 LD1 这个 Led 灯的亮度变化；

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；



图 1 PWM1A 上链接有 LD1

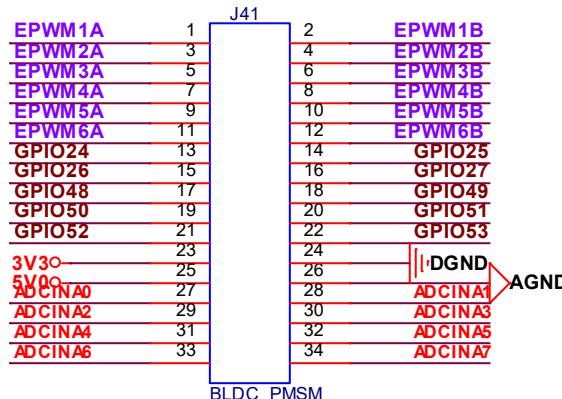


图 2 PWM 外扩接口

本实验是 PWM 通用定时器以向上计数的方式计数。触发 PWM 中断。在中断函数中改变比较值的大小，从而改变占空比。您可以通过示波器看波形，占空比从设定的最小值增到最大值。然后再从最大值减小到最小值。也可以通过观察 LD1 这个灯。会发现这个灯由暗逐渐变亮。然后又从亮逐渐变暗的过程。

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 在上电前先链接好示波器与开发板。最好不要带电插拔示波器探针。如果是隔离的示波器可以带电插拔示波器探针。给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调试；
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的

界面。

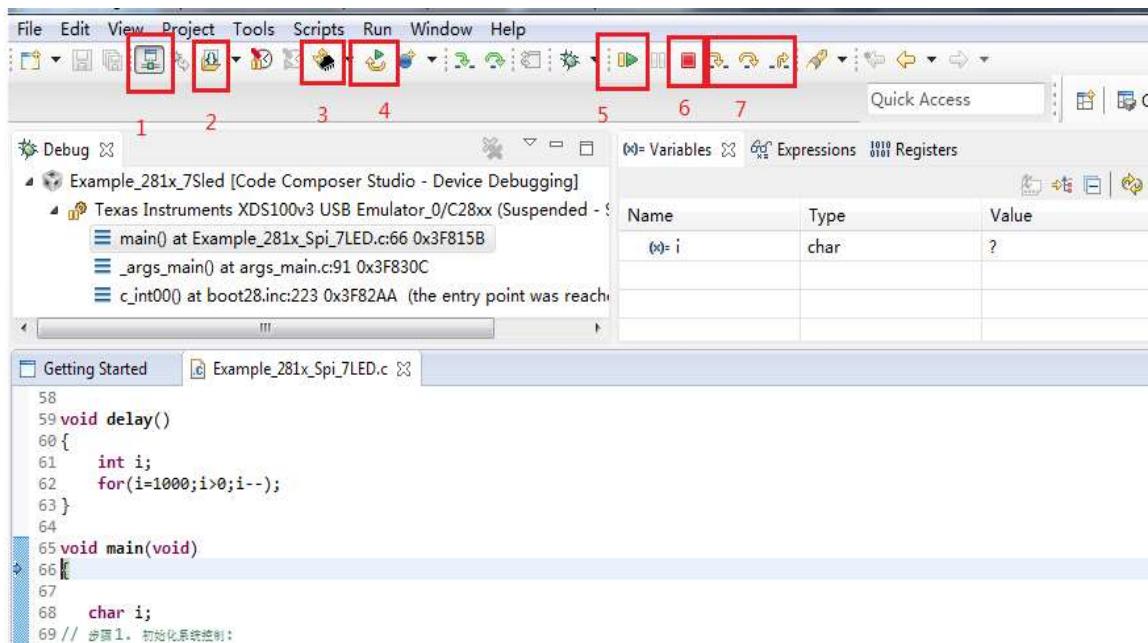
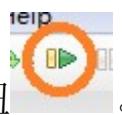


图3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；

单击全速运行按钮 。

试验现象：

用示波器测量 PWM1A\PWM1B、PWM2A\PWM2B 或 PWM3A\PWM3B。可以发现波形的占空比在设定的最大值和最小值之间来回变换。

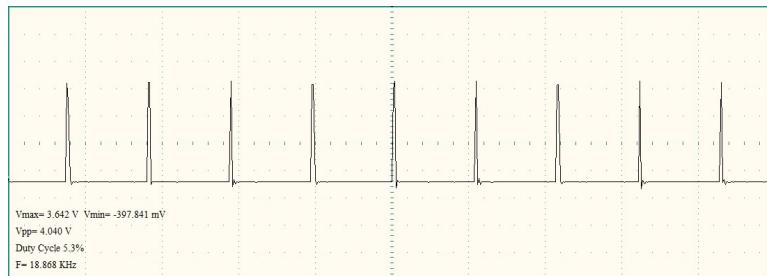


图 4 占空比最小时

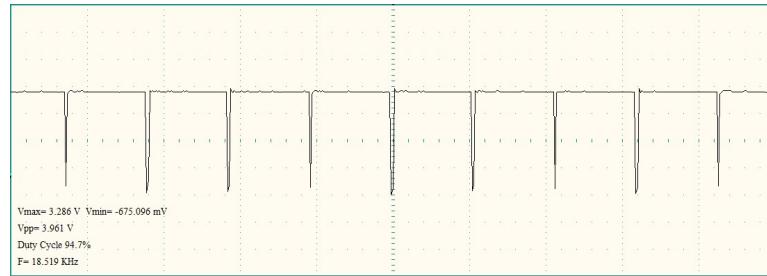
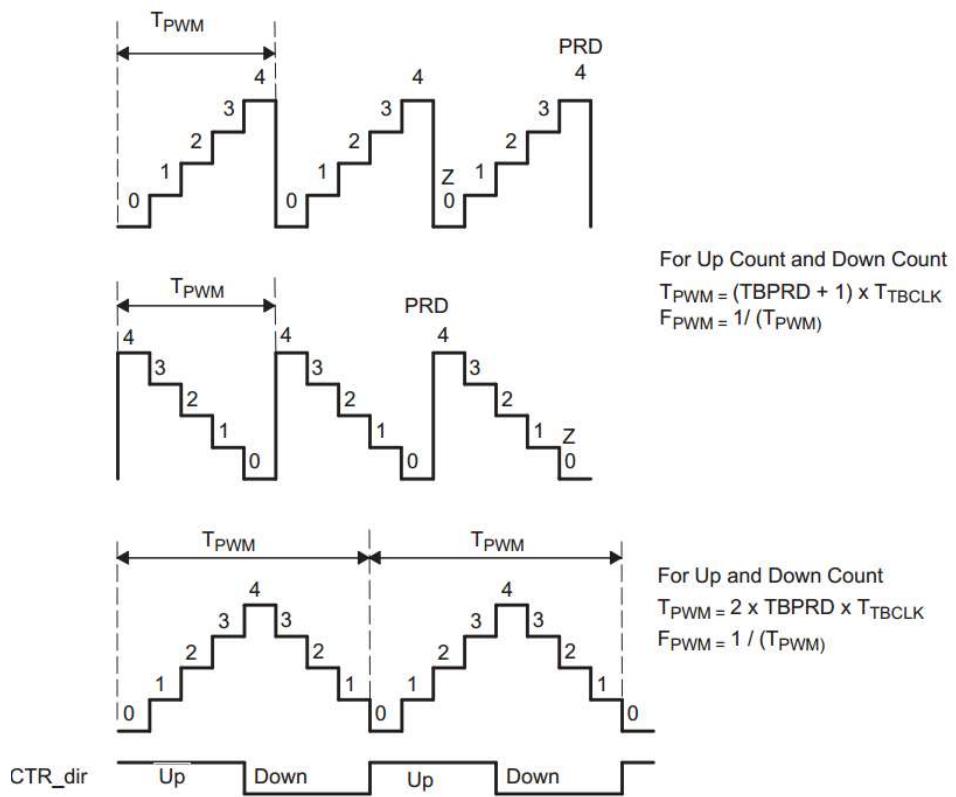


图 4 占空比最大时

也可以观察 LD1 的亮暗程度变化。

原理解读：

下图是 PWM 的几种计数方式，分别为向上计数、向下计数和上下计数。本程序就是使用第一种计数方式。



这些 PWM 的工作方式都差不多，接下来以 PWM1 为例介绍一下程序的实现过程。

```
#define EPWM1_TIMER_TBPRD 2000 // 周期寄存器，决定了 PWM 波形的频率
#define EPWM1_MAX_CMPA 1950 // PWM1A 的最大比较值
#define EPWM1_MIN_CMPA 50 // PWM1A 的最小比较值，最大和最小值影响着占空比的最大和最小值，可以改变这两个比较值改变站空比
#define EPWM1_MAX_CMPB 1950
#define EPWM1_MIN_CMPB 50
// PWM1B 的最小比较值，最大和最小值影响着占空比的最大和最小值，可以改变这两个比较值改变站空比
EALLOW; // 将 PWM 中断函数映射到中断向量表中
PieVectTable.EPWM1_INT = &epwm1_isr;
PieVectTable.EPWM2_INT = &epwm2_isr;
PieVectTable.EPWM3_INT = &epwm3_isr;
EDIS;
EALLOW; // 对 PWM 的工作模式进行配置，请您自己对照一下 PWM 手册看一下配置过程
SysCtrlRegs.PCLKCRO.bit.TBCLKSYNC = 0;
EDIS;
InitEPwm1Example();
InitEPwm2Example();
InitEPwm3Example();
EALLOW;
SysCtrlRegs.PCLKCRO.bit.TBCLKSYNC = 1;
EDIS;
// 使能 PWM 中断对应的 CPU 级中断
IER |= M_INT3;
// 使能 PIE 级中断
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
PieCtrlRegs.PIEIER3.bit.INTx2 = 1;
PieCtrlRegs.PIEIER3.bit.INTx3 = 1;
//PIE 中断函数
_interrupt void epwm1_isr(void)
{
    // 通过更改 CMP 来更改 PWM 的占空比
    update_compare(&epwm1_info);
    // Clear INT flag for this timer
    EPwm1Regs.ETCLR.bit.INT = 1;
    // Acknowledge this interrupt to receive more interrupts from group 3
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

实验 43：PWM 用于定时功能实验 (Example_2833xEPwmTimerInt)

一 实验目的：

- ✧ 了解 28335 的 pwm 外设模块的使用方法；
- ✧ 了解 PWM 定时器工作原理；

二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套；

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；此程序不需要其它硬件，只需要将下面的变量添加到观察窗口即可：

```
EPwm1TimerIntCount  
EPwm2TimerIntCount  
EPwm3TimerIntCount  
EPwm4TimerIntCount  
EPwm5TimerIntCount  
EPwm6TimerIntCount
```

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调试；
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

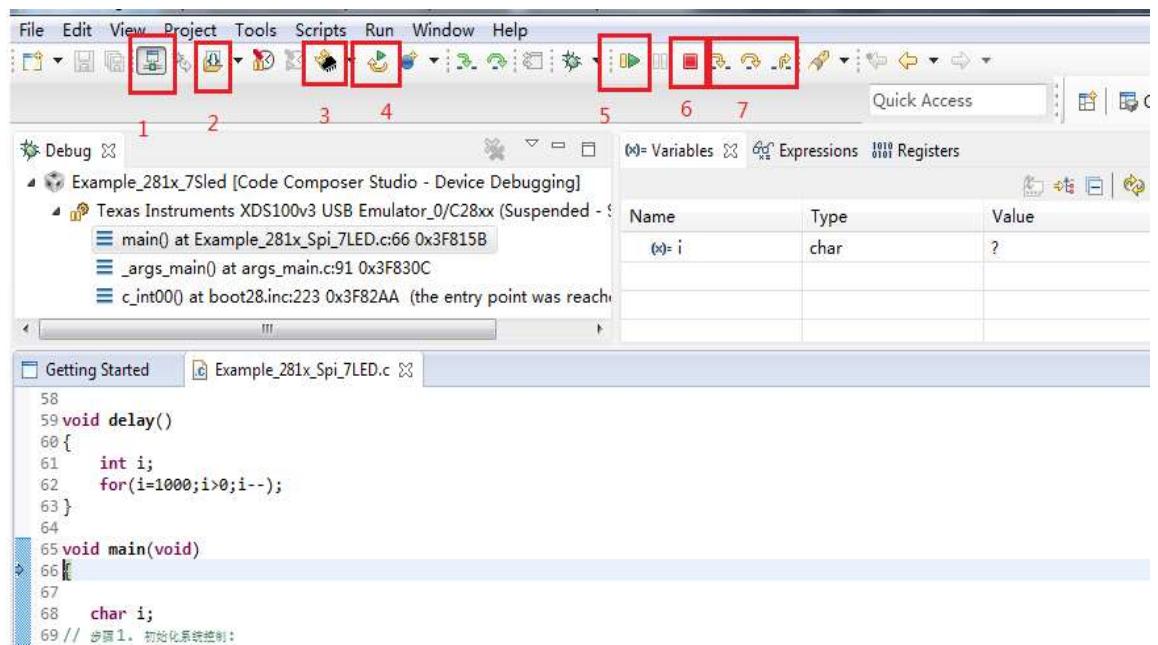


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t ；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；



单击全速运行按钮。

试验现象：

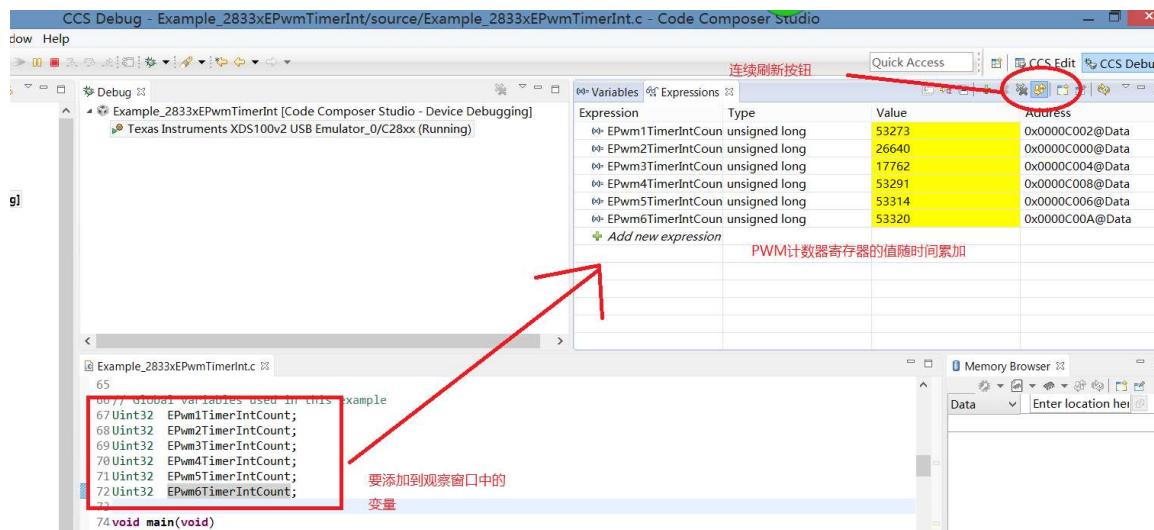


图 4 实验结果

现在是不是已经蒙圈了，28335 已经有通用定时器了，怎么 PWM 模块又有定时器。其实很好理解，就是通用定时器为别的功能计数。PWM 这块用 PWM 定时器提供时间基准了。

Figure 4. Time-Base Submodule Block Diagram

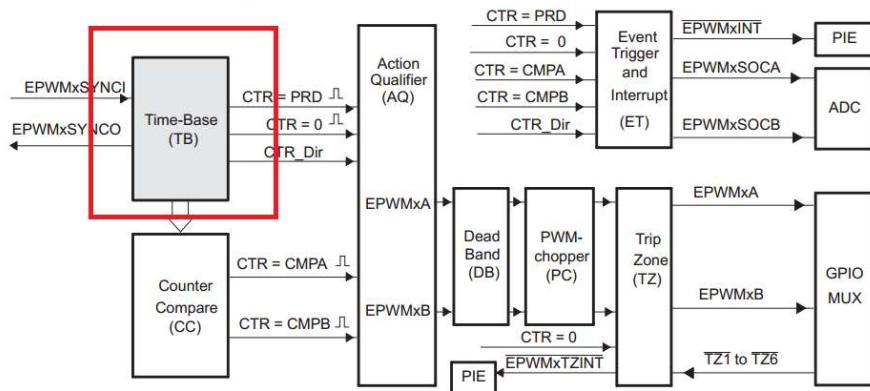


图 5 时间基准在 PWM 模块中的地位

上图是 PWM 定时器在 PWM 模块中的地位。那么它的内部具体结构如下图所示：

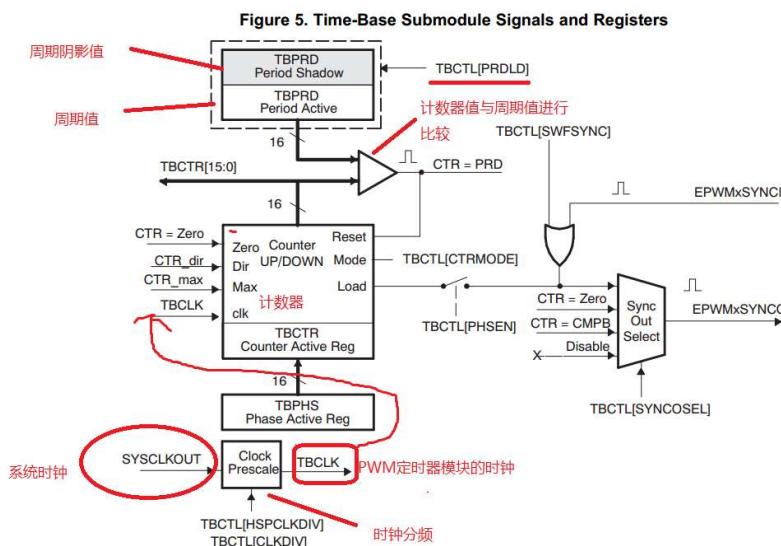


图 6 时间基准的内部具体实现

从上图可以看出 SYSCLKOUT 经过分频后得到 TBCLK (pwm 定时器时钟)。如果我们要使 PWM 计数器计数变慢，我们可以调整图中的 TBCTL 的 HSPCLKDIV 和 CLKDIV 来实现。

下面这个函数是将 pwm 定时器中断映射到中断向量表中。这样当发生中断则会跳到 PIE 中断向量表中找到相应的中断函数去执行。

```
EALLOW; // This is needed to write to EALLOW protected registers
PieVectTable.EPWM1_INT = &epwm1_timer_isr;
PieVectTable.EPWM2_INT = &epwm2_timer_isr;
PieVectTable.EPWM3_INT = &epwm3_timer_isr;
PieVectTable.EPWM4_INT = &epwm4_timer_isr;
PieVectTable.EPWM5_INT = &epwm5_timer_isr;
PieVectTable.EPWM6_INT = &epwm6_timer_isr;
EDIS; // This is needed to disable write to EALLOW protected registers
//这是使能 PIE 级的中断
PieCtrlRegs.PIEIER3.bit.INTx1 = PWM1_INT_ENABLE;
PieCtrlRegs.PIEIER3.bit.INTx2 = PWM2_INT_ENABLE;
PieCtrlRegs.PIEIER3.bit.INTx3 = PWM3_INT_ENABLE;
PieCtrlRegs.PIEIER3.bit.INTx4 = PWM4_INT_ENABLE;
PieCtrlRegs.PIEIER3.bit.INTx5 = PWM5_INT_ENABLE;
PieCtrlRegs.PIEIER3.bit.INTx6 = PWM6_INT_ENABLE;
```

这是汇编语言实现的周期延时函数

```
_asm("    NOP");
EPwm1Regs.TBPRD = PWM1_TIMER_TBPRD; // 装入定时器的周期值
EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; // 向上计数
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO; // 中断条件
EPwm1Regs.ETSEL.bit.INTEN = PWM1_INT_ENABLE; //使能中断
EPwm1Regs.ETPS.bit.INTPRD = ET_1ST; // 每次都产生中断
// 中断函数
_interrupt void epwm1_timer_isr(void)
{
    EPwm1TimerIntCount++;
    // 清除中断标志
    EPwm1Regs.ETCLR.bit.INT = 1;
    // Acknowledge this interrupt to receive more interrupts from group 3
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

实验 44：带死区的 PWM 实验 (Example_2833xEpwmDeadBand)

一 实验目的：

- ✧ 了解 28335 的 pwm 外设模块的使用方法；
- ✧ 了解死区产生原理；

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套;
- ✧ 示波器或是逻辑分析仪一台;

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

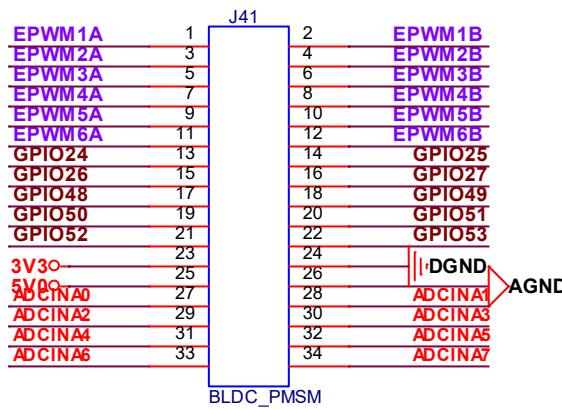


图 1 PWM 外扩接口图

本程序输出 3 对互补同时带有死区的 PWM 波; 其中 PWM1A 和 PWM1B 是一对、PWM2A 和 PWM2B 是一对、PWM3A 和 PWM3B 是一对。在用示波器观察时请成对观察, 才能体会死区的存在。

注: 测试时要注意不要短路。如果短路则可能将开发板烧毁;

- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程, 并进行在线调试:

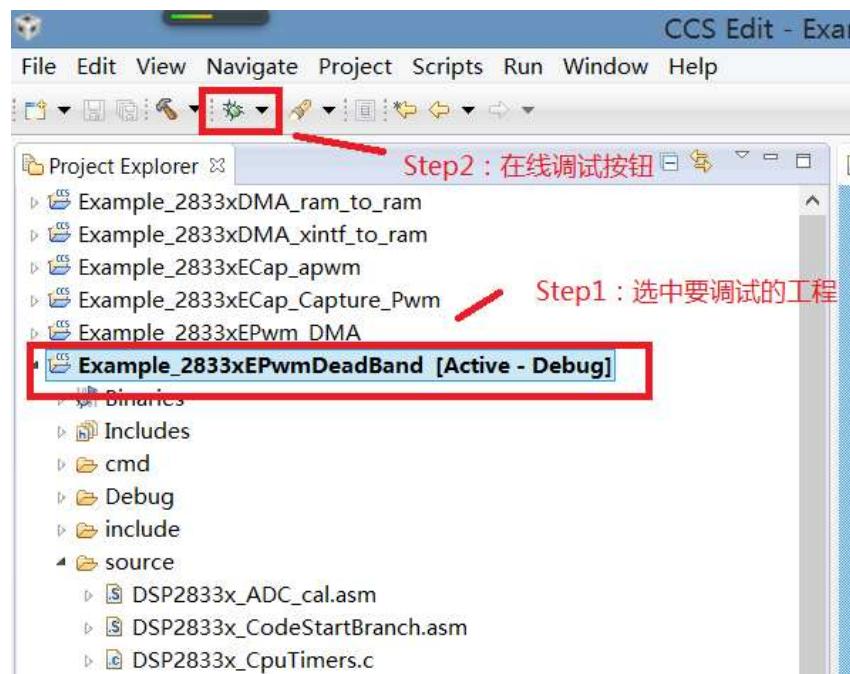


图 2 调试过程

然后单击图中绿色的方框处的调试按钮，进行调试。

- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

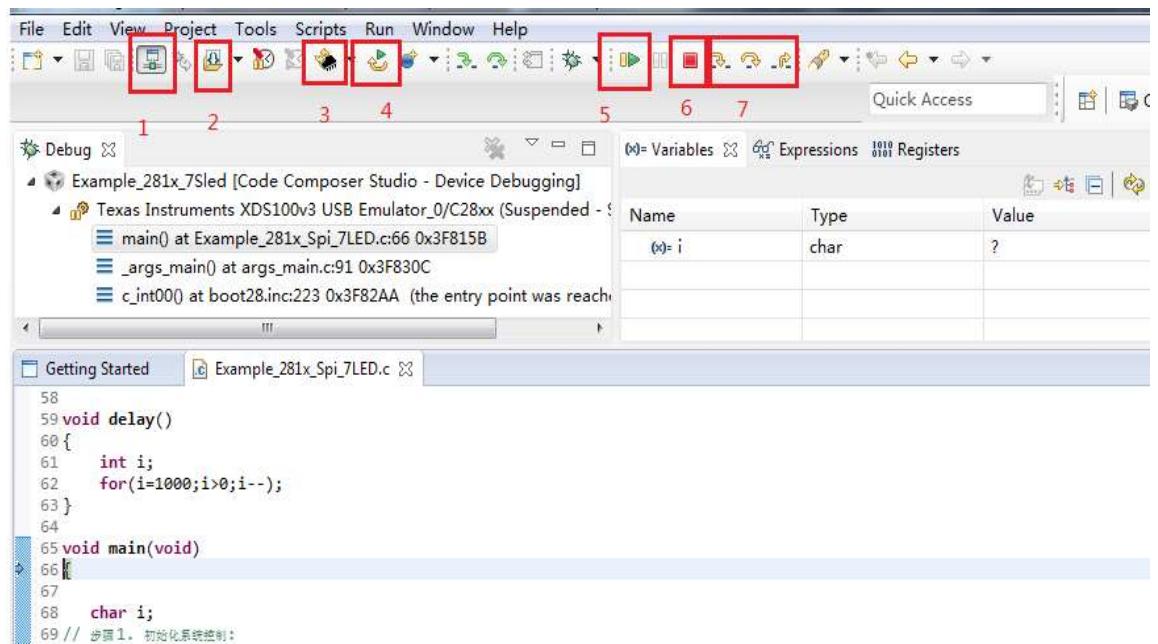


图 3 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的

- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；

单击全速运行按钮 。

试验现象：

如下图所示，3 对带有死区的 PWM 波；

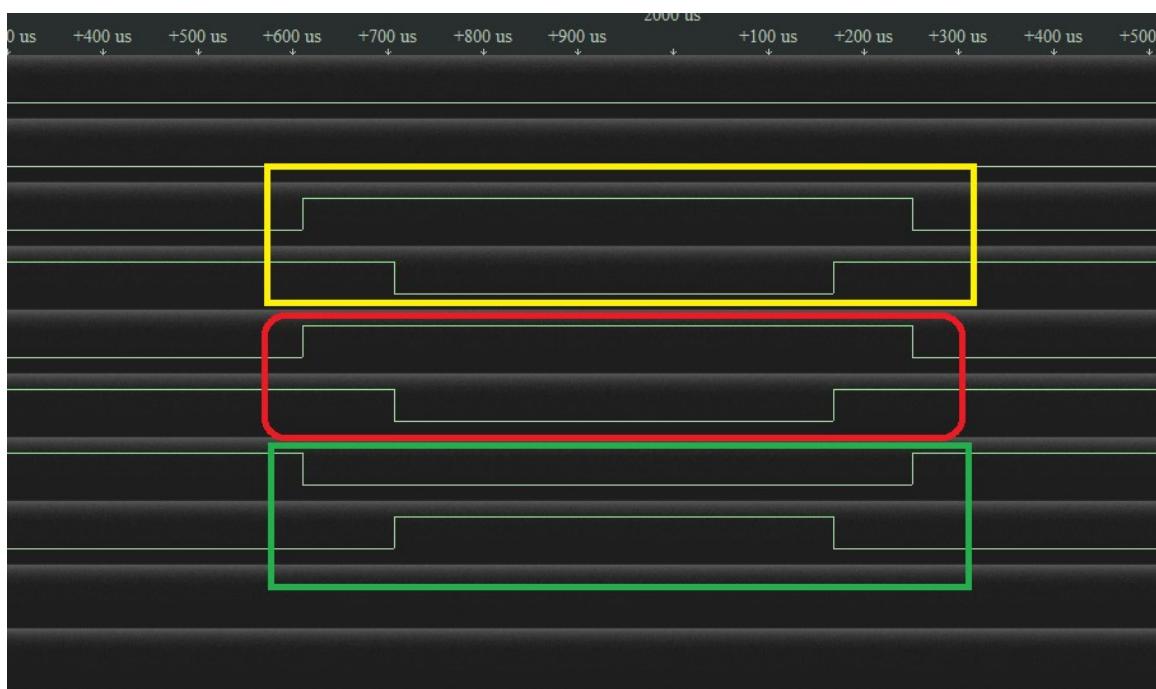


图 4 带有死区的 PWM 波形

如果不带死区则应该是两辆互补的 PWM 波，其上升沿和下降沿是同时产生的，如下图紫色所示：



图 5 带死区与不带死区的图像差别

如果这样看不明白，那么我们也可以改改程序，**禁止死区**，然后看看没有死区时的互补波形是什么样的：

那么如何禁止死区呢？首先看一下图 6；从图中可以发现要想旁路掉死区必须将下图中圆圈里的 S1 和 S0 都设置为 0，但是我们怎么配置才能使其都打到 0 那边呢？方法就在图中。可以看到 DBCTL 控制寄存器的 OUT_MODE 决定了开关 S1 和 S0 的状态。

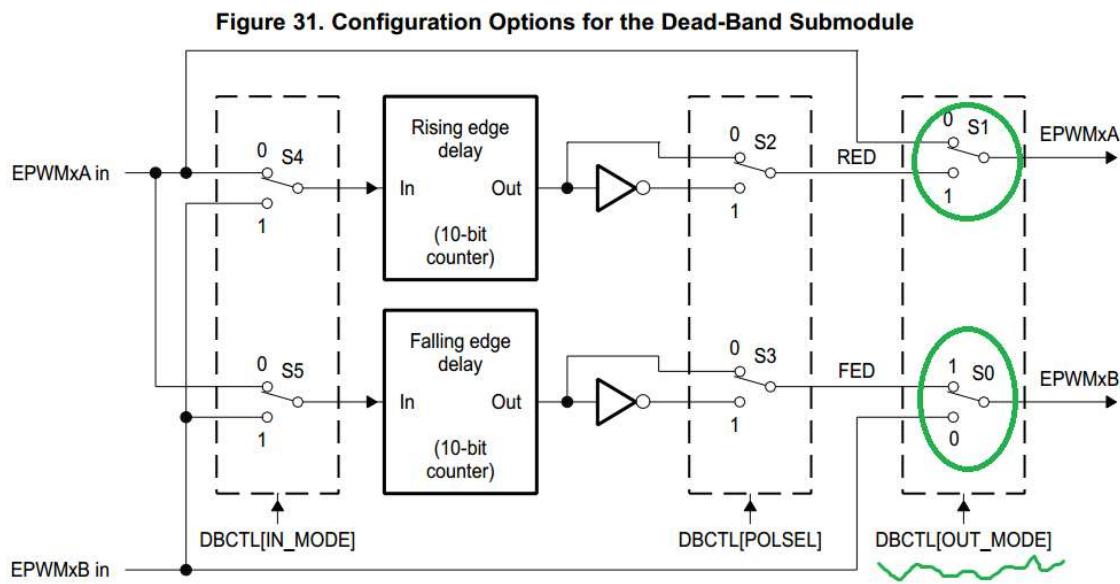


图 6 死区实现模块

下图是 OUT_MODE 位的定义。这是我们可以发现“00”这个状态时就是我们想要的状态。则我们只需要把 DBCTL 的最低两位设置为“00”

即可。

表 1 OUT_MODE 位的定义

1-0	OUT_MODE		Dead-band Output Mode Control Bit 1 controls the S1 switch and bit 0 controls the S0 switch shown in Figure 31. This allows you to selectively enable or bypass the dead-band generation for the falling-edge and rising-edge delay.
		00	Dead-band generation is bypassed for both output signals. In this mode, both the EPWMxA and EPWMxB output signals from the action-qualifier are passed directly to the PWM-chopper submodule. In this mode, the POLSEL and IN_MODE bits have no effect.
		01	Disable rising-edge delay. The EPWMxA signal from the action-qualifier is passed straight through to the EPWMxA input of the PWM-chopper submodule.
		10	The falling-edge delayed signal is seen on output EPWMxB. The input signal for the delay is determined by DBCTL[IN_MODE].
		11	Disable falling-edge delay. The EPWMxB signal from the action-qualifier is passed straight through to the EPWMxB input of the PWM-chopper submodule.
			Dead-band is fully enabled for both rising-edge delay on output EPWMxA and falling-edge delay on output EPWMxB. The input signal for the delay is determined by DBCTL[IN_MODE].

现在我们在程序中找到 PWM1A 和 PWM1B 的配置函数。我们对其进行配置：

```
void InitEPwm1Example()
{
    EPwm1Regs.TBPRD = 6000;                                // Set timer period
    EPwm1Regs.TBPHS.half.TBPHS = 0x0000;                      // Phase is 0
    EPwm1Regs.TBCTR = 0x0000;                                // Clear counter

    // Setup TBCLK
    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;           // Disable phase loading
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;          // Clock ratio to SYSCLKOUT
    EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV4;

    /*
        EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;      // Load registers every ZERO
        EPwm1Regs.CMPCTL.bit.SHDBWMODE = CC_SHADOW;
        EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;
        EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;
    */

    // Setup compare
    EPwm1Regs.CMPA.half.CMPA = 3000;
    // Set actions
    EPwm1Regs.AQCTLA.bit.CAU = AQ_SET;                  // Set PWM1A on Zero
    EPwm1Regs.AQCTLA.bit.CAD = AQ_CLEAR;
    EPwm1Regs.AQCTLB.bit.CAU = AQ_CLEAR;                // Set PWM1A on Zero
    EPwm1Regs.AQCTLB.bit.CAD = AQ_SET;

    // 就下面这句话决定了死区的旁路与否。我们设置为 0 则旁路
}
```

死区

```
EPwm1Regs.DBCTL.bit.OUT_MODE = 0; //DB_FULL_ENABLE;  
EPwm1Regs.DBCTL.bit.POLSEL = DB_ACTV_LOC; //DB_ACTV_LO;  
EPwm1Regs.DBCTL.bit.IN_MODE = DBA_ALL;  
EPwm1Regs.DBRED = EPWM1_MIN_DB;  
EPwm1Regs.DBFED = EPWM1_MIN_DB;  
EPwm1_DB_Direction = DB_UP;  
// Interrupt where we will change the Deadband  
EPwm1Regs.ETSEL.bit.INTSEL = ET_CTR_ZERO; // Select INT on Zero event  
EPwm1Regs.ETSEL.bit.INTEN = 1; // Enable INT  
EPwm1Regs.ETPS.bit.INTPRD = ET_3RD; // Generate INT on 3rd event  
}
```

修改后的波形如下图所示：



图 7 第一对 PWM 去掉死区后的图形

从图中可以看到第一对 PWM 的上升沿和下降沿已经同时变化了。没有死区功能了。

死区一般用在电机控制方面。请您自己掌握与学习一下死区相关的知识。

实验 45：CAP 捕捉 PWM 边沿实验 (Example_2833xECap_Capture_Pwm)

一 实验目的：

- ✧ 了解 28335 的 pwm 外设模块的使用方法；
- ✧ 了解 Ecap 如何配置使其能捕获 PWM 边沿；

二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套；

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；实验原理是用 ECAP 捕获 PWM 输出的波形，本实验就是验证这个功能，硬件连接关系：将 GPIO4/EPWM3A 与 GPIO24 链接到一起：

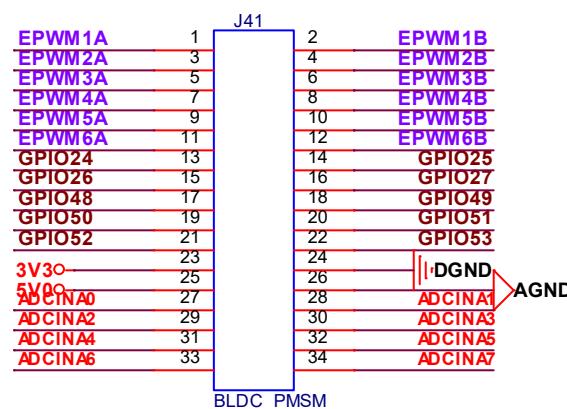


图 1 GPIO24 可以用作 eCAP1 输出管脚

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调试：

- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

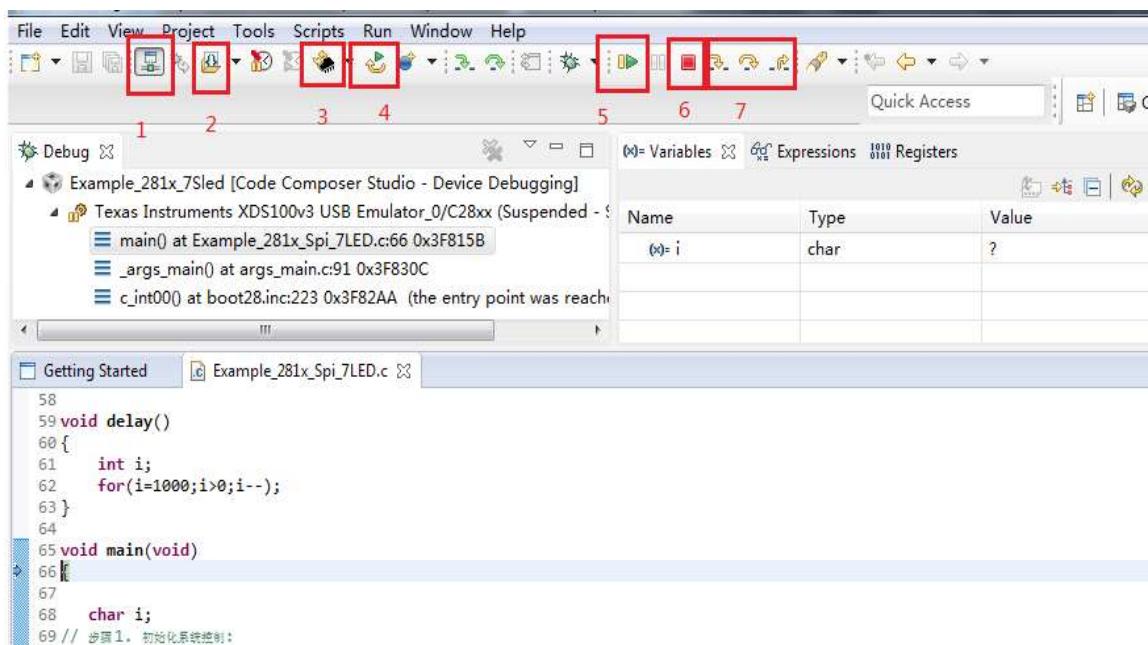


图 2 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
✧ 图中 3 是 C P U 软 R e s e t；
✧ 图中 4 是调试时恢复到程序的开始处。
✧ 图中 5 是全速运行；
✧ 图中 6 是停止调试；
✧ 图中 7 是用于单步调试的；



单击全速运行按钮。

试验现象：

首先将

```
Uint32 ECap1IntCount;  
Uint32 ECap1PassCount;
```

这两个变量添加到观察窗口中。

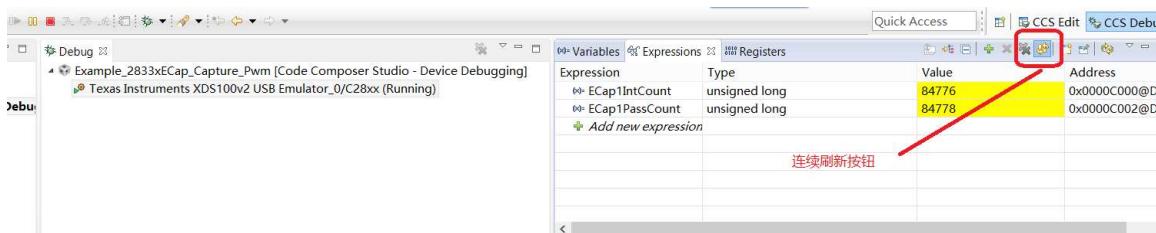


图 3 实验结果

ECAP 模块简介：

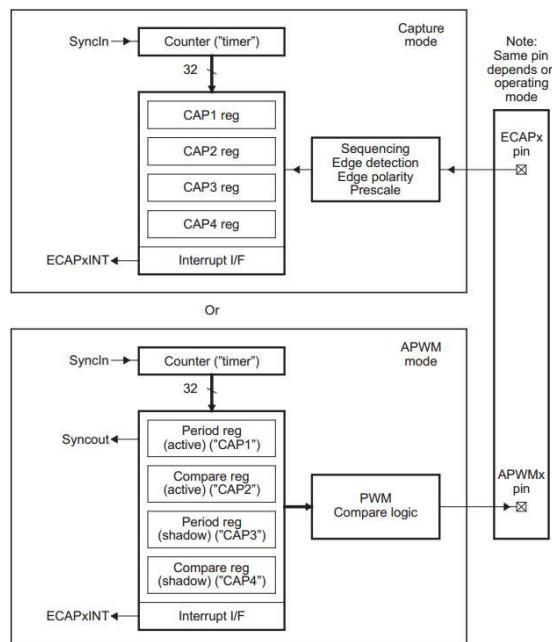


图 4 CAP 结构简介

TMS320F28335 上面有 6 路增强型捕捉模块 eCAP，CAP 模块是应用定时器来实现事件捕获 功能，主要应用在速度测量，脉冲序列周期测量等方面。其原理框图如上图所示。

从图中可以看出 Ecap 模块有两大模块，一个是做 CAP（捕获单元使用）。另一模块是 APWM 功能。本实验使用的是 ECAP 的 CAP 功能，捕获 PWM 波形的上升沿和下降沿等功能。

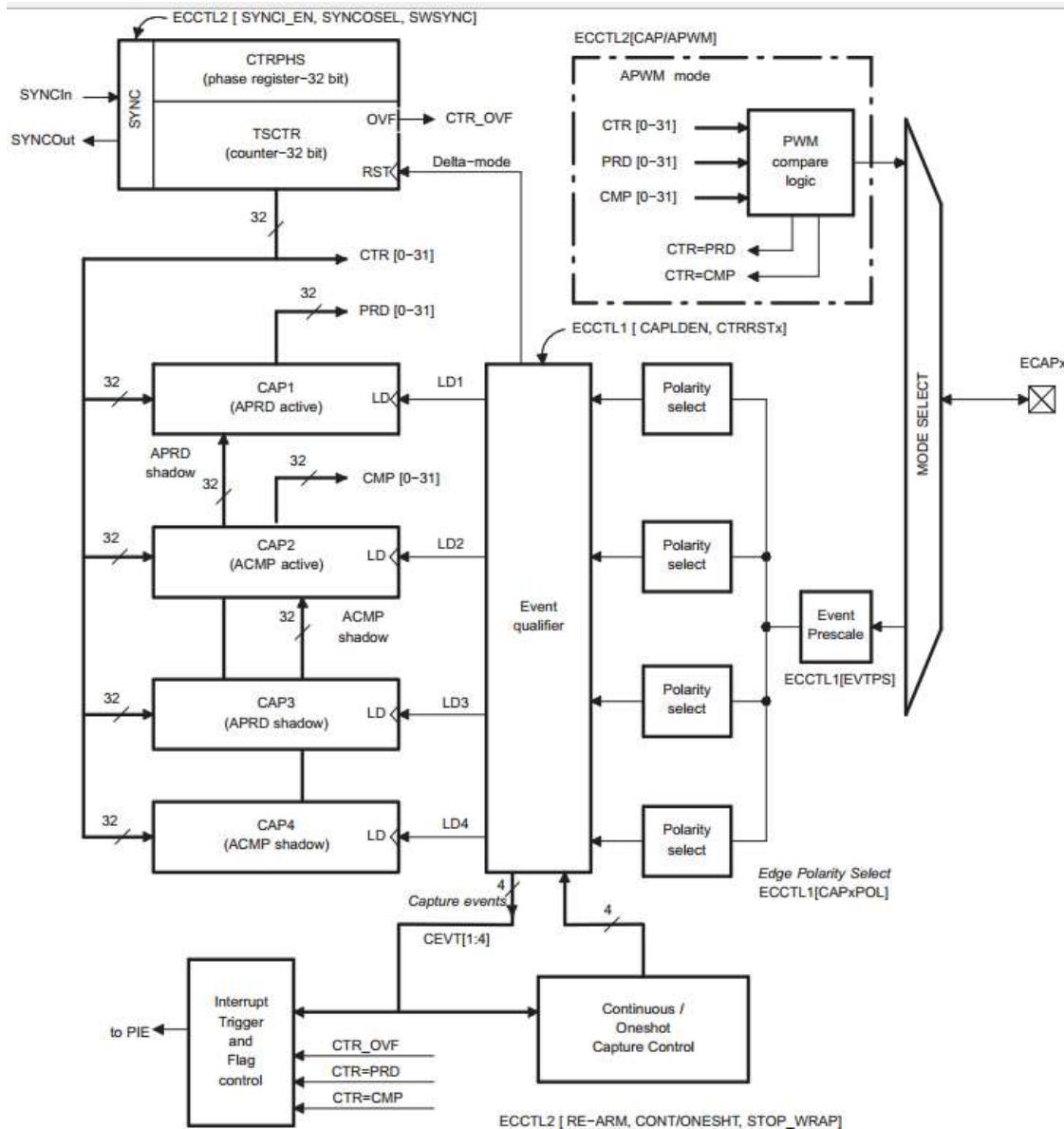


图 5 CAP 内部结构图

eCAP 模块包括以下的资源:

- (1) 可分配的输入引脚;
- (2) 32 bit 时间基准 (计数器);
- (3) 4 个 32 bit 时间窗捕获控制寄存器;
- (4) 独立的边缘极性选择;
- (5) 输入信号分频 ($2^{~62}$);
- (6) 4Capture event 均可引起中断;

eCAP 模块功能分析:

eCAP 模块可以设置为捕捉模式或者是 APWM 模式，一般而言前者比较常用，因此在这里 只对第一种情况进行分析介绍。在捕捉模式下，一般可以将 eCAP 模块分为以下几个模块：事件分频、边沿极性选择与验证、中断控制。

(1) 事件分频 输入事件信号可通过分频器分频处理 (分频系数 $2^{~62}$)，或直接跳过分频器。这个功能 通常针对输入事件信号频率很高的情况下适用。

(2) 边沿极性选择与验证

4 个独立的边沿极性（上升沿/下降沿）选择通道； Modulo 4 序列发生器对 Each edge (共 4 路) 进行事件验证； CAPx 通过 Mod4 对事件边沿计数，CAPx 寄存器在下降沿时被装载。32 bit 计数器 (TSCTR)。此计数器为捕捉提供时钟基准，而时钟的计数则是基于系统时钟的。当此计数器计数超 范围时，则会产生相应的溢出标志，若溢出中断使能，则产生中断。此计数器在计算事件周期时非常有效，详细关于 CAP 的资料请参数据手册。

(3) 中断控制 中断能够被捕获事件所 (CEVT1–CEVT4, CTROVF) 触发，计数器 (TSCTR) 计数溢出同样会 产生中断。事件单独地被极性选择部分以及序列验证部分审核。这些事件中的一个被选择用 来作为中断源送入 PIE。设置 CAP 中断的过程：

关闭全局中断；

停止 eCAP 计数；

关闭 eCAP 的中断；

设置外设寄存器；

清除 eCAP 中断标志位；

使能 eCAP 中断；

开启 eCAP 计数器；

使能全局中断；

eCAP 模块原理的加深理解：

配置好 eCAP 模块的引脚后，外部事件由引脚输入，首先通过模块的分频部分，分频系数 为 $2^{~62}$ ，也可以选择跳过分频部分，此功能主要是针对输入事件信号频率很高的情况。经过 分频后的信号（通常频率会降低），送至边沿及序列审核部分，边沿审核即设置为上升沿或 下降沿有效，序列审核则是指分配当前对哪个寄存器 (CAP1~CAP4) 作用的问题，之后就是 中断执行控制部分。此实验利用 CAP 捕捉信号发生器产生方波，T1 和 T2、T3 和 T4 所测值的 公式为： $T=TMS320F28335 \text{ 工作频率}/\text{所测信号频率}$ ；TMS320F28335 工作频率为 150MHz，信号发生器产生方波的频率为 10KHz，所以 T 的值应该是 15000。

程序解析：

```
EALLOW; //将捕获中断函数映射到中断向量表中
PieVectTable.ECAP1_INT = &ecap1_isr;
EDIS;
__interrupt void ecap1_isr(void)
{
    // 捕获中断函数
    // 判断一下捕获寄存器的值是否在两倍（定时器的计数方式）PWM 周期寄存器的正负 1 范围内。如果不在则认为错误
    if(ECap1Regs.CAP2 > EPwm3Regs.TBPRD*2+1 || ECap1Regs.CAP2 < EPwm3Regs.TBPRD*2-1)
    {
        Fail();
    }
    if(ECap1Regs.CAP3 > EPwm3Regs.TBPRD*2+1 || ECap1Regs.CAP3 < EPwm3Regs.TBPRD*2-1)
    {
```

```
Fail();  
}  
if(ECap1Regs.CAP4 > EPwm3Regs.TBPRD*2+1 || ECap1Regs.CAP4 < EPwm3Regs.TBPRD*2-1)  
{  
    Fail();  
}  
ECap1IntCount++;  
// 调整 PWM 周期  
if(EPwm3TimerDirection == EPWM_TIMER_UP)  
{  
    if(EPwm3Regs.TBPRD < PWM3_TIMER_MAX)  
    {  
        EPwm3Regs.TBPRD++;  
    }  
    else  
    {  
        EPwm3TimerDirection = EPWM_TIMER_DOWN;  
        EPwm3Regs.TBPRD--;  
    }  
}  
else  
{  
    if(EPwm3Regs.TBPRD > PWM3_TIMER_MIN)  
    {  
        EPwm3Regs.TBPRD--;  
    }  
    else  
    {  
        EPwm3TimerDirection = EPWM_TIMER_UP;  
        EPwm3Regs.TBPRD++;  
    }  
}  
ECap1PassCount++;  
ECap1Regs.ECCLR.bit.CEVT4 = 1;  
ECap1Regs.ECCLR.bit.INT = 1;  
ECap1Regs.ECCTL2.bit.REARM = 1;  
// Acknowledge this interrupt to receive more interrupts from group 4  
PieCtrlRegs.PIEACK.all = PIEACK_GROUP4;  
}  
// 错误函数  
void Fail()  
{  
    __asm("    ESTOPO");  
}
```

实验 46：CAP 输出 PWM 波实验 (Example_2833xECap_apwm)

一 实验目的：

- ✧ 了解 28335 的 pwm 外设模块的使用方法；
- ✧ 了解 Ecap 如何配置实现 PWM 功能；

二 实验设备：

- ✧ PC 机一台；
- ✧ XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- ✧ SXD28335 开发板一套，示波器或逻辑分析仪一台；

三 实验步骤：

- ✧ 首先将 CCS6.0 开发环境打开；实验原理是由于 ECAP 具有输出 PWM 波的功能，本实验就是验证这个功能，下面是 Ecap 的接口：

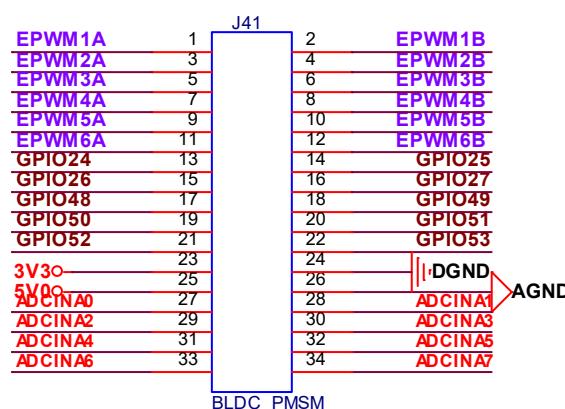


图 1 GPIO24 可以用作 eCAP1 输出管脚

用示波器或逻辑分析仪链接 GPIO24 时注意不要和周边的引脚短路。

- ✧ 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- ✧ 给开发板上电。单击鼠标左键选择要调试的工程，并进行在线调

试：

- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

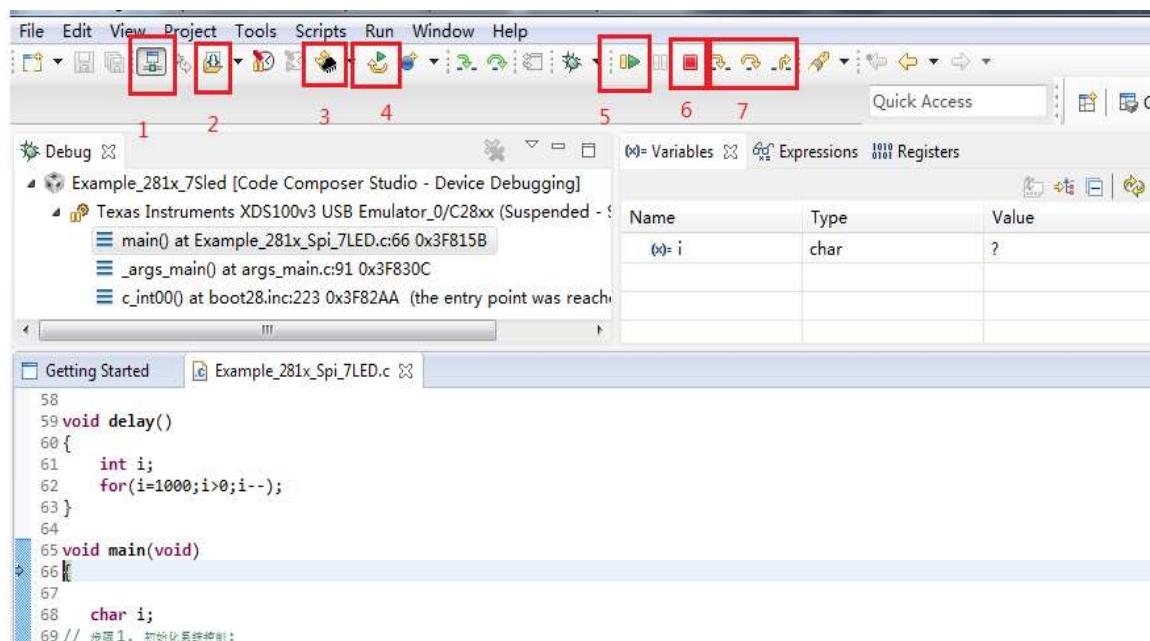


图 2 调试界面

- ✧ 图中 1 图标是用来进行与开发板进行连接的按钮；
- ✧ 图中 2 是用来下载 Debug 文件下的.out 文件的
- ✧ 图中 3 是 C P U 软 R e s e t；
- ✧ 图中 4 是调试时恢复到程序的开始处。
- ✧ 图中 5 是全速运行；
- ✧ 图中 6 是停止调试；
- ✧ 图中 7 是用于单步调试的；

单击全速运行按钮

试验现象：

GPIO24 管脚会输出 PWM 波形；如下图所示，从图中可以看到波形是稀疏变化的。

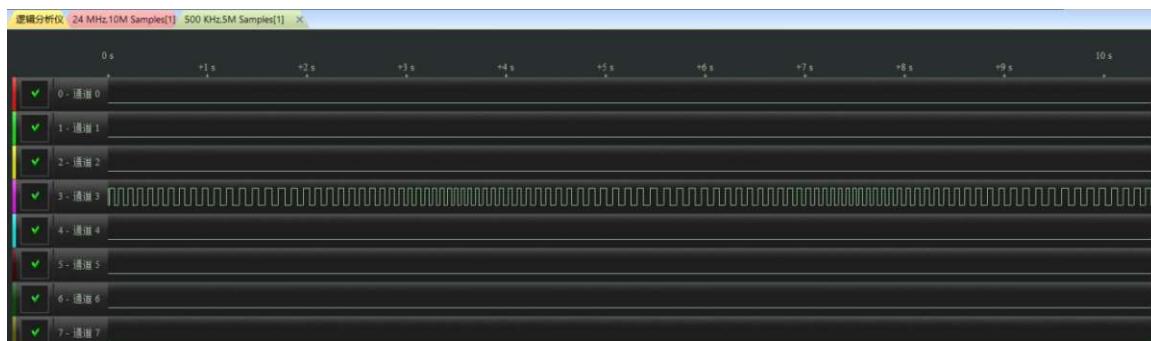


图 3 实验结果

ECAP 模块简介：

TMS320F28335 上面有 6 路增强型捕捉模块 eCAP，CAP 模块是应用定时器来实现事件捕获 功能，主要应用在速度测量，脉冲序列周期测量等方面。其原理框图如下图 所示：

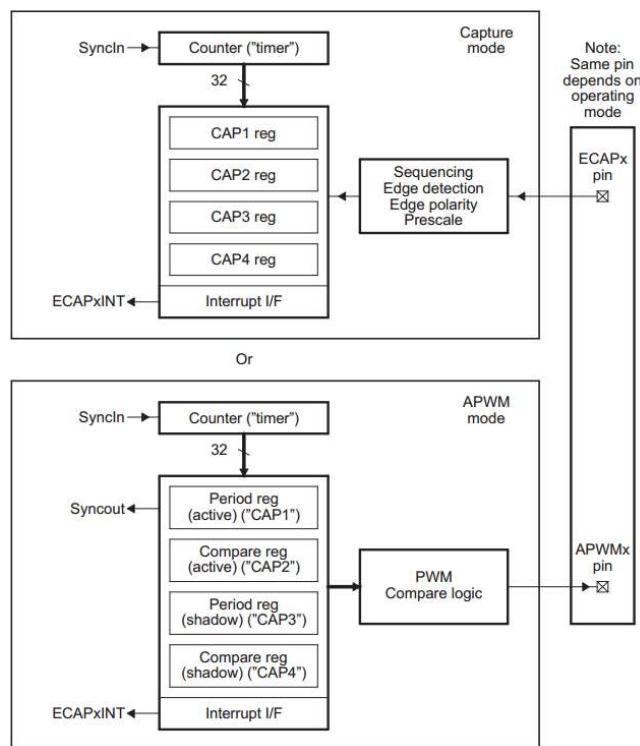


图 4 CAP 结构简介

从图中可以看出 Ecap 模块有两大模块，一个是做 CAP（捕获单元使用）。另一模块是 APWM 功能。本实验使用的是 ECAP 的 APWM 功能，产生 PWM 波形。

APWM 模式操作：

- 1、当 CAP1/2 寄存器不再用于捕获模式，则可以在 APWM 模式下用做为周期和比较值。
- 2、在 APWM 模式下，向 CAP1/CAP2 Active 寄存器写数据的同时，同样的值也被写到 Shadow 寄存器 CAP3/CAP4 中。
- 3、初始化阶段我们要向 Active 周期和 Active 比较寄存器写值。这些值会自动的 COPY 到 Shadow 寄存器中。

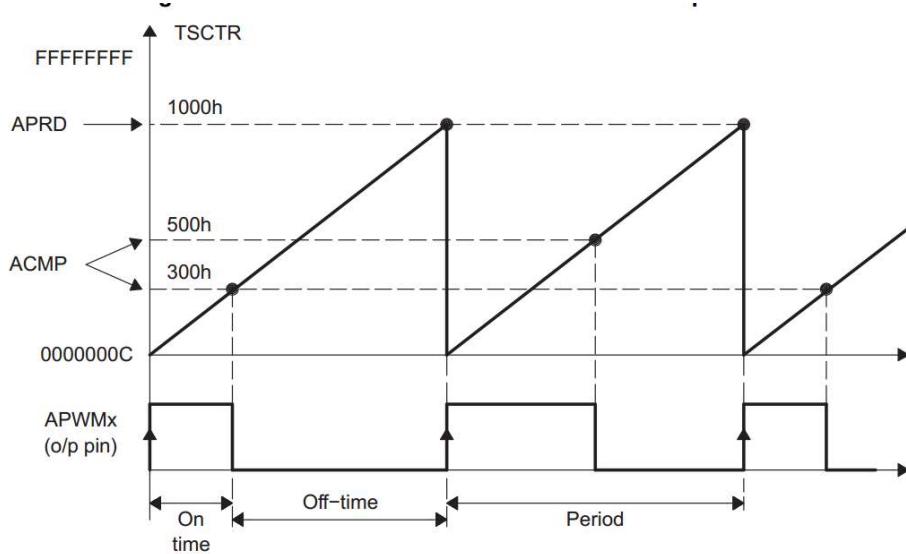


图 5 CAP 的 APWM 波形实现过程

程序解析：

```
// Setup APWM mode on CAP1, set period and compare registers
ECap1Regs.ECCTL2.bit.CAP_APWM = 1; // 使能 APWM 模式
ECap1Regs.CAP1 = 0x01312D00;           // 设置周期值
ECap1Regs.CAP2 = 0x00989680;           // 设置比较值
ECap1Regs.ECCLR.all = 0xFF;             // 清除中断标志
ECap1Regs.ECEINT.bit.CTR_EQ_CMP = 1; // 使能比较相等事件

for(;;)
{
    // 通过右移，占空比减半
    ECap1Regs.CAP4 = ECap1Regs.CAP1 >> 1;
    // 根据周期值设置增减方向
    if(ECap1Regs.CAP1 >= 0x01312D00)
    {
        direction = 0;
    } else if (ECap1Regs.CAP1 <= 0x00989680)
    {
        direction = 1;
    }
    // 根据方向设置周期大小
    if(direction == 0)
    {
        ECap1Regs.CAP3 = ECap1Regs.CAP1 - 500000;
    } else
    {
        ECap1Regs.CAP3 = ECap1Regs.CAP1 + 500000;
    }
}
```

}

实验 47：eQEP 频率测量（Example_2833x_Eqep_freqcal）

一、实验目的：

◆ 了解 eQEP 测量频率原理。

二、实验设备

(1) PC 机一台；

(2) XDS100v2 或 XDS100V3(隔离) 仿真器一套；

(3) SXD28335 开发板一套；

三、实验步骤：

(1) 首先将 CCS6.0 开发环境打开；看一下如下原理图：

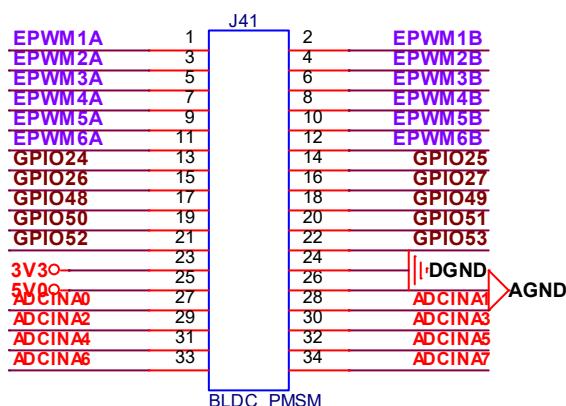


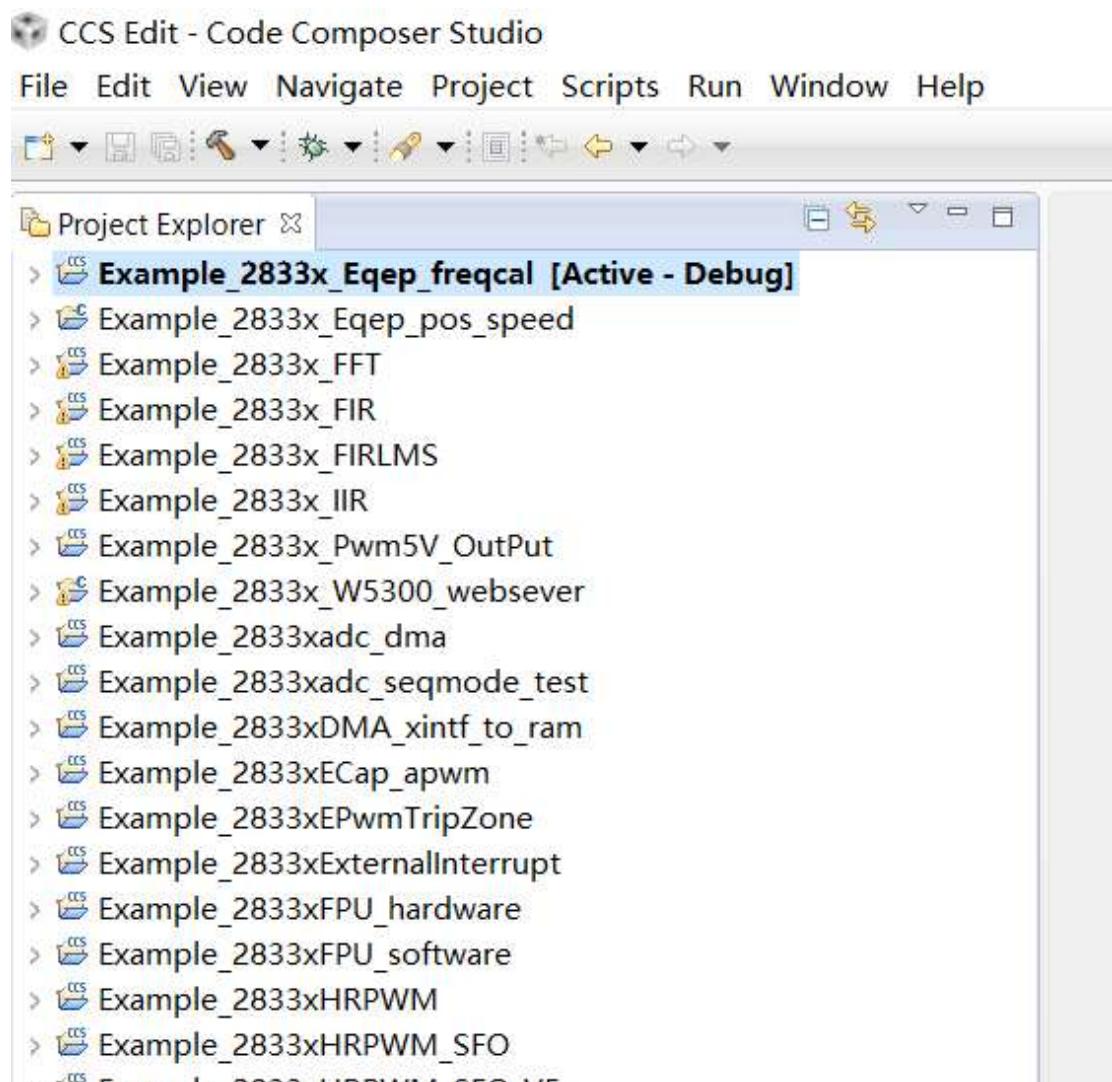
图 1 连接关系图

外部链接

链接 GPIO50 和 GPIO0/EPWM1A；

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：



然后单击图中红色的方框处的调试按钮

(3) 实验现象如下图所示:

Expression	Type	Value	Address
freq.freqhz_fr	long	5000	0x0000C00E@Data
freq.freqhz_pr	long	4989	0x0000C008@Data
Add new expression			

图 3 实验现象

程序说明:

本程序演示增强型正交编码脉冲单元测量 EPWM 频率示例。此示例将利用增强型正交编码脉冲单元模块计算输入信号的频率和周期，被测试信号为 EPWM1A 输出。其中 EPWM1A 配置为 5KHz 频率输出。关于频率计算详细见本例的频率计算公式。

本示例特殊说明：

- 本示例除了主函数外，还需要包含下述函数
 - 1) **Example_freqcal.c**， 包含所有增强型正交编码脉冲单元函数
 - 2) **Example_EPwmSetup.c**， 本示例所用 EPWM 配置函数
 - 3) **Example_freqcal.h**， 频率结构的初始化值

实验 48: EQEP 位置速度测量 (Example 2833x Eqep pos speed)

一、实验目的:

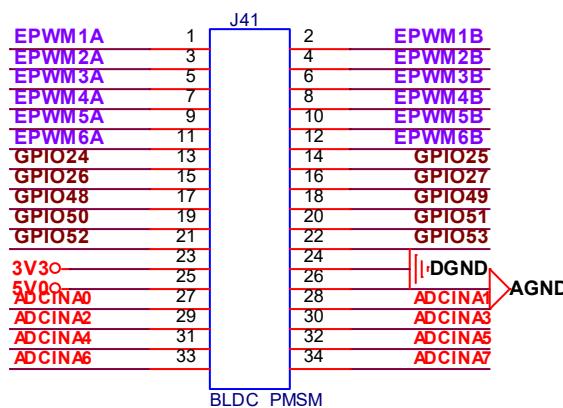
✧ 了解 eQEP 位置速度测量原理。

二、实验设备

- (1) PC 机一台；
- (2) XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- (3) SXD28335 开发板一套；

三、实验步骤：

(1) 打开 CCS 软件，这个例子用 CAP 单元实现位置测量，速度测量。并且使用了 IQMath 库。它仅用于简化高精度计算。

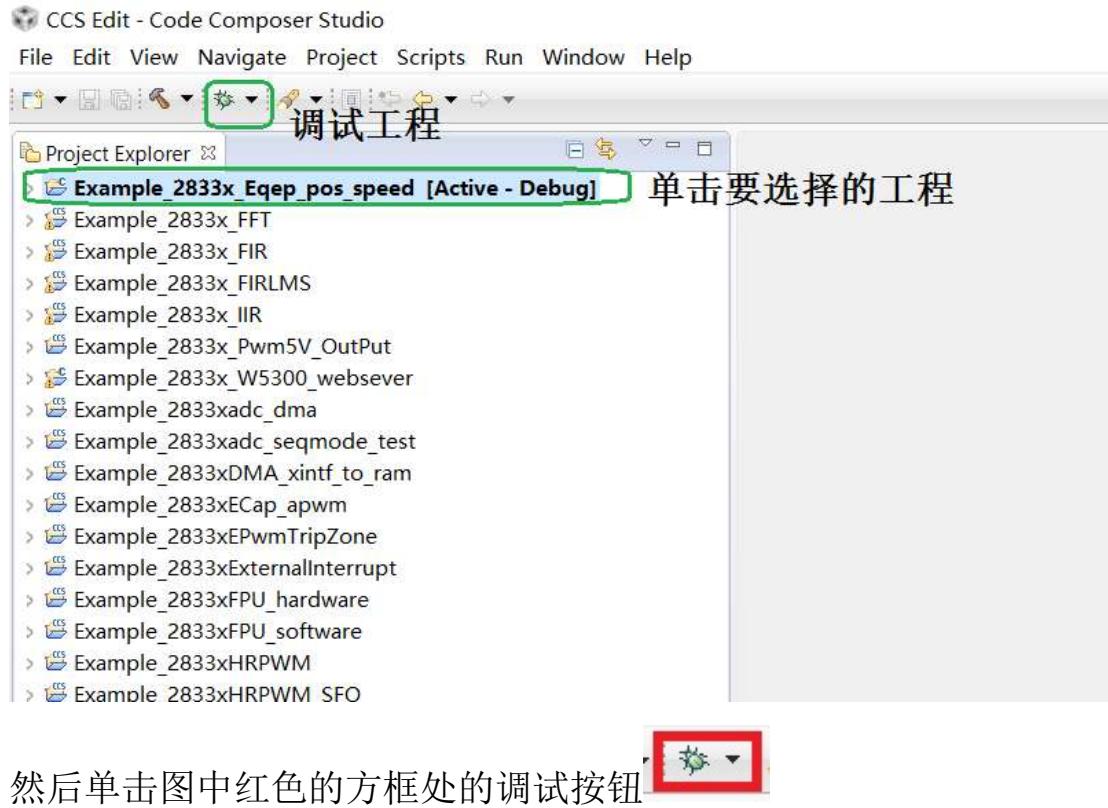


外部链接

- 链接 GPIO50/EQEP1A/XD29 与 ePWM1A(GPIO0)
- 链接 GPIO51/EQEP1B/XD28 与 ePWM1B(GPIO1)
- 链接 GPIO53/EQEP1I/XD26 与 GPIO4/EPWM3A

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：



然后单击图中红色的方框处的调试按钮

(4) 实验现象如下图所示：

Watch Variables

- `qep_posspeed.SpeedRpm_fr` - 速度测量。利用 QEP
- `qep_posspeed.SpeedRpm_pr` - 速度测量。利用 CAP
- `qep_posspeed.theta_mech` - 电机机械角 (Q15)
- `qep_posspeed.theta_elec` - 电机电角度 (Q15)

Expression	Type	Value	Address
<code>qep_posspeed.SpeedRpm_fr</code>	long	299	0x0000C018@Data
<code>qep_posspeed.SpeedRpm_pr</code>	long	300	0x0000C012@Data
<code>qep_posspeed.theta_mech</code>	int	25646	0x0000C003@Data
<code>qep_posspeed.theta_elec</code>	int	18656	0x0000C002@Data
Add new expression			

图 3 实验现象

实验 49：SXD28335 开发板 TF 卡实验 (Example_2833x_SD_FAT32)

一、实验目的：

- ✧ 了解 SD 卡的读写原理。
- ✧ 了解文件系统的实现原理。

二、实验设备

- (1) PC 机一台；
- (2) XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- (3) SXD28335 开发板一套；TF 卡一个；

三、实验步骤：

- (1) 首先将 CCS6.0 开发环境打开；看一下如下原理图：

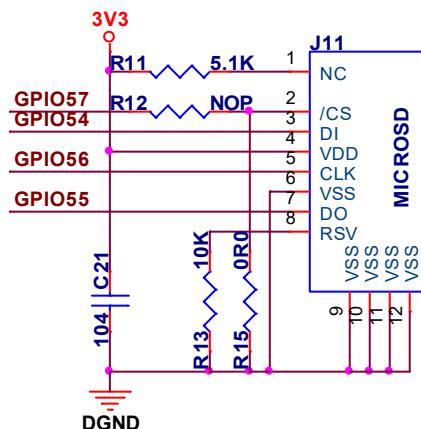


图 1 TF 卡的硬件连接原理图

从图中可以看出 TF 卡是用 SPI 总线控制的。

- (2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；
- (3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：



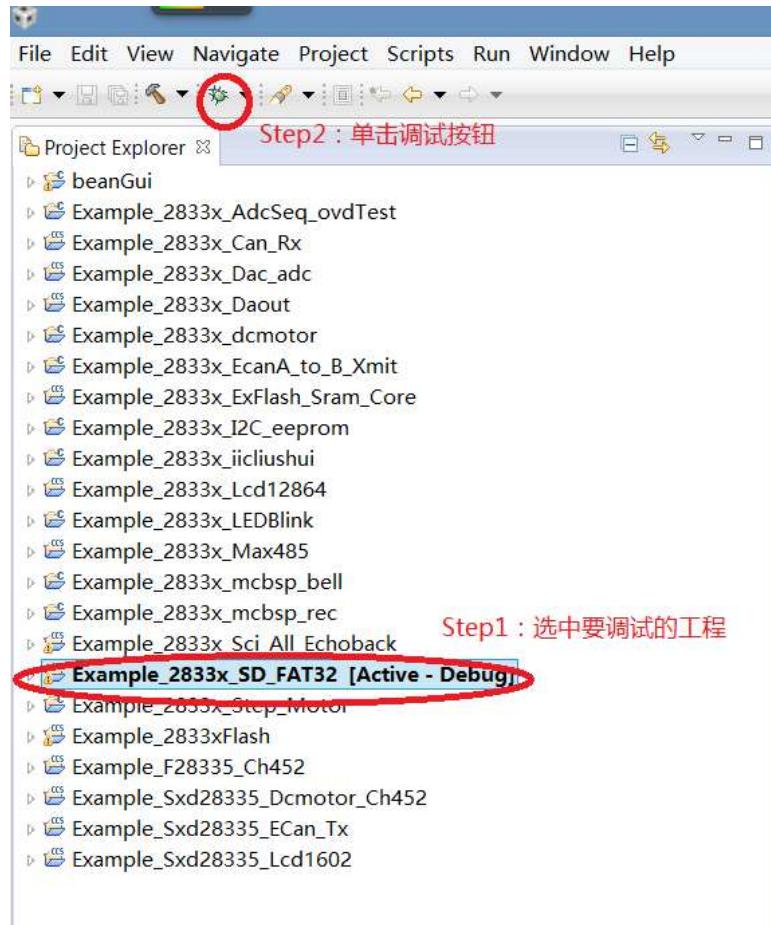


图 2 调试方法

程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。（注意：在开发板上电前请将 TF 卡插到 TF 卡座里，最好不要带电插拔）

全速运行 ；如果跳到 `asm ("ESTOP0");` 这个语句并停止

说明没有成功。这是我们要做的是单击图中的红色方框 ，进行 reset，然后再全速运行 ；直到程序停止在下图所示的语句处：

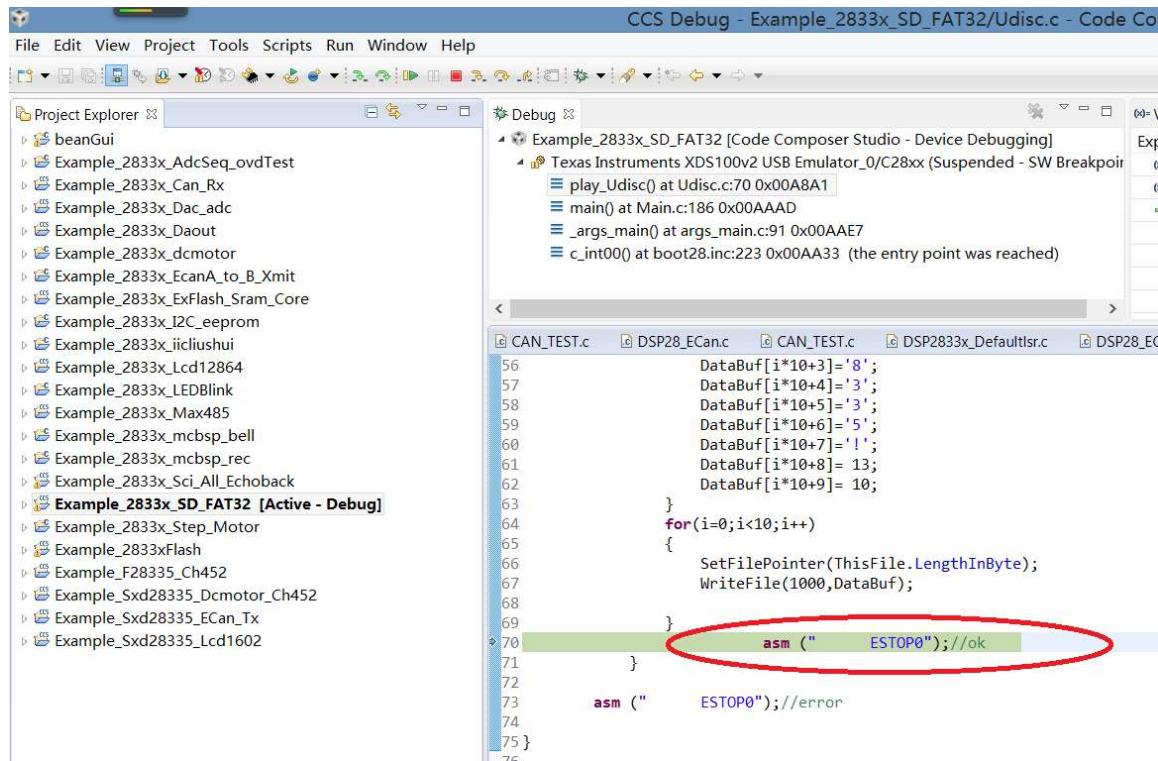


图 3 实验现象

这时在 TF 卡里已经建立了一个文件，打开文件会看到程序中输入的那个字符串；

注：关于文件系统的原理请参考我们提供的文件系统资料。

```

CreateFile("demofat.txt", 0x20); //创建文件名
for (i=0;i<100;i++)
{
    DataBuf[i*10+0]='C'; //向文件里写字母
    DataBuf[i*10+1]=' ';
    DataBuf[i*10+2]='2';
    DataBuf[i*10+3]='8';
    DataBuf[i*10+4]='3';
    DataBuf[i*10+5]='3';
    DataBuf[i*10+6]='5';
    DataBuf[i*10+7]='!';
    DataBuf[i*10+8]=13;
    DataBuf[i*10+9]=10;
}

```

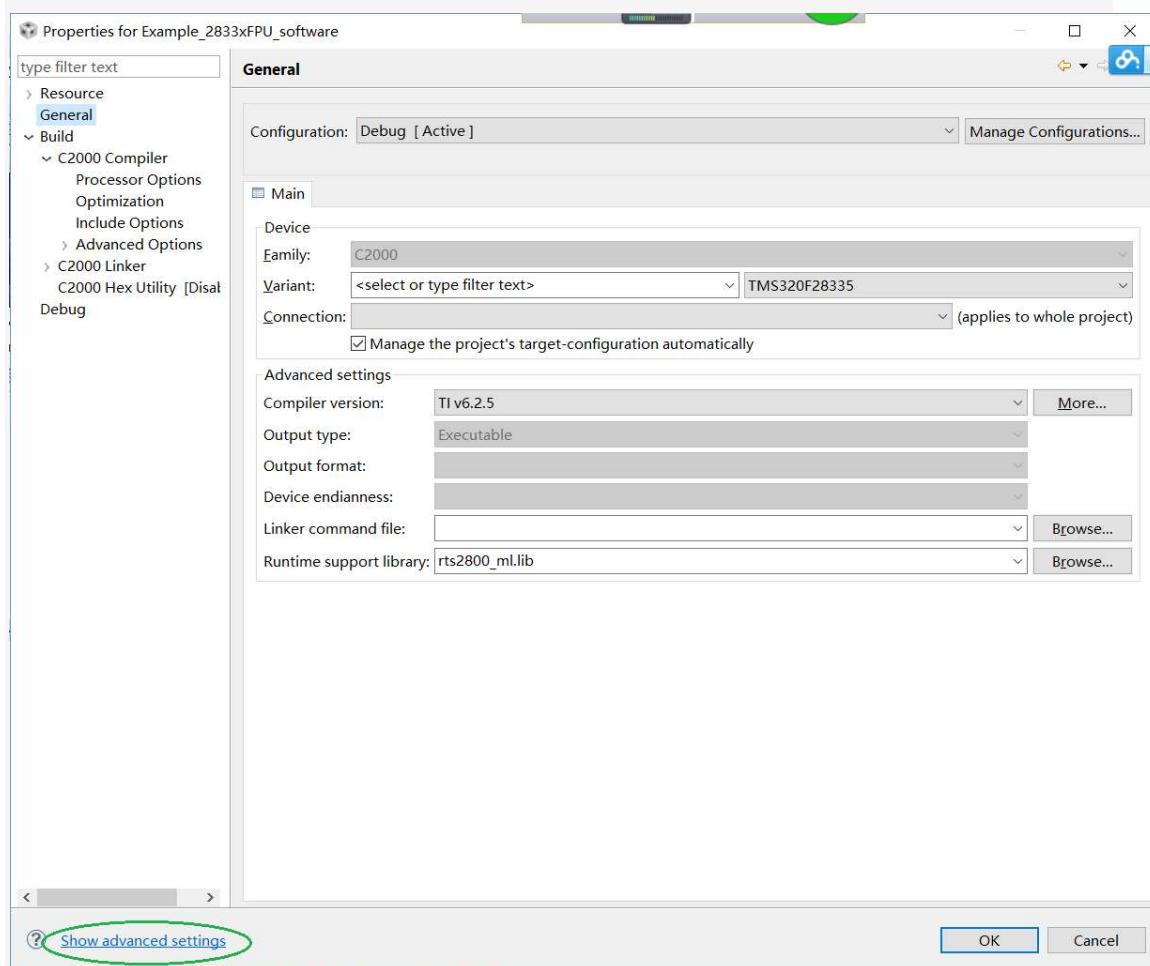
实验 50：单精度浮点运算软件实现 (Example_2833xFPU_software)

本例程示例代码计算两个 $y=mx+b$ 方程。所有的值都是 32-bit floating-point。编译器将只使用定点指令。这意味着运行支持库将被用来模拟浮点。这也没有浮点运算单元在 C28x 设备运行。要编译项目为固定点，要进行下面的配置

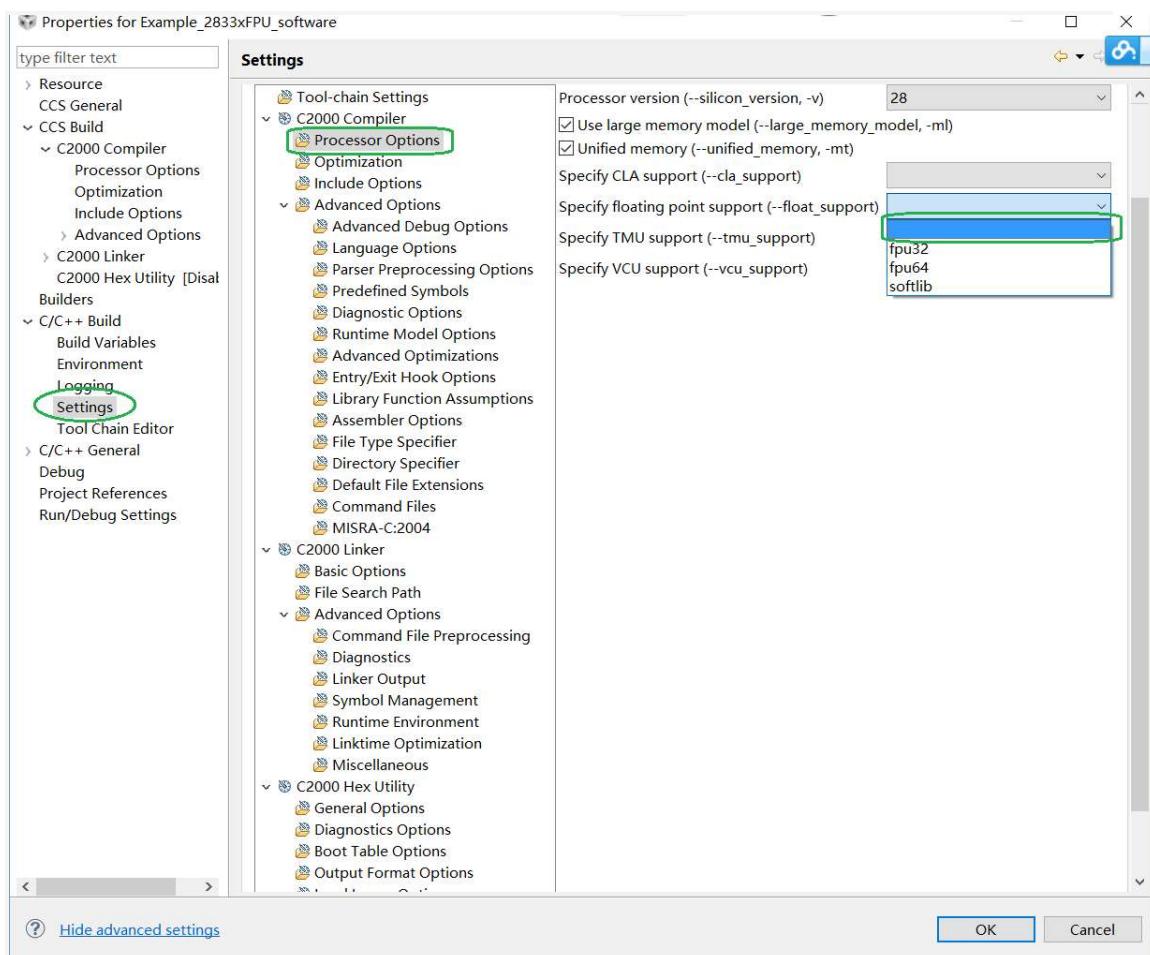
1. Project->Properties-> C/C++ Build window-> Basic Settings-> C2000 Compiler Vx.x

在选项框中：将“-float_support=fpu32”移除：

Step1: show advanced settings

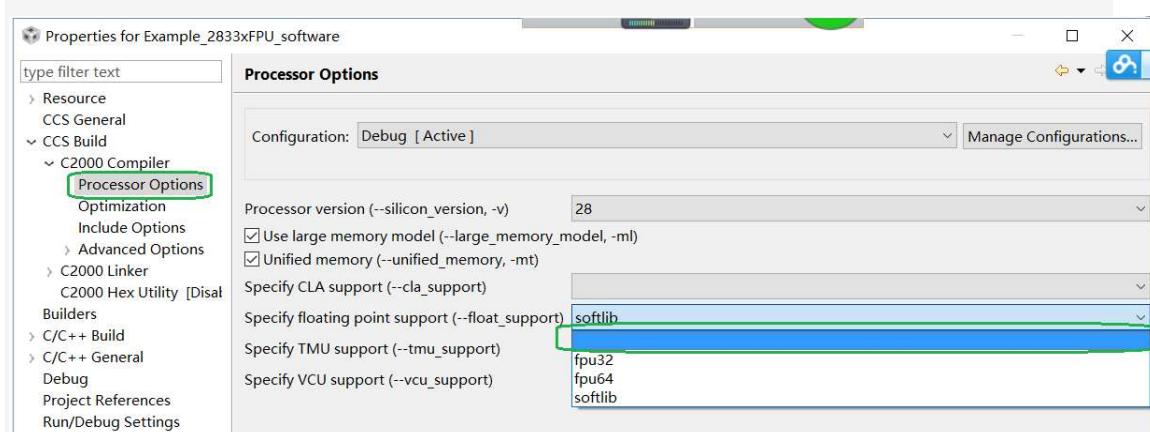


Step2: Specify floating point support->none



OR in Runtime Model Options, under "Specify floating point support (-float_support) pull-down

menu: Select "None".

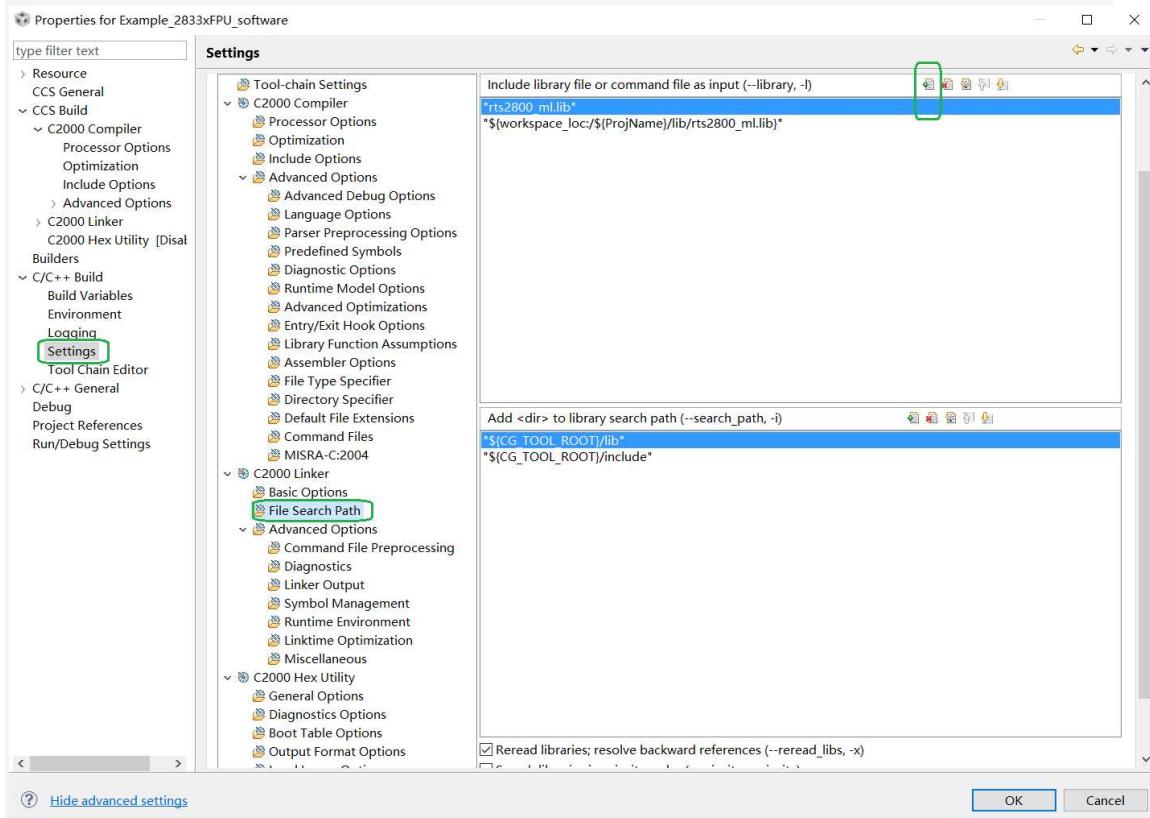


2. Project->Properties-> C/C++ Build window-> Basic Settings-> C2000 Linker Vx.x-> File Search Path

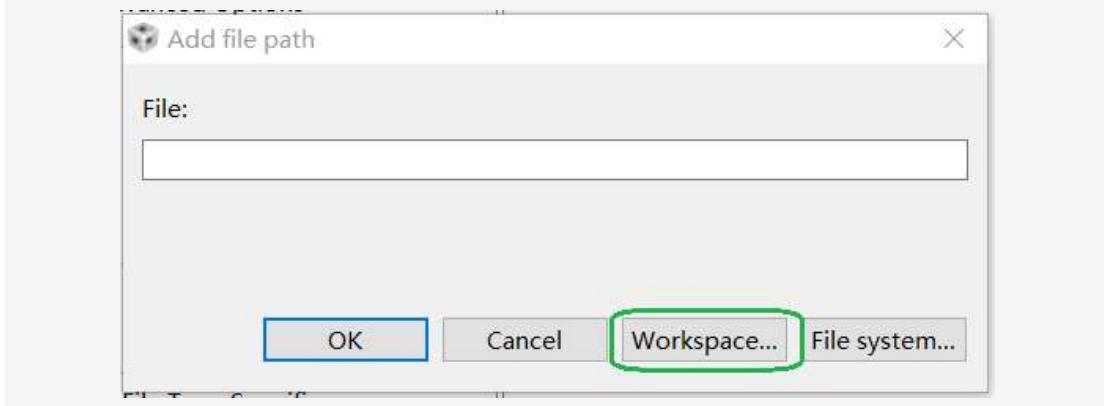
玻尔电子致力于 C2000 全系列开发平台及应用方案的推广

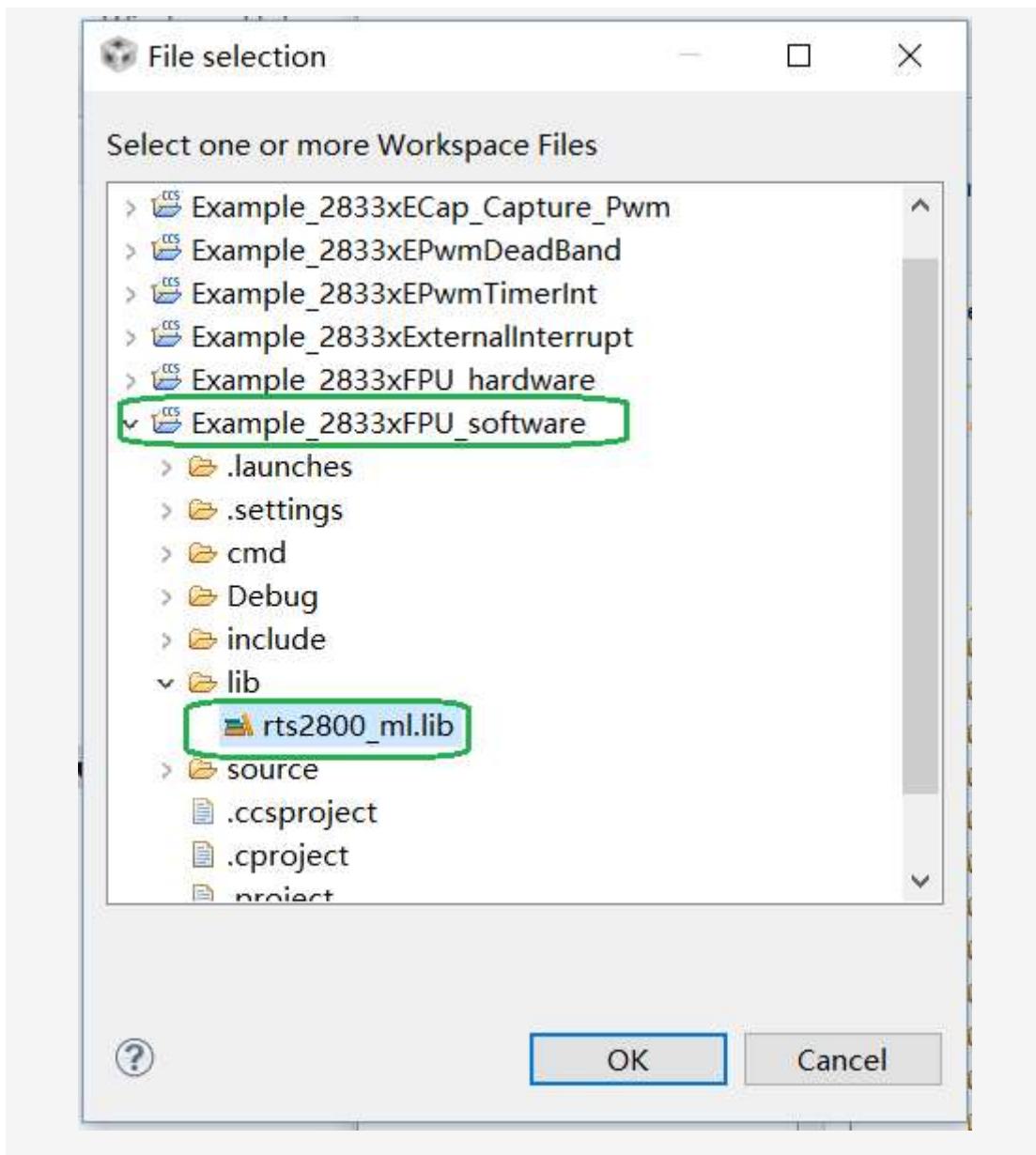
在 "Include linker file or command file as input (-library, -l)" 框中，单击绿色加号添加 rts2800.lib 或 rts2800_ml.lib (run-time support library).

Step1:向工程中添加库



Step2:select lib ->rts2800.lib or rts2800_ml.lib





3. Not included in this example: If the project includes any other libraries, they must also be compiled with fixed point instructions.

观察变量

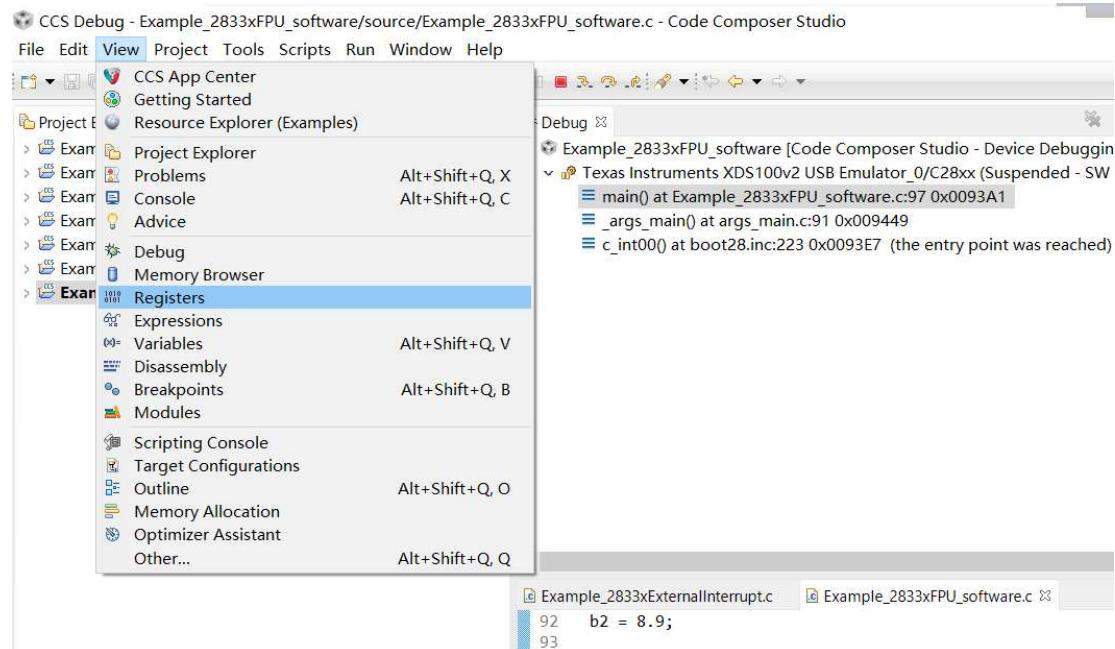
y1

y2

FPU registers (optional)

现象：

Step1:watch FPU registers



Step2:选中 FPU registers

Name	Value	Description
> Core Registers		Core Registers
✓ FPU		FPU Registers
> RB	0x00000000	Repeat Block Register [Memory]
> STF	0x00000000	Floating-Point Status Register [I]
> R0H	0x00000000	Memory Mapped
> R1H	0x00000000	Memory Mapped
> R2H	0x00000000	Memory Mapped
> R3H	0x00000000	Memory Mapped
> R4H	0x00000000	Memory Mapped
> R5H	0x00000000	Memory Mapped
> R6H	0x00000000	Memory Mapped
> R7H	0x00000000	Memory Mapped
> ADC		ADC Registers
> ADCMIRROR		ADC Mirror Registers
> SYSCTRL		System Control Registers
> CSM		Code Security Module

Step3:观察变量

Expression	Type	Value	Address
<code>y1</code>	float	5.9	0x0000C00A@Data
<code>y2</code>	float	13.28	0x0000C008@Data
Add new expression			

Figure 4.47.3 Phenomenon

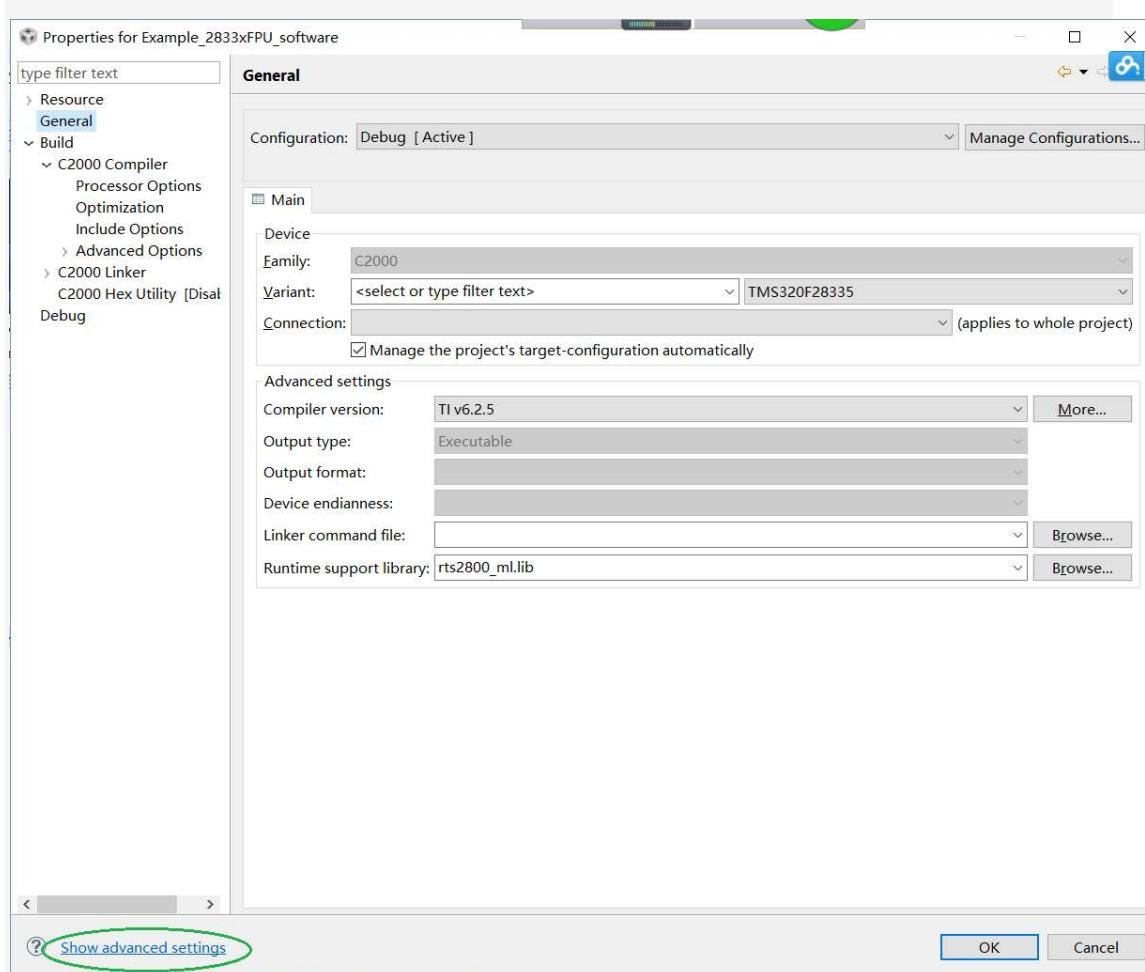
实验 51：单精度浮点单元硬件实现 (Example_2833xFPU_hardware)

本例程代码计算两个 $y=mx+b$ 方程。所有变量都是 32-bit floating-point。编译器将利用浮点指令进行计算。为了用浮点编译此工程，需要进行如下设置：

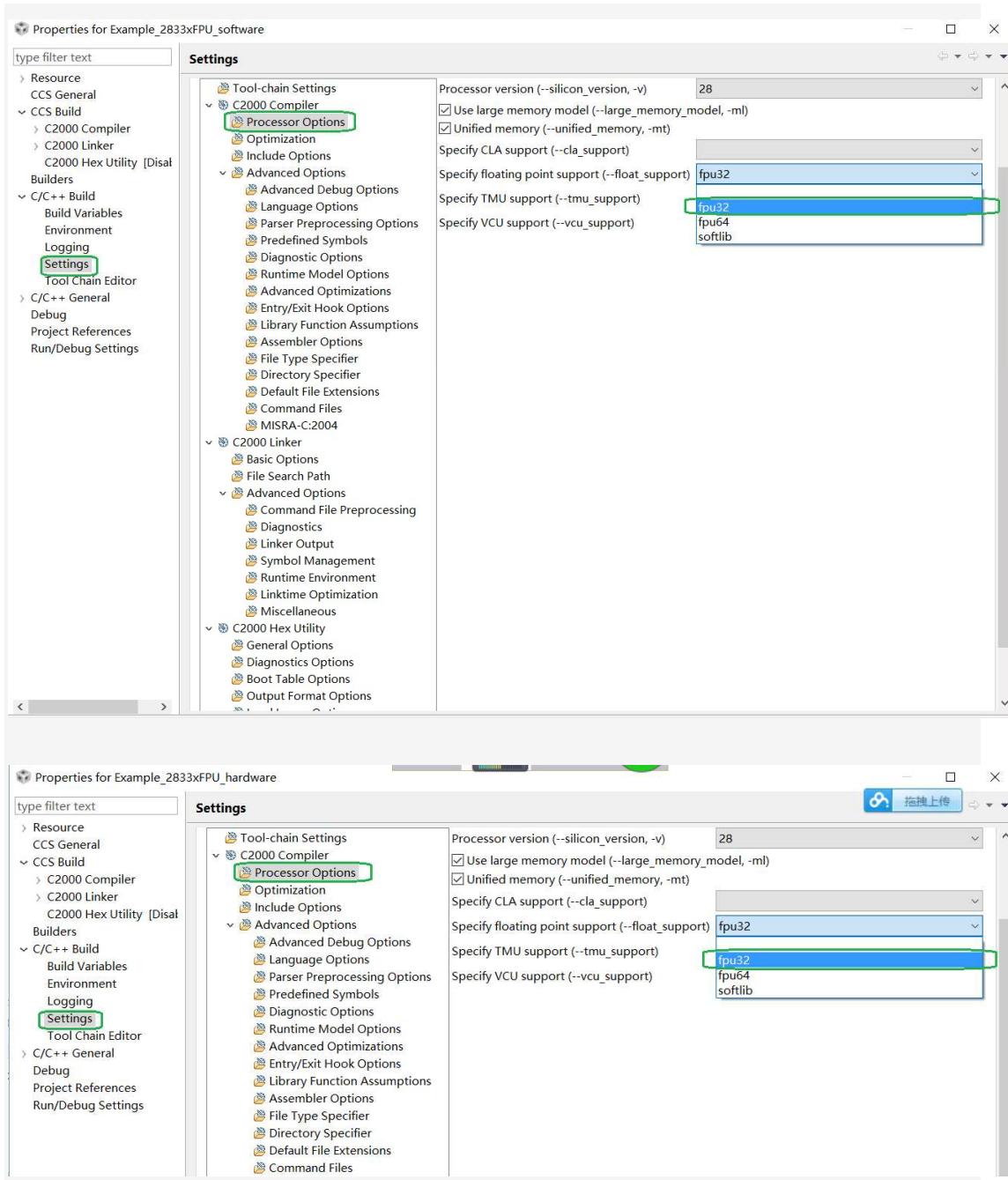
1. Project->Properties-> C/C++ Build window-> Basic Settings-> C2000 Compiler Vx.x

在文本框中：选择“- float_support=fpu32”。

Step1: show advanced settings

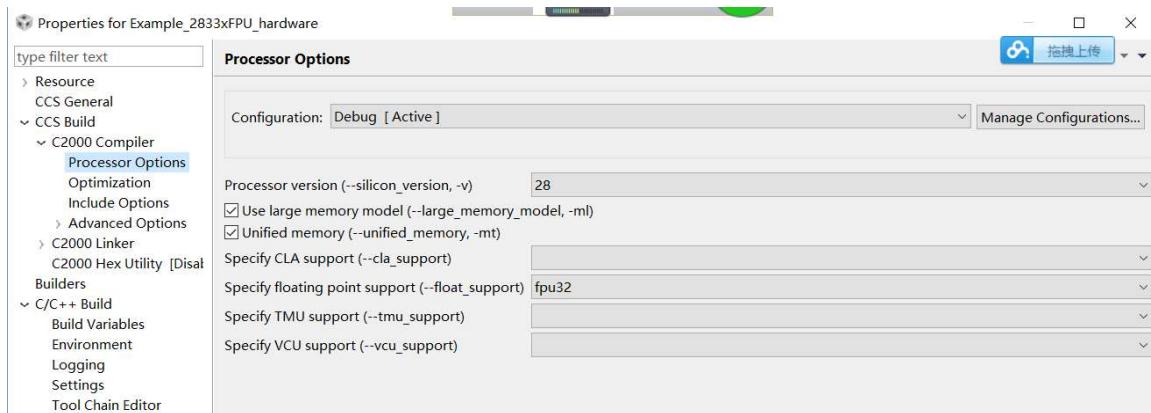


Step2: Specify floating point support-> - float_support=fpu32



或者在 Runtime Model Options, 在"Specify floating point support (-float_support)" 下拉菜单: 选择"fpu32".

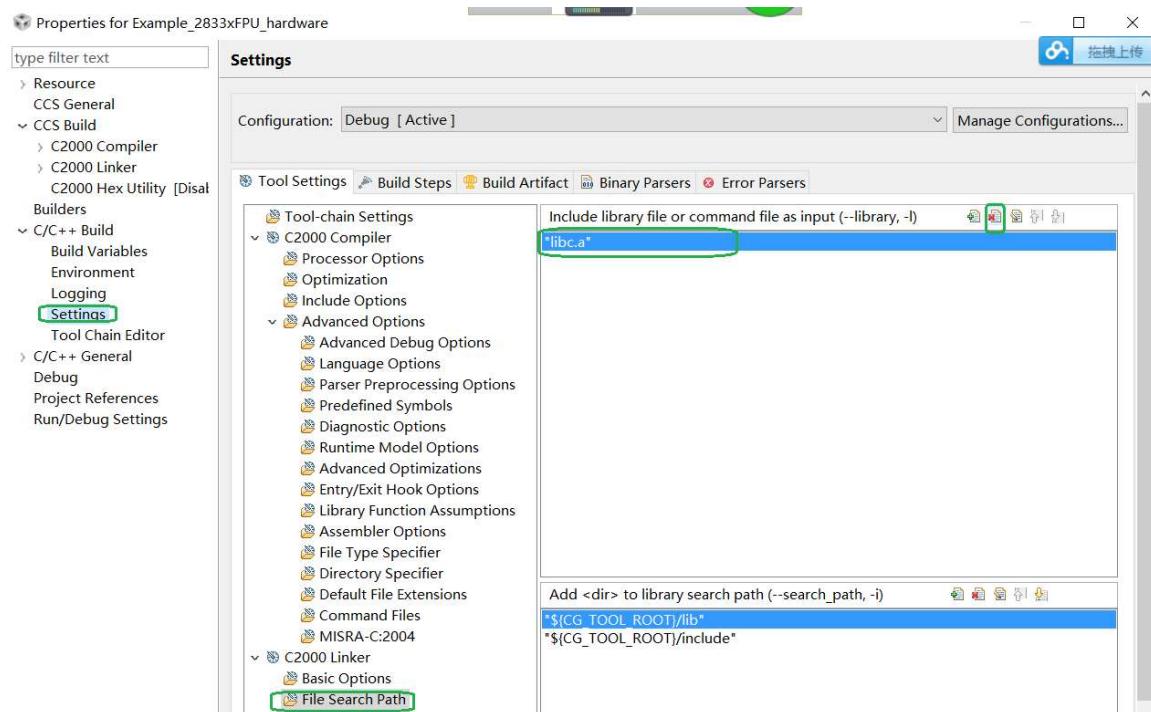
玻尔电子致力于 C2000 全系列开发平台及应用方案的推广



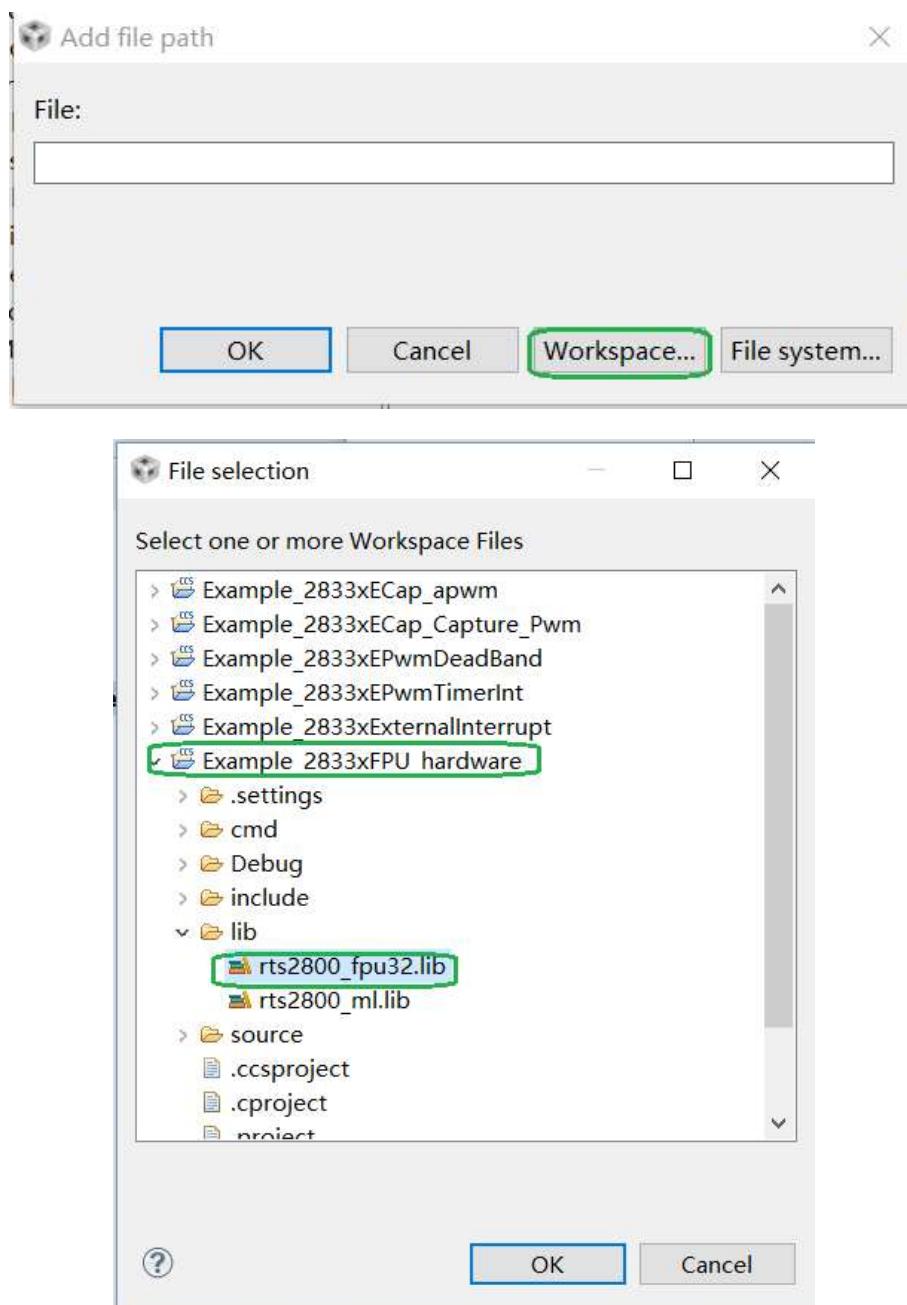
2. Project->Properties-> C/C++ Build window-> Basic Settings-> C2000 Linker Vx.x-> File

Search Path In "Include linker file or command file as input (-library, -l)" 选项, 单击绿色按钮添加 rts2800_fpu32.lib (run-time support library).

Step1:删除 libc.a 库



Step2:添加 rts2800_fpu32.lib



观察变量

y1

y2

FPU registers (optional)

实验 52：高分辨率 PWM 演示示例 (Example_2833xHRPWM)

程序说明：

这个例子修改 MEP 控制寄存器显示 edge displacement。由于 HRPWM 控制逻辑是 EPWM 模块的扩展，所有 EPwm 通道 (GPIO0, gpio2, gpio4, gpio6) 将具有良好的运动边缘。

1. 15MHz PWM (如果是 150 MHz SYSCLKOUT) 或者是 10MHz PWM (如果是 100MHz SYSCLKOUT)
ePWM1A 切换低/高在 MEP 上升沿
ePWM1B 在无 HRPWM 控制情况下切换低/高
2. 7.5MHz PWM (如果是 150 MHz SYSCLKOUT) 或者 5MHz PWM (如果是 100MHz SYSCLKOUT),
ePWM2A 切换低/高在 MEP 上升沿
ePWM2B 在无 HRPWM 控制情况下切换低/高
3. 15MHz PWM (如果是 150 MHz SYSCLKOUT) 或者是 10MHz PWM (如果是 100MHz SYSCLKOUT)
ePWM3A 切换为高/低当 MEP 控制在下降沿
ePWM3B 在无 HRPWM 控制情况下切换低/高
4. 7.5MHz PWM (如果是 150 MHz SYSCLKOUT) 或者 5MHz PWM (如果是 100MHz SYSCLKOUT),
ePWM4A 切换高/低在 MEP 控制的下降沿
ePWM4B 在无 HRPWM 控制情况下切换低/高

External Connections

用示波器观察 ePWM1–ePWM4 管脚：

ePWM1A/GPIO0
ePWM1B/GPIO1
ePWM2A/GPIO2
ePWM2B/GPIO3
ePWM3A/GPIO4
ePWM3B/GPIO5
ePWM4A/GPIO6
ePWM4B/GPIO7

实验 53：High Resolution PWM SF0 V5

This example modifies the MEP control registers to show edge

displacement due to the HRPWM control extension of the respective ePWM module.

Note:

By default, this example project is configured for floating-point math. All included libraries must be pre-compiled for floating-point math. Therefore, SF0_TI_Build_V5B_fpu.lib (compiled for floating-point) is included in the project instead of the SF0_TI_Build_V5B.lib (compiled for fixed-point). To convert the example for fixed-point math, follow the instructions in sfo_readme.txt in the /doc directory of the header files and peripheral examples package.

This program requires the DSP2833x header files, which include the following files required for this example: SF0_V5.h and SF0_TI_Build_V5B_fpu.lib (or SF0_TI_Build_V5B.lib for fixed point). This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V5 functions:

int SFO_MepEn_V5(int i); updates MEP_ScaleFactor[i] dynamically when HRPWM is in use.

Returns

- 1 when complete for the specified channel
- 0 if not complete for the specified channel
- 2 if there is a scale factor out-of-range error (MEP_ScaleFactor[n] differs from seed MEP_ScaleFactor[0] by more than +/-15)

int SFO_MepDis_V5(int i); updates MEP_ScaleFactor[i] when HRPWM is not used Returns

- 1 when complete for the specified channel
- 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details. All ePWM1A-6A channels will have fine edge movement due to the HRPWM logic

5MHz PWM (for 150 MHz SYSCLKOUT), ePWMxA toggle high/low with MEP control on rising edge

3.33MHz PWM (for 100 MHz SYSCLKOUT), ePWMxA toggle high/low with MEP control on rising edge Running the Application

1. **!!IMPORTANT!!** – in SF0_V5.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F28335, the maximum number of HRPWM channels is $6+1=7$, so set #define PWM_CH 7 in

- SFO_V5.h. (Default is 7)
2. Run this example at 150/100MHz SYSCLKOUT
 3. Load the Example_2833xHRPWM_SFO.gel and observe variables in the watch window
 4. Activate Real time mode
 5. Run the code
 6. Watch ePWM1-6 waveforms on a Oscilloscope
 7. In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilities (default). Observe the duty cycle of the waveform changes in fine MEP Steps

 8. In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilities. Observe the duty cycle of the waveform changes in coarse steps of 10nsec.

External Connections

Monitor the following pins on an oscilloscope:

ePWM1A is on GPIO00
ePWM2A is on GPIO02
ePWM3A is on GPIO04
ePWM4A is on GPIO06
ePWM5A is on GPIO08
ePWM6A is on GPIO10

Watch Variables

UpdateFine
MEP_ScaleFactor
EPwm1Regs.CMPA.all
EPwm2Regs.CMPA.all
EPwm3Regs.CMPA.all
EPwm4Regs.CMPA.all
EPwm5Regs.CMPA.all
EPwm6Regs.CMPA.all

实验 54: High Resolution PWM Symmetric Duty Cycle SFO V5

This example modifies the MEP control registers to show symmetric edge displacement due to the HRPWM control extension of the respective ePWM module.

Note:

By default, this example project is configured for floating-point math. All included libraries must be pre-compiled for floating-point

math. Therefore, SF0_TI_Build_V5B_fpu.lib (compiled for floating-point) is included in the project instead of the SF0_TI_Build_V5B.lib (compiled for fixed-point). To convert the example for fixed-point math, follow the instructions in sfo_readme.txt in the /doc directory of the header files and peripheral examples package.

This program requires the DSP2833x header files, which include the following files required for this example: SF0_V5.h and SF0_TI_Build_V5B_fpu.lib (or SF0_TI_Build_V5B.lib for fixed point). This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V5 functions:

int SFO_MepDis_V5(int i); updates MEP_ScaleFactor[i] when HRPWM is not used

Returns

- 1 when complete for the specified channel
- 0 if not complete for the specified channel

Channel ePWM1A will have fine edge movement due to the HRPWM logic when the duty cycle is altered. Channel ePWM1B has a fixed 50% duty cycle.

5MHz PWM (for 150 MHz SYSCLKOUT), ePWM1A toggle high/low with MEP control on rising edge

3.33MHz PWM (for 100 MHz SYSCLKOUT), ePWM1A toggle high/low with MEP control on rising edge

Running the Application

1. !!!IMPORTANT!! - in SF0_V5.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F28335, the maximum number of HRPWM channels is 6+1=7, so set #define PWM_CH 7 in SF0_V5.h. (Default is 7)

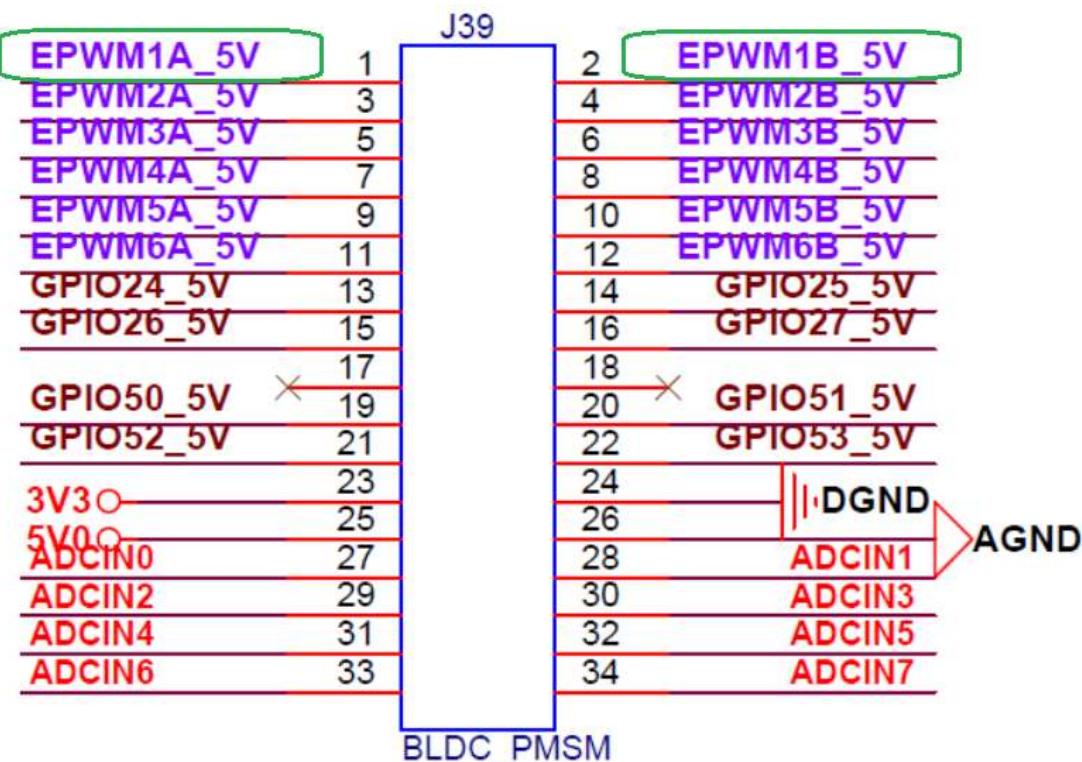
2. Run this example at 150/100MHz SYSCLKOUT
3. Watch ePWM1 waveforms on a Oscilloscope

External Connections

Monitor the following pins on an oscilloscope:

ePWM1A is on GPIO0

ePWM1B is on GPIO1



Watch Variables

MEP_ScaleFactor
EPwm1Regs.CMPA.all
EPwm1Regs.CMPB.all

实验 55: FFT 例子 (Example_2833x_FFT)

傅立叶变换是一种线性的积分变换，常在将信号在时域（或空域）和频域之间变换时使用，在物理学和工程学中有许多应用。因其基本思想首先由法国学者约瑟夫·傅里叶系统地提出，所以以其名字来命名以示纪念。在不同的研究领域，傅立叶变换具有多种不同的变体形式，如连续傅立叶变换和离散傅立叶变换。最初傅立叶分析是作为热过程的解析分析的工具被提出的。

傅里叶变换是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，比如正弦波、方波、锯齿波等，傅里叶变换用正弦波作为信号的成分。

定义

$f(t)$ 是 t 的函数，如果 t 满足狄里赫莱条件：具有有限个间断点；具有有限个极值点；绝对可积。则有①式成立。称为积分运算 $f(t)$ 的 傅立叶变换，

②式的积分运算叫做 $F(\omega)$ 的 傅立叶逆变换。 $F(\omega)$ 叫做 $f(t)$ 的 像函数，
 $f(t)$ 叫做

$F(\omega)$ 的 像原函数。 $F(\omega)$ 是 $f(t)$ 的像。 $f(t)$ 是 $F(\omega)$ 原像。

$$\textcircled{1} \quad F(\omega) = \mathcal{F}[f(t)] = \int_{-\infty}^{\infty} f(t)e^{-i\omega t} dt.$$

$$\textcircled{2} \quad f(t) = \mathcal{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega)e^{i\omega t} d\omega.$$

Phenomenon

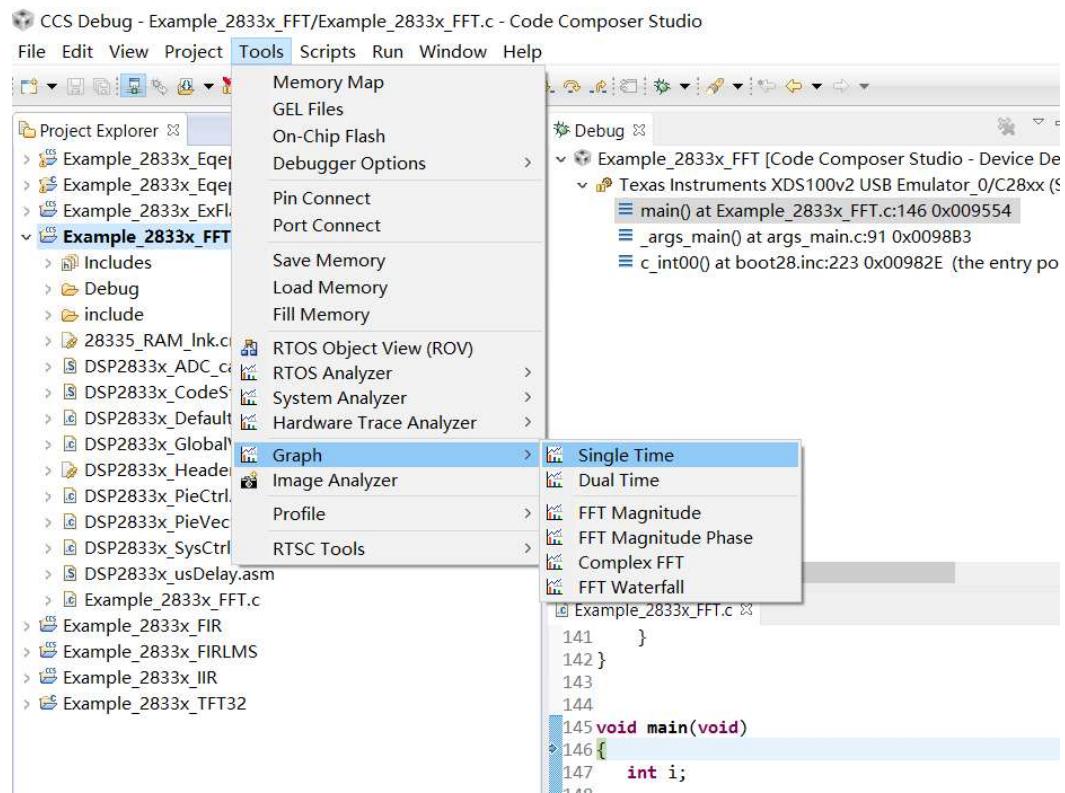
Step1: 首先进入调试界面

```
#define SAMPLENUMBER 128
int INPUT[SAMPLENUMBER],DATA[SAMPLENUMBER];

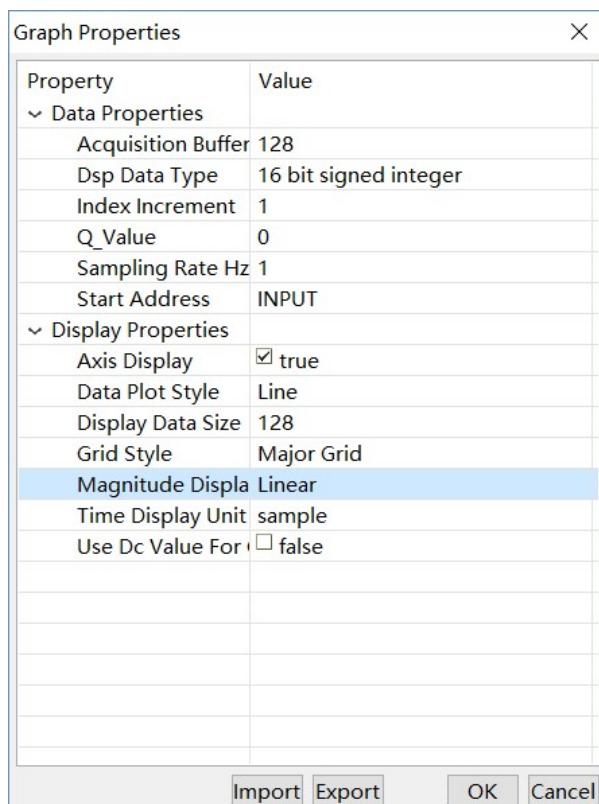
CCS Debug - Example_2833x_FFT/Example_2833x_FFT.c - Code Composer Studio
File Edit View Project Tools Scripts Run Window Help
Project Explorer Debug Variables Registers
Example_2833x_Eqep_freqcal Example_2833x_Eqep_pos_speed Example_2833x_ExFlash_Sram_Core Example_2833x_FFT [Active - Debug]
Includes Debug Name
Debug include i
28335_RAM_Ink.cmd DSP2833x_ADC_cal.asm DSP2833x_CodeStartBranch.asm DSP2833x_DefaultLsr.c DSP2833x_GlobalVariableDefs.c DSP2833x_Headers_nonBIOS.cmd DSP2833x_PieCtrl.c DSP2833x_PieVect.c DSP2833x_SysCtrl.c DSP2833x_usDelay.asm Example_2833x_FFT.c Example_2833x_FIR Example_2833x_FIRLMS Example_2833x_IIR Example_2833x_TFT32
Texas Instruments XDS100v2 USB Emulator_0/C28xx (Suspended)
main() at Example_2833x_FFT.c:146 0x009554
_args_main() at args_main.c:91 0x0098B3
c_int00() at boot28.inc:223 0x00982E (the entry point was
Example_2833x_FFT.c
141 }
142 }
143
144
145 void main(void)
146 {
147     int i;
148
149 // Step 1. Initialize System Control:
150 // PLL, WatchDog, enable Peripheral Clocks
151 // This example function is found in the DSP2833x_SysCtrl.c file.
```

Step2: 添加变量到观察窗口

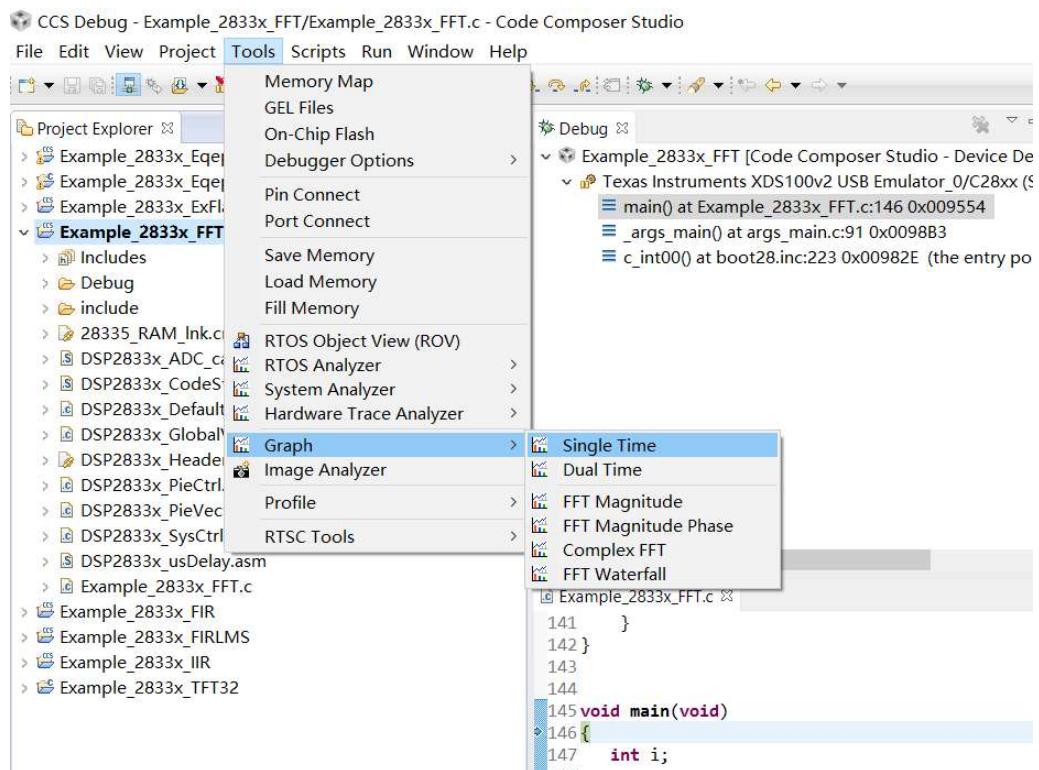
添加 `INPUT[SAMPLENUMBER]` 到观察窗口



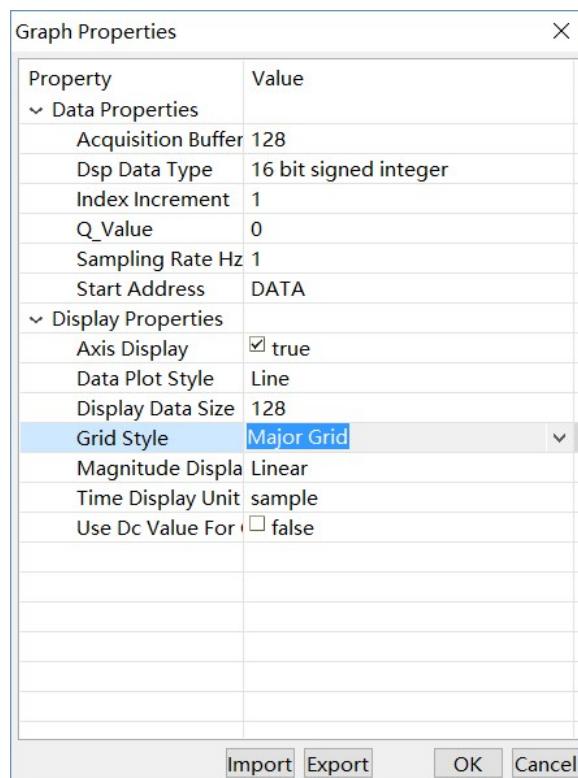
单击 OK 按钮



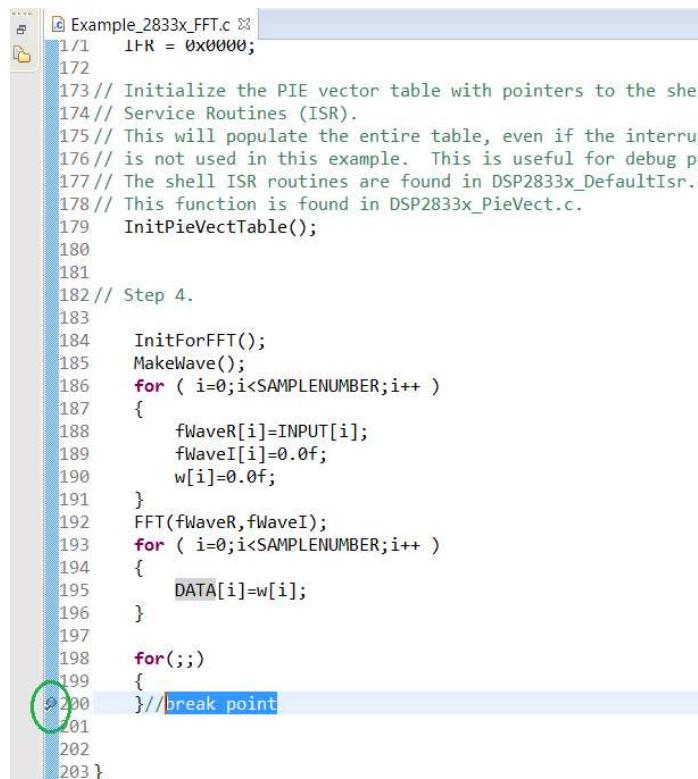
添加 DATA[SAMPLENUMBER] 到图像窗口



单击 OK 按钮



Step3：设置断点（双击断点处即可）

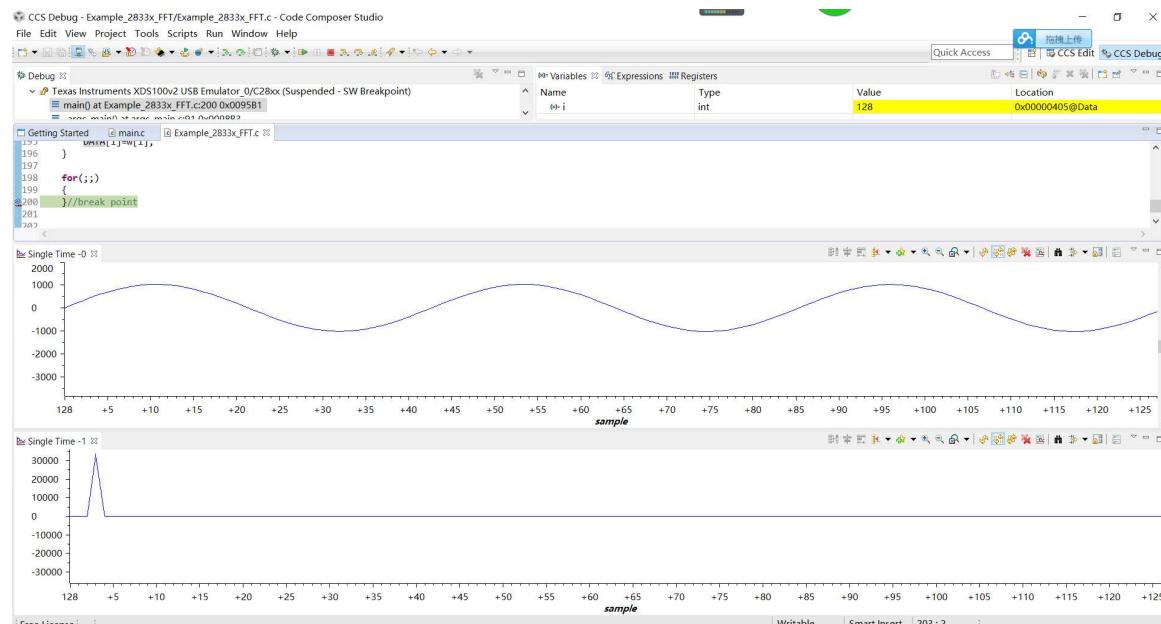


```

1/1    IFR = 0x0000;
172
173 // Initialize the PIE vector table with pointers to the shell
174 // Service Routines (ISR).
175 // This will populate the entire table, even if the interrupt
176 // is not used in this example. This is useful for debug purposes.
177 // The shell ISR routines are found in DSP2833x_DefaultIsr.c
178 // This function is found in DSP2833x_PieVect.c.
179     InitPieVectTable();
180
181
182 // Step 4.
183
184     InitForFFT();
185     MakeWave();
186     for ( i=0;i<SAMPLENUMBER;i++ )
187     {
188         fWaveR[i]=INPUT[i];
189         fWaveI[i]=0.0f;
190         w[i]=0.0f;
191     }
192     FFT(fWaveR,fWaveI);
193     for ( i=0;i<SAMPLENUMBER;i++ )
194     {
195         DATA[i]=w[i];
196     }
197
198     for(;;)
199     {
200         } //break point
201
202 }
203

```

Step4：运行程序



实验 56：有限长单位冲击响应 Example_2833x_FIR

FIR(Finite Impulse Response)滤波器：有限长单位冲激响应滤波器，是数字信号处理系统中最基本的元件，它可以在保证任意幅频特性的同时具有严格的线性相位。

线性相频特性，同时其单位抽样响应是有限长的，因而滤波器是稳定的系统。因此，FIR 滤波器在通信、图像处理、模式识别等领域都有着广泛的应用。

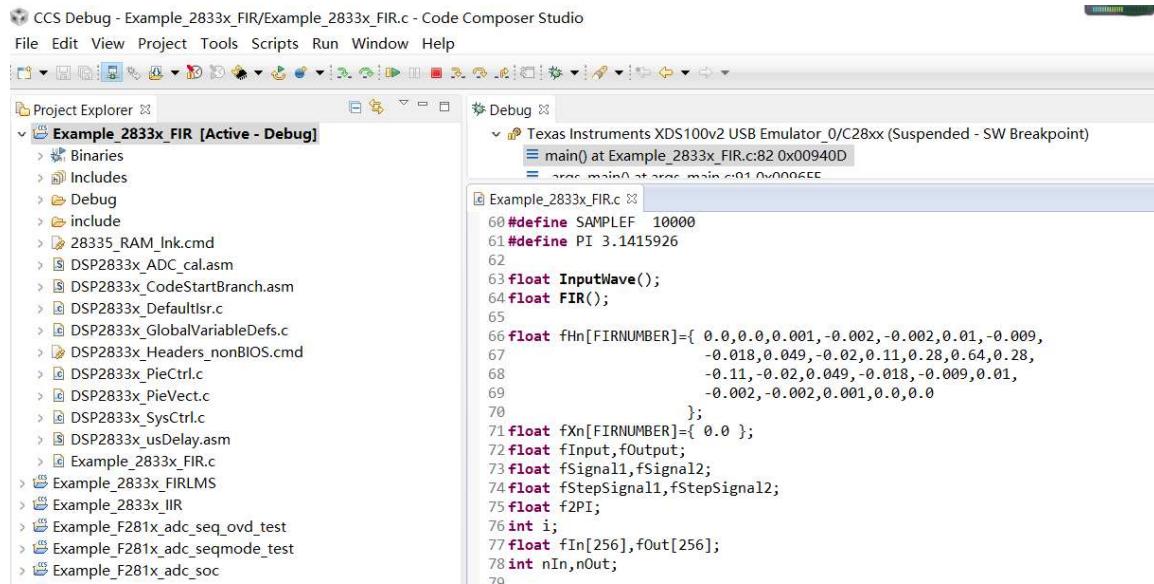
算法原理：离散信号序列通过一个离散滤波系统，得到离散输出信号，如果滤波系统的单位脉冲响应为 $h(n)$ ，信号序列为 $x(n)$ ，输出信号为 $y(n)$ ，则使用运算关系式表达他们之间的关系如下：

$$y(n) = x(n) * h(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m)$$

实验现象

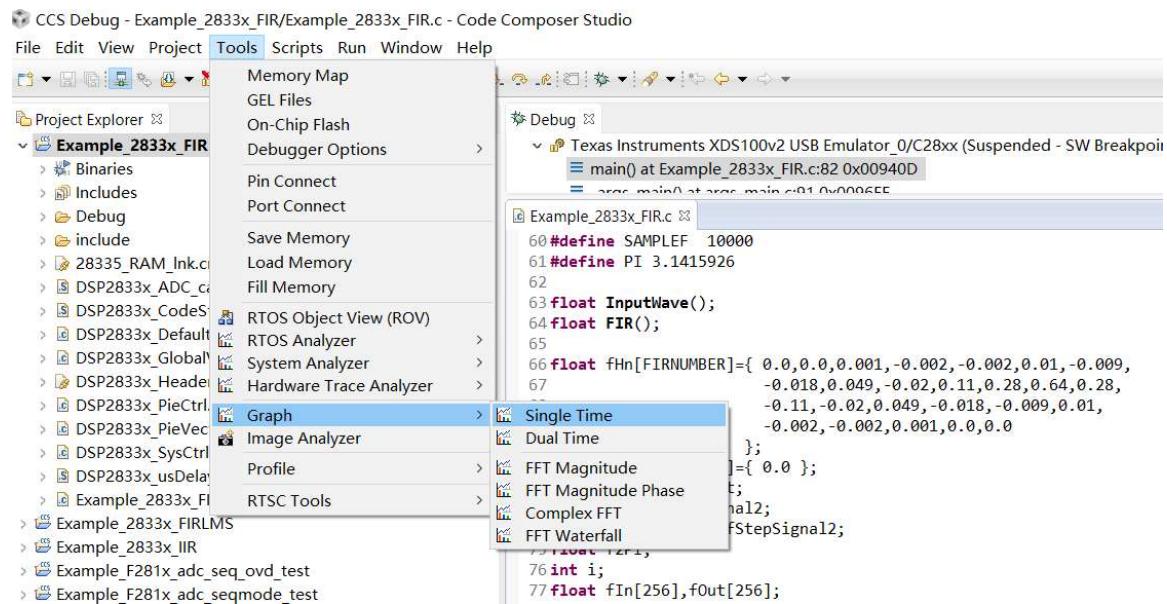
Step1：进入调试接口

```
float fIn[256], fOut[256];
```

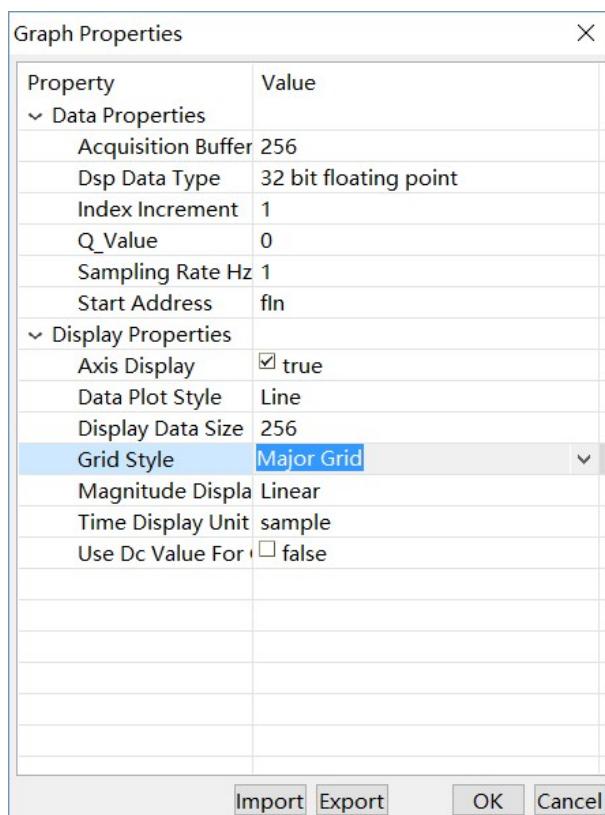


Step2：添加变量到观察窗口

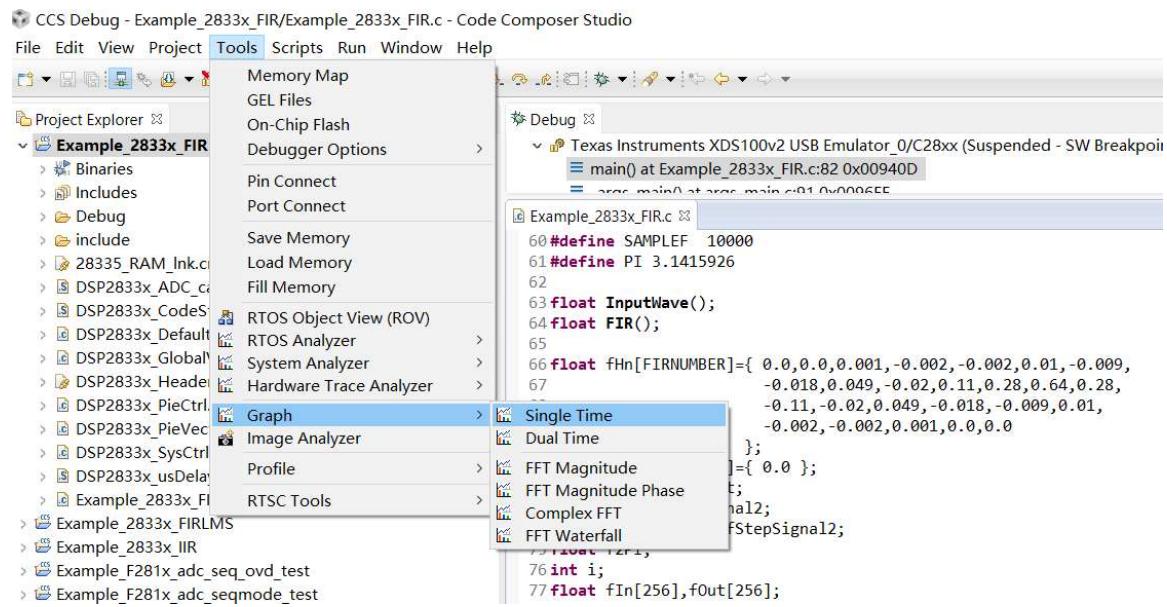
添加 `fIn[256]` 到图像窗口



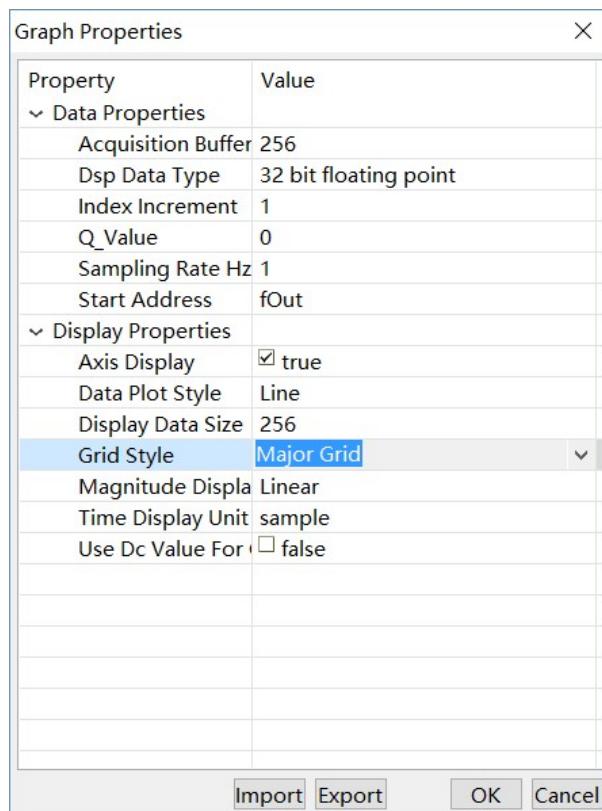
进行如下图像配置：



添加 `fOut[256]` 到图形窗口

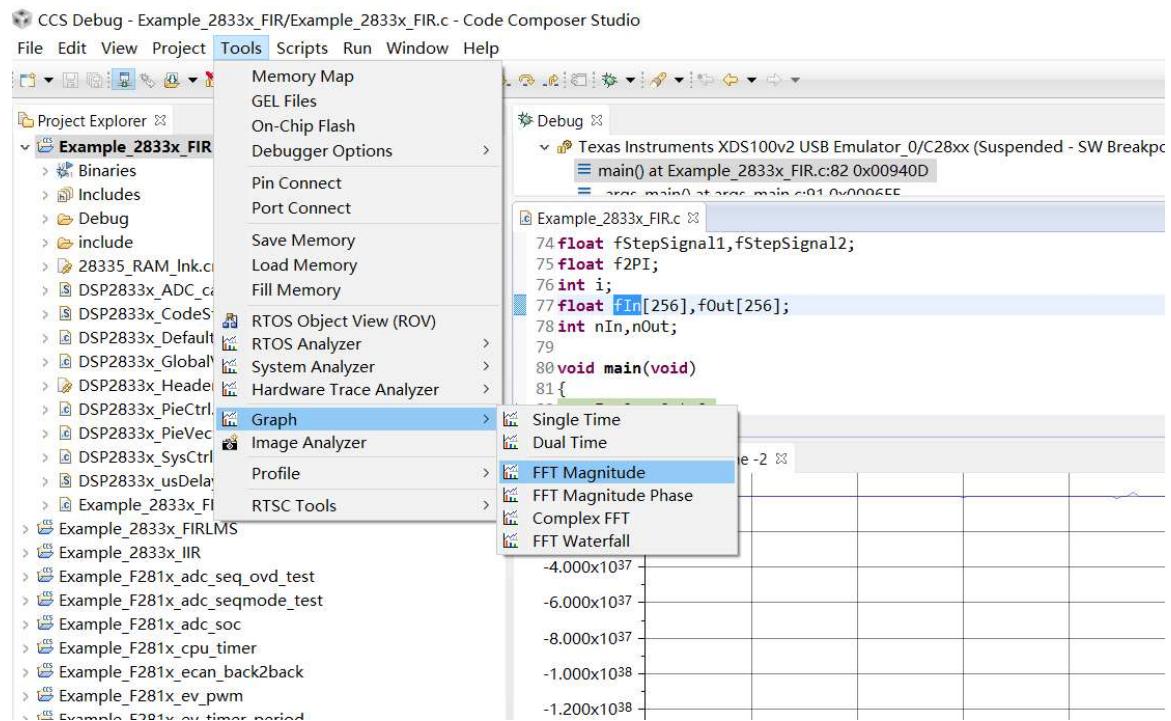


配置相应的设置如下：

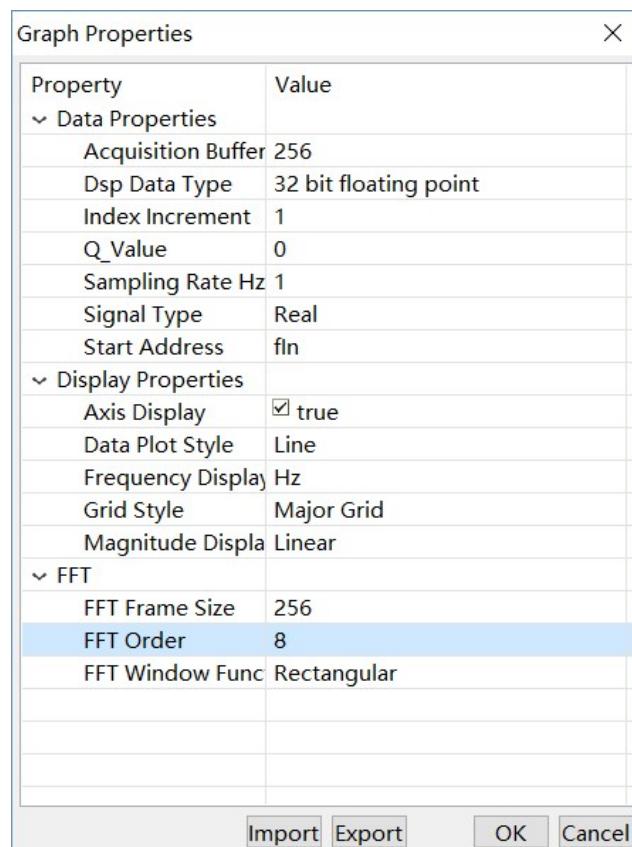


Step3：添加变量到 FFT Magnitude

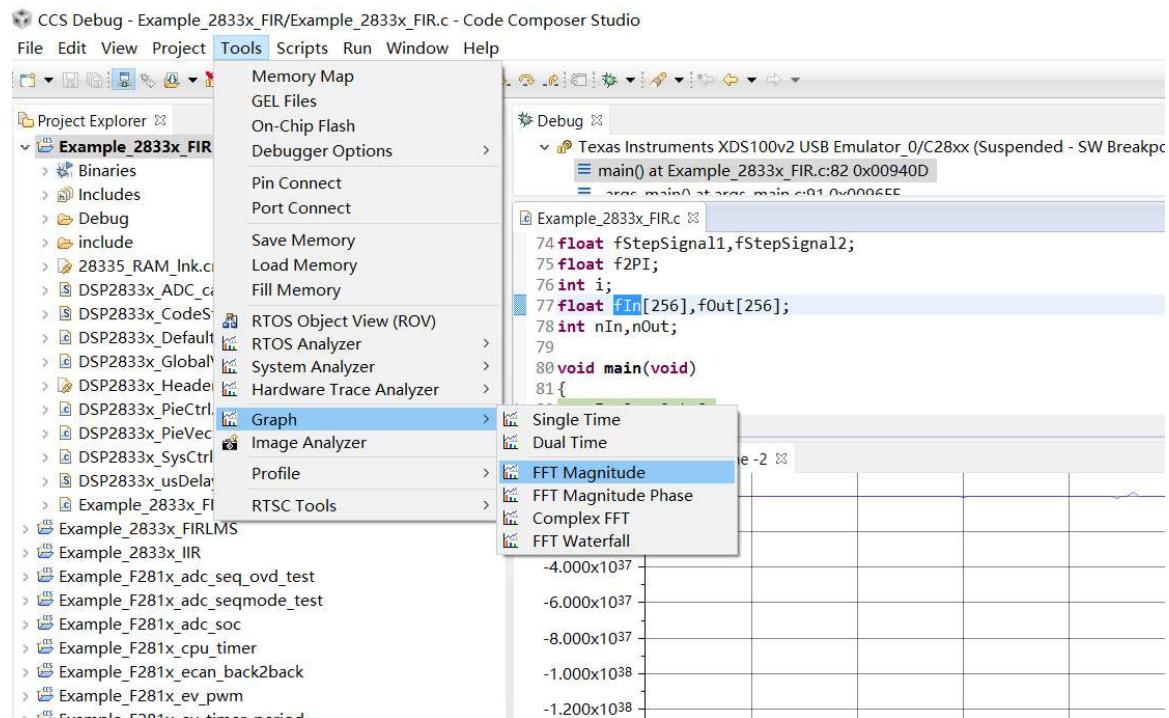
添加 `fIn[256]` 到 FFT Magnitude



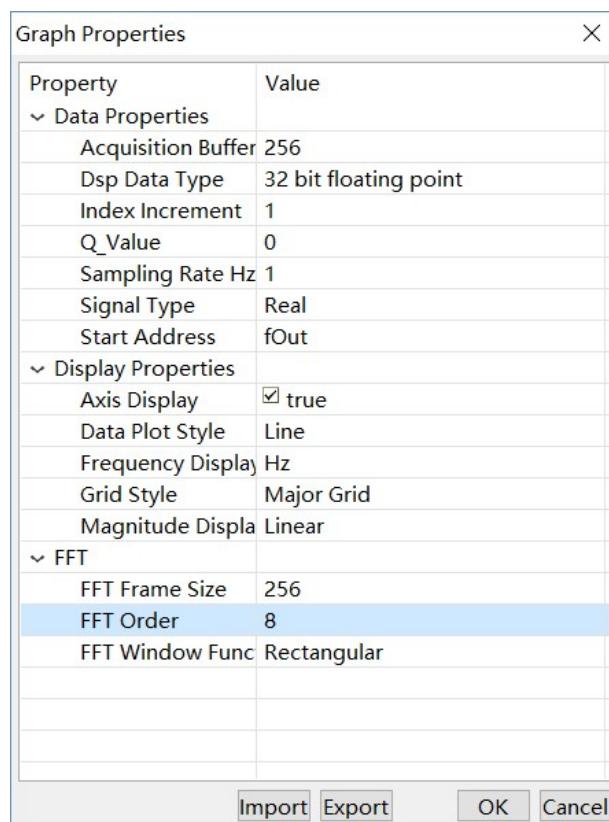
进行如下图所示的配置：



添加 `fOut[256]` 到 FFT Magnitude



按照如下图片进行配置：



Step4：设置断点

CCS Debug - Example_2833x_FIR/Example_2833x_FIR.c - Code Composer Studio

File Edit View Project Tools Scripts Run Window Help

Project Explorer: Example_2833x_FIR [Active - Debug]

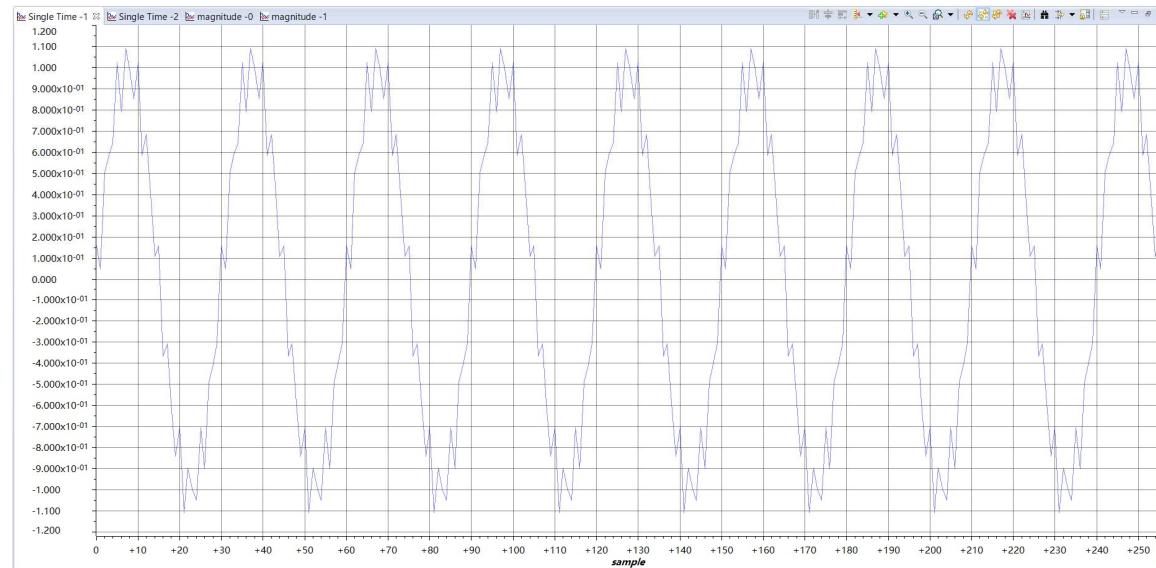
Debug: Texas Instruments XDS100v2 USB Emulator_0/C28xx (Suspended - SW Breakpoint)

```

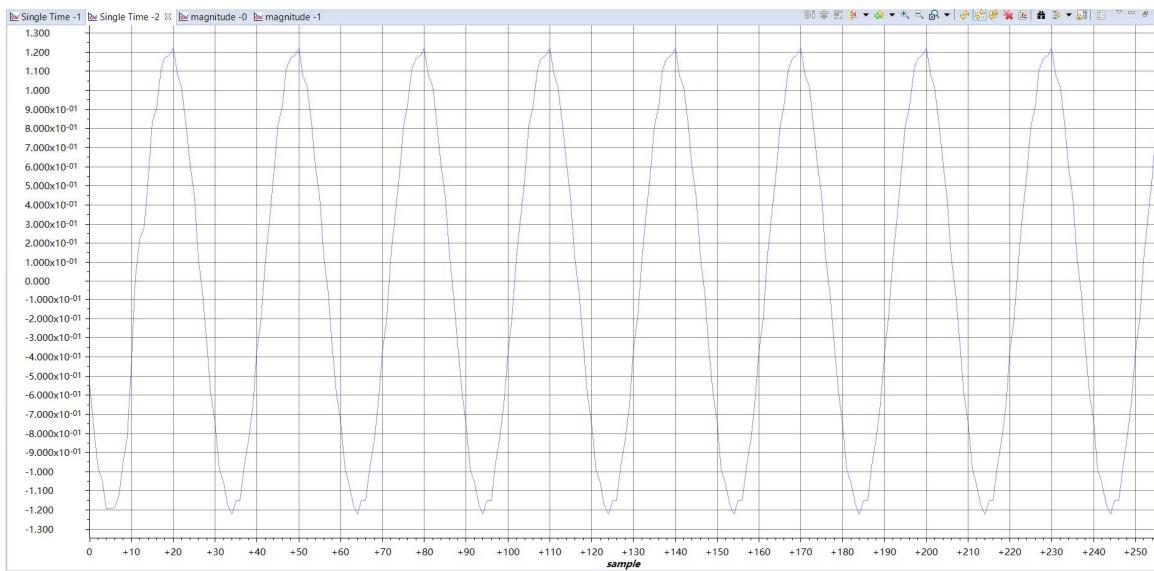
123 // Step 4.
124     while(1)
125     {
126         fInput=InputWave();
127         fIn[nIn]=fInput;
128         nIn++; nIn%256;
129         fOutput=FIR();
130         fOut[nOut]=fOutput;
131         nOut++;
132         if ( nOut>=256 )
133         {
134             nOut=0; /* 请在此句上设置软件断点 */
135         }
136     }
137
138
139
140 }
```

Step5：全速运行

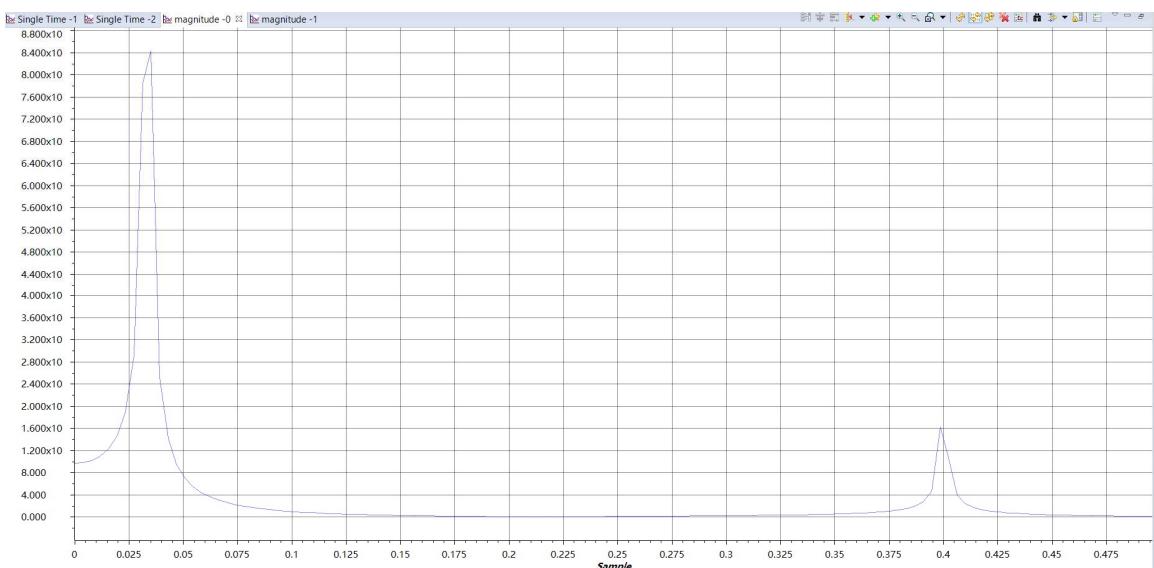
fIn 的时域波形如下：



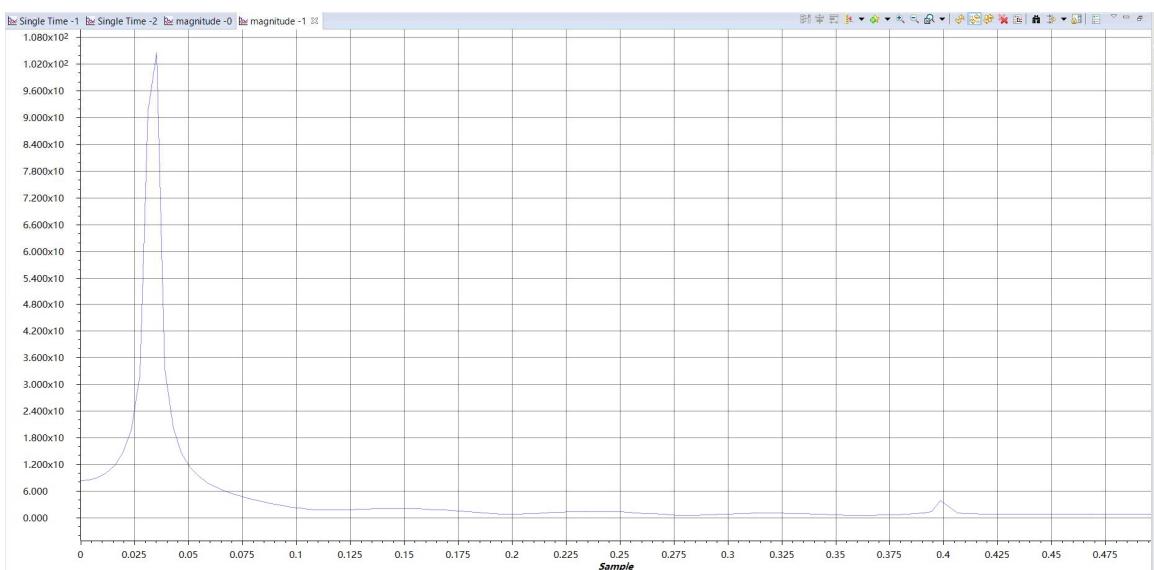
fOut 的时域波形如下：



fIn 的频率波形如下



fOut 的频率波形如下



实验 57：无限长脉冲响应（Example_2833x_IIR）

IIR 滤波器即无限长脉冲响应（Infinite Impulse Response）滤波器，它具有反馈，一般认为具有无限的脉冲响应。IIR 滤波器为非线性相位（双线性变换法），对于非线性相位会造成影响。

IIR 滤波器的特点

1 iir 数字滤波器的系统函数可以写成封闭函数的形式。

2 iir 数字滤波器采用递归型结构，即结构上带有反馈环路。iir 滤波器运算结构通常由延时、乘以系数和相加等基本运算组成，可以组合成直接型、正准型、级联型、并联型四种结构形式，都具有反馈回路。由于运算中的舍入处理，使误差不断累积，有时会产生微弱的寄生振荡。

3 iir 数字滤波器在设计上可以借助成熟的模拟滤波器的成果，如巴特沃斯、契比雪夫和椭圆滤波器等，有现成的设计数据或图表可查，其设计工作量比较小，对计算工具的要求不高。在设计一个 iir 数字滤波器时，我们根据指标先写出模拟滤波器的公式，然后通过一定的变换，将模拟滤波器的公式转换成数字滤波器的公式。

4 iir 数字滤波器的相位特性不好控制，对相位要求较高时，需加相位校准网络。

现象

Step1：进入调试界面

```
Float fIn[256], fOut[256];
```

```

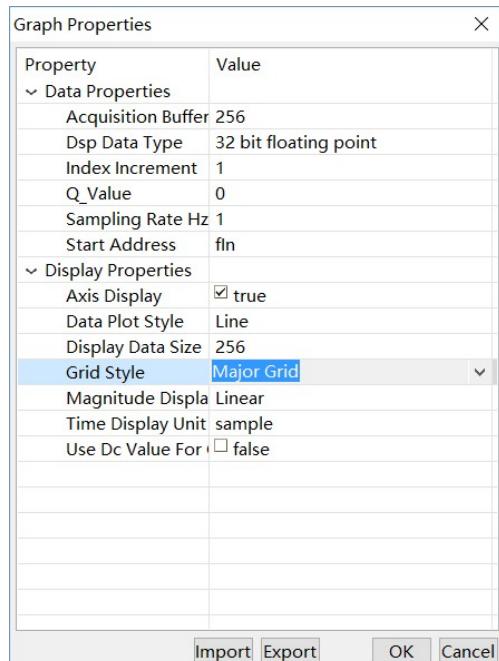
CCS Debug - Example_2833x_IIR/Example_2833x_IIR.c - Code Composer Studio
File Edit View Project Tools Scripts Run Window Help
Project Explorer Debug
Example_2833x_IIR [Active - Debug]
Binaries Includes Debug include
28335_RAM.Ink.cmd DSP2833x_ADC_cal.asm DSP2833x_CodeStartBranch.asm DSP2833x_Defaults.r
DSP2833x_GlobalVariableDefs.c DSP2833x_Headers_nonBIOS.cmd DSP2833x_PieCtrl.c DSP2833x_PieVect.c
DSP2833x_SysCtrl.c DSP2833x_usDelay.asm Example_2833x_IIR.c Example_F281x_adc_seq_ovd_test
Example_F281x_adc_seqmode_test Example_F281x_adc_soc Example_F281x_cpu_timer Example_F281x_ecan_back2back
Example_F281x_ev_pwm
Texas Instruments XDS100v2 USB Emulator _0/C28xx (Suspended - SW Breakpoint)
main() at Example_2833x_IIR.c:76 0x0009316
boot28inc Example_2833x_IIR.c
62 float fAn[IIRNUMBER]={ 0.1122,0.1122 };
63 float fXn[IIRNUMBER]={ 0.0 };
64 float fStepSignal1,fStepSignal2;
65 float fInput,fOutput;
66 float fSignal1,fSignal2;
67 float fStepSignal1,fStepSignal2;
68 float f2PI;
69 int i;
70 float fin[256],fout[256];
71 int nIn,nOut;
72
73 void main(void)
74 {
75     nIn=0; nOut=0;
76     f2PI=2*PI;
77     fSignal1=0.0;
78     fSignal2=PI*0.1;
79 //    fStepSignal1=2*PI/30;
80 //    fStepSignal2=2*PI*1.4;
81 //    fStepSignal1=2*PI/50;
82     fStepSignal2=2*PI/2.5;
83 }

```

Step2：添加变量到图形窗口

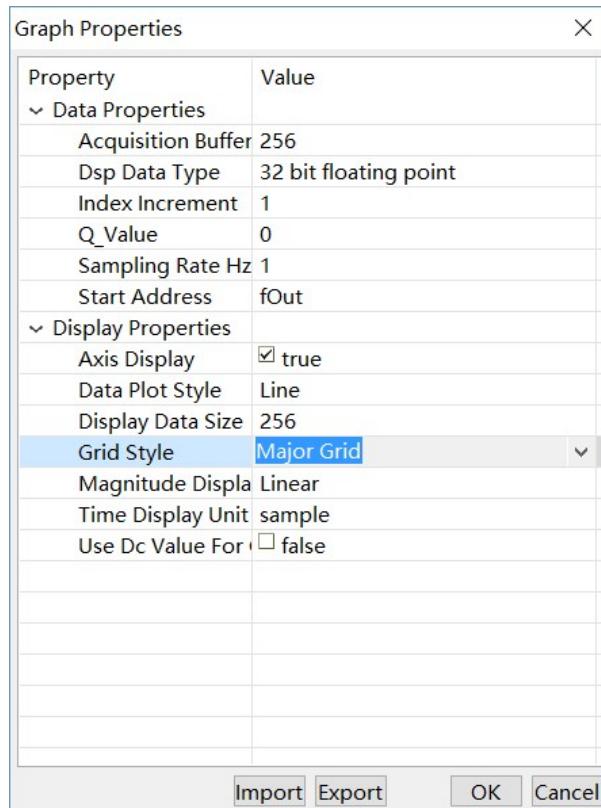
添加 `fIn[256]` 到图像窗口

按照如下图形进行配置：



添加 `fOut[256]` 到图形窗口

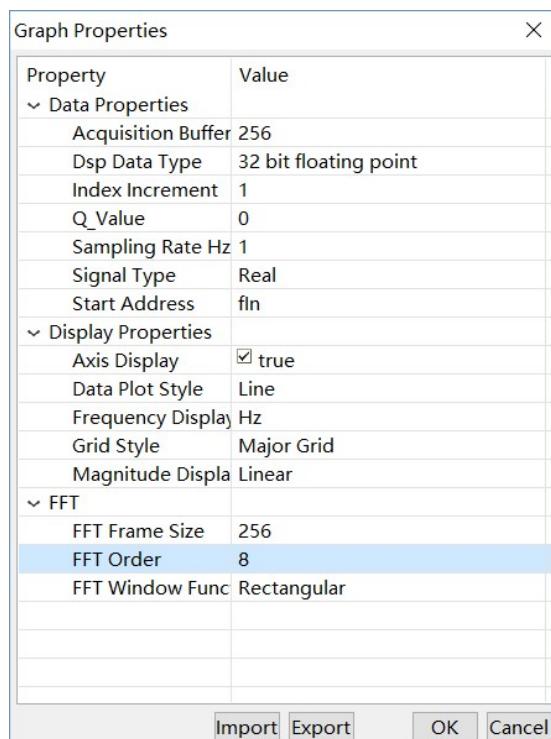
按照如下图形进行配置：



Step3：添加变量到 FFT Magnitude

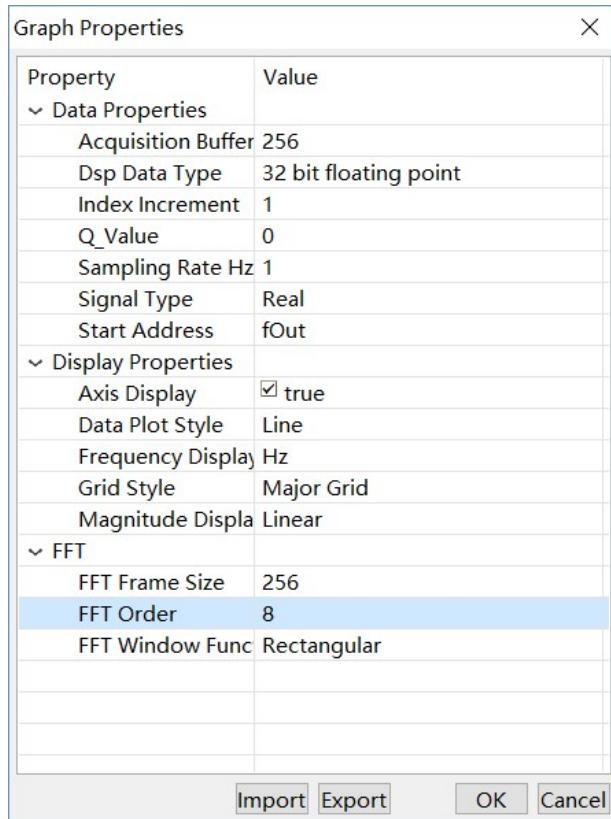
添加 `fIn[256]` 到 FFT Magnitude

按照如下图像进行配置：

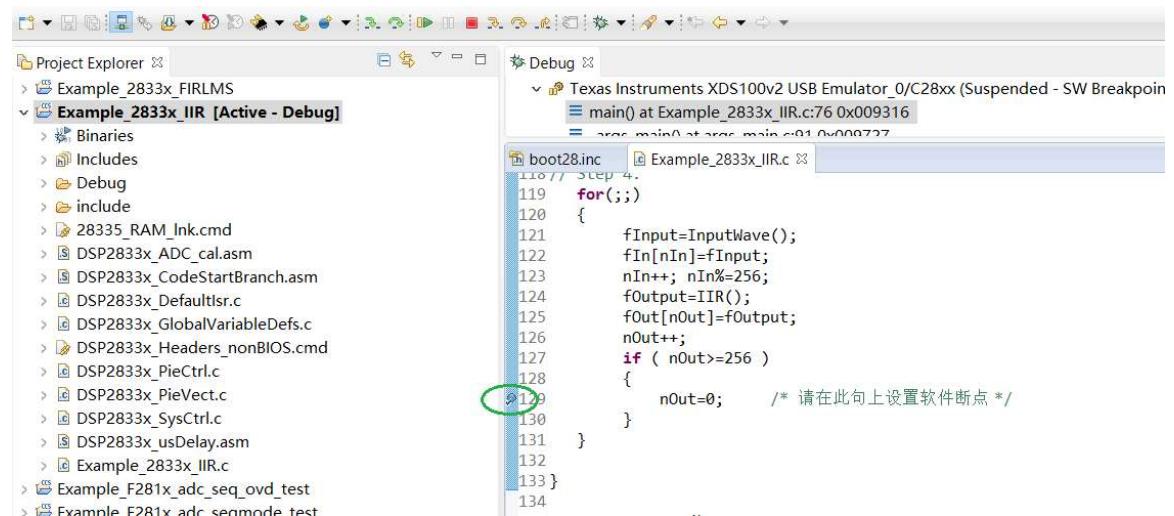


添加 `fOut[256]` 到 FFT Magnitude

按照如下图像进行配置：

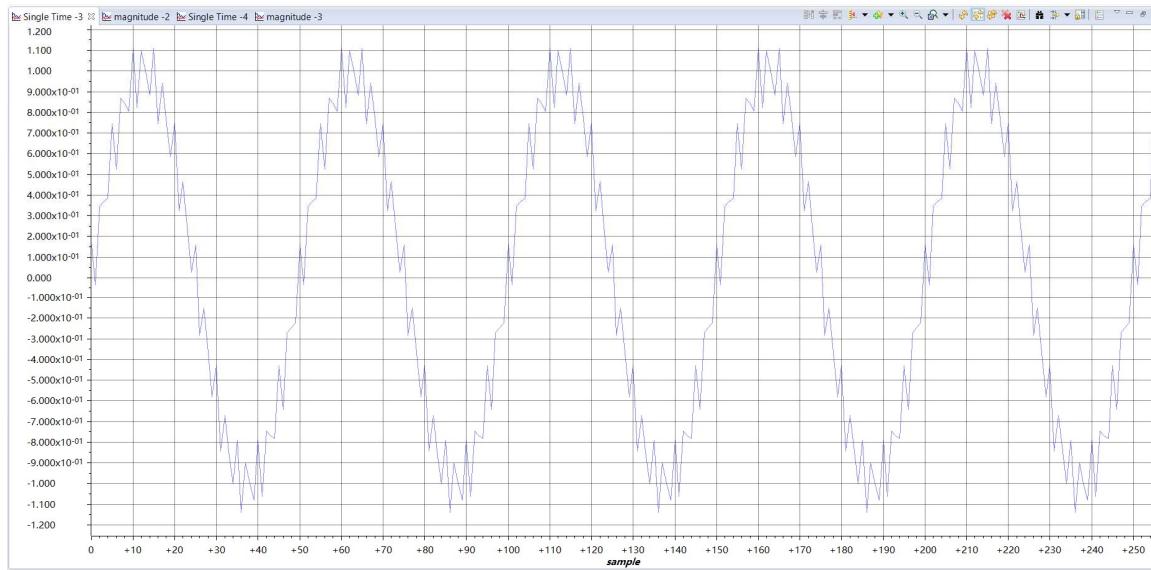


Step4：添加调试断点（在断点处双击）

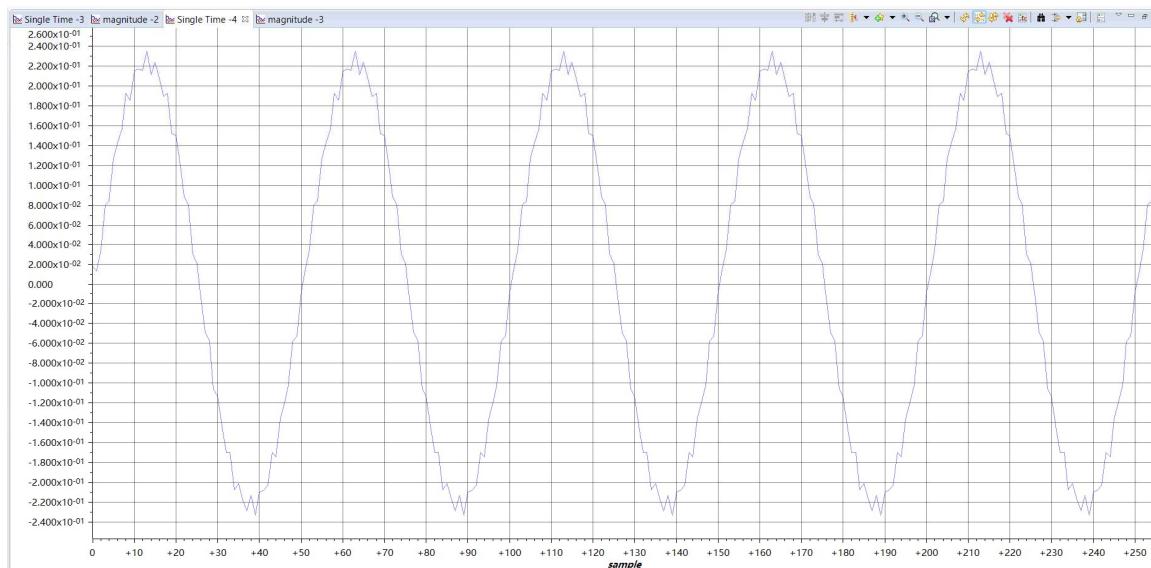


Step5：全速运行

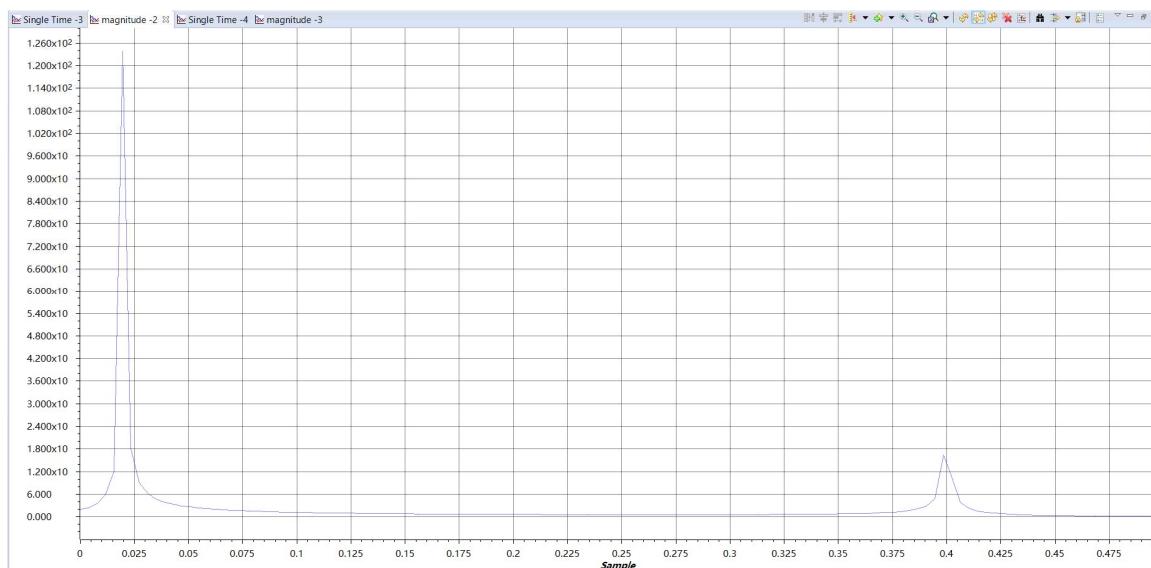
fIn 的时域波形如下图所示：



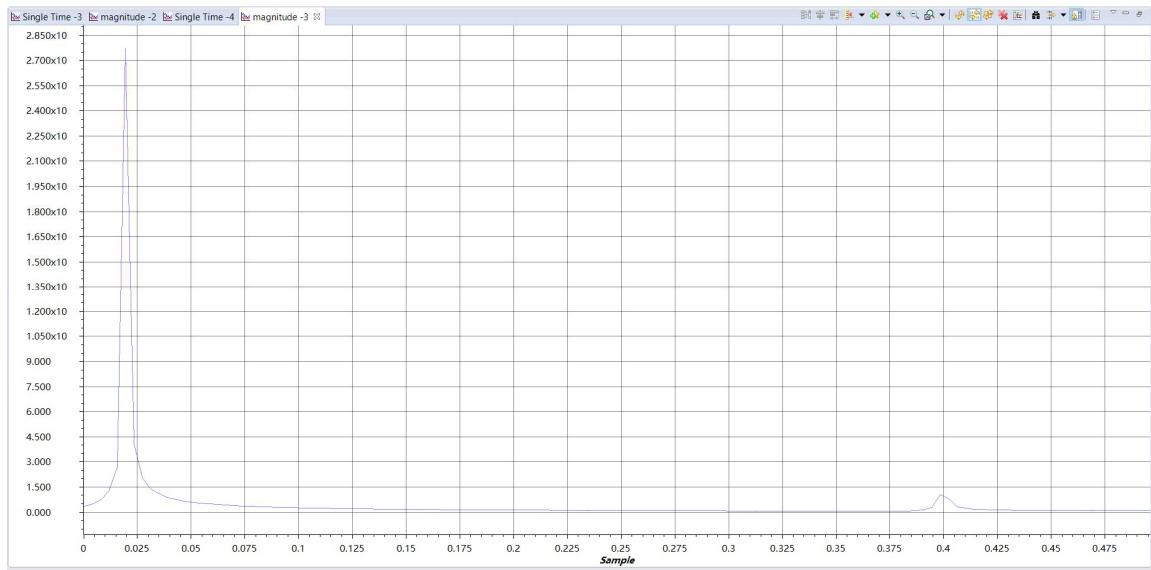
fOut 的时域波形如下图所示：



fIn 的频率波形如下图所示



fOut 的频率波形如下图所示



实验 58：自适应滤波器（Example_2833x_FIRLMS）

自适应滤波器实际上是一种能够自动调整本身参数的特殊维纳滤波器，在设计时不需要预先知道关于输入信号和噪声的统计特性，它能够在工作过程中逐步“了解”或估计出所需的统计特性，并以此为依据自动调整自身的参数，以达到最佳滤波效果。一旦输入信号的统计特性发生变化，它又能够跟踪这种变化，自动调整参数，使滤波器性能重新达到最佳。自适应滤波器由参数可调的数字滤波器(或称为自适应处理器)和自适应算法两部分组成。参数可调数字滤波器可以是 FIR 数字滤波器或 IIR 数字滤波器，也可以是格型数字滤波器。输入信号 $x(n)$ 通过参数可调数字滤波器后产生输出信号(或响应) $y(n)$ ，将其与参考信号(或称期望响应) $d(n)$ 进行比较，形成误差信号 $e(n)$ ，并以此通过某种自适应算法对滤波器参数进行调整，最终使 $e(n)$ 的均方值最小。尽管自适应滤波器具有各种不同的算法和结构，但是，其最本质特征是始终不变的。这种最本质的特

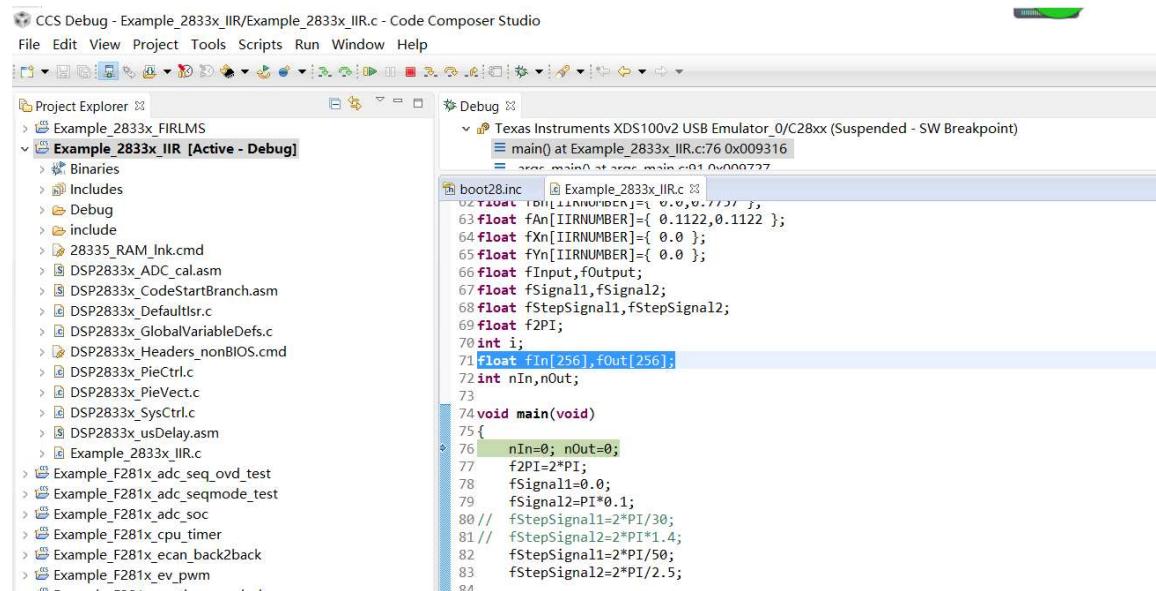
征可以概括为：自适应滤波器依据用户可以接受的准则或性能规范，在未知的而且可能是时变的环境中正常运行，而无须人为的干预。本章主要讨论的是基于维纳滤波器理论的最小均方(LMS)算法，可以看到 LMS 算法的主要优点是算法简单、运算量小、易于实现；其主要缺点是收敛速度较慢，而且与输入信号的统计特性有关。自适应线性滤波器是一种参数可自适应调整的有限冲激响应(FIR)数字滤波器，具有非递归结构形式。因为它的分析和实现比较简单，所以在大多数自适应信号处理系统中得到了广泛应用。

(更多的理论知识参考工程下的：自适应理论原理.pdf)

现象：

Step1：进入调试界面

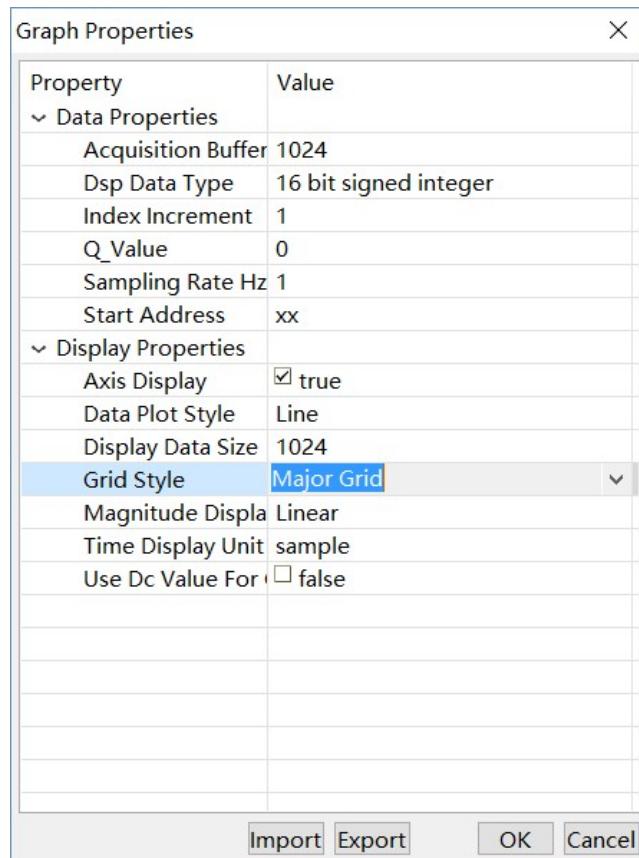
```
#define INPUTNUMBER 1024
int xx[INPUTNUMBER],rr[INPUTNUMBER],wc[INPUTNUMBER];
```



Step2：添加变量到图形窗口

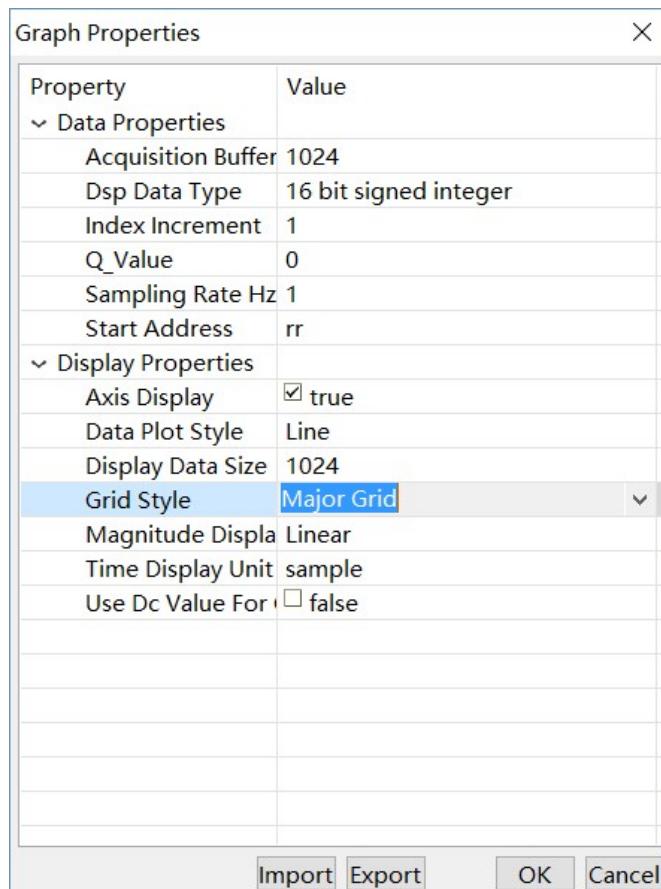
添加 xx[INPUTNUMBER] 到图形窗口

按照如下图形进行配置



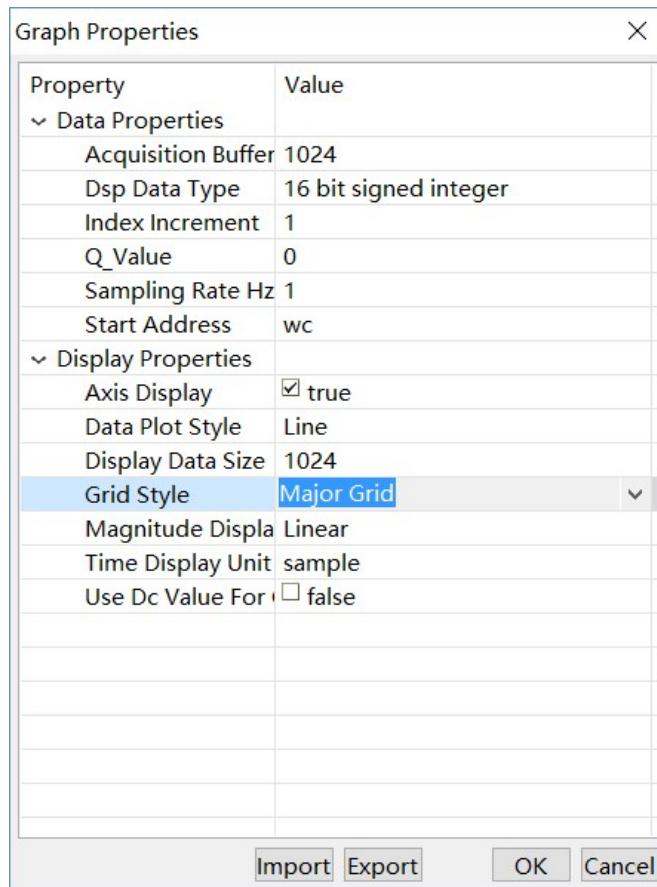
添加 rr[INPUTNUMBER] 到图形窗口

按照如下图片进行配置：



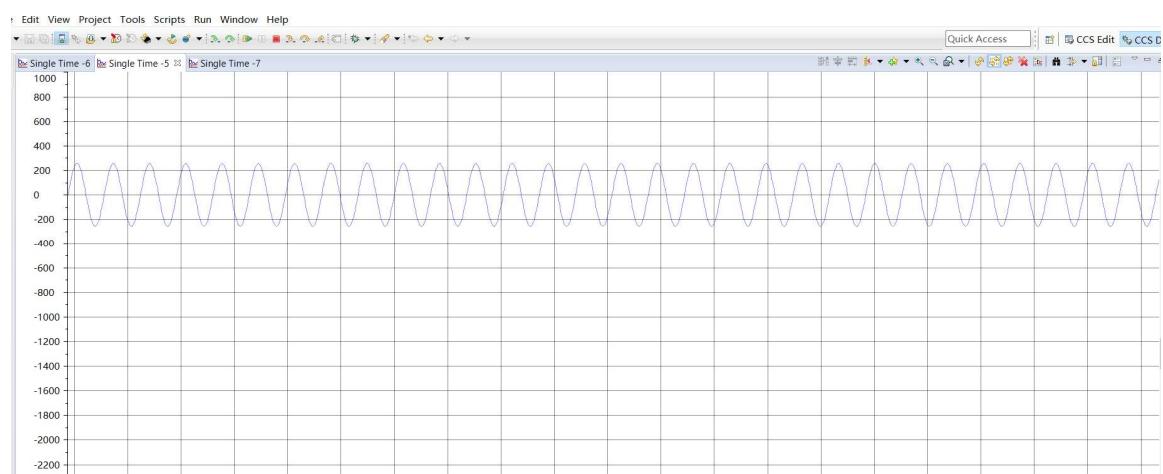
添加 wc[INPUTNUMBER] 到图形窗口

按照如下图形进行配置：

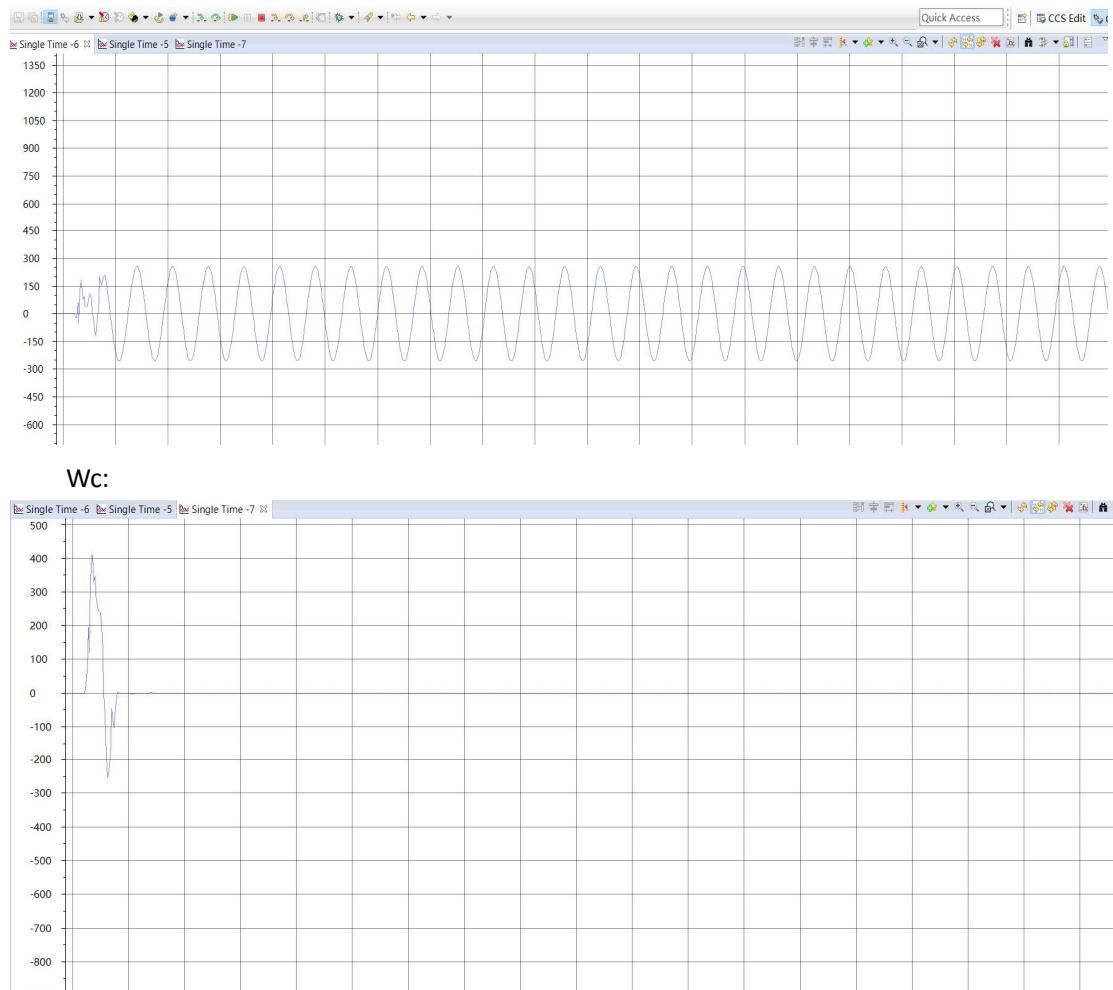


Step3: 全速运行()，等待 10 s。暂停运行().

Xx:



Rr:



实验 59: FIR 滤波器实验 (Example_2833x_FIR_V2)

一、实验目的:

✧ FIR 滤波器原理。

二、实验设备

(1) PC 机一台;

(2) XDS100v2 或 XDS100V3(隔离)仿真器一套;

(3) SXD28335 开发板一套;

三、实验步骤：

FIR(Finite Impulse Response) 滤波器：有限长单位冲激响应滤波器，是数字信号处理系统中最基本的元件，它可以在保证任意幅频特性的同时具有严格的线性相频特性，同时其单位抽样响应是有限长的，因而滤波器是稳定的系统。因此，FIR 滤波器在通信、图像处理、模式识别等领域都有着广泛的应用。

算法原理：离散信号序列通过一个离散滤波系统，得到离散输出信号，如果滤波系统的单位脉冲响应为 $h(n)$ ，信号序列为 $x(n)$ ，输出信号为 $y(n)$ ，则使用运算关系式表达他们之间的关系如下：

$$y(n) = x(n) * h(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m)$$

输入信号与系统响应之间的运算关系为线性卷积。

根据这样的关系，我们至少有两种方法实现 FIR：

1、直接卷积法：根据卷积的迭代运算直接计算输入信号序列与滤波系数的卷积，得到滤波输出信号，此法运算关系简单，但耗时较长；

2、快速卷积法：有时域与频域的对于关系，使用 FFT 算法实现 FIR，此法与直接卷积法相比，相同的点数下运算时间短，而且点数越多相对时间就越短，但是算法关系复杂。

本实验使用第一种方法

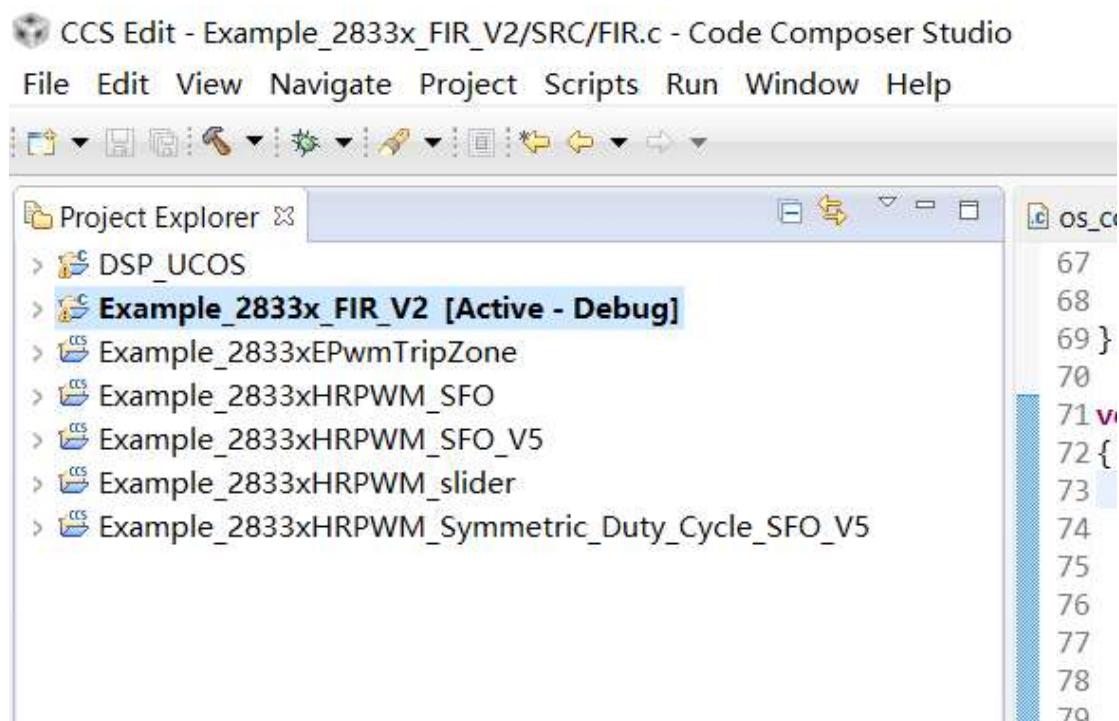
验证方法：使用一段合成的 256 点三次方波信号作为采样后的信号序列，设计一个窗函数滤波除掉二次与三次谐波，输出基次谐波应

该为一段正弦波。为确定合成信号的输出频率，将该合成信号经过 DA 输出，在示波器上观察到其频率为 420Hz，即其基波频率为 420Hz。

确定滤波器的截止频率为 500Hz，采样频率为 128000Hz，滤波器阶数为 32 阶，采用汉宁窗。

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

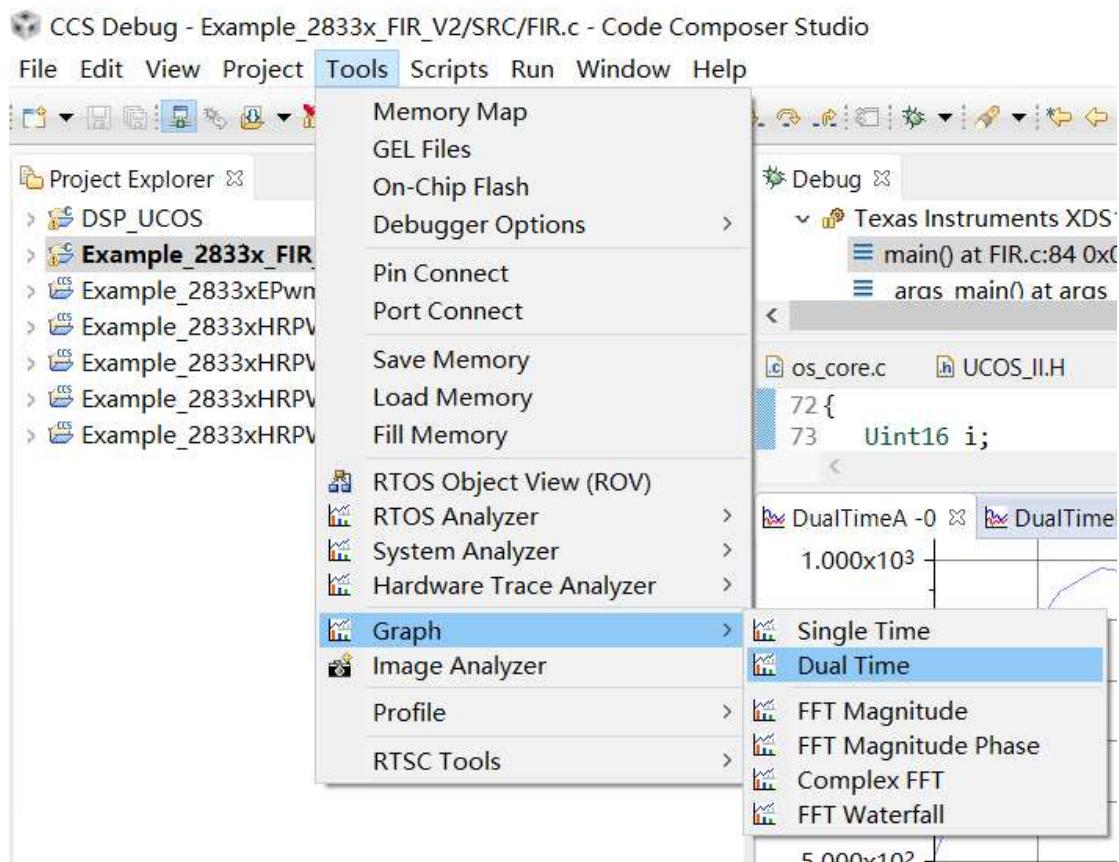


然后单击图中红色的方框处的调试按钮

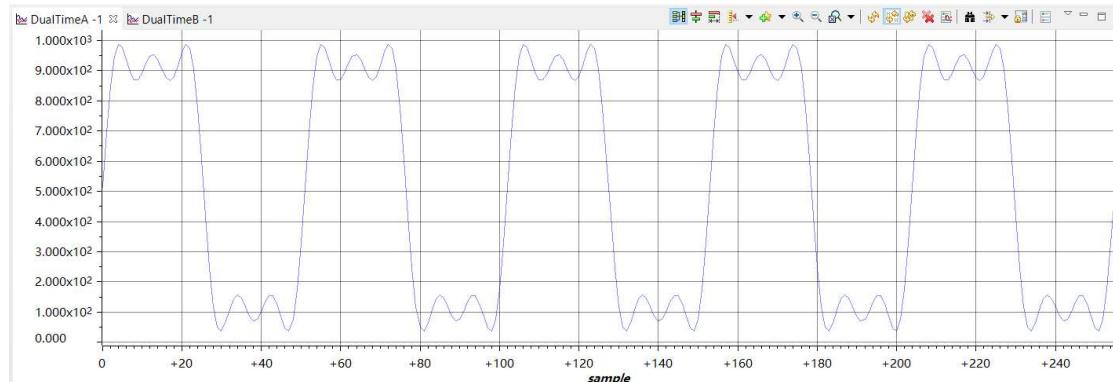


(3) 实验现象如下图所示：

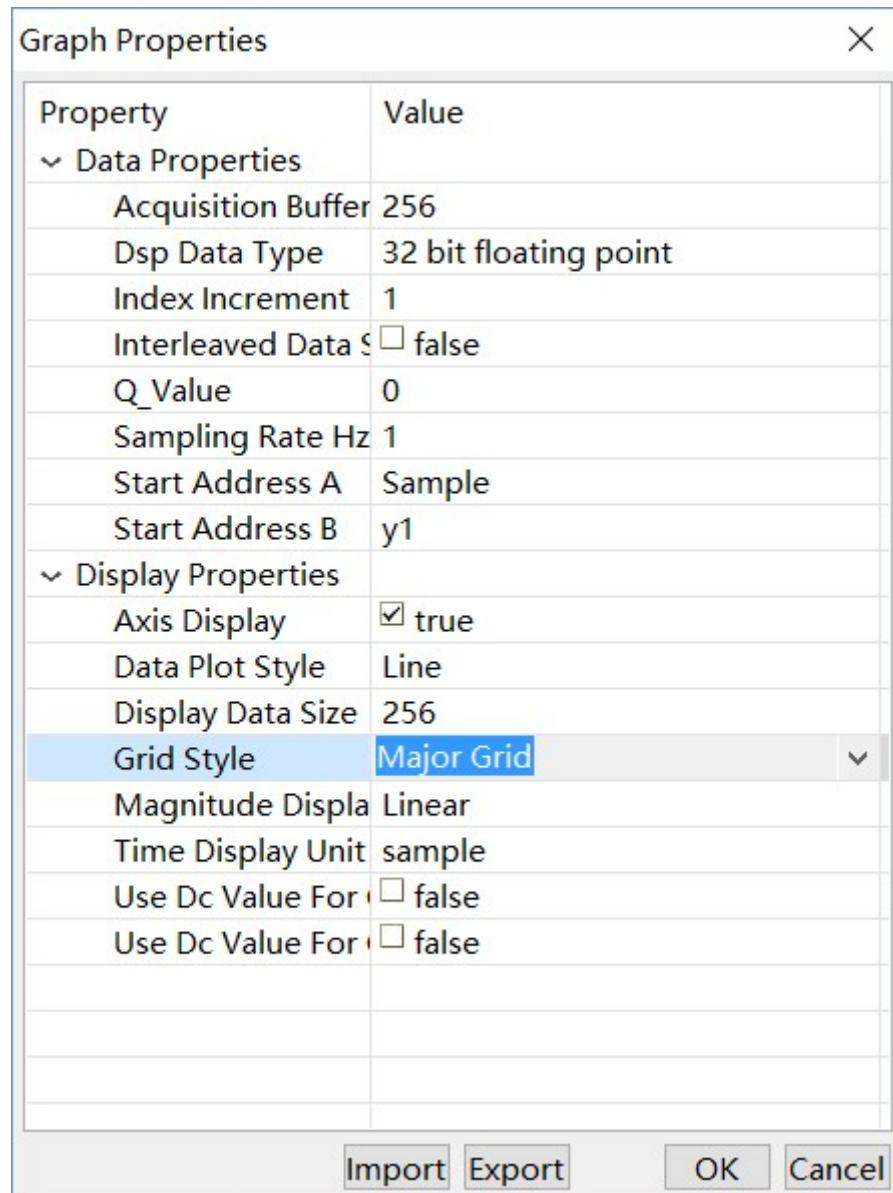
打开 FIR 滤波器例程 FIR 加载到开发板中运行，设置如下：



该信号为合成的三次方波信号。



接着我们设置如下：



上面的波形为输入的三次方波信号，下面的波形为经过 32 阶 FIR 滤波后的输出正弦信号波形。

我们可以清晰的看到，一个周期的方波与一个周期的输出正弦信号是对应的，即基波频率就是方波频率。该实验说明该 FIR 算法已经滤除了二次与三次信号。

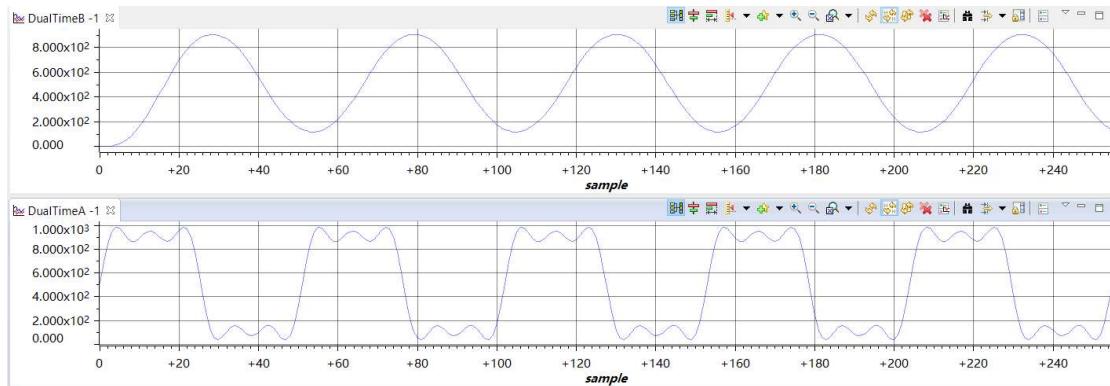


图 3 实验现象

实验 60：卷积实验 (Example_2833x_Convolution)

一、实验目的：

✧ 了解卷积原理。

二、实验设备

- (1) PC 机一台；
- (2) XDS100v2 或 XDS100V3(隔离) 仿真器一套；
- (3) SXD28335 开发板一套；

三、实验步骤：

在泛函分析中，卷积(卷积)、旋积或摺积(英语：Convolution)是通过两个函数 f 和 g 生成第三个函数的一种数学算子，表征函数 f 与经过翻转和平移与 g 的重叠部分的累积。如果将参加卷积的一个函数看作区间的指示函数，卷积还可以被看作是“滑动平均”的推广。

算法原理：该例程实现的是离散线性卷积。两个卷积因子 $x(n)$ 、 $y(n)$ ，输出卷积结果 $f(n)$ ，若 $x(n)$ 序列长度为 L 、 $y(n)$ 序列长度为

M，则输出序列 f(n) 长度为 L+M-1。

离散卷积公式如下：

$$f(n) = x(n) * y(n) = \sum_{m=-\infty}^{\infty} x(m)y(n-m)$$

或

$$f(n) = x(n) * y(n) = \sum_{m=-\infty}^{\infty} y(m)x(n-m)$$

卷积是一种运算关系，例如，FIR 滤波器的单位脉冲响应与输入信号序列的时域运算关系即为线性卷积。

由于实际的信号都是因果序列，长度是有限的，因此上面的公式可以变换为如下所示：

$$f(n) = x(n) * y(n) = \sum_{m=0}^{L-1} x(m)y(n-m)$$

或

$$f(n) = x(n) * y(n) = \sum_{m=0}^{M-1} y(m)x(n-m)$$

如果直接将求和公式展开计算将会变得比较繁琐，因此，可以用一种计算简便、易于手工计算的方法——对位相乘求和法计算卷积。具体计算方法可见本文件夹 PPT 格式资料《使用对位相乘求和法求卷积》。

本算法例程就使用了对位相乘求和法计算卷积。以《使用对位相乘求和法求卷积》资料里面的例 7-2-6 为例进行计算。

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：

The screenshot shows the CCS Edit interface. The Project Explorer panel on the left lists several projects under the active project 'Example_2833x_Convolution [Active - Debug]'. The code editor on the right displays a file named 'boot28.inc' with C code. A red box highlights the green play button icon in the toolbar at the top.

```
boot28.inc
72 {
73     Uint16 :
74     InitSys
75     InitPie
76     IER = 0;
77     IFR = 0;
78     for(i=0
79     {
80         Samp
81         // DE
82     }
83     LinearC
84     while(1
85     {
86         // S
87         /* d
88         yn=
89         for
90         for
91         out
92         if(
93         else
94     }
95 }
96
97
98 //=====
99 // No more
```

然后单击图中红色的方框处的调试按钮

(3) 实验现象如下图所示：

打开卷积例程 Convolution 加载到开发板中运行，将变量 y1 添加到观察窗口，由于例程将卷积输出放在数组 y1 中，通过观察 y1，就可以看见卷积输出序列。如下图所示：

Expression	Type	Value	Address
y1	unsigned int[6]	0x0000C11F@Data	0x0000C11F@Data
↳ [0]	unsigned int	12	0x0000C11F@Data
↳ [1]	unsigned int	17	0x0000C1200@Data
↳ [2]	unsigned int	16	0x0000C121@Data
↳ [3]	unsigned int	10	0x0000C122@Data
↳ [4]	unsigned int	4	0x0000C123@Data
↳ [5]	unsigned int	1	0x0000C124@Data
Add new expression			

图 3 实验现象

计算结果与例 7-2-6 的计算结果是一致的。

如果用户需要对例程进行使用，只需要修改以下几项：

1、#define a 3 //x(n) 的序列长度

#define b 4 //h(n) 的序列长度

修改 a 与 b 的预定义值，该为新的两个卷积序列长度。

2、Uint16 x1[a]={3, 2, 1}; //卷积序列 1

Uint16 h1[b]={4, 3, 2, 1}; //卷积序列 2

在初始化的时候将两个卷积序列进行赋值，或者在进行运算之前进行赋值。重新编译一下程序即可；

实验 61：UCOS 操作系统例程(Example_2833x_ucoyii_run_in_flash_nonBIOS)

一、实验目的：

◆ 了解 UCOS 操作系统

二、实验设备

(1) PC 机一台；

(2) XDS100v2 或 XDS100V3(隔离)仿真器一套；

(3) SXD28335 开发板一套；

三、实验步骤：

(1) 首先将 CCS6.0 开发环境打开；

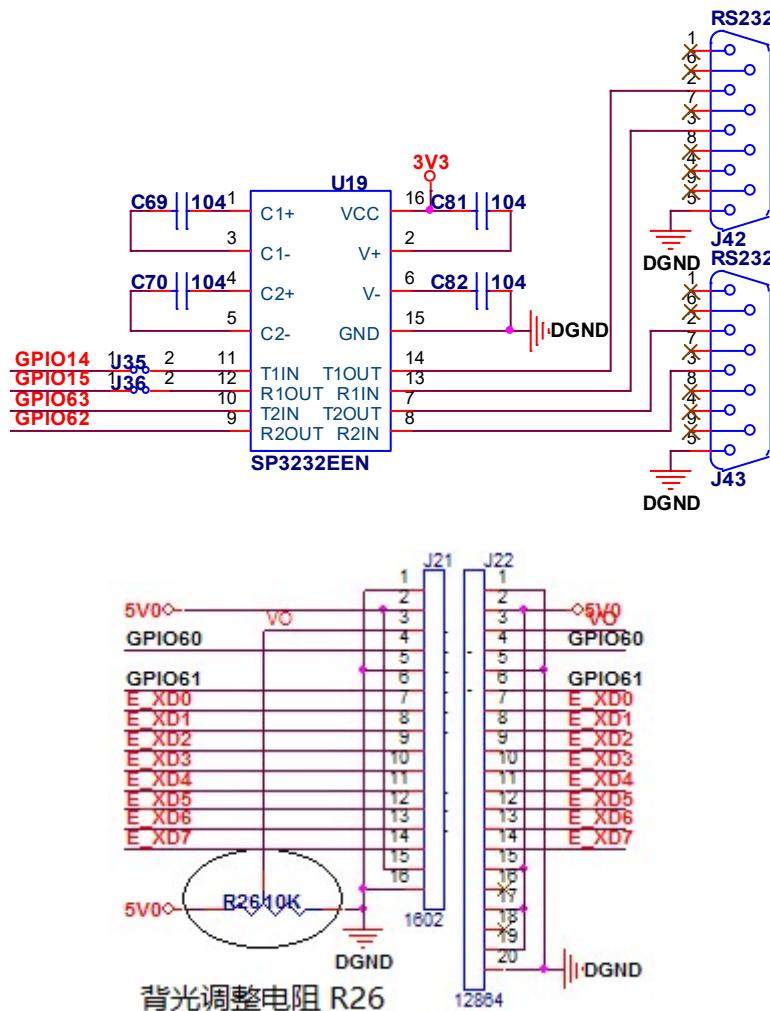


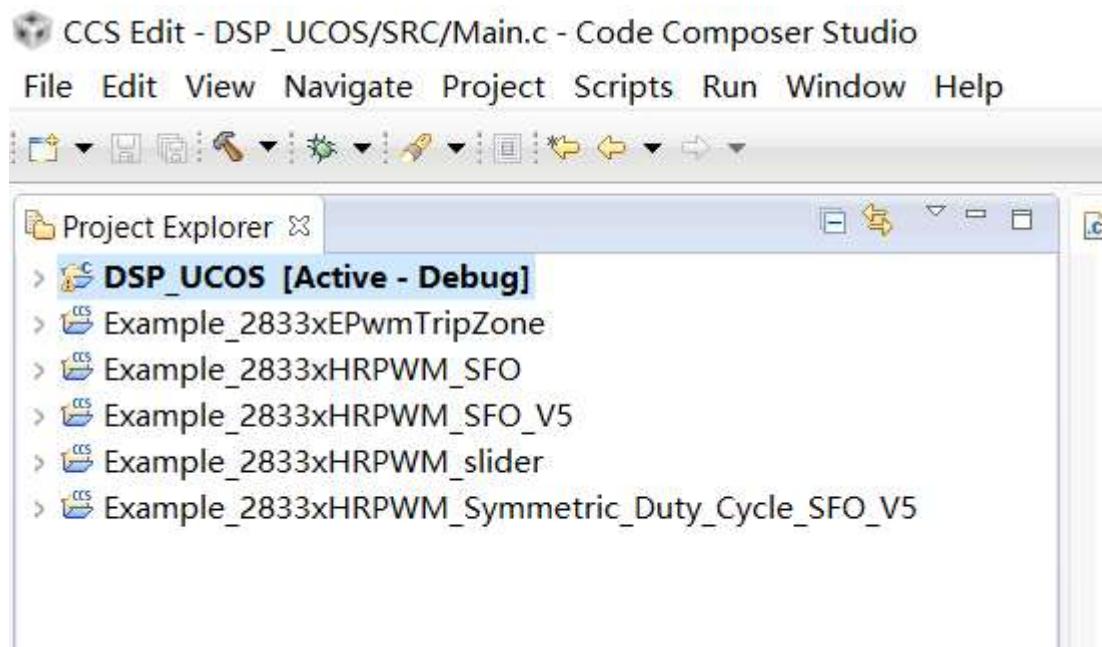
图 1 连接关系图

将 12864 插到此接口上；

(2) 接着把仿真器的 USB 与电脑进行连接，将仿真器的另一端 JATG

端插到 SXD28335 开发板的 JATG 针处；

(3) 给开发板上电。单击鼠标左键选择要调试的工程，如下图所示：



然后单击图中红色的方框处的调试按钮

(3) 实验现象如下图所示：

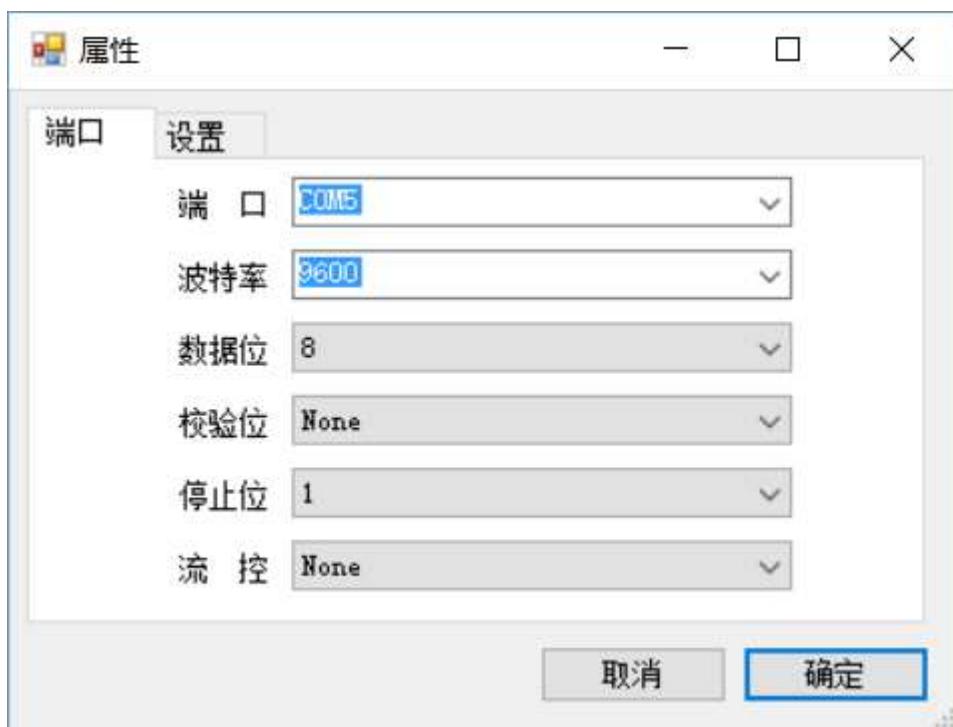
Step1: 全速运行，在下图所示地方暂停：

Main.c

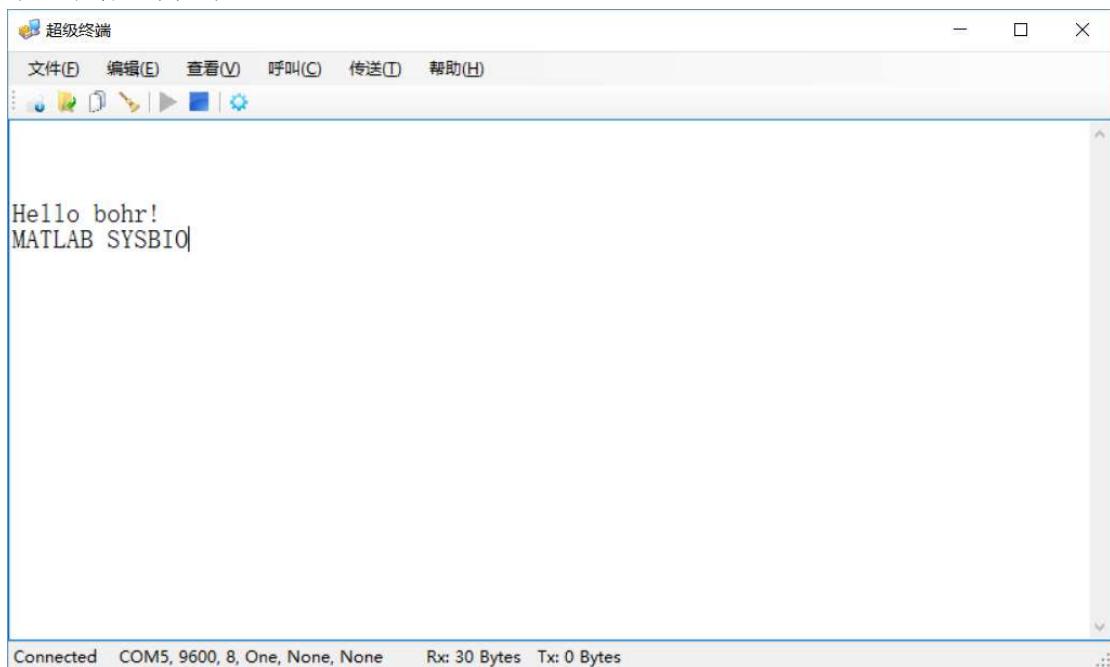
```
231 ****
232 *
233 ****
234 */
235 void Task1(void *data)
236 {
237     msg = "\r\n\nHello bohr!\0";
238     OS_ENTER_CRITICAL();
239     scib_msg(msg);
240
241     msg = "\r\nMATLAB SYSBIOS\r\0";
242     scib_msg(msg);
243     OS_EXIT_CRITICAL();
244 }
```

程序会在此处暂停

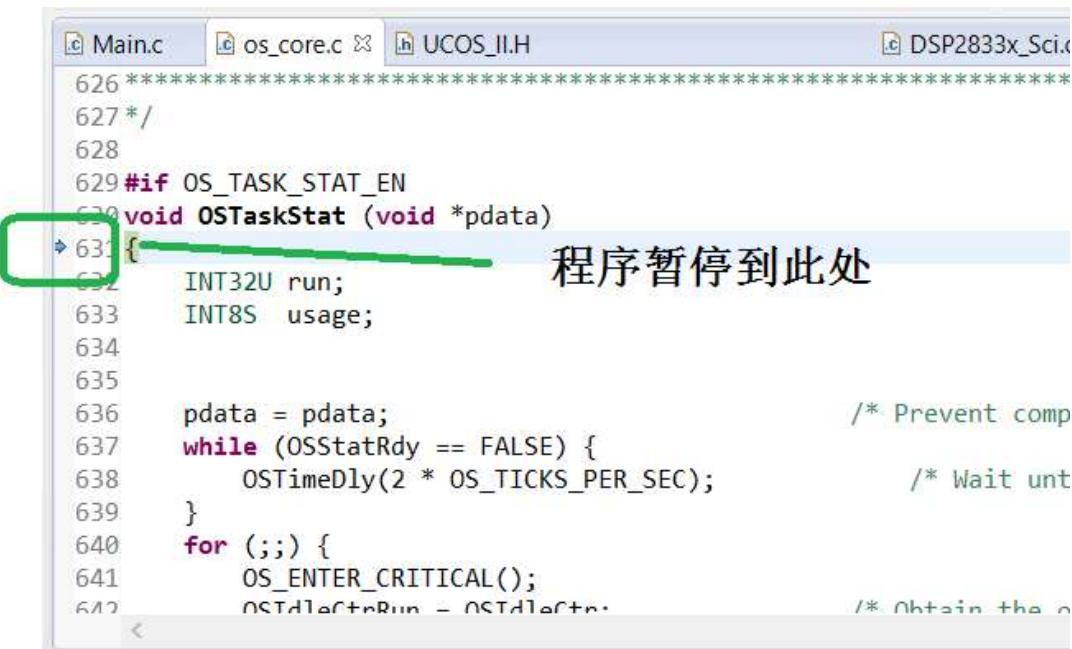
Step2: 按如下图设置串口



串口中断显示如下：



Step3：程序暂停到此处：



```
626 ****
627 */
628
629 #if OS_TASK_STAT_EN
630 void OSTaskStat (void *pdata)
631 {
632     INT32U run;
633     INT8S usage;
634
635
636     pdata = pdata; /* Prevent compiler warning */
637     while (OSStatRdy == FALSE) {
638         OSTimeDly(2 * OS_TICKS_PER_SEC); /* Wait until ready */
639     }
640     for (;;) {
641         OS_ENTER_CRITICAL();
642         OSTdlectrRun = OSTdlectrRun + 1; /* Obtain the number of tasks */
643     }
644 }
```

全速运行：



```
587 * Returns      : none
588 ****
589 */
590
591 void OSTaskIdle (void *pdata)
592 {
593     pdata = pdata;
594     // Enable global Interrupts and higher priority real-time tasks
595     /* Prevent compiler warning for unused variable */
596     for (;;) {
597         //OS_ENTER_CRITICAL();
598         OSIdleCtr++;
599         //OS_EXIT_CRITICAL();
600     }
601 }
```

再全速运行，会看到 LCD12864 显示汉字：

玻尔电子有限公司
三兄弟嵌入式
SYSBIOS MATLAB
344394490@QQ.COM

一分钟后显示一个图片

实验 62: TFT3.2 寸屏实验 (Example_2833x_TFT32)

一 实验目的:

- ✧ 了解 TMS320F28335 的 XINTF 工作原理;
- ✧ 了解 TFT3.2 寸液晶屏;

二 实验设备:

- ✧ PC 机一台;
- ✧ XDS100v2 或 XDS100V3(隔离)仿真器一套;
- ✧ SXD28335 开发板一套, TFT3.2 寸屏一块;

三 实验步骤:

- ✧ 首先将 CCS6.0 开发环境打开; 看一下如下原理图:

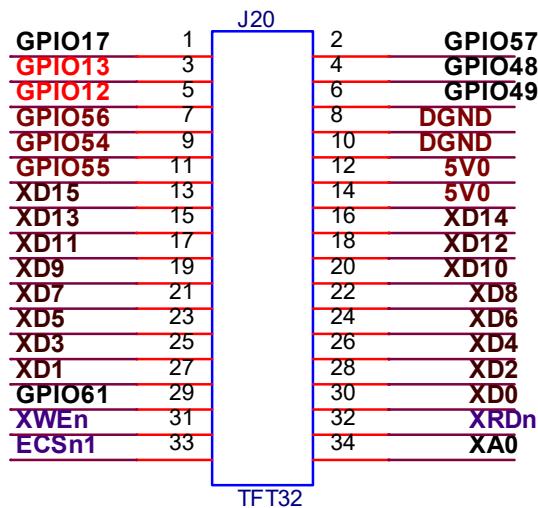


图 1 3.2 寸屏的接口

从上图可以看出 TFT32 用 ECSn1 作为 TFT32 的片选信号。而 ECSn1 则是由 XZCS7n 经地址译码得到的。

- ✧ 接着把仿真器的 USB 与电脑进行连接, 将仿真器的另一端 JATG 端插到 SXD28335 开发板的 JATG 针处;
给开发板上电。单击鼠标左键选择要调试的工程, 进行调试。
- ✧ 程序通过仿真器会加载到 SXD28335 开发板里。这是会出现下面的界面。

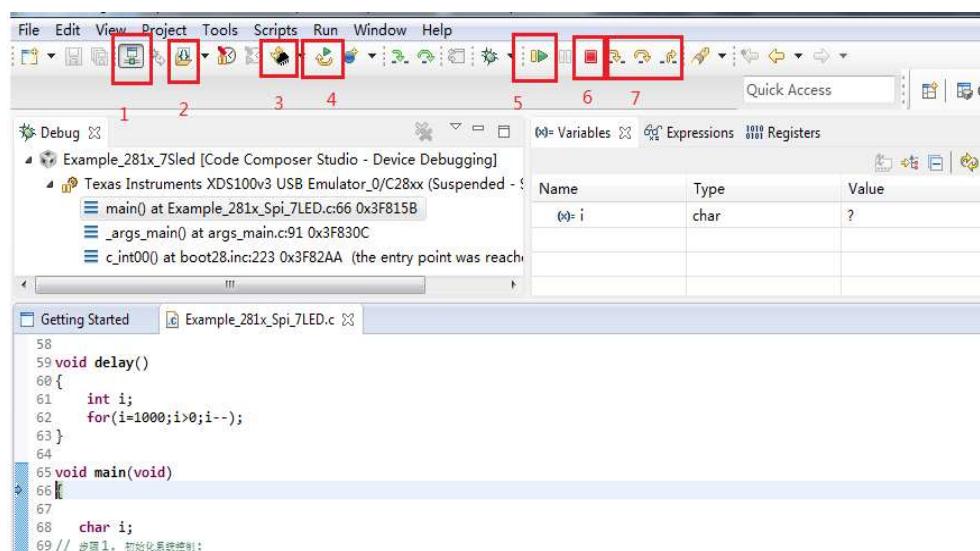


图 5 调试界面

图中 1 图标是用来进行与开发板进行连接的按钮；

图中 2 是用来下载 Debug 文件下的.out 文件的

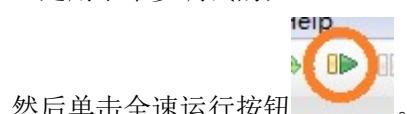
图中 3 是 C P U 软 R e s e t ；

图中 4 是调试时恢复到程序的开始处。

图中 5 是全速运行；

图中 6 是停止调试；

图中 7 是用于单步调试的；



然后单击全速运行按钮。

四 试验现象：

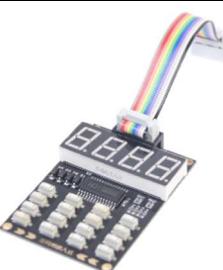
屏幕上会显示一个界面

第三章 宙斯盾版之 SXD28335_QFP 标配及选配清单

宙斯盾版之 SXD28335_QFP(标配清单)

序号	配件图片	配件说明	配件数量
1		宙斯盾版之 SXD28335_QFP	1
2		电源适配器 5V/2A	1
3		USB TO RS232 转换器	1
4		20-20-14 JTAG 电缆	1
5		高品质杜邦线	1
6		双通道 CAN 通信电缆	1
7		跳线帽(出厂已经插到 板子上)	1
11		配套光盘或网盘	1

宙斯盾版之 SXD28335_QFP (选配清单)

序号	配件图片	配件说明	单价
1		步进电机: 5V 28BYJ-48	10 元
2		直流电机: RF-300FA-12350	8 元
3		显示屏: LCD12864	55 元
4		显示屏: LCD1602	20 元
5		显示屏: TFT3.2 寸带触摸	180 元
6		按键显示模块: 4*4 键盘+4 位一体 数码管	55 元
7		转换器: USB 转 485	16 元

8		TF 卡: 2G 16G	18 元 60 元
9		TF 卡读卡器	15 元
10		仿真器: XDS100V2	128 元
11		仿真器: XDS100V3	158 元
12		仿真器: XDS100V3 隔离	598 元
13		仿真器: XDS200	698 元
14		仿真器: XDS200 隔离	898 元

第四章 宙斯盾版之 SXD28335_QFP 包装



第五章 宙斯盾版之 SXD28335_QFP 其它说明

1. 购买提示

本店铺所有商品默认顺丰快递，顺丰未开通地点协商处理。
因淘宝商品不含税，所以需要开发票的顾客朋友，请联系客服，补充发票税点款项；
本开所开发票均为甘肃省工商业统一正规机打发票，分为普通发票或增值税发票；
普通发票收取货款的 6%，增值税发票收取货款的 17%（金额在 1 万元以上）；
补充税点后请留下您的开票信息，增值税发票按规定提供详细信息。

2. 售后服务

1. 自本店发货日期起 7 天内（包含发货日），质量出现问题可享受包退换服务
2. 本店商品保修期为十二个月（自发货之日起），保修期内出现故障免费维修，来回邮费买家与买家各半
3. 易损易耗品及 CPU 主芯片元器件不在保修范围内（如液晶屏、DSP 芯片、AD、DA 等），其余可免费维修，收取成本费
4. 本公司对售出产品实行终身维修制，超过保修期需酌情收取零配件及维修费，邮费由买家全部承担。

3. 特殊声明

本公司产品受商标法及知识产权保护，出现盗用本公司配套资料、视频、图纸、标识、主图、描述等造成的损失，本公司有不告知的情况下提起诉讼的权利，敬请谅解。
本公司产品出现售后质量问题，按售后服务条款执行，若肆意进行中差评，本公司终止相应技术支持，并列入本公司及代理商产品采购黑名单，敬请谅解，如不能接受上述说明，请勿购买本公司产品，谢谢合作。

4. 玻尔答谢

玻尔电子会在忠诚用户的呵护下茁壮成长，您的支持是我们前进的动力，谢谢光临本店。

