

- 
- 1、 **CCS4(包括 4.1 和 4.2 等等)内部已经集成了 XDS100V1 和 XDS100V2 的仿真器驱动程序，所以不用安装 XDS100 驱动程序，而 CCS3.3 就必须安装 XDS100 的驱动程序，这也算是 CCS4 比 CCS3 方便的一点。**
  - 2、 **XDS100V2 仿真器需要在 CCS4 及以上版本(包括 CCS4.1 和 CCS4.2 等等)才能使用，不能在 CCS3.3 版本下使用，不管哪家生产的，只要是 XDS100V2 就一定如此。**
  - 3、 **本店的 TMS320F2808 开发板和 TMS320F2802 开发板，除了 DSP 芯片不同以外，一个是 TMS320F2808，另外一个 TMS320F2802，其他的都一样，连 DSP 引脚都是兼容的。**

在进行后面的实验之前，需要做好以下 3 个步骤：


- 1、安装好 CCS4.2
- 2、获取 license

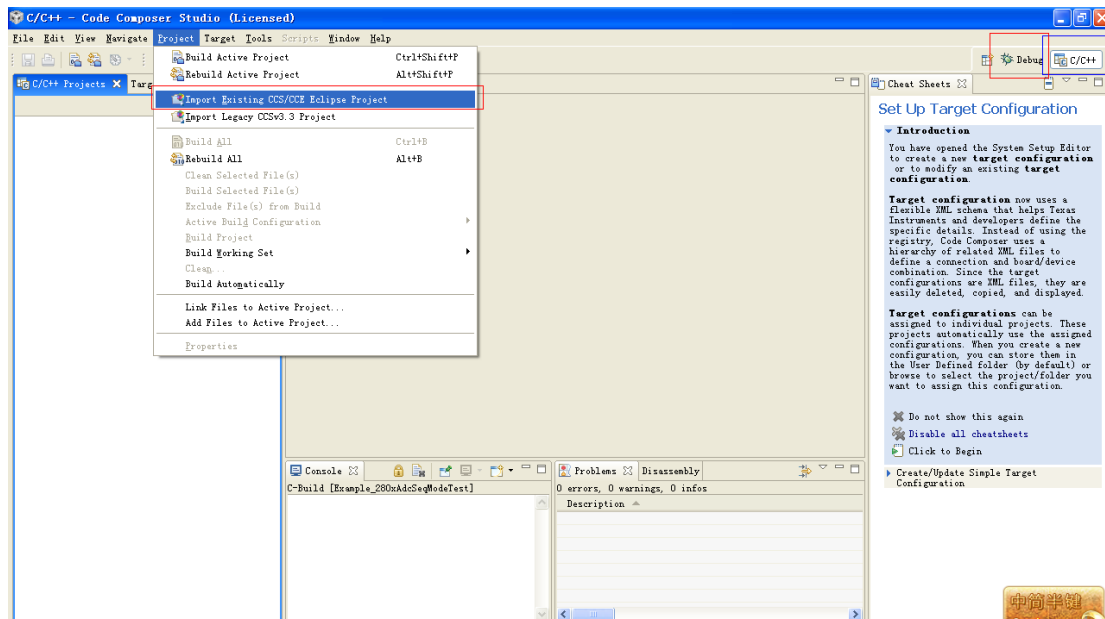
上面这两个步骤在《张掌柜讲 DSP 系列之 CCS4.2 XDS100 在 CCS4.2 环境下仿真编程新手入门》都有详细的讲解，照做就可以了。

- 3、把我的光盘中的源代码文件夹“Code of TMS320F280x CCS4”拷贝到电脑中，注意一定不要把放在有中文路径的文件夹里面，包括电脑的桌面。也不要把 Code of TMS320F280x CCS4 内部的某个文件夹拷贝出来再打开，比如把 Buzzer 这个文件夹从 Code of TMS320F280x CCS4 文件夹拷贝出来，然后再打开 Buzzer 内的 project，这样是不对的，会提示缺少文件的。
- 4、先把 DSP 仿真器的 JTAG 线和 DSP 开发板的 JTAG 口连接起来，然后将 DSP 仿真器的 USB 口插到电脑上，最后给 DSP 开发板上电。这个上电顺序是推荐的上电顺序，不代表只能这样做，你按照其他顺序上电，也不会损坏开发板的。掉电的顺序正好跟上电顺序相反。

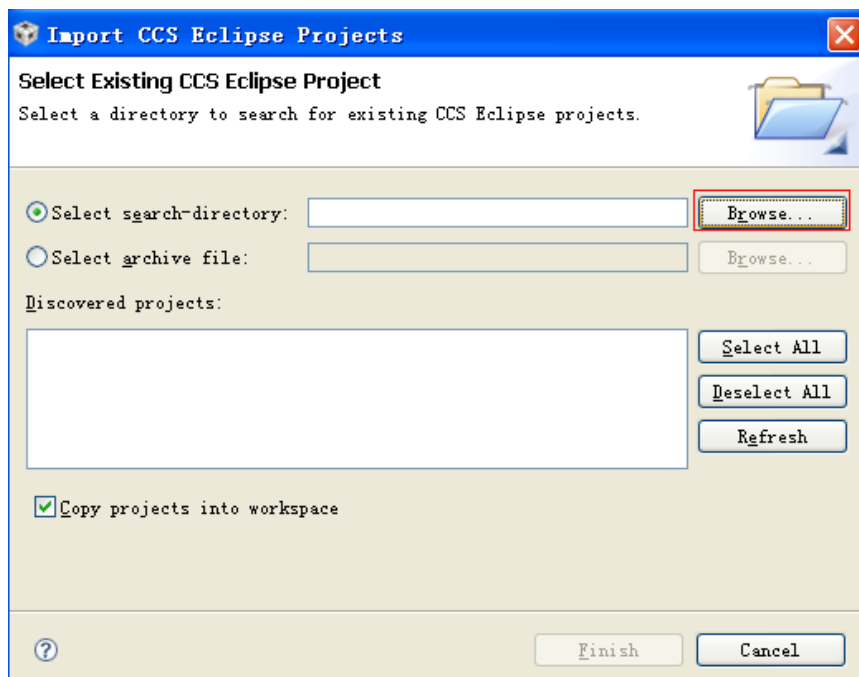
## 第一章 蜂鸣器唱歌实验

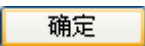
---

(1): 在  C/C++ 界面下，打开 project，如下图

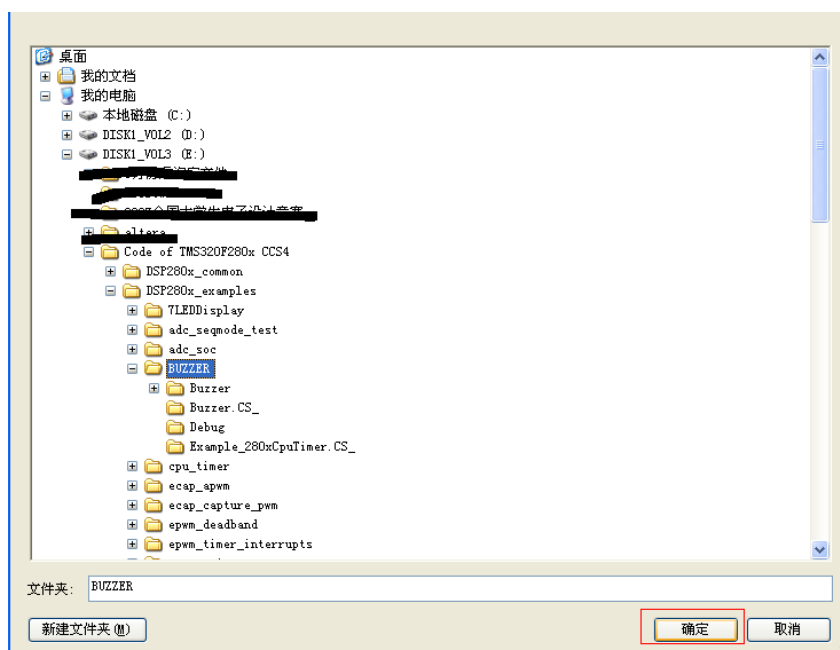


(2) 出现浏览框，然后点 

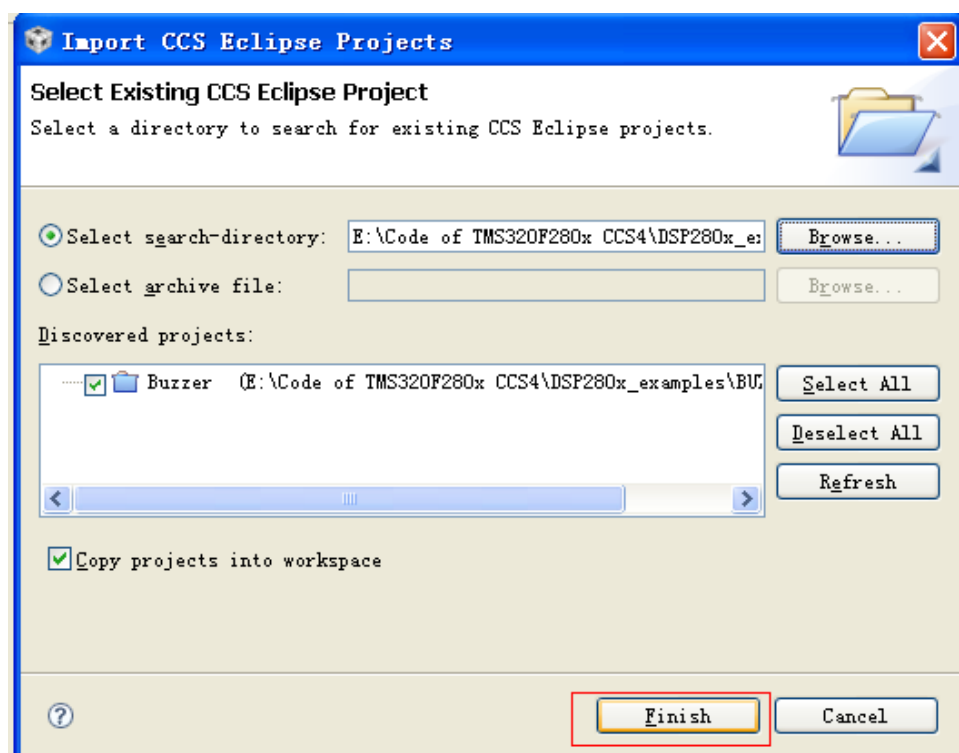


(3) 浏览找到 E:\Code of TMS320F280x CCS4\DSP280x\_examples\BUZZER 这个文件夹，然后点 。如果打开 project 的时候出现错误提示无法正常打开，90%的可能性是把 project 保存在有中文路径的地方了，包括桌面。或者 10%的可能性是把 BUZZER 这个文件夹从原来的 Code of TMS320F280x CCS4 拷贝出去

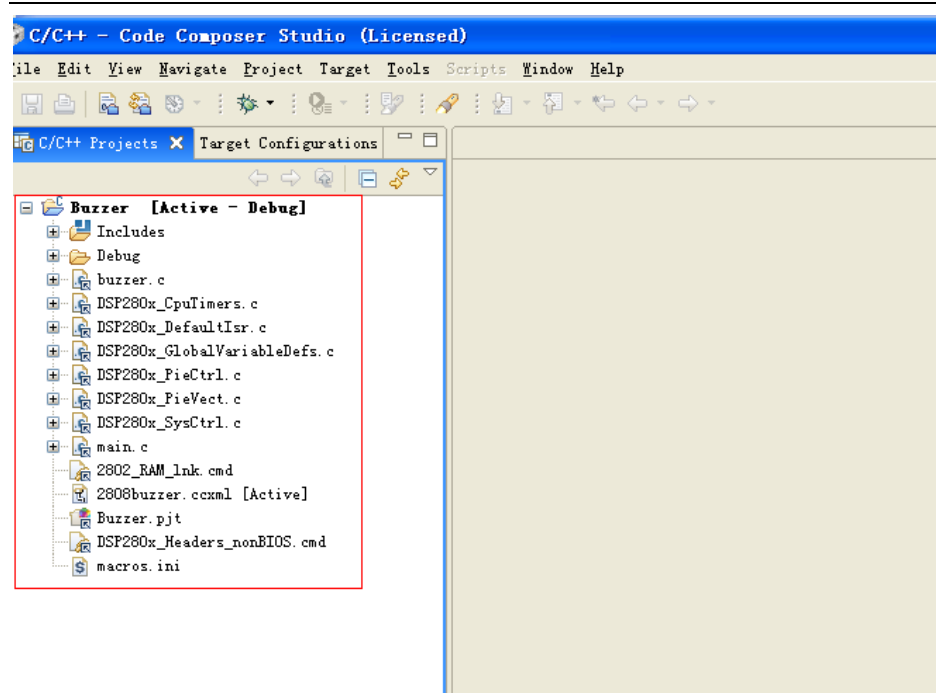
放在另外一个地方，那么也会打不开。注意，请不要把任何一个 DSP280x\_examples 中的子文件 copy 到另外的地方去打开，那样是错误的。想要移动的话，请把 Code of TMS320F280x CCS4 整个文件夹都移动。



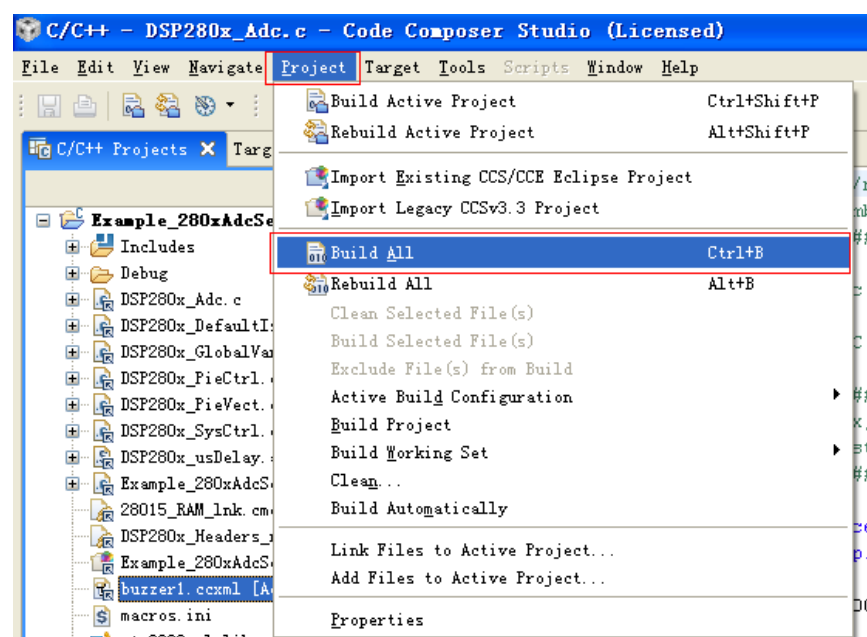
(4) 出现下图后，点 **Finish**



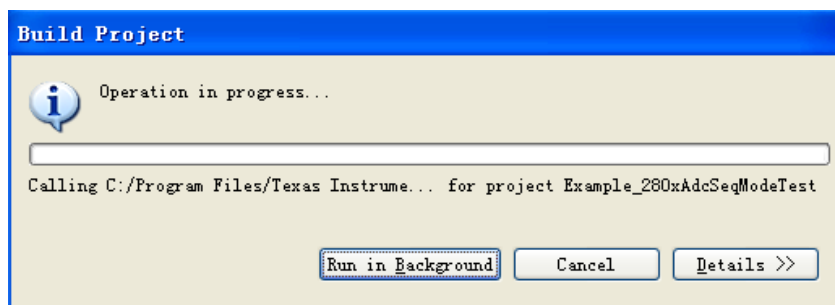
(5) 出现下图红圈处，这就代表已经打开了一个 project，也就是打开了 Buzzer.pjt 这个 project。



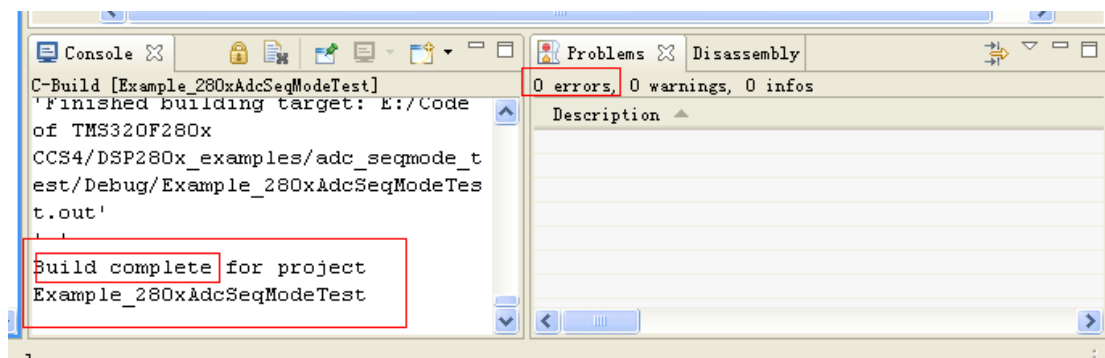
打开 project 后，一定需要**编译（Build）** project，点击 project->Build ALL（快捷键 Ctrl+B）或者 Rebuild ALL（快捷键 Alt+B），如果不编译（Build）project 就无法生成.Out 文件，也就没法进行后面的 Load Program 这个步骤。



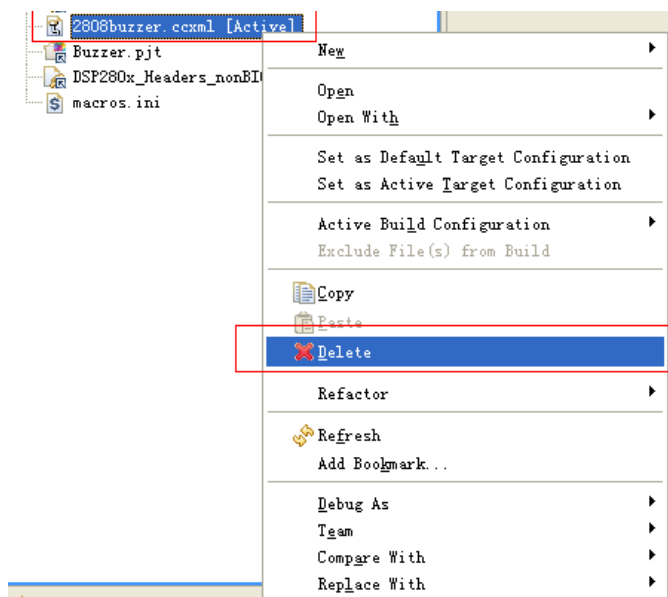
点击编译（Build）后，会出现下面的进度条。



稍等一些时间后，会在下方出现编译的结果，看到“Build complete”并且“0 errors”就代表编译成功，就生成。如果有 errors，那么就编译失败，说明你的程序编写语句有 error，需要找到这个错误，并且修改掉，直至没有 error，否则就无法生成 .out 文件，无法进行后面的 Load program。注意：有 waring 是不影响 OUT 文件生成的。

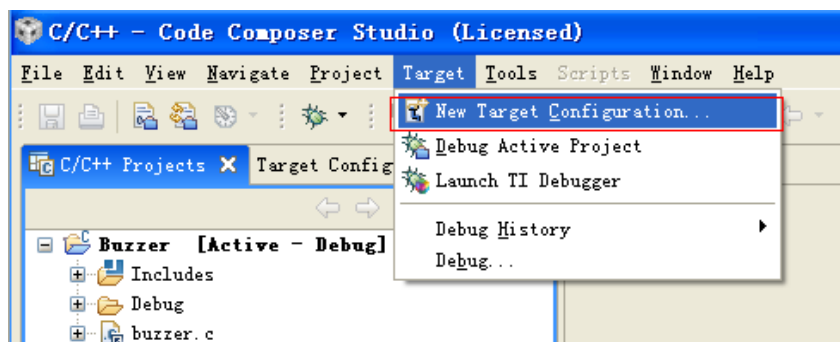


(6) 移除 project 中原来的.ccxml 文件，也就是 target configuration 文件，移除方法，右键点击 2808buzzer.ccxml,然后选择 delete 如下图

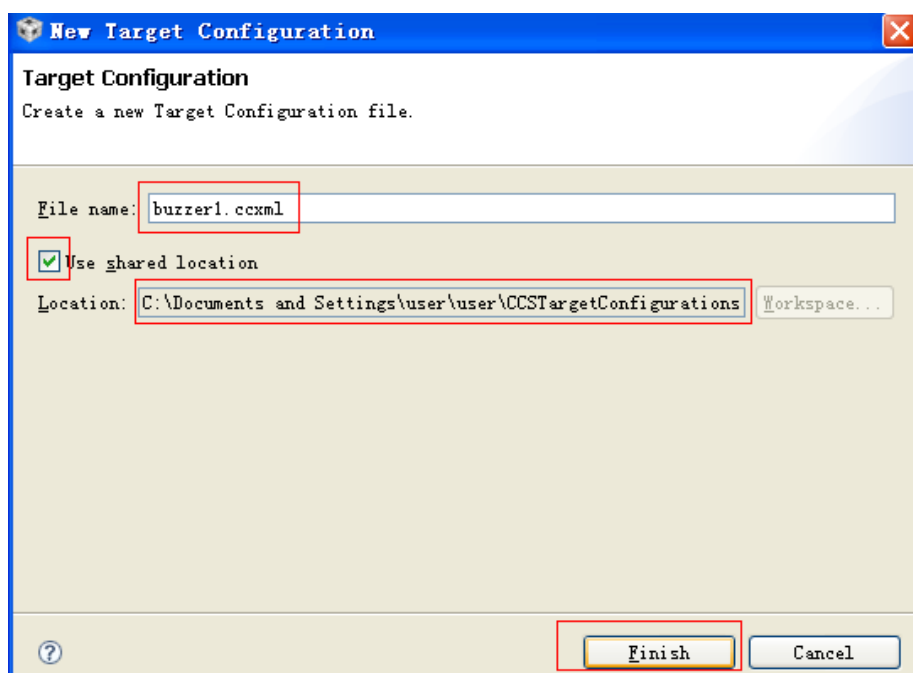


出现提示款后，点 YES，这样就移除完成了

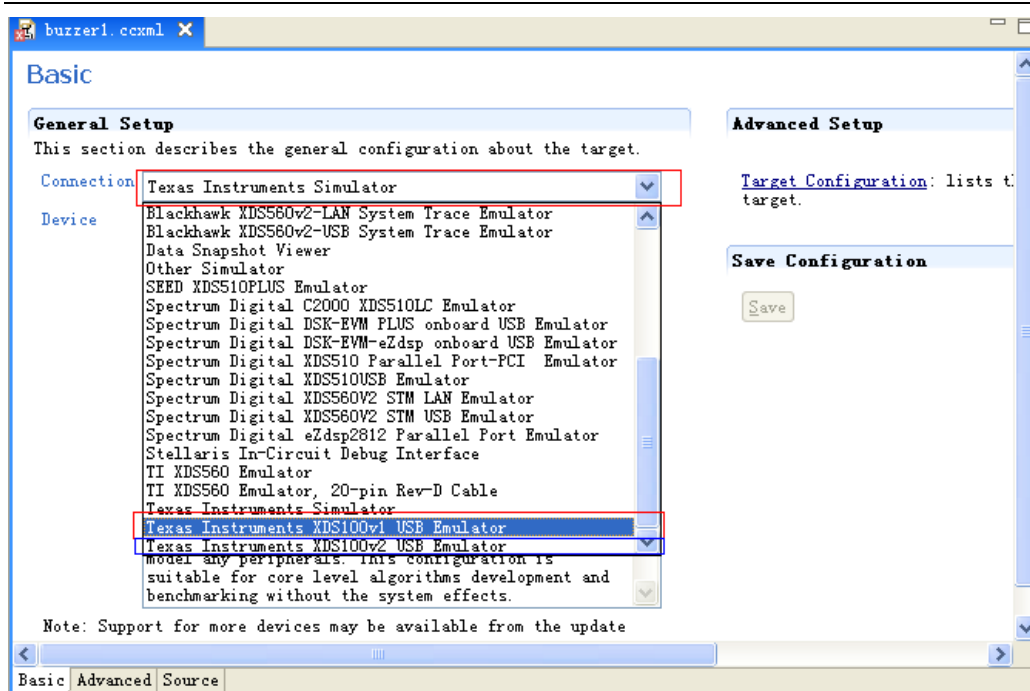
(7) 移除旧的.ccxml 文件后，再新建一个.ccxml 文件，新建方法如下



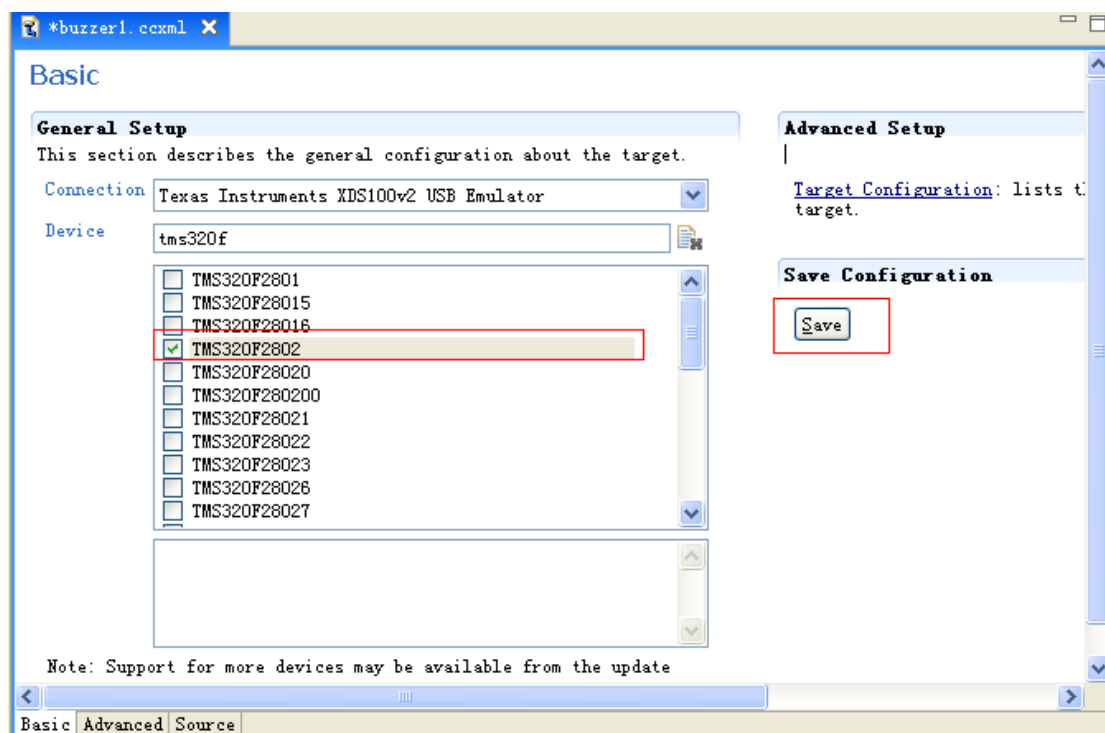
(8) 出现下图后，给新建的 ccxml 文件起个名字，再点上 ☒ Use shared location 前面的小对勾（注意，这个务必要点上），推荐不要修改 location C:\Documents and Settings\user\user\CCSTargetConfigurations，避免不必要的麻烦，尤其是不要保存在有中文路径的地方。然后点 **Finish**



(9) 出现下图，首先选择 DSP 仿真器的型号，如果你买的是本店 100 元的 XDS100 仿真器，那么就选红圈处 **Texas Instruments XDS100v1 USB Emulator**，如果你买的是本店 198 元的 XDS100V2 仿真器，那么就选蓝圈处 **Texas Instruments XDS100v2 USB Emulator**，也就是说你实际使用的仿真器型号，要在这个界面正确的选择。



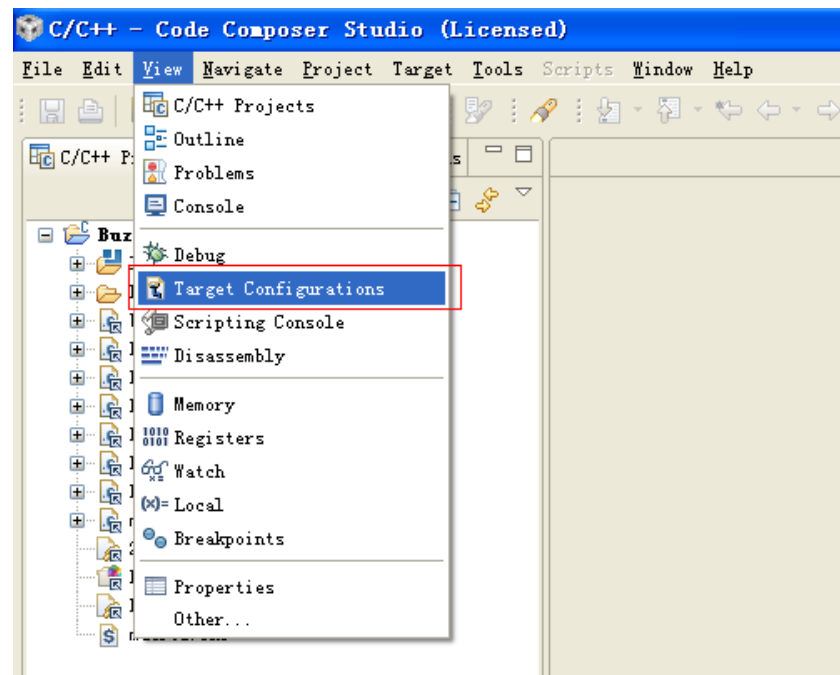
(10)选择DSP开发板上的DSP芯片的型号,本店的DSP开发板有TMS320F2808和TMS320F2802两个型号,请正确对应选择,下图就是以TMS320F2802为例,点上前面的小对勾,然后点save。



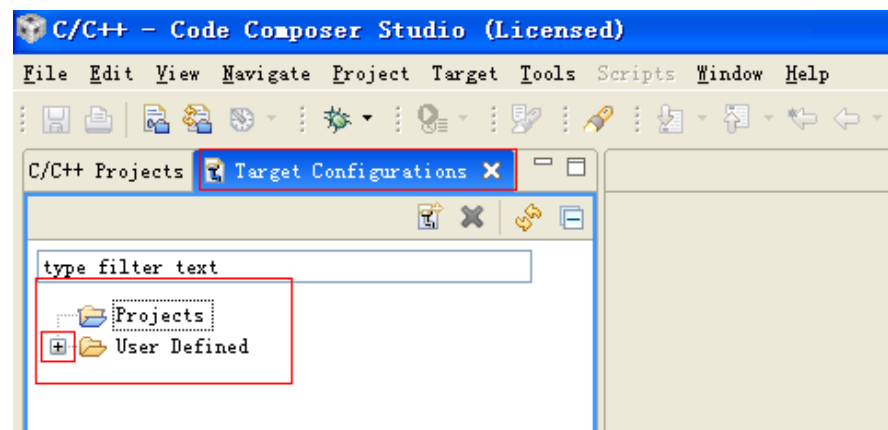
在这里我要格外强调一下,很多朋友在建立.ccxml文件的时候,会不小心建立多个,而且自己还没有发现,于是再往后面进行,就会出错。所以下面我插入一段,告诉大家如何检查自己是不是建立了多个.ccxml文件,如果建立了多个,请删除

其他没有用的，只留下一个。

检查的方法，点击 View->Target Configuraiton,如下图

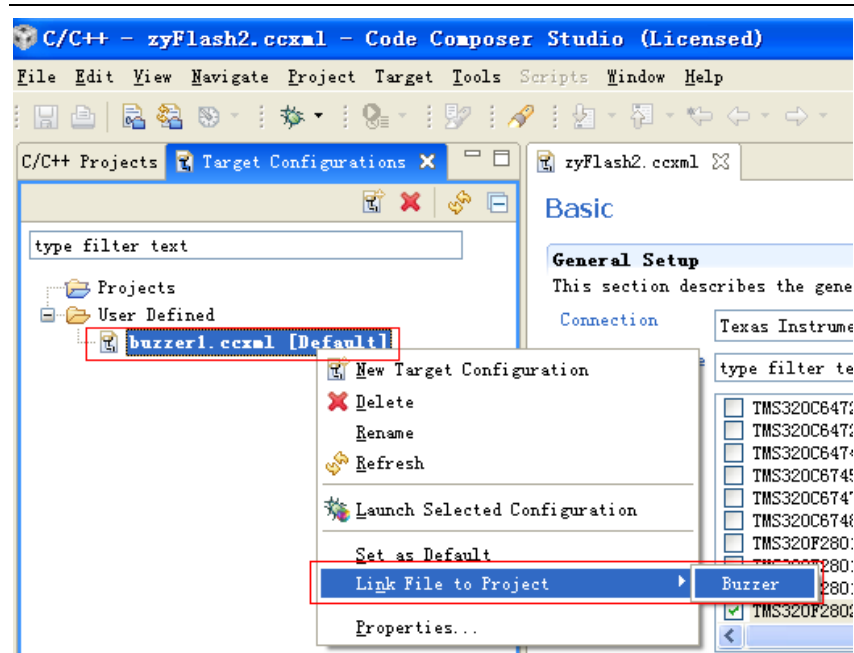


然后就出现下图，需要点开所有的小加号，然后看是不是只有唯一的一个 ccxml 文件，如果有多，请用右键点击，然后选 delete 删除掉。这个很重要，很多朋友都是在这个问题上弄的不好，于是导致“连不上”（也就是无法 Connect）等等问题。

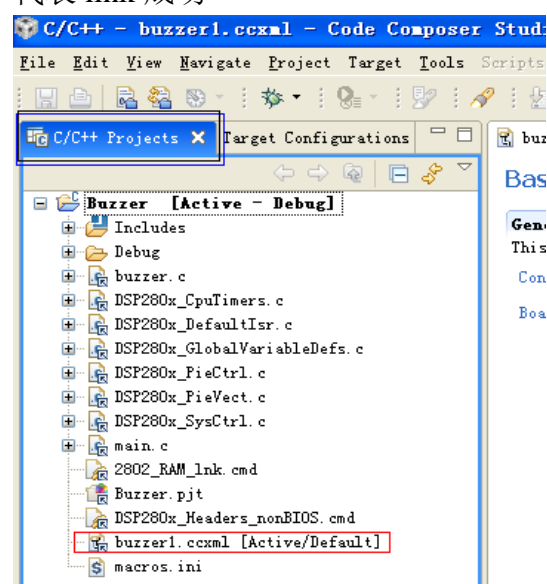


(11) 把 buzzer1.ccxml 文件 link 到 buzzer.pjt。方法如下，在 Target Configuration 界面下，右键点击 buzzer1.ccxml 然后选择 Link File to project，然后选 Buzzer，如下图

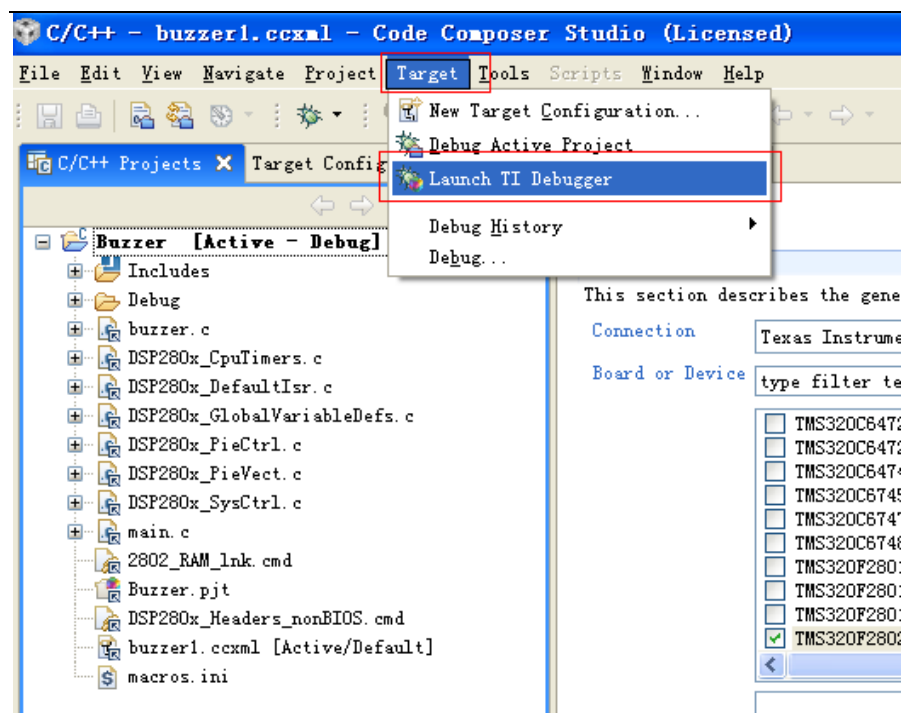




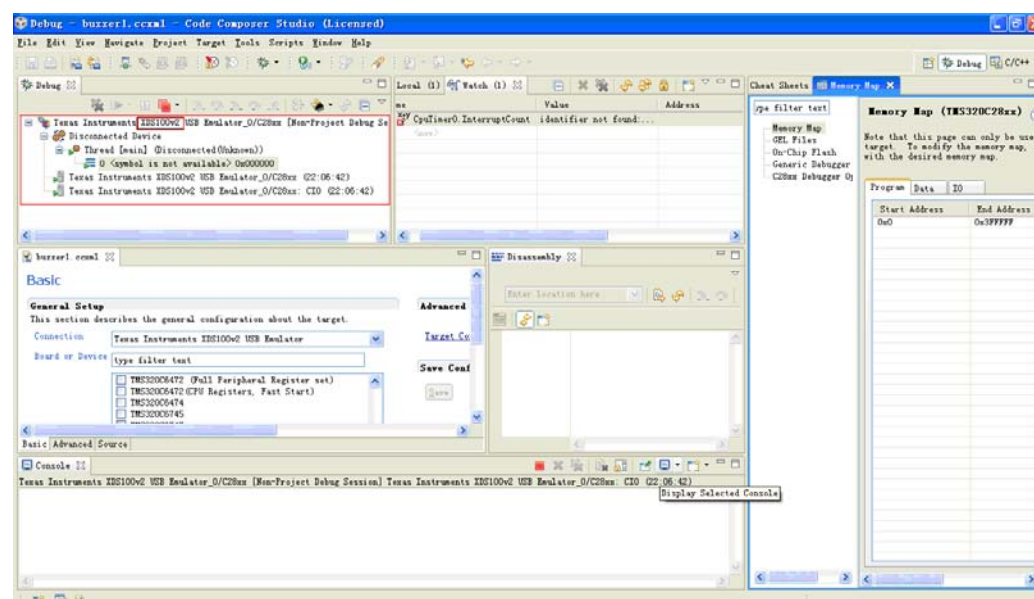
(12) 完成上面的 link 后，可以在 C/C++ Project 界面看到下图红圈处，这样就代表 link 成功



(13) 启动 TI 的 Debugger (Launch TI Debugger)，按下图操作



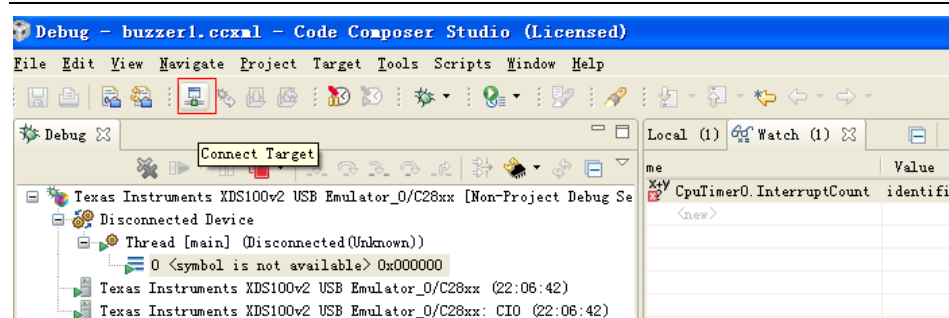
(14) 进入 CCS4 的 Debug 界面，如下图



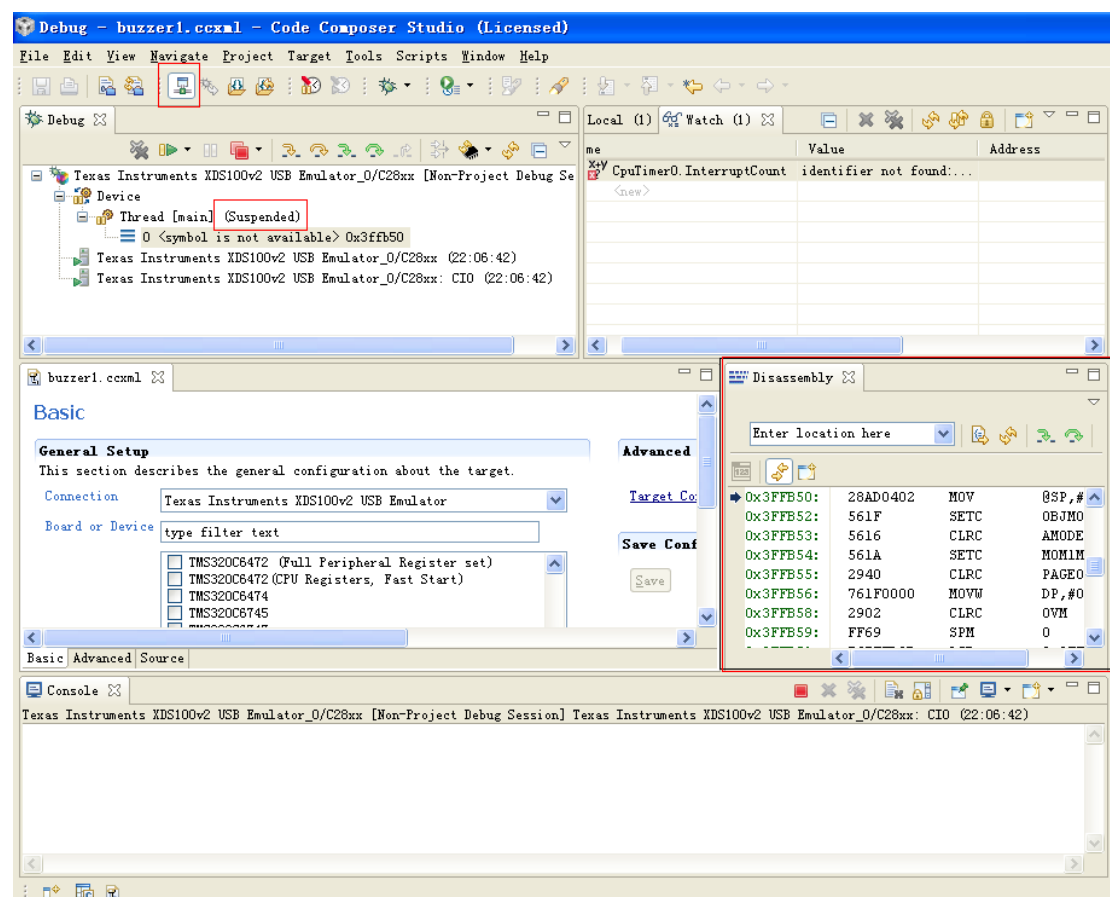
在上图红圈处是可以看出来你究竟选择的是什么的 DSP 仿真器，本实验选择的是 XDS100V2。

插曲：

(15)连接 DSP 仿真器和 DSP 开发板(Conenct)，点击红圈处，或者快捷键 Alt+C



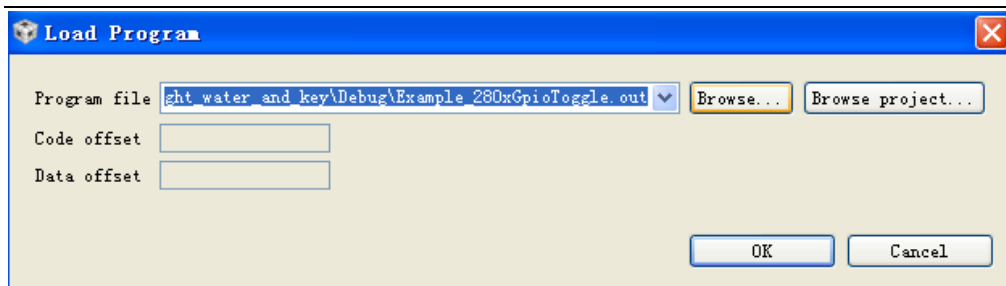
(16)出现下图红圈处，就代表 Connect 成功。



如果不成功，出现错误提示，请做以下几个方面的检查，

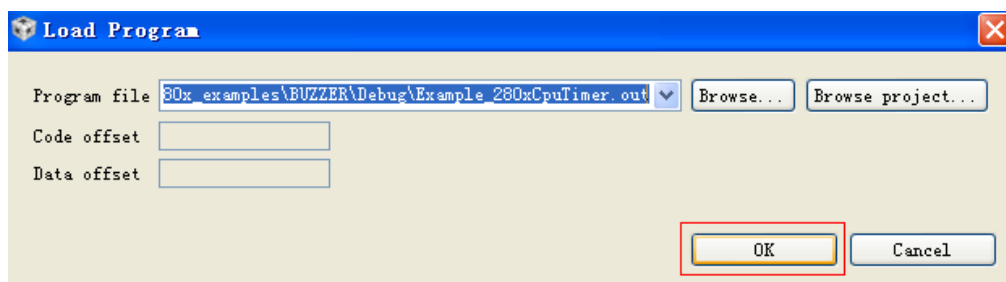
- A、 ccxml 文件是否设置正确，包括仿真器型号选择，DSP 型号选择。  
注意自己有没有点 save
- B、 ccxml 文件是否设置多个，如果不是请删除不必要的 ccxml。
- C、 开发板和仿真器连接是否可靠，仿真器 USB 和电脑是否连接可靠(可以换个 USB 口试试)
- D、 开发板有没有上电

(17) 装载编译好的程序到 DSP 的 RAM 中，也就是 Load Program，方法：点击 ，或者 ctrl+alt+L，然后出现下面浏览框

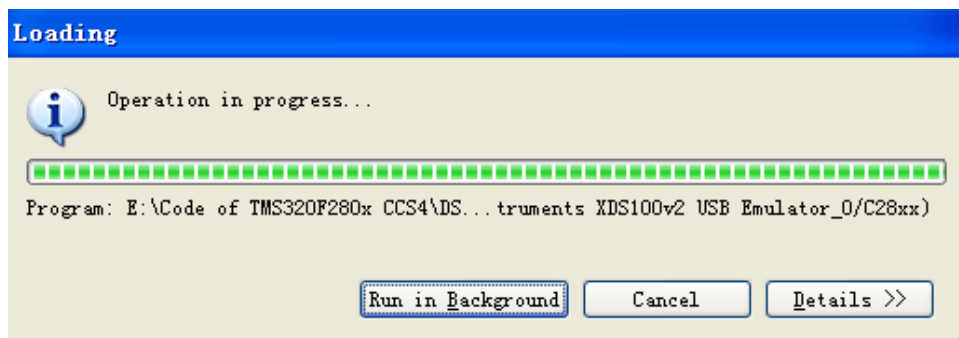



点 Browse，找到编译 project 后生成的 **Example\_280xCpuTimer.out** 文件，路径如下 E:\Code of TMS320F280x CCS4\DSP280x\_examples\BUZZER\Debug，然后点打开

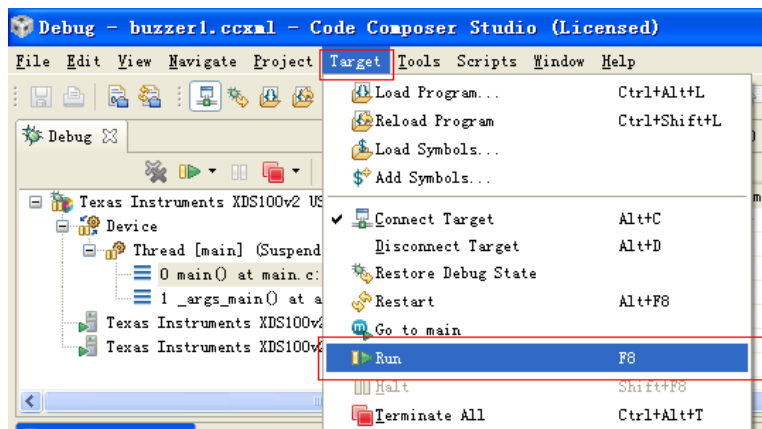
(18) 选择好要 Load 的文件了，就会出现下面界面，然后点 OK





(19) 出现下面的进度条，代表正在 load






(20) Load 完成后，运行 project，方法：点 ，或者快捷键 F8,或者如下图

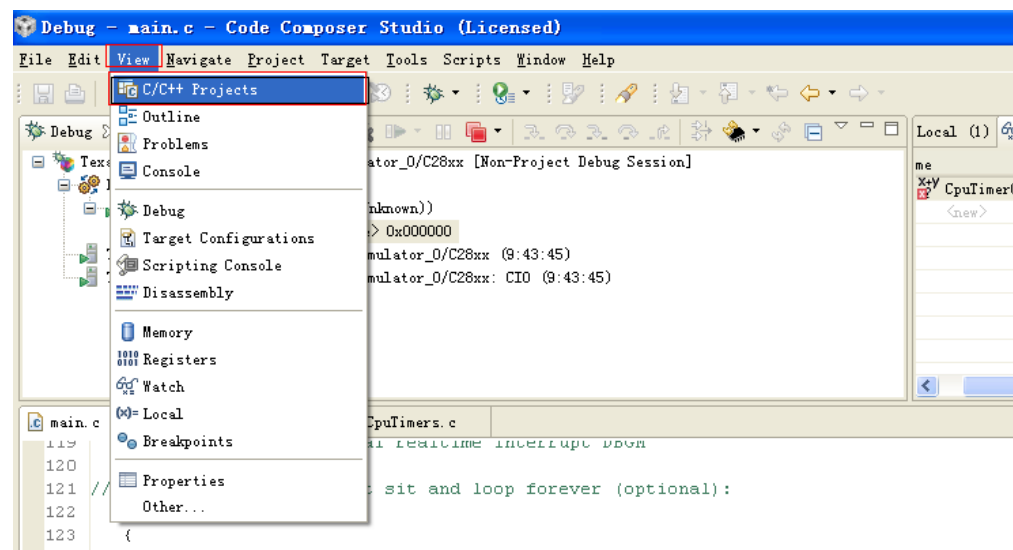


注意：如果前面没有成功 connect，那么 Run 这个图标就是灰色状态，无法点击。

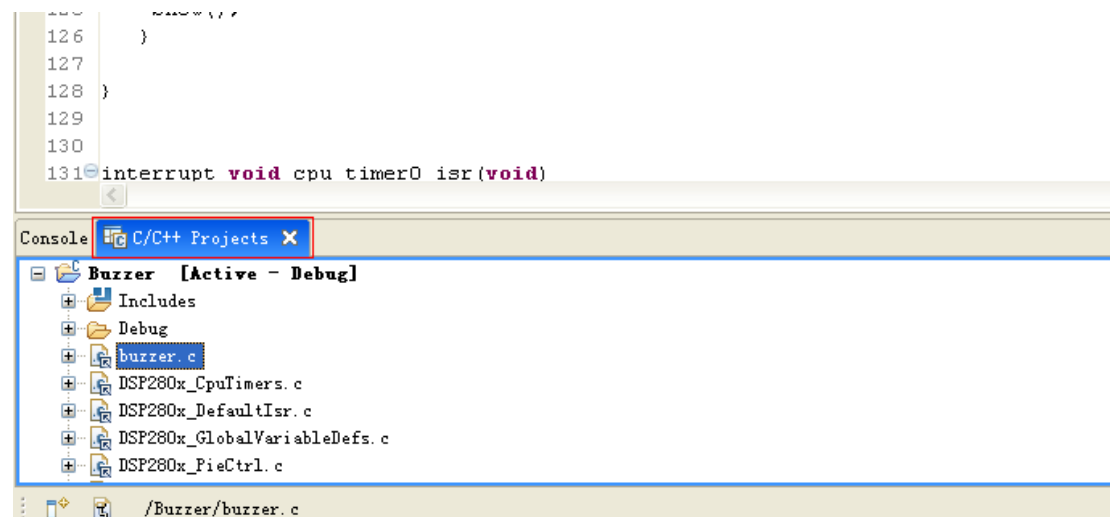
(21) 运行就会听到开发板上的蜂鸣器会唱“祝你生日快乐”，点击，就暂停程序的运行。点击就退出 Debug 的界面，回到 C/C++编程界面。

暂停后，点是 reset cpu，点restart，然后再点就可以重新开始运行了。

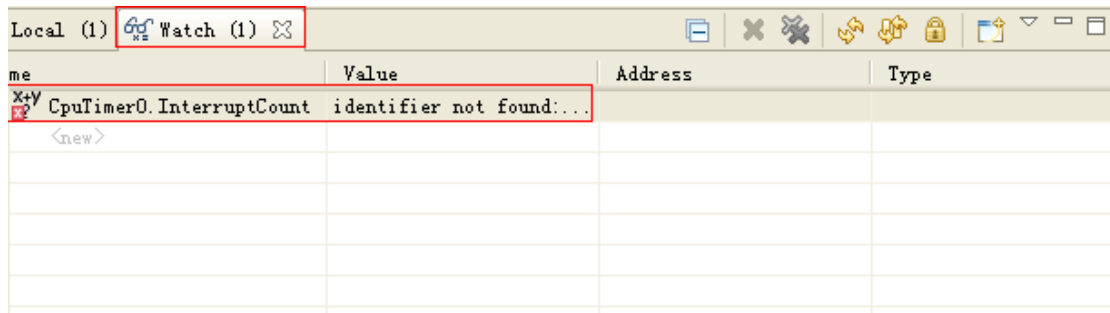
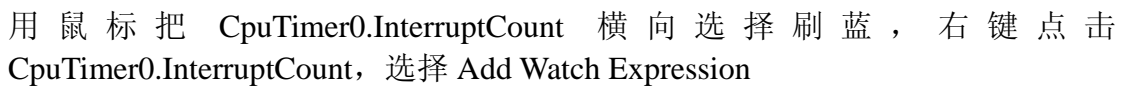
(22) 观察 buzzer.pjt 的变量如何变化，方法如下  
点击 View 然后选择 C/C++ Project



出现下图，然后双击 main.c，就打开了 main.c 这个文件



然后出现下面这个界面，



(23) 运行程序，再暂停程序，就可以看到 CpuTimer0.InterruptCount 的数值会发生变化

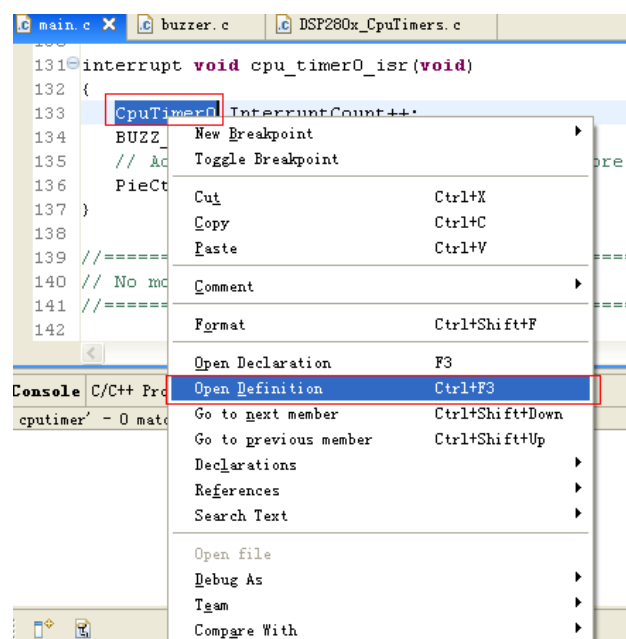
CpuTimer0.InterruptCount 这个数值就代表进入 Cputimer0 中断的次数。至于中断的原理会在后面有专门的讲解，这里第一个章节就是给大家介绍基本的操作，不涉及过多的 DSP 原理。我一直坚持的观点是先有感性认识，再上升到理论高度。

(24) 下面给大家介绍一些 CCS4 的基本操作

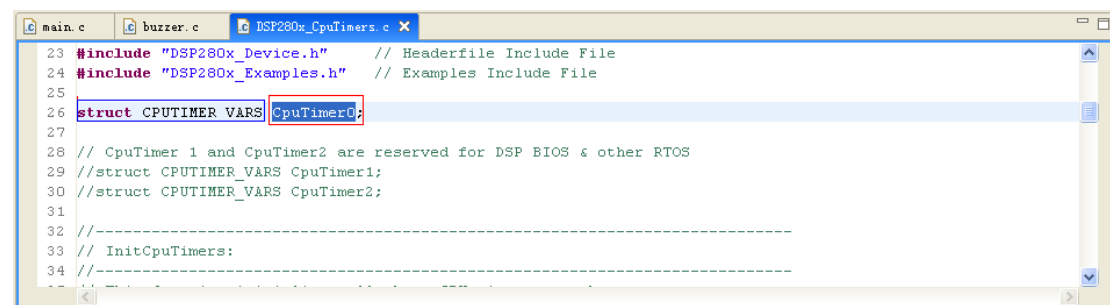
A、跳转到程序中变量的定义和声明的位置的方法

举例，我想跳转到 CpuTimer0.InterruptCount 这个里面的 CpuTimer0 这个结构体的变量定义处，那么需要将 CpuTimer0 用鼠标左键选择（刷蓝），然后右键点击 CpuTimer0 选择 Open

Definition（快捷键 Alt+F3），如下图。



然后就跳到下图红圈处，这个位置就是 CpuTimer0 这个结构体变量的定义，而按照前面的方法点 CPUTIMER\_VARS 跳转到 CPUTIMER\_VARS 的定义，这个就是结构体本身的定义。



跳转到 CPUTIMER\_VARS 的定义如下图

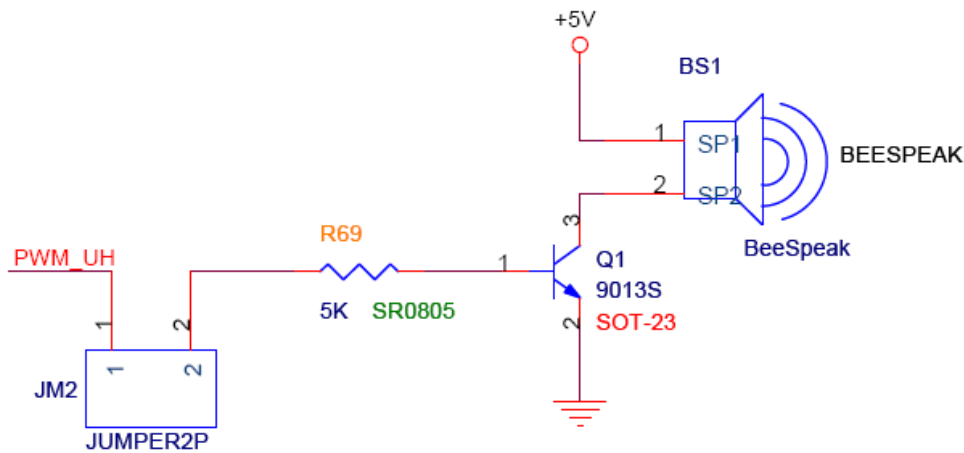
```
108 //-----
109 // CPU Timer Support Variables:
110 //
111 struct CPUTIMER_VARS {
112     volatile struct CPUTIMER_REGS *RegsAddr;
113     Uint32    InterruptCount;
114     float     CPUFreqInMHz;
115     float     PeriodInUSec;
116 };
117
118 //-----
119 // Function prototypes and external definitions:
120
```

\*\*\*\*\*  
\*\*\*\*\*

下面我就简单的给大家介绍一下 buzzer.pjt 的唱歌原理。  
给蜂鸣器输入不同的频率的时候，蜂鸣器就会发出不同音调的声音，下图就是蜂鸣器发出do, re, mi, fa, so, la等音调与激励频率之间的对应关系。

	do	re	mi	fa	so	la
频 率 /Hz	262	294	330	349	392	440
周 期 /us	3816	3401	3030	2865	2551	2273

蜂鸣器的激励电路



JUMPER是跳线子的意思

PWM\_UH 这个是连接到 74LVX3245 (U6) 的 Pin21，也就相当于连接到 DSP2808 或者 2802 的 PWN1A (Pin47)，在这里不是使用 Pin17 的 PWM 引脚功能，而是使用 Pin17 作为 GPIO (通用 IO)，然后在 Pin17 上产生不同频率的方波就可以使蜂鸣器产生不同音调的声音。  
74LVX3245 (U6) 是一个 3.3V 转 5V 的芯片，因为 DSP 引脚的电压是 3.3V 的，而蜂鸣器的驱动电路是 5V 的，所以需要这样一个电压转换，而且



---

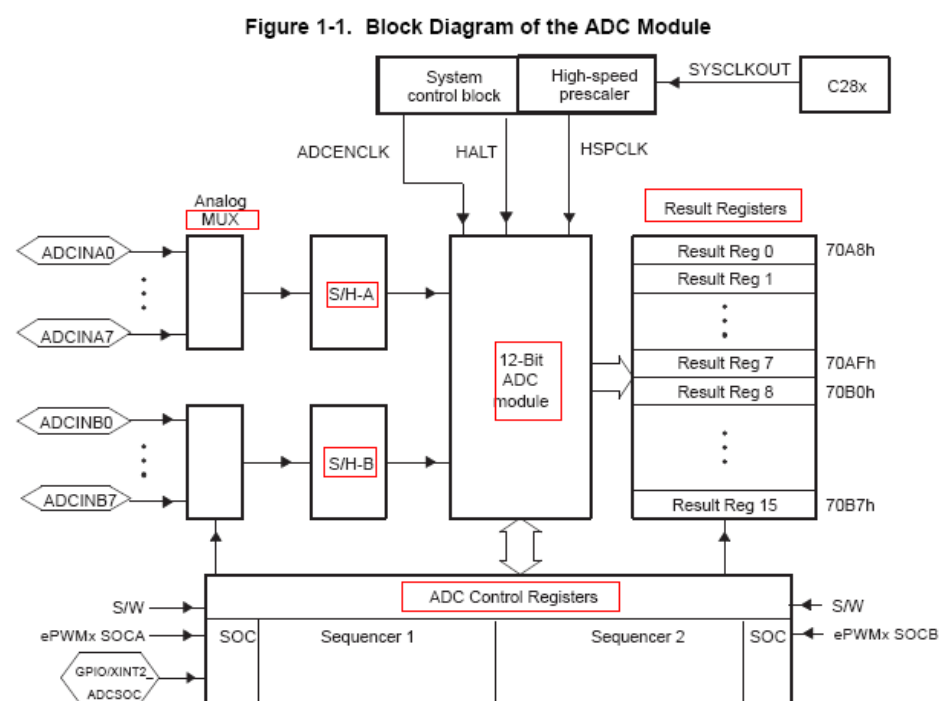
74LVX3245 还有增加电路驱动能力的作用。

---


## 第二章 ADC 采样实验

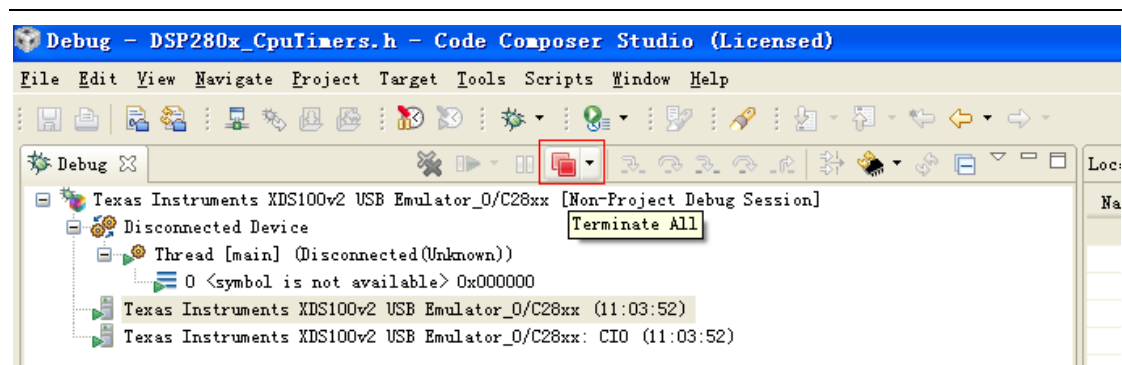
下面我来给各位讲解 TI DSP TMS320F2802 或者的 ADC（模拟数字转换）模块的内容，并配合 2802 开发板做电压测量实验。

大家对于 ADC 应该不会陌生，ADC 的作用就是将模拟量转换为数字量。ADC 一般分两类，一类为 SAR 型，也就是逐次逼近型；另一类为  $\Sigma$ - $\Delta$  型（sigma-delta 型），一般的可以认为前者转换速度快，后者稍慢一点，但是精度可以做的更到一点。在 TMS320F2802DSP 中采用的是 12bit 的 SAR 型 ADC 核心。之所以叫“核心”，而不是说“全部”，那是因为 DSP 的 ADC 模块不仅仅只是有一个将模拟量转换为数字量的核心部分，还包括多路数字选择器(MUX)、采样保持器(S/H)、结果寄存器（Result Registers）和各个 ADC 控制寄存器，如下图所示。

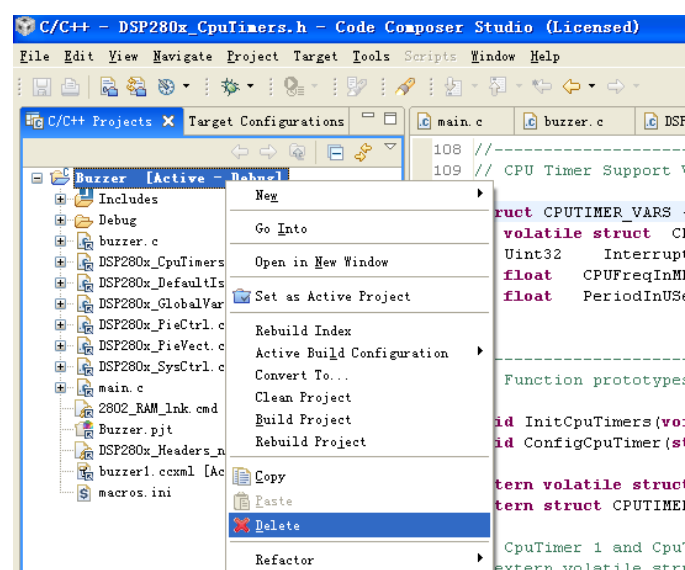


(1) 首先关掉前面打开的 project

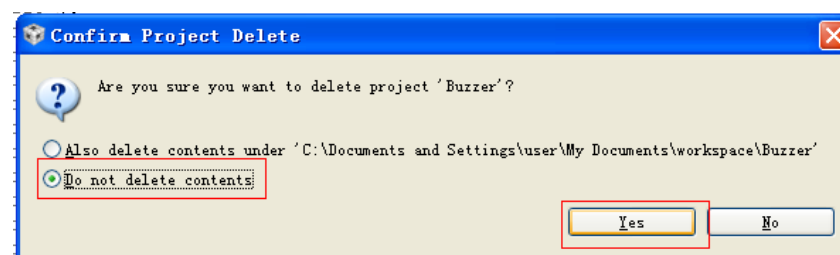
如果前面是刚刚做了别的 project 的实验，那么就一定要点击 ，这样就退出了前面的 project 的 debug，如果不点击，你后面的实验就很有可能出现很奇怪的现象。



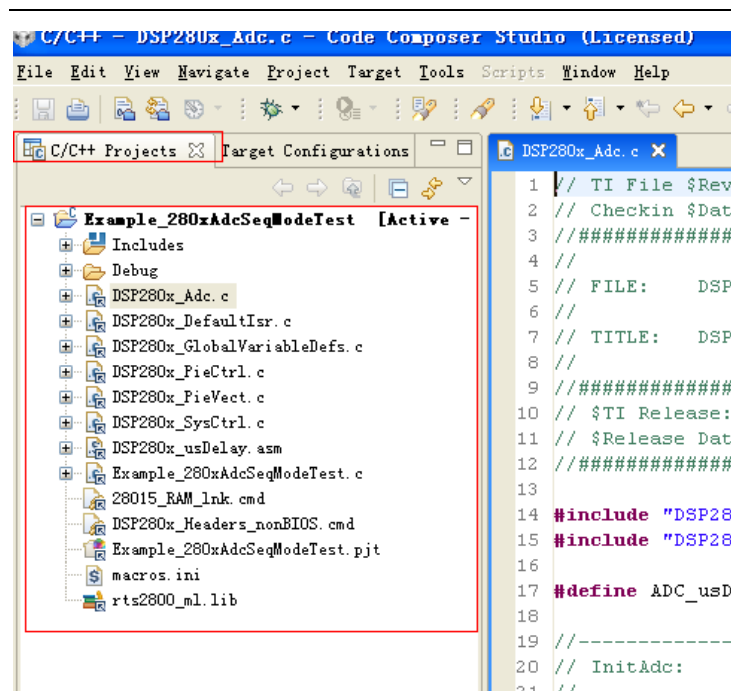
(2) 在 C/C++ project 窗口关掉前面打开的 project 关掉，方法，右键点击 project，然后选择 Delete，如下图



然后出现提示框，请选择 No，这个的意思是说只是关闭 project，而不是将这个 project 从电脑中删除掉。如果选 YES 就是把这个 project 在电脑中删除了，务必不要选择 Yes。



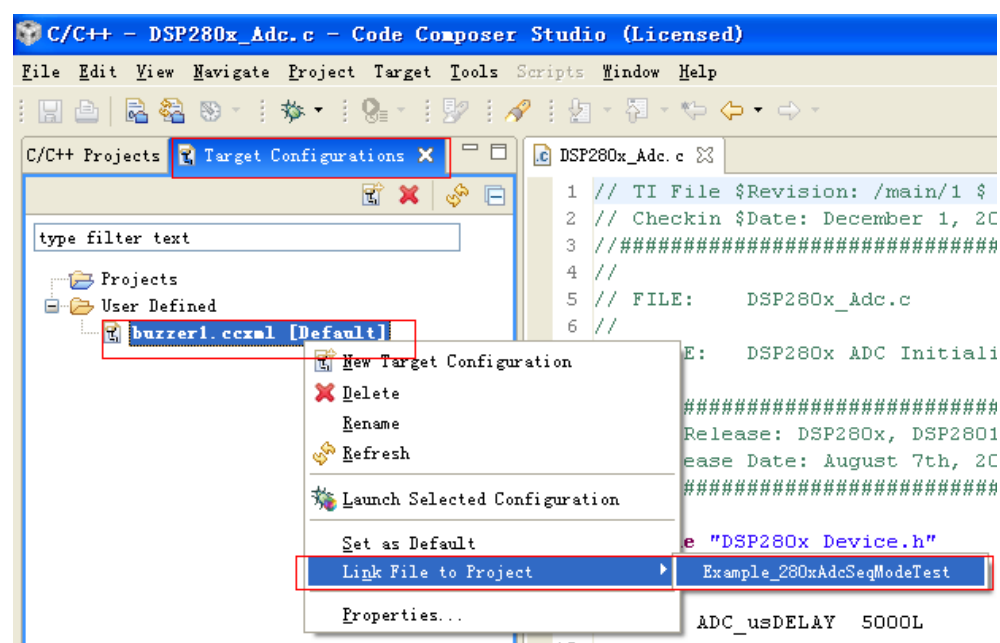
(3) 再打开 ADC 的 project，文件夹名 adc\_seqmode\_test。路径 E:\Code of TMS320F280x CCS4\DSP280x\_examples。具体的打开方法这里不赘述了。参见第一章的第 (1) 步至第 (5) 步。打开后，出现下图



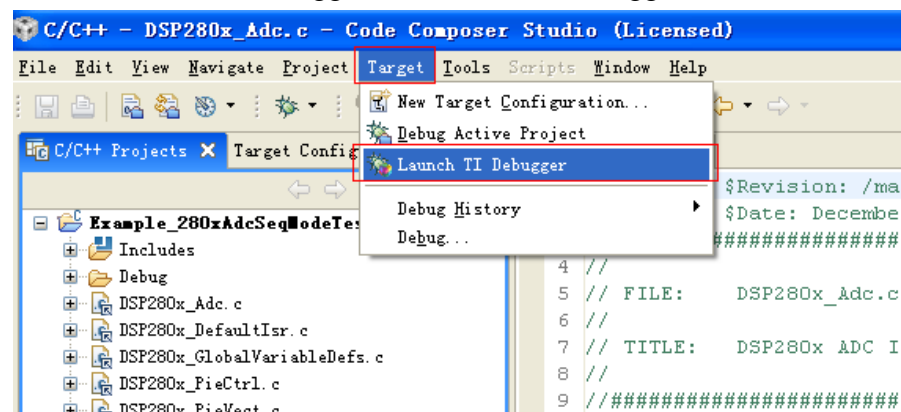
(4) 如果 project 中不包括 ccxml 文件，那么就不用移除了，如果 project 包括有 ccxml，咱们就把这个陈旧的 ccxml 移除掉，移除方法参见第一章步骤（6）。上面那个图可以看到，是没有 ccxml 这个后缀的文件的，所以不用移除了。

(5) 因为在第一章的第（7）步到第（10）步已经创建了一个新的并且与我们开发板相对应的 target configuration（也就是.ccxml 文件），所以我们这里就不需要再重新创建了。

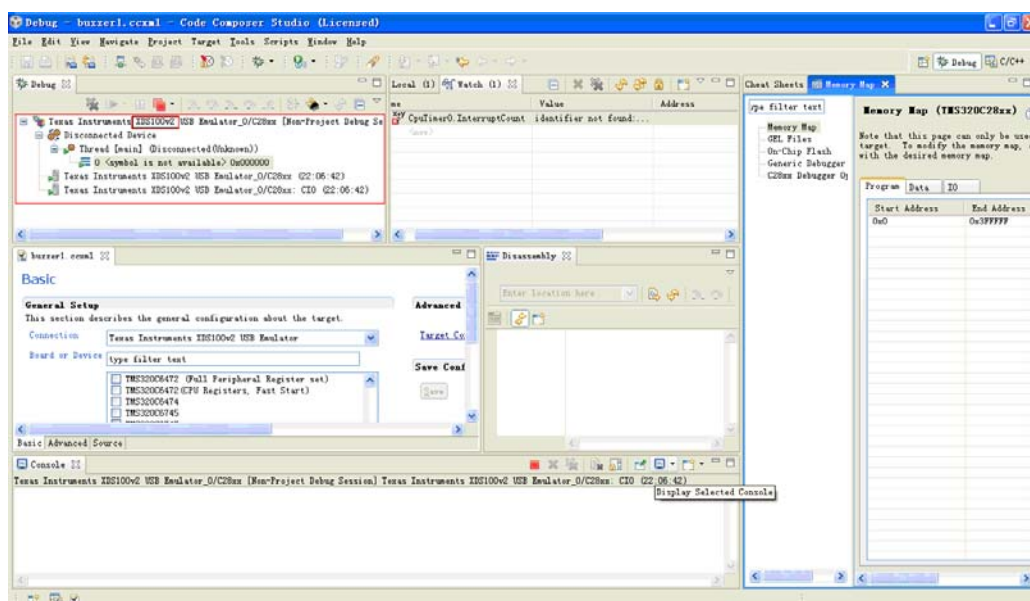
(6) 将前面已经创建好的 target configuration（也就是.ccxml 文件）link 到当前的 project Example\_280xAdcSeqModeTest，方法如下图所示，也可以参照第一章的第 11 至 12 步骤。



(7) 启动 TI 的 Debugger （Launch TI Debugger），按下图操作




然后就会进入 CCS4 的 Debug 界面，如下图




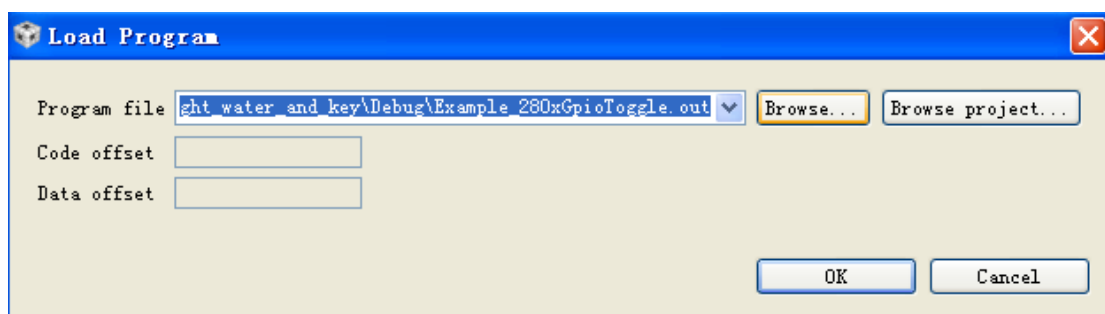
在上图红圈处是可以看出来你究竟选择的是什么的 DSP 仿真器，本实验选择的是 XDS100V2,在这里 XDS100V2 只是举例，并不代表只有 XDS100V2 才是正确的。

注意：CCS4 有两个界面，一个是 C/C++界面，一个是 Debug 界面，这两个界

面的切换，可以通过点击屏幕右上角的  来实现。

(8) 连接 DSP 仿真器和 DSP 开发板 (Conenct)，点击红圈处 ，或者快捷键 Alt+C，具体方法可以参见第一章的第 (15) 和 (16) 步骤

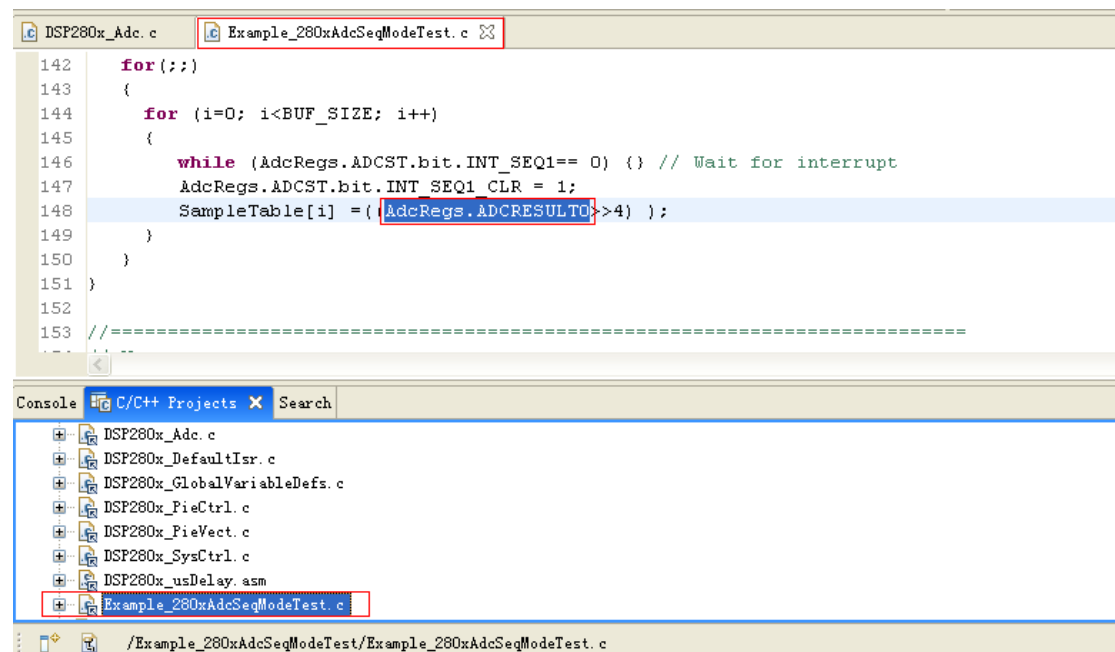
(9) 装载编译好的程序到 DSP 的 RAM 中，也就是 Load Program，方法：点击 ，或者 ctrl+alt+L，然后出现下面浏览框



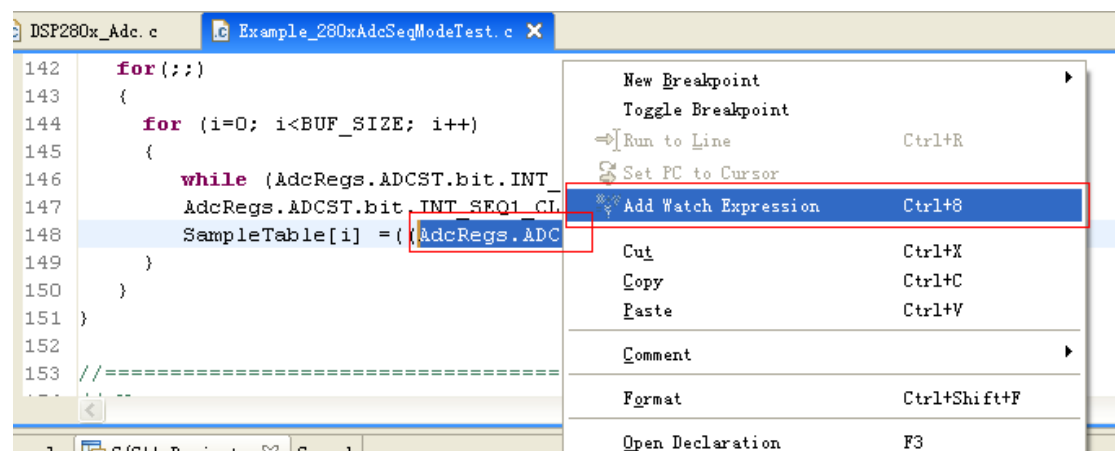
点 Browse，找到编译 project 后生成的 **Example\_280xAdcSeqModeTest.out** 文件，路径如下 E:\Code of TMS320F280x CCS4\DSP280x\_examples\adc\_seqmode\_test\Debug，然后点打开

(10) Load 完成后，还要把 Example\_280xAdcSeqModeTest.c 这个 C 文件中的变量 `AdcRegs.ADCRESULT0` 添加到 Watch Window，这样才能方便大家直观的观察到 `AdcRegs.ADCRESULT0` 的数值，操作如下：

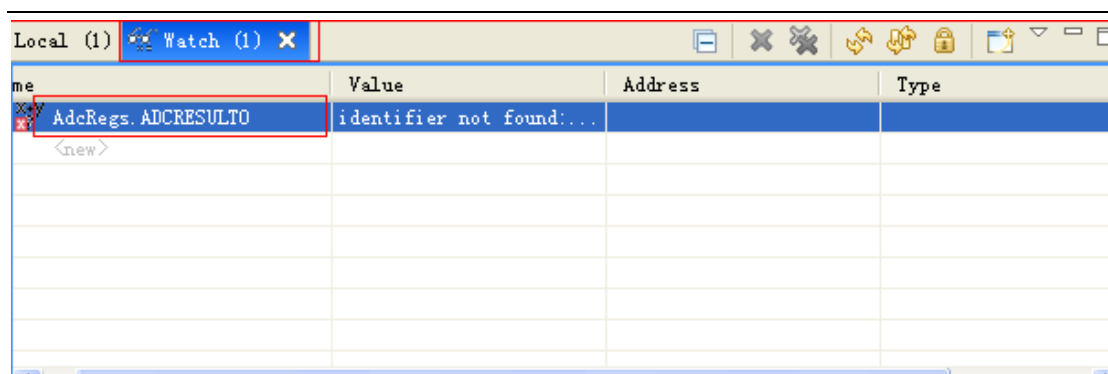
在 Debug 界面打开 C/C++ project 这个浏览框，然后双击 Example\_280xAdcSeqModeTest.c 这个文件，就会出现下图红圈处的显示，这样代表 Example\_280xAdcSeqModeTest.c 在 Debug 界面被打开了。



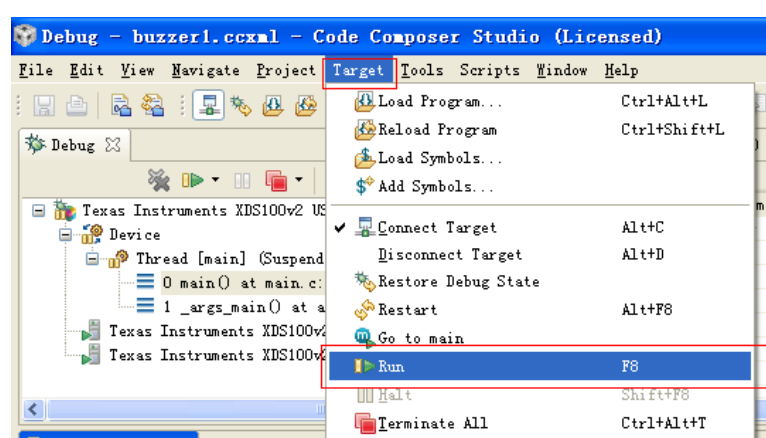
然后找到 `AdcRegs.ADCRESULT0` 用鼠标左键横扫刷成蓝色，再右键点击，选择 Add Watch Expression




添加成功后，就会出现下图所示








(11) 运行程序，点，或者快捷键 F8,或者如下图

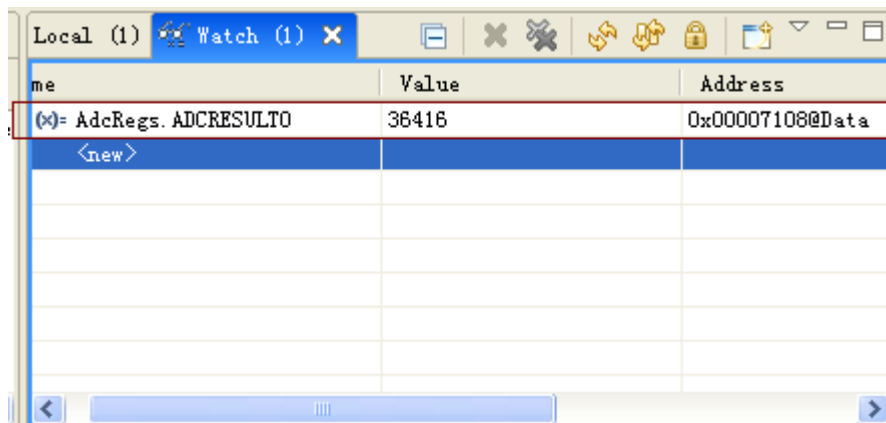


注意：如果前面没有成功 connect，那么 Run 这个图标就是灰色状态，无法点击。

运行 10-20 秒钟后，点击，就暂停程序的运行，就可以看到 watch window 中 AdcRegs.ADCRESULT0 是有数值的，这个数值就是 ADC 采样电压经过模数转换后的转换值。

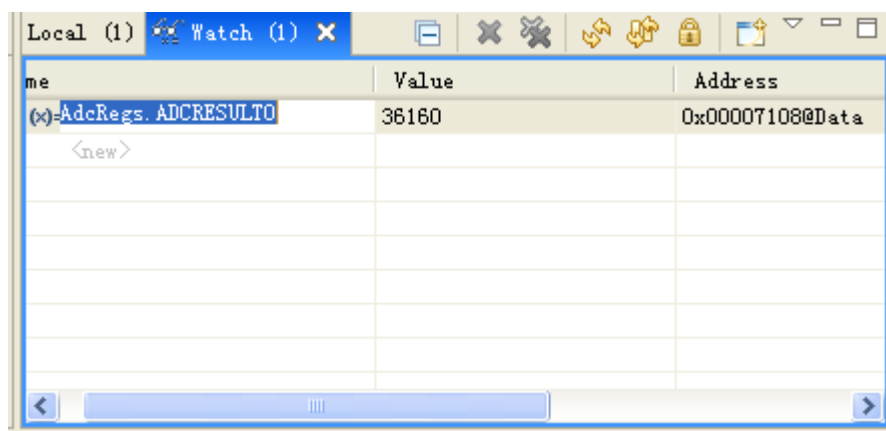
(12) 用小一字螺丝刀拧动开发板上的电位器来调节电压值，电位器在 PCB 上连接到 DSP 的 ADC 的 Pin27，也就是 ADCINB0 脚。用小的一字螺丝刀拧电位器，可以改变 ADCINB0 脚上的电压值，然后会发现 ADC 的采样值就会相应的改变，也就是说 AdcRegs.ADCRESULT0 的数值也会相应的变化。拧动后，点（reset cpu），再点（restart）再点，经过 10~20 秒钟再点击，再看 watch window 中 AdcRegs.ADCRESULT0 的数值，就会发现与前面相比是有变化的。如下图



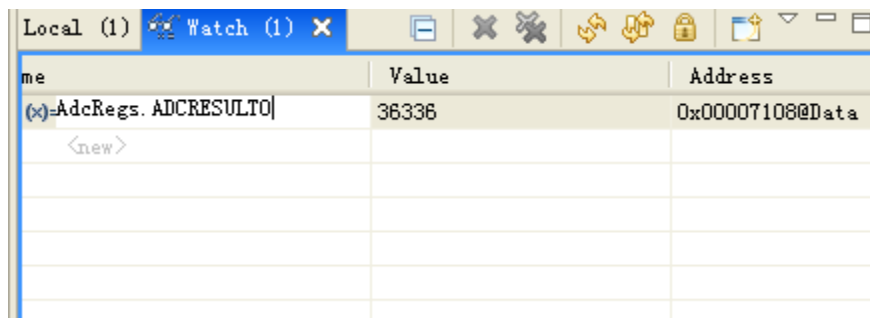


在给大家多介绍一个小技巧，可以将数学运算与 watch window 中的变量数据相结合，比如，TMS320F2808 或者 2802DSP 的内部 ADC 是 12bit 分辨率的，但是要放在 16 位的 **ADCRESULT** 寄存器中，那么就存在一个左对齐和右对齐的问题，在上面的例子中是左对齐的，所以应该有一 4 位才是真实的 ADC 数值，那么就可以如下操作。

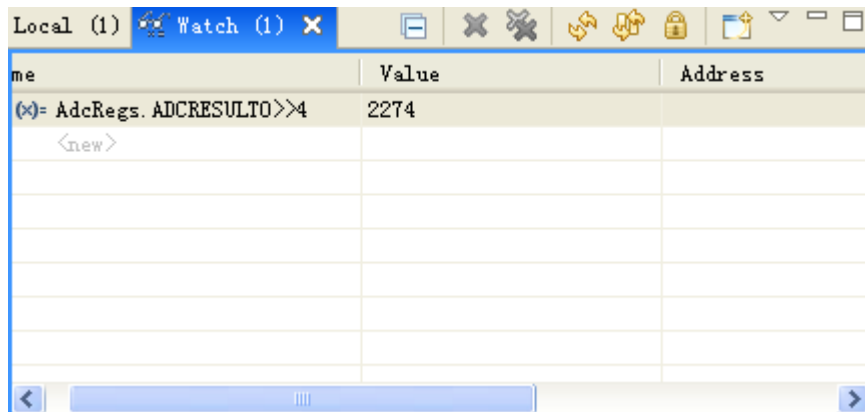
点击 **AdcRegs.ADCRESULT0**，然后出现下图那种状态



然后再点击一下 **AdcRegs.ADCRESULT0**，出现下图



在上图这种状态，可以添加数学运算符号，比如我们输入>>4,就是将 `AdcRegs.ADCRESULT0` 左移 4 位



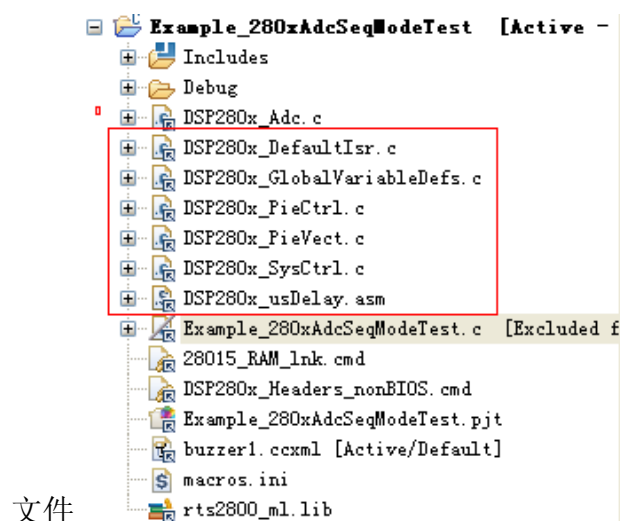
而且不仅仅是左移右移,只要是 C 语言允许的数学运算都可以在 Watch Window 进行添加。

格外注意：在运行程序一段时间有停止程序后，并且修改程序再重新 Load Program, 这样如果要再次运行，一定要 Reset CPU 和 Restart, 如下图。否则 Load Program 后直接运行，就会出现 `AdcRegs.ADCRESULT0` 变量的数值观察全都是错误的值的现象。

\*\*\*\*\*  
\*\*\*\*\*

至此，初步的实验就可以告一段落。授之以鱼，不如授之以渔。下面我就给大家讲讲，如何自学 TI DSP 的编程。我这里就是把 TI DSP 在 CC2S4.开发环境下的编程的一般性、有规律性的东西介绍给大家。下面这段内容比较重要，都是精华啊！简短捷说，切入正题

大家看多了 DSP 的程序就会发现,几乎每一个 project 都包含如下这么几个.c



文件

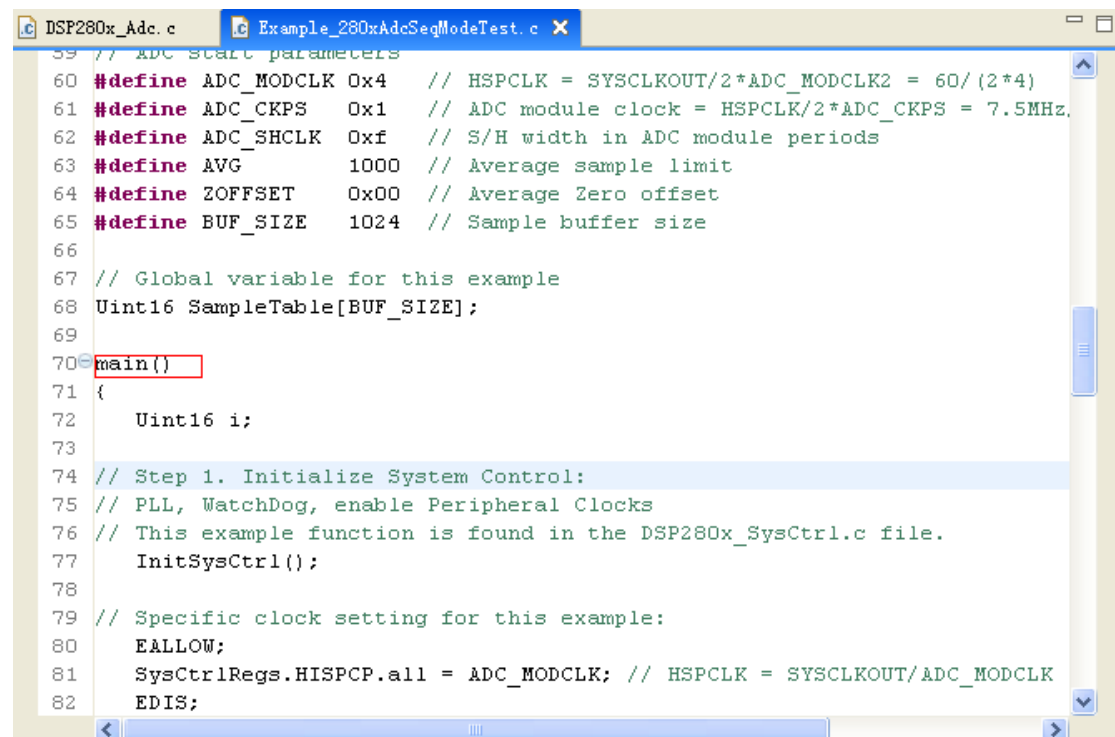
在 Example\_280xAdcSeqModeTest.pjt 这个 project 中最关键的.c 文件是 Example\_280xAdcSeqModeTest.c, 其他的.c 文件, 比如:

DSP280x\_GlobalVariableDefs.c 、 DSP280x\_PieCtrl.c 、 DSP280x\_PieVect.c 、

DSP280x\_SysCtrl.c 、 DSP280x\_usDelay.asm 可以说是“必备通用配件”, 就是

说少了这些.c 文件还不行, 不同的 project 还都用这些相同的.c 文件。我建议 DSP 的初学者入门时先不要理会这些.c 文件, 以后 DSP 水平进阶了再仔细研究。我比较推崇的就是学东西一定要分清主次, 因为时间和精力是有限的, 要把有限的精力用在最重要的事情上, 而不能浪费在边边角角的事情上。

双击打开 Example\_280xAdcSeqModeTest.c, 看到如下图



```
39 // ADC start parameters
60 #define ADC_MODCLK 0x4 // HSPCLK = SYSCLKOUT/2*ADC_MODCLK2 = 60/(2*4)
61 #define ADC_CKPS 0x1 // ADC module clock = HSPCLK/2*ADC_CKPS = 7.5MHz.
62 #define ADC_SHCLK 0xf // S/H width in ADC module periods
63 #define AVG 1000 // Average sample limit
64 #define ZOFFSET 0x00 // Average Zero offset
65 #define BUF_SIZE 1024 // Sample buffer size
66
67 // Global variable for this example
68 Uint16 SampleTable[BUF_SIZE];
69
70 main()
71 {
72     Uint16 i;
73
74     // Step 1. Initialize System Control:
75     // PLL, WatchDog, enable Peripheral Clocks
76     // This example function is found in the DSP280x_SysCtrl.c file.
77     InitSysCtrl();
78
79     // Specific clock setting for this example:
80     EALLOW;
81     SysCtrlRegs.HISPCP.all = ADC_MODCLK; // HSPCLK = SYSCLKOUT/ADC_MODCLK
82     EDIS;
```

找到 main()函数,这是主函数,是 Example\_280xAdcSeqModeTest.c 这个 C 文件中最主要的函数。main()函数中主要包括以下几个部分

(1) **InitSysCtrl()**; 这个函数的原型在 DSP280x\_SysCtrl.c 文件中。这个函数是初始化 DSP 的内核系统控制的(Initialize System Control),包括 锁相环(PLL),看门狗(WatchDog)和片内外围模块的时钟(Peripheral Clocks)的初始化。初学者,根本不用管 InitSysCtrl()内部是什么,以后 DSP 水平进阶了再仔细研究。

(2) **InitPieCtrl()**;这个函数的原型在 DSP280x\_PieCtrl.c 文件中。这个函数用来将 PIE 控制寄存器初始化为默认状态。所谓的默认状态就是指禁用所有的 PIE,并将所有的 PIE 中断标志位清零。还是建议初学者不用管 InitPieCtrl()内部是什么,以后 DSP 水平进阶了再仔细研究。

**小贴士:** PIE 是 Peripheral Interrupt Expansion (外围模块中断扩展)的缩写,

详细内容请看附赠光盘中的文档 [spru712g\(TMS320x280x, 2801x, 2804x DSP System Control and Interrupts Reference Guide\).pdf](#) 。

(3) [InitPieVectTable\(\)](#);这个函数的原型在 `DSP280x_PieVect.c` 文件中。这个函数是初始化 PIE 矢量表，通过函数指针将中断服务例程 (ISR) 的与中断的发生联系起来，举例来说当中断 `SEQ1INT_ISR` 发生了，通过这个 PIE 矢量表，程序就会自动跳转到 `SEQ1INT_ISR` 相对应的子函数。这里暂时不太懂没关系的，因为今天暂时用不到。那为什么用不到还要调用这个 [InitPieVectTable\(\)](#)呢？那是因为 [InitPieVectTable\(\)](#)对于 CCS3.3 下的 Debug 是必需的。

(4) [InitAdc\(\)](#);的作用是将 DSP 内部的 Reference 和 bandgap 上电，不重要。关键的内容在下面那几句话，如下图所示，对于这几行语句我会较详细的介绍一下。

```
// Specific ADC setup for this example:
AdcRegs.ADCCTRL1.bit.ACQ_PS = ADC_SHCLK;
AdcRegs.ADCCTRL3.bit.ADCCLKPS = ADC_CKPS;
AdcRegs.ADCCTRL1.bit.SEQ_CASC = 1;           // 1 Cascaded mode
AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x8;      //zy
AdcRegs.ADCCTRL1.bit.CONT_RUN = 1;          // Setup continuous run
```

我以下面这句话 `AdcRegs.ADCCTRL1.bit.ACQ_PS = ADC_SHCLK;`为例，给各位仔细讲解一下，如何看懂 DSP 的 C 程序。首先，上面是一个结构体写法，这个大家都明白。其次，`AdcRegs` 代表 ADC 寄存器，`.ADCCTRL1` 代表 ADC 的 Control Register 控制寄存器，`ADCCTRL1` 与 TI 官方文档 [spru716d.pdf《TMS320x280x, 2801x, 2804x DSP Analog-to-Digital Converter \(ADC\)Reference Guide》](#) 中的写法是一致的，TI 的所有程序都是以这样的形式来写的，也就是说，看到程序中的 `ADCCTRL1`，再到文档中搜索 `ADCCTRL1`，就能找到 `ADCCTRL1` 这个寄存器的详细描述，具体到 `ADCCTRL1` 的每一个 bit 是什么意思，干什么用的。`ACQ_PS` 代表 `ADCCTRL1` 寄存器中的 `ACQ_PS[3:0]`，在文档 [spru716d.pdf](#) 中第 34 页有如下内容。

		11	Mode 3. Sequencer and Outer wrapper logic stops immediately on emulation suspend.
11-8	ACQ_PS[3:0]		Acquisition window size. This bit field controls the width of SOC pulse, which, in turn, determines for what time duration the sampling switch is closed. The width of SOC pulse is <code>ADCCTRL1[11:8] + 1</code> times the <code>ADCLK</code> period.

强调一点，大家没有必要强求自己记住 ADC 模块的每一个寄存器，只需要做到了解，在需要使用 ADC 寄存器的时候，能够在文档中找到详细描述这个 ADC 寄存器相应的位置即可。

按照上面所讲的道理，下面这句话 `AdcRegs.ADCCTRL1.bit.SEQ_CASC = 1;`就是说将 `ADCCTRL1` 的 `SEQ_CASC` 位赋值为 1.至于 `SEQ_CASC` 位赋值为 1 究竟代表什么意思，看下图

4	SEQ_CASC		Cascaded sequencer operation. This bit determines whether SEQ1 and SEQ2 operate as two 8-state sequencers or as a single 16-state sequencer (SEQ).
		0	Dual-sequencer mode. SEQ1 and SEQ2 operate as two 8-state sequencers.
		1	Cascaded mode. SEQ1 and SEQ2 operate as a single 16-state sequencer (SEQ).

再讲讲在程序中 `AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x8` 这句话。  
**ADCCHSELSEQ1** 是 **ADC Input Channel Select Sequencing Control Registers1** 的缩写, 就是 **ADC** 输入通道选择定序控制器, 一共有四个类似 **ADCCHSELSEQ1** 这样的寄存器, 分别是 **ADCCHSELSEQ1**、**ADCCHSELSEQ2**、**ADCCHSELSEQ3**、**ADCCHSELSEQ4**, 详细内容在 `spru716d.pdf` 文档中第 43 页。

**Table 2-11. CONVnn Bit Values and the ADC Input Channels Selected**

CONVnn Value	ADC Input Channel Selected
0000	ADCINA0
0001	ADCINA1
0010	ADCINA2
0011	ADCINA3
0100	ADCINA4
0101	ADCINA5
0110	ADCINA6
0111	ADCINA7
1000	ADCINB0
1001	ADCINB1
1010	ADCINB2
1011	ADCINB3
1100	ADCINB4
1101	ADCINB5
1110	ADCINB6
1111	ADCINB7

通过开发板的原理图可以看到, **TMS320F2802** 或者 **2808** 开发板上电位器是连到 **ADC** 的 **B0** 通道, 而 **B0** 通道对应 **1000**(二进制), 所以才将 **ADCCHSELSEQ1** 寄存器的 **CONV00** 赋值为 **0x8**。假如果 **TMS320F2802** 或者 **2808** 开发板上电位器是连到 **ADC** 的 **A0** 通道, 那么就赋值为 **0x0**。

IA_ADIN	23	ADCINA0
AJ1	27	ADCINB0
IB_ADIN	22	ADCINA1
VDC_IN	28	ADCINB1
AD_1	21	ADCINA2
AD_2	29	ADCINB2
AD_3	20	ADCINA3
AD_4	30	ADCINB3
AD_5	19	ADCINA4
AD_6	31	ADCINB4
AD_7	18	ADCINA5
AD_8	32	ADCINB5
	17	ADCINA6
	33	ADCINB6
AD_10	16	ADCINA7
	34	ADCINB7

至此，DSP 的 ADC 基本内容就介绍到这里，后面就需要大家自己去实践去看数据手册来深入学习了，要想研究透彻 DSP 的 ADC 模块的详细内容请看 [spru716d\(TMS320x280x, 2801x, 2804x DSP ADC Reference Guide\).pdf](#), 路径在 光盘：（非常重要）TI DSP TMS320x280x 系列芯片资料\TI TMS320F280xDSP 的用户指南\。

---

## 第三章 ePWM 实验

### 1、导言

我前面强调过TI的TMS320F2802和TMS320F2808 DSP 是TMS320F2812的增强版，在PWM模块这里就是一个非常明显体现。

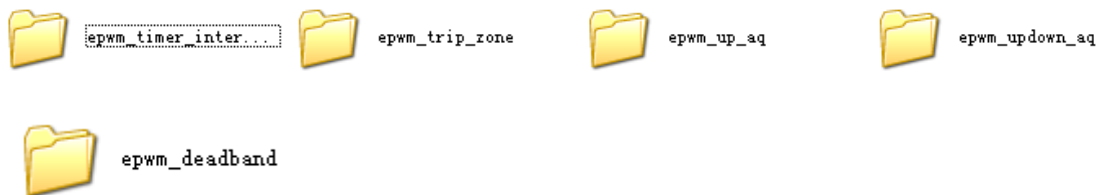
首先，在名字上TMS320F2802和TMS320F2808的PWM模块叫ePWM，这个e是Enhanced的头字母，就是增强的意思。

其次，在2812中只有2个Event Manager，其中包括PWM、CAP和QEP三部分，这三部分是共享16bitTimer，而且CAP和QEP之间会共用3个的IO通道，这些对于实际项目都是非常不方便的。相比较，在TMS320F2802和TMS320F2808中是将PWM、CAP和QEP分别独立开来，这三者之间再也没有干扰和重叠，而且性能也有所加强。

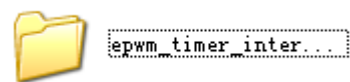
### 2、TMS320F280x 系列 DSP 的 ePWM 总体介绍

今天我来给各位讲解TI DSP TMS320F2802和TMS320F2808的ePWM（ePWM是 Enhanced Pulse Width Modulator 的缩写）模块的内容，并配合TMS320F2802或TMS320F2808开发板做电压测量实验。本教程要将3个实验。其中第一个实验需要使用示波器来观察TMS320F2802或TMS320F2808开发板产生的PWM波，这样就可以直观的看到PWM波，另外2个实验不需要示波器。

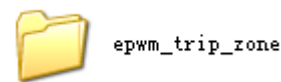
在本教程中总共有5个实验中，分别使用的project如下



这五个project的实验目的分别是



掌握DSP的PWM模块的中断（Interrupt）原理和设置方法。



掌握DSP的PWM模块的跳闸（trip zone）的原理和设置方法，所谓的跳闸（trip）就可以理解成“刹车”，简单的说，在DSP上有6个引脚

分别是TZ1、TZ2到TZ6，在这些引脚上有电平信号的跳变，就会立即触发DSP的PWM模块停止输出PWM波，这个对于电机控制来说是非常重要和实用的。在实际应用中，一般是将DSP的TZ1到TZ6的其中的一个引脚（比如TZ1）与IPM功率模块的过电流报警信号相连接，当IPM发生电流过大情况时，IPM模块会产生一个过电流信号（比如一个2us的低电平）传到DSP的TZ1，这时就会触发DSP的PWM的trip zone，于是PWM模块立即停止输出PWM波，这样IPM模块内的IGBT

---

桥臂就关断了，于是电流没有了，也就避免了因为电流过大而烧坏IPM模块。



epwm\_up\_aq

掌握DSP的PWM模块的AQ模块（Action-Qualifier的缩写）的原理和设置方法。通俗的讲，就是通过AQ模块改变PWM波形的形状，但是AQ模块的作用不仅仅是这些。

Specify the type of action taken when a time-base or counter-compare submodule event occurs 这是AQ模块的一句话概括，翻译过来就是，当时基子模块（time-base submodule）或者计数比较子模块（counter-compare submodule）有事件（event）发生时，由AQ模块指定PWM发生的行为类型。现在对这句话会感觉非常费解，通过后面的学习就理解了。



epwm\_updown\_aq

掌握DSP的PWM模块的上下计数（也叫中心计数）方式的原理和设置方法。



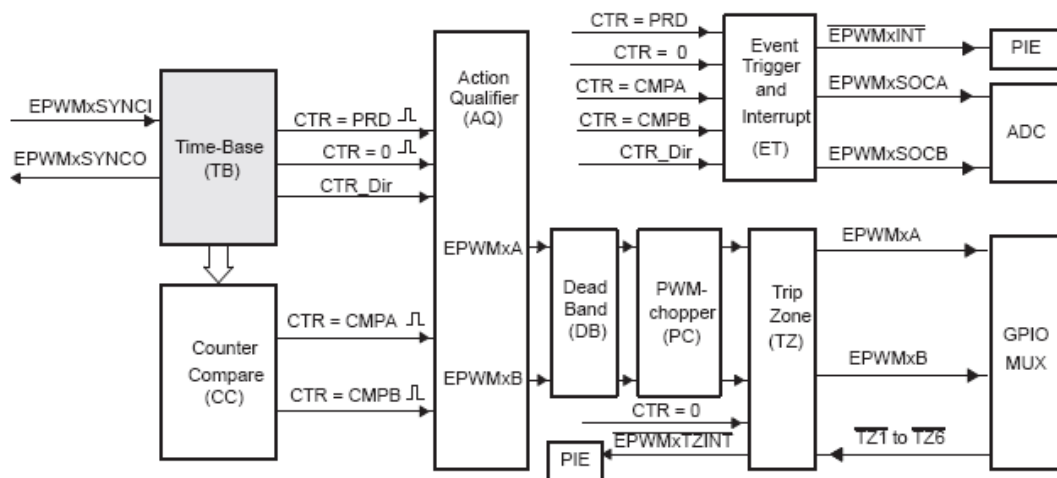
epwm\_deadband

掌握DSP的PWM模块的死区（deadband）子模块设置。

PWM模块是DSP非常重要和实用的一个部分。PWM模块包括7个子模块，如下

- (1) Time-base (TB) 时基子模块
- (2) Counter-compare (CC) 计数比较模块
- (3) Action-qualifier (AQ) 行为指定模块
- (4) Dead-band (DB) 死区模块
- (5) PWM-chopper (PC) PWM斩波模块
- (6) Trip-zone (TZ) 跳闸模块
- (7) Event-trigger (ET) 事件触发模块

各个模块的结构概况图如下



从上图中可以看出，PWM的每个子模块“消费”哪些信号，又“生产”出哪些



---

信号。

对DSP的PWM模块有一个大概的了解后，咱们开始做实验，有一个感性的认识后，咱们再详细的讲解PWM模块的各个子模块。

### 3、实验步骤

因为在第一章和第二章对于CCS4.2的操作步骤已经介绍的很详细了，这里就不赘述了，只是大体上描述一下实验的步骤，如果各位朋友有疑惑，请参照第一章的内容。

(1) 首先关掉前面打开的 project

(2) 再打开 PWM 的 project，文件夹名 epwm\_updown\_aq\_1。路径 E:\Code of TMS320F280x CCS4\DSP280x\_examples\。具体的打开方法这里不赘述了。参见第一章的第 1 步至第 5 步。打开后，出现下图


(3) 如果 project 中不包括 ccxml 文件，那么就不用移除了，如果 project 包括有 ccxml，咱们就把这个陈旧的 ccxml 移除掉，移除方法参见第一章步骤 (6)。上面那个图可以看到，是没有 ccxml 这个后缀的文件的，所以不用移除了。

(4) 因为在第一章的第 (7) 步到第 (10) 步已经创建了一个新的并且与我们开发板相对应的 target configuration (也就是.ccxml 文件)，所以我们这里就不需要再重新创建了。

(5) 将前面已经创建好的 target configuration (也就是.ccxml 文件) link 到当前的 project，Example\_280xEPwmUpDownAQ.pjt 方法如下图所示，也可以参照第一章的第 11 至 12 步骤。


(6) 启动 TI 的 Debugger (Launch TI Debugger)，点 Target——>Launch TI Debugger

(7) 连接 DSP 仿真器和 DSP 开发板 (Conenct)，点击红圈处 ，或者快捷键 Alt+C，具体方法可以参见第一章的第 (15) 和 (16) 步骤

(8) 装载编译好的程序到 DSP 的 RAM 中，也就是 Load Program，方法：点击 ，或者 ctrl+alt+L，编译好的 out 文件路径

E:\Code of TMS320F280x CCS4\DSP280x\_examples\epwm\_updown\_aq\_1\Debug

(9) Load 完成后，还要把 Example\_280xEPwmUpDownAQ.c 这个 C 文件中的结构体变量 epwm1\_info 添加到 Watch Window，并且添加后点开 epwm1\_info 前面的小+，这样就可以看到这个结构体内部的各个子部分。

(10) 点 ，运行 project，再暂停，会发现，在 Watch Window 中 EPwmTimerIntCount 的数据由 0 到 10 进行变化。

---

Local (1) Watch (1)		
Name	Value	Address
epwm1_info	{...}	0x00008914@Data
EPwmRegHandle	0x00006800	0x00008914@Data
EPwm_CMPA_Direction	0	0x00008916@Data
EPwm_CMPB_Direction	1	0x00008917@Data
EPwmTimerIntCount	7	0x00008918@Data
EPwmMaxCMPA	1950	0x00008919@Data
EPwmMinCMPA	50	0x0000891A@Data
EPwmMaxCMPB	1950	0x0000891B@Data
EPwmMinCMPB	50	0x0000891C@Data
<new>		

然后请有条件的朋友，将示波器的探头连接到板子右上方的20针JTAG口的Pin2、4、6、8、10、12，然后就可以看到6路占空比变化的PWM波了。

20针JTAG的Pin2电路上是通過SN74LVX3245芯片连接到DSP的PWM3A

20针JTAG的Pin4电路上是通過SN74LVX3245芯片连接到DSP的PWM3B

20针JTAG的Pin6电路上是通過SN74LVX3245芯片连接到DSP的PWM2A

20针JTAG的Pin8电路上是通過SN74LVX3245芯片连接到DSP的PWM2B

20针JTAG的Pin10电路上是通過SN74LVX3245芯片连接到DSP的PWM1A

20针JTAG的Pin12电路上是通過SN74LVX3245芯片连接到DSP的PWM1B

SN74LVX3245芯片是一个非常常用并且重要的3.3V转5V芯片，不仅仅是电压转换，还提高了电流驱动能力，还起到了隔离和保护的作用。

\*\*\*\*\*

#### 4、讲解 Example\_280xEPwmUpDownAQ.pjt 中的各个c语句的含义

(1) **InitSysCtrl();** 是系统控制初始化函数，主要是时钟设置、看门狗设置等等。

**InitEPwm1Gpio();**

**InitEPwm2Gpio();**

(2) **InitEPwm3Gpio();**

是将PWM模块对应的GPIO0、GPIO1、GPIO2、GPIO3、GPIO4、GPIO5口设置成在PWM模式，而不是做一般的IO口模式。

**InitEPwm1Example();**

**InitEPwm2Example();**

(3) **InitEPwm3Example();**

是用来初始化EPWM1、EPWM2、EPWM3模块的函数，本质上就是对EPWM的各个寄存器进行设置，从而实现用户想要的效果。

---

```
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
PieCtrlRegs.PIEIER3.bit.INTx2 = 1;
PieCtrlRegs.PIEIER3.bit.INTx3 = 1;
```

(4)

使能ePWM1、ePWM2、ePWM3这三个中断

```
for(;;)
{
    asm("        NOP");
}
```

(5)

无限循环空函数，所有的DSP程序都必须是一个无限循环函数。绝大多数都是通过一个定时中断，来不断的触发中断，然后进入中断服务子程序进行相应的程序运算等等。

(6) 在本程序中一共使能了ePWM1、ePWM2、ePWM3三个中断，中断服务子函数分别如下3个，可以发现，中断服务子函数的前面一定要用 **interrupt** 来定义这个函数的类型，这是TI的DSP程序所特有的一种写法。

```
interrupt void epwm1_isr(void)
```

```
interrupt void epwm2_isr(void)
```

```
interrupt void epwm3_isr(void)
```

(7) 在这个pj1中，通过下面这三句话，将三个中断服务子函数的地址映射到中断向量表中的相应地址位置。**注意**：必须要有下面这三句话，否则当中断发生了，DSP的CPU内核也不知道应该要**跳转**到哪个函数去运行，于是程序就跑飞了。

```
EALLOW; // This is needed to write to EALLOW protecte
PieVectTable.EPWM1_INT = &epwm1_isr;
PieVectTable.EPWM2_INT = &epwm2_isr;
PieVectTable.EPWM3_INT = &epwm3_isr;
EDIS;    // This is needed to disable write to EALLOW.
```

(8) 中间插播一段，经常有买家朋友问我的一个问题，就是EALLOW; EDIS; 这两句话是什么意思。借此机会给大家讲解一下，EALLOW是用来解除寄存器的写操作保护的，也就是说，想要对一些关键的寄存器进行写操作，那么就必须在前边先加上EALLOW这句话，否则你对寄存器的写操作就是无效的，并且这个在project进行build的时候是不会提示error的。如果你不注意到这点，那么可能会发生，明明自己写了寄存器，但是程序运行起来，什么反应都没有，于是开始郁闷，开始满程序的去找自己是哪里写错了，呵呵，这个事情在我刚刚接触DSP的时候就经历过。EDIS是EALLOW的相反操作，也就是使DSP的寄存器的写操作被禁止，为什么要有EDIS这个呢，这个是DSP为了防止程序运行中出现意外和错误是，错误的对DSP的寄存器进行一些随机性的操作，从而导致不可预料的后果。DSP的关键寄存器在DSP的reset后的默认状态就是禁止寄存器写操作。

---

```

interrupt void epwm1_isr(void)
{
    // Update the CMPA and CMPB values
    update_compare(&epwm1_info);
    // Clear INT flag for this timer
    EPwm1Regs.ETCLR.bit.INT = 1;
    // Acknowledge this interrupt to receive mo.
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
(9)

```

黑圈处的语句的作用是清除DSP的中断标志位，这个是进中断首先要做的第一件事，否则后面就会连续不停的进中断，就是说，会发生本来不应该进中断的时候，又进中断了。

红圈处的语句的作用是应答（Acknowledge）这个中断，这样才能继续接收到后面发生的中断，也就是说，如果没有红圈处语句，那么以后真正发生中断了，应该进中断的时候，又进不去中断了。

## 5、ePWM模块的各个寄存器设置简要介绍

下面我以 `void InitEPwm1Example()` 为例，给大家讲解一下EPWM的各个寄存器设置，其实所谓的弄懂DSP的PWM模块，就是把PWM的这些寄存器弄懂。

### 5.1 PWM的TB子模块（时基子模块）

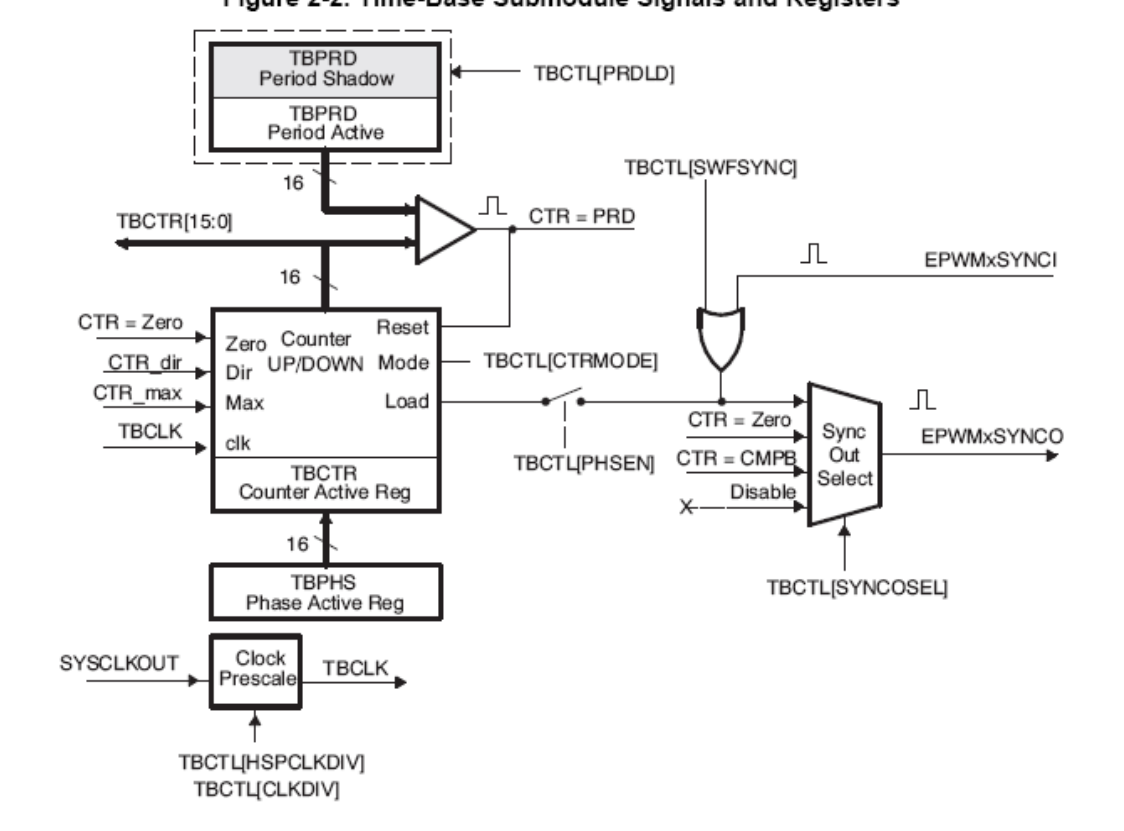
```

EPwm1Regs.TBPRD = EPWM1_TIMER_TBPRD;
EPwm1Regs.TBPHS.half.TBPHS = 0x0000;
EPwm1Regs.TBCTR = 0x0000;

// Setup counter mode
EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up
EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE; // Disable phase loading
EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1; // Clock ratio to SYSCLKOUT
EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;

```

上面这几句话是对PWM的TB子模块（时基子模块）进行设置，TB子模块的详细框图如下



TB子模块一共有6个寄存器，其中比较重要的是TBCTL寄存器，每一个寄存器的每一位是什么意思，我就不给大家讲了，自己可以自己看我给大家的光盘中的《spru791f(TMS320x280x, 2801x, 2804x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide).pdf》这个手册中的第96页到第99页。

Table 2-2. Time-Base Submodule Registers

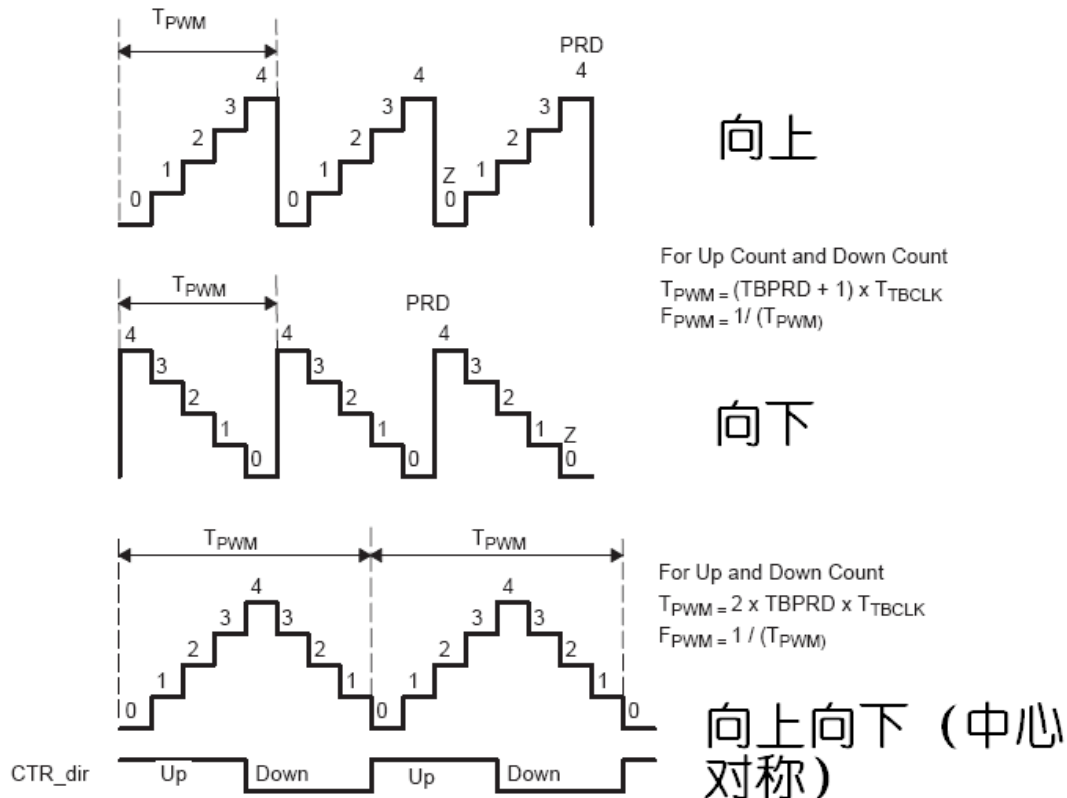
Register	Address offset	Shadowed	Description
TBCTL	0x0000	No	Time-Base Control Register
TBSTS	0x0001	No	Time-Base Status Register
TBPMSHR	0x0002	No	HRPWM Extension Phase Register <sup>(1)</sup>
TBPHS	0x0003	No	Time-Base Phase Register
TBCTR	0x0004	No	Time-Base Counter Register
TBPRD	0x0005	Yes	Time-Base Period Register

我就挑其中比较重要的一点讲一下，就是下面这句话

```
EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;  
#define TB_COUNT_UPDOWN 0x2
```

从TB\_COUNT\_UPDOWN这个英文字面意思就可以看出来一些门道，就是TB向上向下计数模式（也叫中心对称模式），与这个向上向下相对应的另外两种模式是，TB\_COUNT\_UP（向上模式），TB\_COUNT\_DOWN（向下模式）。通过下图，大家就比较好理解了。

Figure 2-3. Time-Base Frequency and Period





因为我们上面打开的project是 Example\_280xEPwmUpDownAQ.pjt，从字面意思就看出来了，我们这个project选择的的就是向上向下模式（中心对称模式），正

是因为 `EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;` `#define TB_COUNT_UPDOWN 0x2` 这句话的存在，我们这个pjt才配得上 Example\_280xEPwmUpDownAQ.pjt 这个名字，否则就是名不符实了，呵呵。

现在大家猜到在 epwm\_up\_aq 文件夹中的 Example\_280xEPwmUpAQ.pjt 这个pjt与上面这个 Example\_280xEPwmUpDownAQ.pjt 之间的区别是什么了吧。就是在 `EPwm1Regs.TBCTL.bit.CTRMODE` 这里的设置不同，所以明白了 Example\_280xEPwmUpDownAQ.pjt 这个，也就自然而然明白了 Example\_280xEPwmUpAQ.pjt。

想要更加直观的理解这个Up计数模式和UpDown计数模式的区别，可以分

别打开  Example\_280xEPwmUpAQ.pjt 和  Example\_280xEPwmUpDownAQ.pjt ，分别观察 EPwm1Regs.TBCTR 这个寄存器的数值变化，就会发现Up计数模式，数值是从0增大到EPwm1Regs.TBPRD再直接变为0，而在UpDown计数模式，数值是从0增大到EPwm1Regs.TBPRD再逐步减小到0，

## 5.2 PWM的CC子模块（计数比较子模块）

下面说说PWM的CC子模块（计数比较子模块），程序中就这么6句话是涉及CC子模块的。

```
EPwm1Regs.CMPA.half.CMPA = EPWM1_MIN_CMPA;    // Set compare A value
EPwm1Regs.CMPB = EPWM1_MAX_CMPB;              // Set Compare B value

// ...
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;
```

其中CC\_SHADOW的意思就是说，对ePWM的CMPA和CMPB寄存器的写数据并不是立即生效的，而是先进入SHADOW寄存器，然后在PWM的计数器的值变为0的时候，或者最大的时候，将数据从SHADOW寄存器转入到CMPA和CMPB寄存器。CC\_CTR\_ZERO这里的意思就是设置为在PWM的计数器的值变为0的时候将数据从SHADOW寄存器转入到CMPA和CMPB寄存器。

通过改变CMPA和CMPB的值就可以改变PWM的占空比，在 `void update_compare(EPWM_INFO *epwm_info)` 这个函数中就是这样应用的，如果你不明白画黑圈处的“->”是什么意思，就说明你的C语言比较薄弱，请回去自己补充一下C语言的结构体的知识。

```
void update_compare(EPWM_INFO *epwm_info)
{
    // Every 10'th interrupt, change the CMPA/CMPB values
    if(epwm_info->EPwmTimerIntCount == 10)
    {
        epwm_info->EPwmTimerIntCount = 0;

        // If we were increasing CMPA, check to see if
        // we reached the max value. If not, increase CMPA
        // else, change directions and decrease CMPA
        if(epwm_info->EPwm_CMPA_Direction == EPWM_CMP_UP)
        {
            if(epwm_info->EPwmRegHandle->CMPA.half.CMPA < epwm_info->EPwmMaxCMPA)
            {
                epwm_info->EPwmRegHandle->CMPA.half.CMPA++;
            }
            else
            {
                epwm_info->EPwm_CMPA_Direction = EPWM_CMP_DOWN;
                epwm_info->EPwmRegHandle->CMPA.half.CMPA--;
            }
        }
    }
}
```

## 6、总结

这里最后强调一下，DSP的内容很多很多，我不可能把所有内容都面面俱

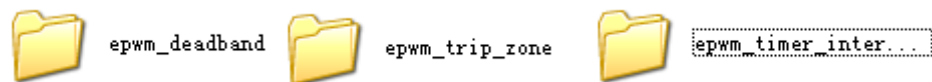


---

到的讲给大家，我只能尽全力将我认为比较重要的内容给大家介绍一下，希望能给大家以帮助。你的DSP究竟可以做多深走多远，最重要的因素还是你自己。我最多只能起到一个辅助的作用，帮助大家更好更快的入门，我不可能把你“教”成一个合格的工程师，只有自己潜心研究，你才会真正的掌握DSP的知识。

因为 是 相 同 的 所 以 `void InitEPwm2Example()` 和 `void InitEPwm3Example(void)` 这两个我就不重复叙述了。

这一篇教程今天就到这里，下面这3个文件夹中的ePWM模块pjt请大家自行研究。



至此，DSP的PWM基本内容就介绍到这里，还是那句老话，我的教程可以帮助你入门DSP，但是没有办法将你“教成”合格的DSP软件工程师，后面就需要大家自己去实践去看数据手册来深入学习了。要想研究透彻DSP的PWM模块的详细内容请看[spru791f\(TMS320x280x, 2801x, 2804x Enhanced Pulse Width Modulator \(ePWM\) Module Reference Guide\).pdf](#)，路径在 光盘：（非常重要）TI DSP TMS320x280x系列芯片资料\TI TMS320F280xDSP的用户指南\


\*\*\*\*\*



---

## 第四章 GPIO 实验（流水灯和按键实验）

### 1、前言

下面由我来给大家讲解一下 TI TMS320F280x 系列 DSP 的 GPIO 模块，使用的 pjt 文件是  Example\_280xGpioToggle.pjt，路径为 E:\Code of TMS320F280x CCS4\DSP280x\_examples\Led\_Light\_water\_and\_key\_ALL。众所周知，GPIO 是 DSP 最最基本的一个模块，也是大家都肯定会用到的一个模块，并且这部分内容也比较简单，但是相对单片机来说还是要稍微复杂一点点。

为了避免重复和啰嗦，在前面几个章节中讲过的 CCS4.2 的操作方法，我在这里就不做介绍了。如果买家朋友在学习本教程的时候，发现有些 CCS 的步骤不会做，请自己重新学习一下前面的[第一章](#)。

### 2、实验步骤

(1) 将仿真器与开发板 JTAG 口连接，再将仿真器的 USB 线连接电脑 USB 口，再给开发板上电。

(1) 首先关掉前面打开的 project


(2) 再打开 PWM 的 project，文件夹名为 [Led\\_Light\\_water\\_and\\_key](#)。路径 E:\Code of TMS320F280x CCS4\DSP280x\_examples\。具体的打开方法这里不赘述了。参见第一章的第 1 步至第 5 步。打开后，出现下图


(3) 如果 project 中不包括 ccxml 文件，那么就不用移除了，如果 project 包括有 ccxml，咱们就把这个陈旧的 ccxml 移除掉，移除方法参见第一章步骤 (6)。


(4) 因为在第一章的第 (7) 步到第 (10) 步已经创建了一个新的并且与我们开发板相对应的 target configuration（也就是 ccxml 文件），所以我们这里就不需要再重新创建了。

(5) 将前面已经创建好的 target configuration（也就是 ccxml 文件）link 到当前的 project，Example\_280xEPwmUpDownAQ.pjt 方法如下图所示，也可以参照第一章的第 11 至 12 步骤。

(6) 启动 TI 的 Debugger（Launch TI Debugger），点 target——>Launch TI Debugger

(7) 连接 DSP 仿真器和 DSP 开发板（[Conenct](#)），点击红圈处 ，或者快捷键 Alt+C，具体方法可以参见第一章的第 (15) 和 (16) 步骤

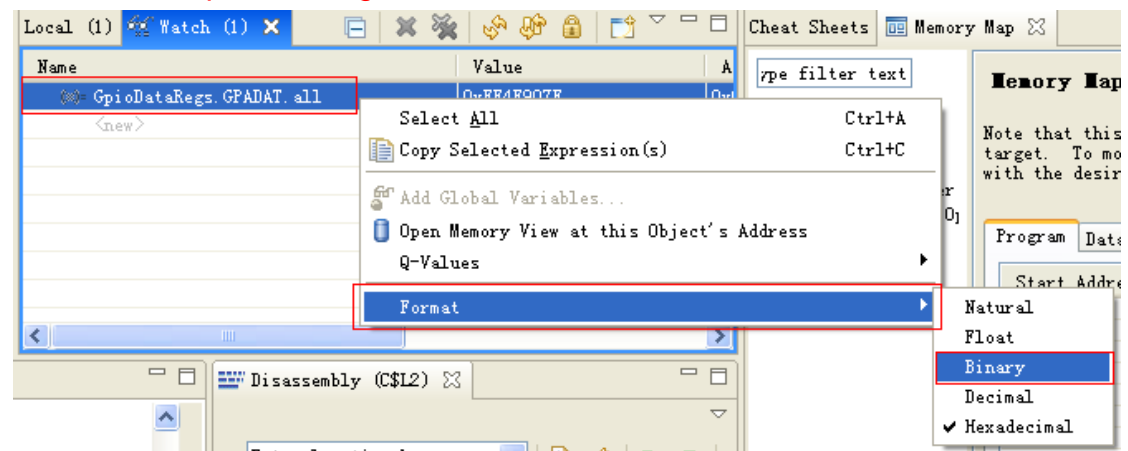
(8) 装载编译好的程序到 DSP 的 RAM 中，也就是 Load Program，方法：点击 ，或者 ctrl+alt+L，编译好的 out 文件路径 E:\Code of TMS320F280x CCS4\DSP280x\_examples\Led\_Light\_water\_and\_key\Debug

(10) 点 ，运行 project，会发现，按下按键，流水灯会逐个闪亮，按键弹起，流水灯全灭。

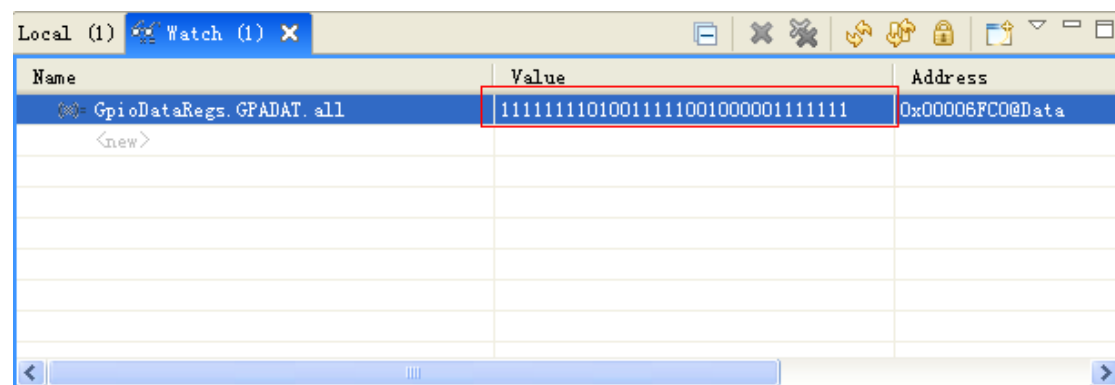
(11) 用 watch window 观察 [GpioDataRegs.GPADAT.all](#) 这个变量的变化，并且改变观察的数据格式，如下图，在 watch window 中右键点击

---

GpioDataRegs.GPADAT.all，然后选择 Format 再选择 Binary，就代表按照二进制数来显示 GpioDataRegs.GPADAT.all 中的数据



然后如下图



### 3、讲解 `Example_280xGpioToggle.pjt` 中的各个 c 语句的含义

下面我给大家讲解一下 `Example_280xGpioToggle.pjt` 这个 project 中的一些关键 C 语句的含义。

```
// For this example use the following configuration:  
Gpio_select();  
// f void Gpio_select(void)  
// f void Gpio_select(void) interrupts and initialize PIE vector table:  
// Disable CPU interrupts  
DINT;
```

这个 Gpio\_select();函数的内部语句如下

---

```
EALLOW;

GpioCtrlRegs.GPAPUD.bit.GPIO06 = 0;
GpioDataRegs.GPASET.bit.GPIO06 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO06 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO06 = 1;

GpioCtrlRegs.GPAPUD.bit.GPIO16 = 0;
GpioDataRegs.GPASET.bit.GPIO16 = 1;
GpioCtrlRegs.GPAMUX2.bit.GPIO16 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO16 = 1;

GpioCtrlRegs.GPAPUD.bit.GPIO17 = 0;
GpioDataRegs.GPASET.bit.GPIO17 = 1;
GpioCtrlRegs.GPAMUX2.bit.GPIO17 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO17 = 1;

GpioCtrlRegs.GPAPUD.bit.GPIO19 = 0;
GpioDataRegs.GPASET.bit.GPIO19 = 1;
GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO19 = 1;

GpioCtrlRegs.GPAPUD.bit.GPIO08 = 0;
GpioDataRegs.GPASET.bit.GPIO08 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO08 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO08 = 1;

GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO31 = 0;

EDIS;
```

| `GpioCtrlRegs.GPAPUD.bit.GPIO16 = 0;` 这句是禁止 GPIO16 脚的内部上拉，

如果设置值 1 的话，就是使能内部上拉。

`GpioDataRegs.GPASET.bit.GPIO16 = 1;` 这句就是初始值设置为高

`GpioCtrlRegs.GPAMUX2.bit.GPIO16 = 0;` 这句比较关键，这个是决定 DSP 的 GPIO16 这个引脚究竟是干什么用的。因为 DSP 的引脚绝大部分都具有复用功能，以这个 GPIO16 为例，它既可以做 TZ5 用，又可以 SPISIMOA 用，又可以做 GPIO16。也就是说如果你把 `GpioCtrlRegs.GPAMUX2.bit.GPIO16` 这个值设置为 0，那么这个脚就当做 GPIO 来用，如果设置为 1，那么就当做 SPISIMOA 来用，如果设置为 3，那么就当做 TZ5 来用。

再多说一些，这样的 GPIO 功能选择寄存器一共有 4 个，分别是 GPAMUX1、GPAMUX2，GPBMUX1，GPBMUX2，这四个寄存器都是 32 位寄存器（TI DSP 的绝大部分寄存器都是 32 位的）每 2 位是分给一个 GPIO 来用，所以就能分给 16 个 GPIO 口，如下图

---

Figure 4-4. GPIO Port A MUX 1 (GPAMUX1) Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
GPIO15		GPIO14		GPIO13		GPIO12		GPIO11		GPIO10		GPIO9		GPIO8	
R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
GPIO7		GPIO6		GPIO5		GPIO4		GPIO3		GPIO2		GPIO1		GPIO0	
R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0	

LEGEND- R/W = Read/Write; R = Read only; -n = value after reset

Figure 4-5. GPIO Port A MUX 2 (GPAMUX2) Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
GPIO31		GPIO30		GPIO29		GPIO28		GPIO27		GPIO26		GPIO25		GPIO24	
R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
GPIO23		GPIO22		GPIO21		GPIO20		GPIO19		GPIO18		GPIO17		GPIO16	
R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0		R/W-0	

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Figure 4-6. GPIO Port B MUX 1 (GPBMUX1) Register

31	6	5	4	3	2	1	0
Reserved		GPIO34		GPIO33		GPIO32	
R-0		R/W-0		R/W-0		R/W-0	

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Figure 4-7. GPIO Port B MUX 2 (GPBMUX2) Register

31	0
Reserved	
R/W-0	

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

`GpioCtrlRegs.GPADIR.bit.GPIO16 = 1;` 这句话也很关键啊，当设置为 1 的时候，代表是做 IO 口输出；当设置为 0 时，代表是做 IO 口输入，比如 `GpioCtrlRegs.GPADIR.bit.GPIO31 = 0;`，这个就是将 GPIO31 设置成输入口。

在 main 函数中的 `void Gpio_example1(void)` 这个是用来控制流水灯按顺序依此循环闪烁的程序。

```
Example_280xGpioToggle.c
130 }
131
132
133 void Gpio_example1(void)
134 {
135     // Example 1:
136     // Toggle I/Os using DATA registers
137     if(GpioDataRegs.GPADAT.bit.GPIO31==1) //key 通过按键切换流水灯亮与不亮
138     {
139         GpioDataRegs.GPADAT.all=0xFFFDFFFF; //GPIO17=0 LED灯亮
140         delay_loop();
141         GpioDataRegs.GPADAT.all=0xFFF7FFFF; //GPIO19=0
142         delay_loop();
143         GpioDataRegs.GPADAT.all=0xFFFFFBBF; //GPIO6=0
144         delay_loop();
145         GpioDataRegs.GPADAT.all=0xFFFFFEFF; //GPIO8=0
146         delay_loop();
147         GpioDataRegs.GPADAT.all=0xFFFFEFFF; //GPIO16=0
148         delay_loop();
149         GpioDataRegs.GPADAT.all=0xFFFFFFFF;
150     }
151
152
153 }
154
```

当 GPIO 作为输出口时。

GpioDataRegs.GPADAT.all=0xFFFDFFFF;就是让 GPIO17=0,也就是是使 GPIO17 输出低电平,最左边的 LED 灯亮。GpioDataRegs.GPADAT.all=0x0FFF7FFFF,就是让 GPIO19=0 也就是让 GPIO19 输出低电平,让左数第二个 LED 灯亮,依此类推。这个都是非常好理解的,就不多说了。

当 GPIO 作为输入口时。

对设置成输入口的 D A T A 寄存器赋值是没有任何意义的,比如对 GpioDataRegs.GPADAT.all 赋值就是没有任何意义的,这时应该读取 GpioDataRegs.GPADAT.all 的值,这个也没什么难理解的。

至此, D S P 的 G P I O 模块就简单介绍完毕了。要想研究透彻 DSP 的 GPIO 模块的详细内容请看[spru712g\(TMS320x280x, 2801x, 2804x DSP System Control and Interrupts Reference Guide\).pdf](#) 路径在 光盘:(非常重要)TI DSP TMS320x280x系列芯片资料\TI TMS320F280xDSP的用户指南\

\*\*\*\*\*