DSP 实验报告

陈鸿煜　测控1902班　$U201914594$

环境配置

环境配置可参考《实验指导书》和《CCS的配置和使用》。

注意：**路径不能有中文，在修改path时，要把原先所有的路径都删掉，不然无法编译成功**

程序烧录和仿真参考链接：[CCS使用教程04：程序烧写与仿真_ccs烧录程序]

其中的关键步骤在于建立目标配置文件：

1. 在项目文件夹下建立`targetConfigs`文件夹，然后对此文件夹新建`Target Configuration File`，注意新建的文件后缀为`ccxml`
2. 在新建的ccxml文件中选择对应的单片机仿真器型号

实验1 定时器和 GPIO 实验

实验1主要是对时钟、GPIO和中断的操作，这一节我们要掌握这三个重要的技能。

以下是实验一的代码，我会从代码中找到对应的知识点，结合代码对知识点进行深度分析。

```
/*
 * main.c
 */
#include "DSP28x_Project.h"
#include "LED_TM1638.h"

interrupt void cpu_timer0_isr(void);  //timer0
interrupt void myXint1_isr(void);     //xint1
interrupt void EPWM4Int_isr(void);    //EPWM4
interrupt void Ecap1Int_isr(void);    //ECAP1
interrupt void MyAdcInt1_isr(void);   //ADCINT1

void InitPWM4Gpio();
void InitPWM4();
void InitCAPGpio();
void InitCAP();
void InitADC();

void HorseRunning(int16 no, int Running);
#define Led0Blink() GpioDataRegs.GPACLEAR.bit.GPIO0 = 1
#define Led1Blink() GpioDataRegs.GPACLEAR.bit.GPIO1 = 1
#define Led2Blink() GpioDataRegs.GPACLEAR.bit.GPIO2 = 1
#define Led3Blink() GpioDataRegs.GPACLEAR.bit.GPIO3 = 1
#define Led0Blank() GpioDataRegs.GPASET.bit.GPIO0 = 1
#define Led1Blank() GpioDataRegs.GPASET.bit.GPIO1 = 1
#define Led2Blank() GpioDataRegs.GPASET.bit.GPIO2 = 1
#define Led3Blank() GpioDataRegs.GPASET.bit.GPIO3 = 1
```

```
int hourH = 0,hourL=0,minH=0,minL=0,secH=0,secL=0,TenmS = 0;
int Running = 0,NewLedEn = 0,KeyDLTime = 0;
int LedFlashCtr;
int period,hightime;
int PWM1Prd;
int ledtmp;
int adcptr;
unsigned int ADC_GD,ADC_FK;
float ADC_GDF,ADC_FKF;
long int li1,li2,li3,li4,PWM_HI,PWM_LO,PWM_PRD;

int PWMIntNo,PWMDuty,Tridir;
int ledkd,leddat;

void  Xint1_Init()
{
    EALLOW;
    GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO12 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO12 = 0;      // Pullup's enabled GPIO4
    GpioIntRegs.GPIOXINT1SEL.bit.GPIOSEL = 12;
    XIntruptRegs.XINT1CR.bit.POLARITY = 0;
    XIntruptRegs.XINT1CR.bit.ENABLE = 1;
    EDIS;
}

void HorseIO_Init()
{
      EALLOW;
    GpioDataRegs.GPASET.bit.GPIO0 = 1;
    GpioDataRegs.GPASET.bit.GPIO1 = 1;
    GpioDataRegs.GPASET.bit.GPIO2 = 1;
    GpioDataRegs.GPASET.bit.GPIO3 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
    EDIS;

}

void main(void) {

    int x,y;
    InitSysCtrl();      //初始化系统时钟，选择内部晶振1，10MHZ，12倍频，2分频，初始化外设时钟，低速外设,4分频
    DINT;               //关总中断
    IER = 0x0000;       //关CPU中断使能
    IFR = 0x0000;       //清CPU中断标志
    InitPieCtrl();      //关pie中断
    InitPieVectTable(); //清中断向量表
    EALLOW;
    GpioDataRegs.GPACLEAR.bit.GPIO0 = 1;
    GpioDataRegs.GPACLEAR.bit.GPIO1 = 1;
```

```
    GpioDataRegs.GPASET.bit.GPIO2 = 1;
    GpioDataRegs.GPASET.bit.GPIO3 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
    EDIS;

    EALLOW;              /**配置中断向量表*****/
    PieVectTable.TINT0 = &cpu_timer0_isr;
    PieVectTable.XINT1 = &myXint1_isr;
    PieVectTable.ECAP1_INT = &Ecap1Int_isr;
    PieVectTable.EPWM4_INT = &EPWM4Int_isr;
    PieVectTable.ADCINT1 = &MyAdcInt1_isr;
    EDIS;

    // MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
    InitFlash();

    InitCpuTimers();    // 初始化定时器
    ConfigCpuTimer(&CpuTimer0, 60,10000);
    CpuTimer0Regs.TCR.bit.TSS = 0;
    CpuTimer0Regs.TCR.bit.TRB = 1;
    CpuTimer0.InterruptCount = 0;

    HorseIO_Init();
    Xint1_Init();
    TM1638_Init();      //初始化LED

    InitPWM4Gpio();
    InitPWM4();
    InitCAPGpio();
    InitCAP();
    InitADC();

    PieCtrlRegs.PIECTRL.bit.ENPIE = 1;      // Enable the PIE block
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;      // TINT0
    PieCtrlRegs.PIEIER1.bit.INTx4 = 1;      // XINT1
    PieCtrlRegs.PIEIER1.bit.INTx1 = 1;      // ADCINT1
    PieCtrlRegs.PIEIER3.bit.INTx4 = 1;      // EPWM4
    PieCtrlRegs.PIEIER4.bit.INTx1 = 1;      // ECAP1

    IER |= M_INT1;      /**使能CPU中断**/
    IER |= M_INT3;      /**使能CPU中断**/
    IER |= M_INT4;      /**使能CPU中断**/


    EINT;
    // ERTM;

    ledkd=0;
    while(1){
      if(NewLedEn==0) {
        if(ledkd==0){
```

```
            LED_Show(1,(TenmS % 10),0);
            LED_Show(2,(TenmS / 10),0);
            LED_Show(3,secL,1);
            LED_Show(4,secH,0);
            LED_Show(5,minL,1);
            LED_Show(6,minH,0);
            LED_Show(7,hourL,1);
            LED_Show(8,hourH,0);
            }
          else if(ledkd==1){
            leddat=PWM_HI;
            LED_Show(4,leddat /1000,0);
            leddat = leddat % 1000;
            LED_Show(3,leddat /100,0);
            leddat = leddat % 100;
            LED_Show(2,leddat /10,0);
            LED_Show(1,(leddat % 10),0);

            leddat=PWM_PRD;
            LED_Show(8,leddat /1000,0);
            leddat = leddat % 1000;
            LED_Show(7,leddat /100,0);
            leddat = leddat % 100;
            LED_Show(6,leddat /10,0);
            LED_Show(5,(leddat % 10),1);
            }

          else if(ledkd==2){
            leddat=ADC_FKF;
            LED_Show(4,leddat /1000,0);
            leddat = leddat % 1000;
            LED_Show(3,leddat /100,1);
            leddat = leddat % 100;
            LED_Show(2,leddat /10,0);
            LED_Show(1,(leddat % 10),0);

            leddat=ADC_GDF;
            LED_Show(8,leddat /1000,0);
            leddat = leddat % 1000;
            LED_Show(7,leddat /100,1);
            leddat = leddat % 100;
            LED_Show(6,leddat /10,0);
            LED_Show(5,(leddat % 10),1);
            }

          NewLedEn = 1;
          }
        }
}

void HorseRunning(int16 no, int Running)
{
    if(Running == 0) {
        if(no & 0x1)Led0Blink();
        else Led0Blank();
        if(no & 0x2)Led1Blink();
        else Led1Blank();
        if(no & 0x4)Led2Blink();
```

```
        else Led2Blank();
        if(no & 0x8)Led3Blink();
        else Led3Blank();
    } else if(Running == 1) {
        if(no & 0x1)Led3Blink();
        else Led3Blank();
        if(no & 0x2)Led2Blink();
        else Led2Blank();
        if(no & 0x4)Led1Blink();
        else Led1Blank();
        if(no & 0x8)Led0Blink();
        else Led0Blank();
    } else if(Running == 2) {
        if(no & 0x1)Led0Blink();
        else Led0Blank();
        if(no & 0x2)Led2Blink();
        else Led2Blank();
        if(no & 0x4)Led1Blink();
        else Led1Blank();
        if(no & 0x8)Led3Blink();
        else Led3Blank();
    }

}


interrupt void myXint1_isr(void)
{
    if((Running == 0)&&(KeyDLTime > 20)){
      EALLOW;
      CpuTimer0Regs.TCR.bit.TSS = 1;
      CpuTimer0Regs.TCR.bit.TRB = 1;
      CpuTimer0Regs.TCR.bit.TSS = 0;
      EDIS;
      Running = 1;
      KeyDLTime = 0;
      }
    else if((Running == 1)&&(KeyDLTime > 20)){
      Running = 2;
      KeyDLTime = 0;
      }
    else if((Running == 2)&&(KeyDLTime > 20)){
      Running = 0;
      hourH = 0;hourL=0;minH=0;minL=0;secH=0;secL=0;TenmS = 0;
      KeyDLTime = 0;
      EALLOW;
      // CpuTimer0Regs.TCR.bit.TSS = 1;
      // CpuTimer0Regs.TCR.bit.TRB = 1;
      EDIS;
      }
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}

interrupt void cpu_timer0_isr(void) {

    KeyDLTime++;

    LedFlashCtr++;
```

```
    if((LedFlashCtr % 10)==0)NewLedEn = 0;
    if(Running == 1){
      TenmS++;
      if(TenmS == 100){
        TenmS = 0;
        secL++;
        }

      if(secL==10){
          secL=0;
          secH++;
          }
        if(secH==6){
            secH=0;
            minL++;
          }
       if(minL==10){
           minL=0;
           minH++;
         }
      if(minH==6){
          minH=0;

          hourL++;
        }
      if(hourL==4 && hourH==2){
          hourL=0;
          hourH=0;
      }
      else if(hourL==10){
          hourL=0;
          hourH++;

        }
    }
  HorseRunning((LedFlashCtr & 0xf0)>>4, Running);
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

一、概念剖析

1. 时钟

在实验一中，涉及时钟的代码只有一行

```
InitSysCtrl();      //初始化系统时钟，选择内部晶振1，10MHZ，12倍频，2分频，初始化外设时钟，低速外设,4分频
```

时钟模块有**三个重点**：

1. 选择晶振，设置晶振频率
2. 设置PLL模块，进行倍频分频
3. 要注意系统时钟和外设时钟都要配置

以下我列出了**学习时钟时概念上的困惑**：

1. 什么是晶振，和时钟有什么关系

   晶振是一种控制频率元件，在电路模块中提供频率脉冲信号源，给电路提供一定频率的稳定的震荡（脉冲）信号，比如石英钟，就是通过对脉冲记数来走时的。在单片机内，晶振用于产生周期性的时钟信号，所以时钟和晶振的关系就是：晶振构成振荡器，振荡器的输出生成时钟脉冲信号。

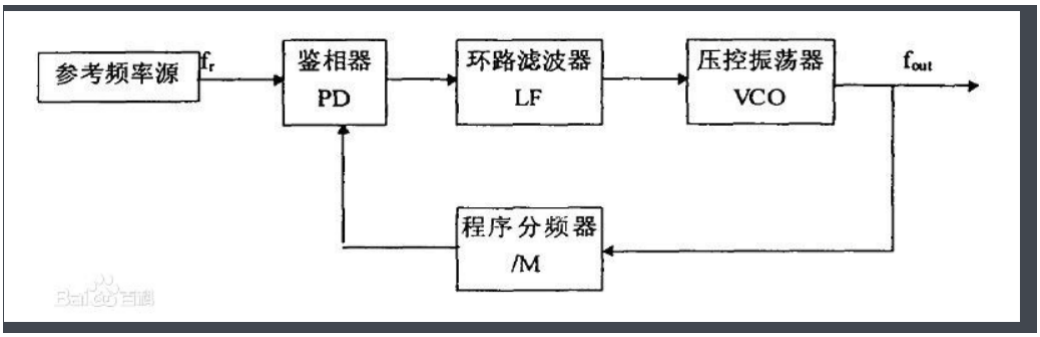
2. 什么是定时器？教材中章节名明明是《时钟与系统控制》，为什么要提到CPU定时器？

   在单片机中，时钟和定时器往往是配套出现的。定时器是一种工作在计数模式下，只计数固定周期脉冲的计数器；由于脉冲周期固定，由计数的值可以计算出时间，所以定时器有定时功能。在使用单片机时，我们往往需要每隔一段时间来触发某个功能，此时就需要使用定时器。

3. PLL锁相环是什么？

   Phase Locked Loop，是一种利用反馈控制原理实现相位和频率同步的技术，一般由鉴相器，滤波器，压控震荡器和分频器构成，它的作用是将电路输出的时钟与其外部的参考时钟保持同步。PLL需要有一个参考频率Fi。输出频率为Fo，参考频率与输出频率同时送入鉴相器。鉴相器的作用是检测输入信号和输出信号的相位差，并将检测出的相位差信号转换成uD（t）电压信号输出。当输出信号的频率与输入信号的频率相等时，输出电压与输入电压保持固定的相位差值，即输出电压与输入电压的相位被锁住，这就是锁相环的名称由来。

   当我们需要倍频时，显然在反馈通路上添加分频器即可。

4. 系统时钟和外设时钟有什么联系？什么是外设？

   这里的外设指的是片内外设，单片机内部的外设一般包括：串口控制模块，SPI模块，I2C模块，A/D模块，PWM模块，CAN模块，EEPROM，比较器模块，等等，它们都集成在单片机内部，有相对应的内部控制寄存器，可通过单片机指令直接控制。 外设指的是单片机外部的外围功能模块，比如键盘控制芯片，液晶，A/D转换芯片，等等。

   为什么有了系统时钟，还需要外设时钟呢？我们可以打个比方：系统是单片机的大脑，外设是单片机的手和脚，有的时候我们只需要动动脑筋就行，这个时候可以给外设时钟设置的慢一些，这样可以减少功耗，所以就给系统和外设分别配置了时钟。学习单片机时经常会遇到不同模块重复配置了一些设备，此时我们可以从耦合度和功耗的角度进行考量。

   在28027中，通过振荡器和PLL获得系统时钟(CLKIN)后，CPU输出系统时钟SYSCLKOUT(SYSCLKOUT=CLKIN)，外部时钟信号(XCLOUT)由系统时钟输出信号提供，其频率为其1/2 或 1/4。

定时器

教材中，CPU定时器涉及四个寄存器：`PRDH:PRD` `TIMH:TIM` `TDDRH:TDDR` `PSCH:PSC`。定时器由两个部分组成：16位分频器和32位计数器。分频器对系统时钟进行分频，计数器减计数，减到0时发出中断TINT。

这四个寄存器中，PRDH:PRD 和 TIMH:TIM 为一组，可以理解为：PRD为用户设置计数器计数值存放处，TIM为计数器本身的计数值。每次开始计数时，先要把用户设置值(PRD)导入到计数器(TIM)，计数器再进行减计数。

TDDRH:TDDR 和 PSCH:PSC 为一组，与计数器的那一组类似：TDDR用于装载用户设置的分频系数，PSC为分频器本身的计数值。使用时TDDR内装载的值写入PSC中。

2. GPIO

GPIO的操作有很多，这里我们只看跑马灯部分

```
void HorseIO_Init()
{
    EALLOW;
    GpioDataRegs.GPASET.bit.GPIO0 = 1;
    GpioDataRegs.GPASET.bit.GPIO1 = 1;
    GpioDataRegs.GPASET.bit.GPIO2 = 1;
    GpioDataRegs.GPASET.bit.GPIO3 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
    EDIS;
}
```

书上对GPIO的引入比较跳跃，这里我们从基础概念开始讲起。

General Purpose Input Output(GPIO)，通用输入输出端口。它的核心功能就是读取外部输入，也可以从此端口向外部输出信号(高低电平)。但是书上列举了一堆别的功能：上拉电阻、多路复用、触发中断和各种寄存器。这些其实都是依据核心功能引申出的花式用法。DSP的开发商帮我们从硬件内部完成了这些功能，就不需要我们开发时通过软件实现，这样做节约了开发时间，不过学习的时候容易摸不着头脑。接下来我将逐一解释这些花式用法。

1) 多路复用

首先我们解释什么是多路复用，以及为什么要引入多路复用。

DSP中使用了流水线技术，可以提高运行速度，那么在IO口，我们也可以使用类似的方式提高系统效率，这就是“多路复用”，因为IO口并不是每时每刻都接收输入/输出，此时可以给它多连几个外设端口以提高效率。但是多路复用会存在一个问题：如果多条通路同时有数据，会存在撞车的问题，此时就需要一位交警来进行交通梳理，这位交警大哥就是多路复用寄存器(MUX)，其实就是一个数据选择器，选谁谁能走，不选就禁能。我们可以从教材中看到，MUX有许多保留位，可以留给我们进行设置。

不过我第一次看MUX寄存器表时有些懵，这里我截取教材中的部分表格进行解说：

	复位时为默认状态基本的I/O功能	外设选择1	外设选择2	外设选择3
GPAMUX1寄存器位	(GPAMUX1 位 = 00)	(GPAMUX1 位 = 01)	(GPAMUX1 位 = 10)	(GPAMUX1 位 = 11)
1, 0	GPIO0	EPWM1A(O)	保留	保留
3, 2	GPIO1	EPWM1B(O)	保留	COMP1OUT(O)

我们看表格的第三行，MUX的1，0位选中了GPIO0，代表这两位用于进行GPIO0的多路复用选择，当MUX = 00时，允许GPIO0通过，当MUX = 01时，允许PWM通过，其余两位保留，并且保留位用户不能自己定义，只能空着。

多路复用在默认情况下不使用(即默认状态都是GPIO导通)。

2) 方向寄存器

IO口可以输入也可以输出，通过方向寄存器进行设置

3) 上拉使能

刚接触上拉和下拉时，容易产生一个误会：如果设置某一位上拉，那么这一位就一直处于上拉使能状态。然而在单片机中，上拉使能是提供了一个默认的高电平状态。如果后续改变为低电平，默认状态结束。

4) 中断选择

有时，IO口伴随着中断，比如按一下按钮，就开始计时。这时我们可以令IO口作为中断源。GPIO内置了触发中断XINT，以下代码举例了配置方式。

```
void  Xint1_Init()
{
    EALLOW;
    GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO12 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO12 = 0;      // Pullup's enabled GPIO4
    GpioIntRegs.GPIOXINT1SEL.bit.GPIOSEL = 12;// 设置GPIO12为XINT1触发源
    XIntruptRegs.XINT1CR.bit.POLARITY = 0;    // 极性为0
    XIntruptRegs.XINT1CR.bit.ENABLE = 1;      // s
    EDIS;
}
```

3. 中断

我们关注中断的开、关，以及中断向量表、中断源的配置

```
EALLOW;     /**配置中断向量表*****/
PieVectTable.TINT0 = &cpu_timer0_isr;
PieVectTable.XINT1 = &myXint1_isr;
PieVectTable.ECAP1_INT = &Ecap1Int_isr;
PieVectTable.EPWM4_INT = &EPWM4Int_isr;
PieVectTable.ADCINT1 = &MyAdcInt1_isr;
EDIS;
```

中断的核心部分有三个：**中断使能**、**中断向量表**、**外设中断扩展(PIE)**。其中PIE内有一个重要的概念：**中断源**。

中断使能

在DSP中，存在可屏蔽中断和非屏蔽中断，只有可屏蔽中断需要中断使能。

对于可屏蔽中断，在我们使用的28027中，受三个控制位控制：IFR, IER, INTM。其中IER和INTM为中断使能位，IFR为中断标志位，表示中断需要响应。

中断向量

中断向量有两个等级：CPU级和外设级(PIE)，CPU级中断是真正的中断，外设级中断是通过外设触发的CPU级中断。

通用的可屏蔽CPU级中断有14个(INT1 - INT14)，PIE中断把一组外设中断连接到对应的CPU级中断

外设中断扩展

外设中断，首先要配置好中断源，对应到相应的外设中断向量，然后对应到CPU级的中断向量。

我在学习PIE中断向量时遇到疑惑：向量表中各种INT1.1, INT1.2, 一直到INT12.8，这些向量怎么配置到我们想要的触发源。然而开发商已经给我们配置好了，或者说触发源已经固定了。在表格的第五列"说明"中，我们可以看到配置好的触发源，配置中断函数时，直接用即可。比如配置定时器0(TINT0)复位时触发中断函数:

```
PieVectTable.TINT0 = &cpu_timer0_isr;// 等号左边是中断向量，右边是中断函数地址，即把中断函数的地址放进zhong'duan'xiang'lin
```

二、代码讲解

因为C语言是面向过程，所以直接对main函数从上往下讲即可。

```
DINT;               //关总中断
IER = 0x0000;       //关CPU中断使能
IFR = 0x0000;       //清CPU中断标志
InitPieCtrl();      //关pie中断
```

这里把所有中断都关闭了，后面应该有打开中断的命令。果然在后面可以找到：

```
IER |= M_INT1;      //使能CPU中断
IER |= M_INT3;      //使能CPU中断
IER |= M_INT4;      //使能CPU中断
```

我们可以了解一下IER寄存器：该寄存器有16位，每一位对应一个可屏蔽中断，置1时允许。这里允许了1，3，4。

接下来进行了GPIO的配置：

```
EALLOW;
GpioDataRegs.GPACLEAR.bit.GPIO0 = 1;
GpioDataRegs.GPACLEAR.bit.GPIO1 = 1;
GpioDataRegs.GPASET.bit.GPIO2 = 1;
GpioDataRegs.GPASET.bit.GPIO3 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;
GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
EDIS;
```

这里涉及了四个寄存器：

1. GPACLEAR

清除寄存器，对GPIO PORT A 的指定引脚清零

2. GPASET

   设置寄存器，指定引脚设置为高

3. GPAMUX1

   多路复用寄存器，设置是否多路复用，为0时不复用

4. GPADIR

   方向寄存器，置1为输出

那么可以看出，程序中讲GPIO0 1 2 3 初始化并置为输出。

接着配置了中断向量：

```
EALLOW;             //配置中断向量表
PieVectTable.TINT0 = &cpu_timer0_isr;
PieVectTable.XINT1 = &myXint1_isr;
PieVectTable.ECAP1_INT = &Ecap1Int_isr;
PieVectTable.EPWM4_INT = &EPWM4Int_isr;
PieVectTable.ADCINT1 = &MyAdcInt1_isr;
EDIS;
```

接下来一堆初始化，这里我们主要看定时器初始化，顺便把上面的时钟初始化也看了：

时钟初始化：

```
InitSysCtrl();      //初始化系统时钟，选择内部晶振1，10MHZ，12倍频，2分频，初始化外设时钟，低速外设,4分频
```

这个程序被封装好了，我们看一下实现晶振选择和分频倍频需要怎么操作：

1. 选择时钟源并关闭其他时钟源

```
EALLOW;
SysCtrlRegs.CLKCTL.bit.INTOSC1OFF = 0;
SysCtrlRegs.CLKCTL.bit.OSCCLKSRCSEL=0;  // Clk Src = INTOSC1
SysCtrlRegs.CLKCTL.bit.XCLKINOFF=1;     // Turn off XCLKIN
SysCtrlRegs.CLKCTL.bit.XTALOSCOFF=1;    // Turn off XTALOSC
SysCtrlRegs.CLKCTL.bit.INTOSC2OFF=1;    // Turn off INTOSC2
EDIS;
```

   这里用到了`CLKCTL`寄存器

2. 配置PLL模块，设置倍频分频

```
void InitPll(Uint16 val, Uint16 divsel)
{
volatile Uint16 iVol;
if (SysCtrlRegs.PLLSTS.bit.MCLKSTS != 0)
{
EALLOW;
SysCtrlRegs.PLLSTS.bit.MCLKCLR = 1;
EDIS;
asm("        ESTOP0");     // Uncomment for debugging purposes
}
if (SysCtrlRegs.PLLSTS.bit.DIVSEL != 0)
{
```

```
 EALLOW;
 SysCtrlRegs.PLLSTS.bit.DIVSEL = 0;
 EDIS;
}
if (SysCtrlRegs.PLLCR.bit.DIV != val)
{
EALLOW;
SysCtrlRegs.PLLSTS.bit.MCLKOFF = 1;
SysCtrlRegs.PLLCR.bit.DIV = val;
EDIS;
DisableDog();
while(SysCtrlRegs.PLLSTS.bit.PLLLOCKS != 1)
{
    // Uncomment to service the watchdog
    // ServiceDog();
}
EALLOW;
SysCtrlRegs.PLLSTS.bit.MCLKOFF = 0;
EDIS;
}
if((divsel == 1)||(divsel == 2))
{
  EALLOW;
  SysCtrlRegs.PLLSTS.bit.DIVSEL = divsel;
  EDIS;
}
if(divsel == 3)
{
  EALLOW;
  SysCtrlRegs.PLLSTS.bit.DIVSEL = 2;
  DELAY_US(50L);
  SysCtrlRegs.PLLSTS.bit.DIVSEL = 3;
  EDIS;
}
}
```

配置PLL时，用到了两个寄存器：`PLLSTS` `PLLCR`。比较简单，看书即可。

3. 配置外设时钟

外设时钟寄存器由PCLKCR0 - PCLKCR3，这三个寄存器分别负责不同的外设，给各种外设模块的时钟使能。

```
void InitPeripheralClocks(void)
{
EALLOW;
GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 3;        // GPIO18 = XCLKOUT，多路复用
SysCtrlRegs.LOSPCP.all = 0x0002;            // 低速外设时钟4分频系统时钟
(SYSCLKOUT)
SysCtrlRegs.XCLK.bit.XCLKOUTDIV=2;          // Set XCLKOUT =
SYSCLKOUT/1 外设时钟不分频
SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 0;       // ADC
SysCtrlRegs.PCLKCR3.bit.COMP1ENCLK = 0;     // COMP1
SysCtrlRegs.PCLKCR3.bit.COMP2ENCLK = 0;     // COMP2
SysCtrlRegs.PCLKCR3.bit.CPUTIMER0ENCLK = 1; // CPU Timer-0
SysCtrlRegs.PCLKCR3.bit.CPUTIMER1ENCLK = 0; // CPU Timer-1
SysCtrlRegs.PCLKCR3.bit.CPUTIMER2ENCLK = 0; // CPU Timer-2
SysCtrlRegs.PCLKCR1.bit.ECAP1ENCLK = 1;     // eCAP1
```

```
SysCtrlRegs.PCLKCR1.bit.EPWM1ENCLK = 1;      // EPWM1
SysCtrlRegs.PCLKCR1.bit.EPWM2ENCLK = 1;      // EPWM2
SysCtrlRegs.PCLKCR1.bit.EPWM3ENCLK = 1;      // EPWM3
SysCtrlRegs.PCLKCR1.bit.EPWM4ENCLK = 1;      // EPWM4
SysCtrlRegs.PCLKCR3.bit.GPIOINENCLK = 1;     // GPIO
SysCtrlRegs.PCLKCR0.bit.HRPWMENCLK=0;        // HRPWM
SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 0;       // I2C
SysCtrlRegs.PCLKCR0.bit.SCIAENCLK = 0;       // SCI-A
SysCtrlRegs.PCLKCR0.bit.SPIAENCLK = 0;       // SPI-A
EDIS;
}
```

定时器初始化:

```
InitCpuTimers();    // 初始化定时器
ConfigCpuTimer(&CpuTimer0, 60,10000);
CpuTimer0Regs.TCR.bit.TSS = 0;
CpuTimer0Regs.TCR.bit.TRB = 1;
CpuTimer0.InterruptCount = 0;
```

看书即可，主要注意定时器的开关，中断允许，以及**计数值要减1**。注意这里的CpuTIMERxRegs指定一个定时器，它其实是一个结构。

实验2 PWM 和 eCAP 实验

实验2主要是对PWM模块(脉冲宽度调制)和eCAP(增强型捕获)模块的使用，当然其中也包括了对时钟、GPIO和中断的操作。

以下为PWM程序

```
#include "DSP28x_Project.h"
extern int PWMIntNo,PWMDuty,Tridir;
void InitPWM4Gpio()
{
    EALLOW;
    GpioCtrlRegs.GPAPUD.bit.GPIO6 = 1;    // Disable pull-up on GPIO6 (EPWM4A)
    GpioCtrlRegs.GPAPUD.bit.GPIO7 = 1;    // Disable pull-up on GPIO7 (EPWM4B)
    GpioCtrlRegs.GPAMUX1.bit.GPIO6 = 1;   // Configure GPIO6 as EPWM4A
    GpioCtrlRegs.GPAMUX1.bit.GPIO7 = 1;   // Configure GPIO7 as EPWM4B
    EDIS;
}

void InitPWM4()
{

  int PWMPRD,DeadTime;

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

  PWMPRD = 3000;
  DeadTime = 480; // ?死区时间
  EPwm4Regs.TBPRD = PWMPRD;
```

```
  EPwm4Regs.TBPHS.half.TBPHS = 0; // Set Phase register to zero
  EPwm4Regs.TBCTL.bit.CLKDIV = 0; //CLKDIV = 0;
  EPwm4Regs.TBCTL.bit.HSPCLKDIV = 0; //HSPCLKDIV = 0;
  EPwm4Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // 0x2, Symmetrical mode
  EPwm4Regs.TBCTL.bit.PHSEN = TB_DISABLE; // 0x0, Master module
  EPwm4Regs.TBCTL.bit.PRDLD = TB_SHADOW;// 0x0
  EPwm4Regs.TBCTL.bit.SYNCOSEL = TB_CTR_ZERO; // 0x1, Sync down-stream module
  EPwm4Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;// 0x0
  EPwm4Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;// 0x0
  EPwm4Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO_PRD; // 0x2, load on CTR=Zero or
PRD
  EPwm4Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO_PRD; // 0x2,, load on CTR=Zero or
PRD
  EPwm4Regs.AQCTLA.bit.CAU = AQ_CLEAR; // 0x1, set actions for EPWM1A
  EPwm4Regs.AQCTLA.bit.CAD = AQ_SET;// 0x2
  EPwm4Regs.AQCTLA.bit.CBU = 0; // set actions for EPWM1A,do nothing
  EPwm4Regs.AQCTLA.bit.CBD = 0;
  EPwm4Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // 0x3, enable Dead-band module
  EPwm4Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC; // 0x2, Active Hi complementary
  EPwm4Regs.DBFED = DeadTime;
  EPwm4Regs.DBRED = DeadTime;

  PWMDuty = 1500;
  EPwm4Regs.CMPA.half.CMPA=PWMDuty;
  EPwm4Regs.CMPB = PWMDuty;

  // Enable CNT_zero interrupt using EPWM1 Time-base
  EPwm4Regs.ETSEL.bit.INTEN = 1;   // Enable EPWM1INT generation
  EPwm4Regs.ETSEL.bit.INTSEL = 1;  // Enable interrupt CNT_zero event
    //EPwm1Regs.ETSEL.bit.INTSEL = 3;  // Enable interrupt CNT_zero & CNT_PRD
event
    EPwm4Regs.ETPS.bit.INTPRD = 1;   // Generate interrupt on the 1st event
    EPwm4Regs.ETCLR.bit.INT = 1;     // Enable more interrupts

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

}

interrupt void EPWM4Int_isr(void)
{

  PWMIntNo++;
  // PWMIntNo &=0x1f;
  if(PWMIntNo & 0x1){
     if(Tridir==0){
       PWMDuty +=3;
       if(PWMDuty >= 750){Tridir=1;PWMDuty=750;}
       }
     else {
       PWMDuty -=3;
       if(PWMDuty <= 0){Tridir=0;PWMDuty=0;}
       }
  }
  // if(PWMIntNo == 0)PWMDuty++;
  // if(PWMDuty > 750)PWMDuty = 0;
  //EPwm4Regs.CMPA.half.CMPA=PWMDuty;
```

```
  EPwm4Regs.ETCLR.bit.INT = 1;
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
  EINT;
}
```

以下为eCAP程序

```
#include "DSP28x_Project.h"
extern long int li1,li2,li3,li4,PWM_HI,PWM_LO,PWM_PRD;

void InitCAPGpio()
{
  EALLOW;
  GpioCtrlRegs.GPAPUD.bit.GPIO5 = 0;    // Enable pull-up on GPIO5 (ECAP1)
  GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 3;   // Configure GPIO5 as ECAP1
  EDIS;
}

void InitCAP()
{

  ECap1Regs.ECEINT.all = 0x0000;             // Disable all capture interrupts
  ECap1Regs.ECCLR.all = 0xFFFF;              // Clear all CAP interrupt flags

  ECap1Regs.ECCTL1.bit.CAP1POL = 0;
  ECap1Regs.ECCTL1.bit.CAP2POL = 1;
  ECap1Regs.ECCTL1.bit.CAP3POL = 0;
  ECap1Regs.ECCTL1.bit.CAP4POL = 1;
  ECap1Regs.ECCTL1.bit.CTRRST1 = 0;
  ECap1Regs.ECCTL1.bit.CTRRST2 = 0;
  ECap1Regs.ECCTL1.bit.CTRRST3 = 0;
  ECap1Regs.ECCTL1.bit.CTRRST4 = 0;
  ECap1Regs.ECCTL1.bit.CAPLDEN = 1;
  ECap1Regs.ECCTL1.bit.PRESCALE = 0;
  ECap1Regs.ECCTL2.bit.CAP_APWM = 0;
  ECap1Regs.ECCTL2.bit.CONT_ONESHT = 0;
  ECap1Regs.ECCTL2.bit.SYNCO_SEL = 2;
  ECap1Regs.ECCTL2.bit.SYNCI_EN = 0;
  ECap1Regs.ECCTL2.bit.TSCTRSTOP = 1; // 允许TSCTR
  ECap1Regs.ECEINT.bit.CEVT4 = 1; // CEVT4
}

interrupt void Ecap1Int_isr(void)
{
  li1=ECap1Regs.CAP1;
  li2=ECap1Regs.CAP2;
  li3=ECap1Regs.CAP3;
  li4=ECap1Regs.CAP4;
  PWM_HI=((li2-li1)+(li4-li3)) >> 1;
  PWM_LO=li3-li2;
  PWM_PRD=((li3-li1)+(li4-li2)) >> 1;

  ECap1Regs.ECCLR.bit.CEVT4 = 1;
  ECap1Regs.ECCLR.bit.INT = 1;
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP4;
  EINT;
```

```
}
```

一、概念剖析

1. PWM

谈及PWM(脉冲宽度调制)，一个重要的概念就是占空比(Duty Ratio)。对于占空比我经常有一个误会：既然是占“空”比，应该是低电平占周期的比值，然而事实与其相反，占空比说的是高电平占周期的比值。占空比展开来说，就是通电时间相对于总时间所占的比例。

PWM的作用，一个是改变平均功率，一个就是提供对应的波形，我们使用DSP芯片，主要就是为了实现脉冲宽度调制，提供不同占空比的波形，当然，其中还有很多细节，比如死区，互补PWM，作为中断源触发中断等等。

PWM比GPIO和CPU时钟都难一些，因为它有很多子模块，不过这些子模块都比较简单，我将一一讲解。

时基子模块(Time Base)

四个ePWM模块都分别拥有自己的时基，并且28027的PWM拥有同步逻辑电路，可以让四个PWM一起工作，拥有相同的时钟。配置时钟同步使用`EPWMxSYNCI`。时基子模块有时钟和计数器(之前讲过时钟模块都有自己的计数器)，那么通过控制其时钟频率和计数器计数模式，就可以控制PWM的周期，并产生事件：

1. CTR = PRD

   时基计数器的计数值等于周期时

2. CTR = Zero

   时基计数器的计数值等于0时

3. CTR = CMPB

   时基计数器的计数值等于比较寄存器的值时

4. CTR = dir

   指示实际计数器的方向，递增时该信号为高电平

5. CTR = max

   计数值等于最大值产生该信号(TBCTR = 0XFFFF)

教材上只讲了会发生这些事件，但是没说这些事件该怎么用，其实这些事件产生后，是通过动作限定器的寄存器表现出来的。动作限定器规定了产生事件时，会触发哪些“动作”，所以我们不需要手动监听事件，交给动作限定器即可。所以教材里说的”事件“其实是相对于动作限定器而言的。

当然，时基子模块也可以进行分频，计数模式也可以设置先递增后递减，它也有自己的状态寄存器，以显示自身状态。

注意：计数器递增或递减模式时，计算计数周期时，计数值要加*1*

学习计数器时，教材中把计数器的计数值称为"CTR"，我一直以为是ctrl而感到疑惑，后来才发现是COUNTER的缩写。

时基周期的映射寄存器(Shadow Register)

教材里把shadow翻译为映射，然而网络上直接译为影子，这里我们就根据教材来，称其为映射寄存器。之所以称为"Shadow"，是因为两个寄存器地址相同，可以认为是“身体和影子”。

为什么要这样设计呢？因为所有真正需要起作用的寄存器(有效寄存器)可以在同一个时间(发生更新事件时)被更新为所对应的预装载寄存器的内容，这样可以保证多个通道的操作能够准确地同步。如果没有影子寄存器，软件更新寄存器时，则直接更新了真正操作的寄存器，因为软件不可能在一个相同的时刻同时更新多个寄存器，结果造成多个通道的时序不能同步，如果再加上例如中断等其它因素，多个通道的时序关系有可能会混乱，造成是不可预知的结果。

那么，什么时候映射寄存器内的值会写入有效寄存器呢？对于时基周期的映射，当时基计数器等于=0时，进行写入。

注意：网络上可能会把影子寄存器叫做预装载寄存器，把有效寄存器称作影子寄存器。

如果要使用映射寄存器，直接操作相应的控制寄存器即可。比如时基周期的映射寄存器，启动方式为：

```
EPwmxRegs.TBCTL.bit.PRDLD = TB_SHADOW;
```

计数器-比较子模块(Count-Compare)

每个ePWM有两个计数器(CMPA & CMPB)，每个计数器都可以产生事件：

1. CTR = CMPA

   时基计数器的值等于比较器A寄存器的值

2. CTR = CPMB

   时基计数器的值等于比较器B寄存器的值

这两个比较器是独立的，产生的事件也互相独立。

比较子模块的映射寄存器

相比于时基子模块，比较子模块的映射寄存器比较特殊：它需要配置有效寄存器装载的时基，可以是时基计数器的值为0，也可以是时基计数器的值为周期值。

动作限定器子模块(Action Qualifier)

刚接触这个模块时，比较疑惑一点：什么是"限定"？根据解释，这个模块是根据不同的事件产生动作，那么为什么要用"限定"这个词？后来知道，因为这个词是直译过来的，英文的Qualifier，更合理应该翻译为"资格"，表示哪个事件有资格触发动作。所以"动作限定器"可以解释为：限定哪个事件可以触发动作的寄存器。这里的动作，就是指ePWMxA和ePWMxB产生的输出

动作限定器可以根据以下事件产生动作(置零、清零、切换)：

1. CTR = PRD
2. CTR = Zero
3. CTR = CMPA
4. CTR = CPMB

死区发生器子模块(Dead-Band Generator)

首先解释一下什么是死区，为什么要有死区。PPT上的解释为："在利用dsp对整流器或逆变器进行控制时，为了避免桥臂的上下管同时导通而导致短路，需要对输出PWM信号的上升沿或者下降沿进行延时，也就是设置PWM死区。"，我们可以通过互补PWM来理解，理想情况下，互补PWM可以保证两个输出信号在同一时刻是互补的，但是在电平交换时，肯定会存在一定的"混乱"状态，即不清楚是高电平还是低电平，那么这时可能会出现两个pwm输出都是高电平的情况。如果pwm作为桥臂上下管的输入电源，同时导通就会导致短路，所以需要一个"时延"来延迟高电平到来的时间，这样就可以确认一个器件关闭后再打开另一个器件。这个"时延"就是死区。

死区发生器中，涉及了"极性控制"，他的功能我理解了，在书中193页有具体讲解，但是其中为什么要命名为"高低电平有效/互补"，我还没搞懂，搞懂了回来写一写。

事件触发器子模块(Event Triggers)

事件触发器用于接收时基模块、计数比较模块等子模块产生的事件，可以设置发生1/2/3个事件后(预分频)，产生中断或者启动ADC。

中断名称为`EPWMxINT`，每个ePWM可以产生一个中断；ADC命令为`ADCTRIG2x+3`和`ADCTRIG2x+4`,每个ePWM可以产生两个ADC命令。

中断直接连接到PIE，直接到PIE中断向量表里就能找到。

PWM用于DAC

因为PWM输出的信号基本都是高频的(系统时钟频率很高，MHz级)，所以我们给PWM输出加一个低通滤波器，高频信号就被滤除，只剩下直流信号，然而我们知道，直流信号就是电压平均值，那么经过低通滤波后的输出电压就正比于占空比，即模拟信号。

2. eCAP

值得吐槽的是，28027只有一个捕获模块，但是厂商热心地给这个模块命名为eCAP1，导致我在学习时一直以为有多个eCAP，就像PWM一样。不过还有一个容易混淆地点：eCAP模块拥有四个时间戳捕捉寄存器(CAP1 - CAP4)，刚开始学时我把cCAP和CAP搞混了。CAP1 - 4是eCAP模块的寄存器，用于存储捕捉的计数值。

增强型捕获模块(Enhanced Capture)，用于对外部事件进行捕捉。这里我要解释两个词：“外部”和“捕捉”。

1. 外部：这里的外部是物理意义的外部，即单片机的外部，这个外部事件是需要通过引脚接入到单片机的，eCAP模块会监听这个引脚，从而进行捕捉，产生记录。
2. 捕捉：其实就是对事件进行标记，记录事件发生的时间。比如PWM，我们可以通过对他的上升沿和下降沿进行捕获从而获取其周期和占空比。

捕捉到事件以后，可以产生动作，不过这里的动作并不只是产生中断这么简单，它可以对事件进行计数、计时。至于如何实现，后面会提到。

既然要统计捕捉到事件的时间，肯定需要时钟和计数器，这里我们直接使用系统时钟，并内置了32位时基计数器。当然，对于高频信号输入，每次事件都捕获没什么意义，所以厂家贴心的内置了预分频器。

我们也可以使用限定器选择触发事件的边沿，因为eCAP有四个CAP寄存器，可以对一个连续事件捕获四次。如果我们只想进行一组捕获(只捕获四次)后就停止，就需要使用eCAP模块中的模四计数器和一个停止寄存器(2位)，停止寄存器用于标志发生几次事件后停止捕捉，和模四计数器的计数值进行比较，相等时停止捕捉，这就是单触发模式。当然也可以使用连续捕获模式，捕获值不停地被写入CAP1-4。这里提到了一个词：循环缓冲区，一开始我以为深度为4的循环缓冲区对应四个CAP寄存器，然而并不是这样的。循环缓冲区是一个独立的功能，简单的理解是：在连续捕获模式中，要不停地进行读(读取时钟计数器的值)和写(写到CAP寄存器中)操作，这样会产生冲突从而丢失数据，设置一个深度为4的缓冲区从而保证数据的可靠性与实时性。

预分频器

分频倍数只能是2的倍数，N = 2 - 62，当N = 1 时分频器被旁路。

边缘极性选择与限定器

这里的“限定”和PWM的动作限定器意思类似，就是限定哪个边沿可以触发事件。边缘极性选择与限定器，就是指这个限定器可以限定是上升沿或/和下降沿可以触发事件。

32位计数器

32位计数器（TSCTR）通过系统时钟为事件捕捉提供时间基准。也就是：该计数器提供了事件捕获的时基，由系统时钟来驱动。

4个事件中的任何一个都可以复位32位计数器，这对于偏差的捕捉非常有用。先捕捉32位计数器的值，之后任何一个LD1~LD4信号都可以将其复位为0。这样做的好处是可以统计绝对时间和相对时间：如果每捕捉一个事件就清零，那么下一次捕捉到事件的时间就是上一个事件发生后的相对时间；如果不清零，统计到的就是绝对时间。

CAP1 - CAP4时间戳捕捉寄存器

每个寄存器有32位，直接连接到时钟定时器上，可以进行计数值的装载。

中断

eCAP可以产生7种中断，在捕捉时间时，可以产生中断。并且可以单独使能/禁止不同的中断事件。在任何其他中断脉冲产生前，中断服务程序必须清除全局中断标志位。

二、代码讲解

主要是寄存器的配置，先对相关GPIO进行初始化(是否上拉，多路复用等等)，然后根据系统时钟的频率，PWM进行配置，再用eCAP进行捕获。具体操作请看代码中的注释文档。

实验4 综合实验

实验4主要是新增了对ADC(模数转换模块)的使用。

以下为ADC程序：

```
#include "DSP28x_Project.h"
#define ADC_usDELAY 10000L
#define ADCSampT 39
#define TrigSelNo 11   //ADCTRIG11 - ePWM4, ADCSOCA
extern int adcptr;
extern unsigned int ADC_GD,ADC_FK;
extern float ADC_GDF,ADC_FKF;
float KP,KI,UK,EK,UK_1,EK_1;
interrupt void MyAdcInt1_isr(void);
void InitADC()
{

    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;
    (*Device_cal)();
    EDIS;

  /* Configure ADC pins using AIO regs*/
  // This specifies which of the possible AIO pins will be Analog input pins.
  // NOTE: AIO1,3,5,7-9,11,13,15 are analog inputs in all AIOMUX1
configurations.
  // Comment out other unwanted lines.
 /*
    EALLOW;
```

```
    GpioCtrlRegs.AIOMUX1.bit.AIO2 = 2;    // Configure AIO2 for A2 (analog input)
operation
    GpioCtrlRegs.AIOMUX1.bit.AIO4 = 2;    // Configure AIO4 for A4 (analog input)
operation
    GpioCtrlRegs.AIOMUX1.bit.AIO6 = 2;    // Configure AIO6 for A6 (analog input)
operation
    GpioCtrlRegs.AIOMUX1.bit.AIO10 = 2;   // Configure AIO10 for B2 (analog
input) operation
    GpioCtrlRegs.AIOMUX1.bit.AIO12 = 2;   // Configure AIO12 for B4 (analog
input) operation
    GpioCtrlRegs.AIOMUX1.bit.AIO14 = 2;   // Configure AIO14 for B6 (analog
input) operation
    EDIS;
  */
    DELAY_US(ADC_usDELAY);  // Delay before converting ADC channels

      EALLOW;
      AdcRegs.ADCCTL1.bit.ADCBGPWD  = 1;        // Power ADC BandGap
      AdcRegs.ADCCTL1.bit.ADCREFPWD = 1;        // Power reference
      AdcRegs.ADCCTL1.bit.ADCPWDN   = 1;        // Power ADC
      AdcRegs.ADCCTL1.bit.ADCENABLE = 1;        // Enable ADC
      // AdcRegs.ADCCTL1.bit.ADCREFSEL = 1;     // Select Outside Reference
Voltage
      AdcRegs.ADCCTL1.bit.ADCREFSEL = 0;        // Select Internal Reference
Voltage
      EDIS;

      DELAY_US(ADC_usDELAY);  // Delay before converting ADC channels

      EALLOW;
      AdcRegs.ADCCTL1.bit.INTPULSEPOS   = 1;    //ADCINT1 trips after AdcResults
latch
      AdcRegs.INTSEL1N2.bit.INT1E     = 1;  //Enabled ADCINT1
      AdcRegs.INTSEL1N2.bit.INT1CONT  = 0;  //Disable ADCINT1 Continuous mode
      AdcRegs.INTSEL1N2.bit.INT1SEL = 0x09; //setup EOC9 to trigger ADCINT1 to
fire

      AdcRegs.ADCSAMPLEMODE.all = 0xff;  //Simultaneous sample for SOCAx and
SOCBx.

      //A01(IW),B01(IU),A2B2(OH),A3B3(VDC),A4B4(ASIN1),A5B5(ASIN2),A67B67(BAK)
      AdcRegs.ADCSOC0CTL.bit.CHSEL  = 0x07;   //A7,Result0;  B7,Result1
      AdcRegs.ADCSOC2CTL.bit.CHSEL  = 0x07;   //A7,Result2;  B7,Result3
      AdcRegs.ADCSOC4CTL.bit.CHSEL  = 0x07;   //A7,Result4;  B7,Result5
      AdcRegs.ADCSOC6CTL.bit.CHSEL  = 0x07;   //A7,Result6;  B7,Result7#define
      AdcRegs.ADCSOC8CTL.bit.CHSEL  = 0x07;   //A7,Result8;  B7,Result9

      AdcRegs.ADCSOC0CTL.bit.TRIGSEL    = TrigSelNo;      //ADCTRIG11- ePWM4,
ADCSOCA
      AdcRegs.ADCSOC2CTL.bit.TRIGSEL    = TrigSelNo;      //ADCTRIG11- ePWM4,
ADCSOCA
      AdcRegs.ADCSOC4CTL.bit.TRIGSEL    = TrigSelNo;      //ADCTRIG11- ePWM4,
ADCSOCA
      AdcRegs.ADCSOC6CTL.bit.TRIGSEL    = TrigSelNo;      //ADCTRIG11 - ePWM4,
ADCSOCA
      AdcRegs.ADCSOC8CTL.bit.TRIGSEL    = TrigSelNo;    //ADCTRIG11 - ePWM4,
ADCSOCA
```

```
      AdcRegs.ADCSOC0CTL.bit.ACQPS  = ADCSampT;   //Sample window is ADCSampT
cycles long
      AdcRegs.ADCSOC2CTL.bit.ACQPS  = ADCSampT;   //Sample window is ADCSampT
cycles long
      AdcRegs.ADCSOC4CTL.bit.ACQPS  = ADCSampT;   //Sample window is ADCSampT
cycles long
      AdcRegs.ADCSOC6CTL.bit.ACQPS  = ADCSampT;   //Sample window is ADCSampT
cycles long
      AdcRegs.ADCSOC8CTL.bit.ACQPS  = ADCSampT;   //Sample window is ADCSampT
cycles long
      EDIS;

   // Assumes ePWM1 clock is already enabled in InitSysCtrl();
    EPwm4Regs.ETSEL.bit.SOCAEN  = 1;        // Enable SOC on A group
    EPwm4Regs.ETSEL.bit.SOCASEL = 1;        // Enable event CTR = ZERO
    EPwm4Regs.ETPS.bit.SOCAPRD  = 1;        // Generate pulse on 1st event
    EPwm4Regs.ETCLR.bit.SOCA = 1;       // Clear SOCA flag
    }

interrupt void MyAdcInt1_isr(void)
{
   adcptr++;
   ADC_GD = AdcResult.ADCRESULT0;
   ADC_GD += AdcResult.ADCRESULT2;
   ADC_GD += AdcResult.ADCRESULT4;
   ADC_GD += AdcResult.ADCRESULT6;
   ADC_GD += AdcResult.ADCRESULT8;

   ADC_FK = AdcResult.ADCRESULT1;
   ADC_FK += AdcResult.ADCRESULT3;
   ADC_FK += AdcResult.ADCRESULT5;
   ADC_FK += AdcResult.ADCRESULT7;
   ADC_FK += AdcResult.ADCRESULT9;

   ADC_GDF= ADC_GDF *0.98 + ADC_GD*0.1611 * 0.02/10;
   ADC_FKF = ADC_FKF *0.98 + ADC_FK*0.1611 * 0.02/10;

   EK=ADC_GDF-ADC_FKF;
   KP=1;
   KI=1;
   UK=UK_1+KP*(EK-EK_1)+KI*EK;
   if(UK<0)
       UK=0;
   else if(UK>1)
       UK=1;
   EPwm4Regs.CMPA.half.CMPA=UK*12000;
   UK_1=UK;
   EK_1=EK;

   DELAY_US(ADC_usDELAY);
   AdcRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

一、概念剖析

提及ADC，我们知道需要经过四个过程：采样、保持、量化、编码。在实际电路中(比如28027的ADC模块)，往往采样保持同时进行，量化编码同时进行，这对应着ADC模块中的“采样/保持(S/H)电路(Sample and Hold)”和”转换内核(多为主次比较型)“。并且ADC模块具有多路复用功能，可以进行通道选择，这样就可以输入多组模拟信号。

教材直接解释了ADC模块的各种寄存器，但是首先我们需要先学习一些ADC的术语。

1. 基本术语

分辨率和通道

当您阅读任何微控制器或ADC IC的参数规格时，ADC的参数信息将使用“通道”和“分辨率（位）”等术语给出。

例如，Arduino UNO的ATmega328有一个8通道10位ADC。术语“8通道”意味着ATmega328微控制器上有8个引脚可以读取模拟电压，每个引脚可以读取10位分辨率的电压。并非微控制器上的每个引脚都可以读取模拟电压，这因不同型号的微控制器而异。

假设我们的 ADC 范围是从 0V 到 5V，并且我们有一个 10 位 ADC，这意味着我们的输入电压 0-5 伏将被分成 1024 级离散模拟值（0000000000—1111111111，2^10 = 1024）。1024 是 10 位 ADC 的分辨率，类似地，8 位 ADC 的分辨率为 512 (2^8)，16 位 ADC 的分辨率为 65536 (2^16)。

这样，如果实际输入电压为 0V，则 MCU 的 ADC 会将其读取为 0，如果为 5V，则 MCU 将读取为 1024，如果是2.5V，则 MCU 将读取为 512。我们可以使用以下公式根据 ADC 的分辨率和参考电压计算 MCU 读取的数字值。

(ADC 分辨率 / 参考电压) = (ADC 数字值 / 实际电压值)。

28027的ADC模块分辨率为12位，有16个通道。

参考电压

您应该了解的另一个重要术语是“参考电压”。在 ADC 转换过程中，通过将其与已知电压进行比较来找到未知电压的值，这个已知电压，称为参考电压。

通常所有 MCU 都有一个设置内部参考电压的选项，这意味着您可以使用软件（程序）在内部将此电压设置为某个可用值。

在 Arduino UNO 板中，参考电压在内部默认设置为 5V，如果需要，用户也可以在软件中进行所需的更改，再通过 Vref 引脚在外部设置此参考电压。

需要注意的是，测量的模拟电压值应始终小于参考电压值，并且参考电压值应始终小于单片机的工作电压值。

采样窗口

当对某个引脚采样时，采样电容的电压需要一段时间才能累积到精确的采样电压，所以这里需要给足够的时间段进行采样，这个时间段就是采样窗口，一般以时钟周期为单位。

采样频率

也被称为采样速率，我们知道，需要事件进行触发，然后才能进行采样。当触发信号产生时，采样才开始。所以两个触发信号之间的间隔就是采样时间，采样时间的倒数就是采样频率。

序列发生器

序列信号是指在同步脉冲作用下循环地产生一串周期性的二进制信号，能产生这种信号的逻辑器件就称为序列信号发生器或序列发生器。比如可以周期产生01001011001序列信号的信号发生器，就是序列发生器。教材中写道：“与之前的那些ADC不同，这个ADC不基于序列发生器(sequencer)，而是基于SOC”。什么是基于序列发生器呢？因为事件触发ad转换动作，需要选择通道，对通道进行排序，这个排序的功能就由序列发生器提供，所以就叫基于序列发生器。当然，28027提供了更先进的方式：SOC。

SOC

一般说来， SoC(System on Chip)称为系统级芯片，也有称片上系统，意指它是一个产品，是一个有专用目标的集成电路，其中包含完整系统并有嵌入软件的全部内容。同时它又是一种技术，用以实现从确定系统功能开始，到软/硬件划分，并完成设计的整个过程。

片上系统是将计算机系统的许多元素组合到单个芯片中的集成电路。SoC 始终包含 CPU，但也可能包括系统内存、外围设备控制器（用于 USB、存储）和更高级的外围设备，如图形处理单元 （GPU）、专用神经网络电路、无线电调制解调器（用于蓝牙或 Wi-Fi）等。片上系统方法与具有CPU芯片和独立控制器芯片，GPU和RAM的传统PC形成鲜明对比，可以根据需要更换，升级或互换。SoC的使用使计算机更小，更快，更便宜，耗电更少。

在28027的ADC模块中，SOC特指用于处理模数转换的系统。它可以配置该模块的触发源、转换通道和采样窗口的尺寸。28027的ADC模块有16个SOC。

2. 具体配置

概述

ADC的配置，大体可以分为八点：

1. 配置引脚和工作（参考）电源
2. 配置转换顺序和采样方式
3. 配置转换速度(采样保持时间，转换时间)
4. 配置触发方式
5. 配置优先级
6. 配置结果读取方式
7. 配置中断方式
8. 其他特殊功能

触发源、通道和采样窗口

28027的ADC模块由SOC驱动，所以触发源、转换通道和采样窗口都是由SOC的寄存器配置。

1. SOCx的触发源由 `ADCSOCxCTL` 寄存器以及 `ADCINTSOCSEL1/ADCINTSOCSEL2` 联合配置
2. 通道和采样窗口尺寸由 `ADCSOCxCTL` 配置

由于输入的不一致性，有的输入需要很短的时间就能把采样电容器充电完成，有的需要更多的时间。为了保证充电完成，每个 `ADCSOCxCTL` 寄存器都包含一个六位字段 `ADQPS` ，可以设置S/H窗的长度。

每个SOC都可以独立配置输入触发器，当然多个不同的SOC也可以配置成使用同一个触发源。当我们需要使用连续转换时，可以配置ADC模块的中断做为触发源。

每个SOC都可以独立配置转换哪个通道的输入。SOC有两种采样模式：

1. 顺序采样

   当被配置为顺序采样时，可以通过寄存器配置要转换的通道

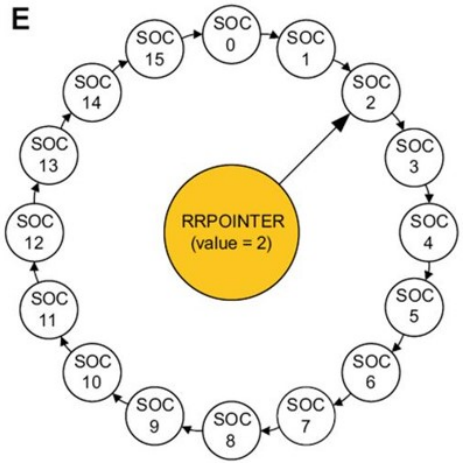
2. 同步采样

   也就是说并行采样，我们知道28027有两个采样保持电路，所以只能同步采样两个通道，可以通过寄存器进行配置。

轮询机制

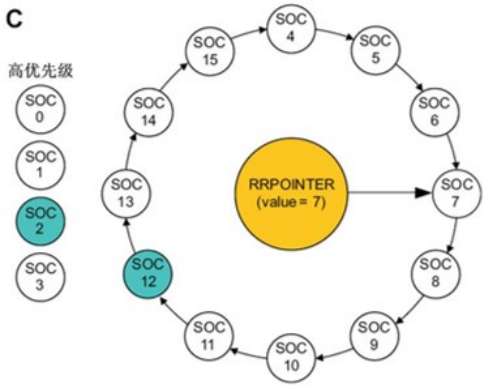
在顺序采样模式中，如果几个SOC同时被触发，但是采样保持电路只能一次运行一个(顺序模式中)，那么就需要一种机制来决定优先级。默认的机制为**轮询机制**，当然也可以使用`ADCSOCPRIORITYCTL`寄存器来分配优先级。

在轮询机制中，各个SOC的优先级在整体上是一样的，当发生优先级的竞争时，谁的优先级高纯靠运气，优先级仅取决于轮询指针(RRPOINTER)。轮询指针的值实时保存到寄存器`ADCSOCPRIORITYCTL`中，它指向上一个被转换的SOC。下一个比轮询指针大的值就是当前优先级最高的SOC，并在SOC15后又绕回SOC0。复位时，RRPOINTER的值为32，此时SOC0为最高优先级。



分配优先级

如果不使用默认优先级机制(轮询机制)，那么可以自主分配优先级。`ADCSOCPRIORITYCTL`寄存器中的`SOCPRIORITY`字段可以为所有SOC分配高优先级。当某个或某些SOC被配置为高优先级时，他们会被从轮询环中取出，单独摆成一列，当发生优先级竞争时，优先被取出的SOC。并且被配置为高优先级的SOC之间也有优先级，标号越小优先级越高。



同步采样模式

有时要求采样的两个信号之间延迟很短，或者最好能够同时得到采样结果，由于28027有两个独立的采样保持电路那么我们可以使用同步采样模式对两个通道进行同时采样。同步采样模式使用`ADCSAMPLEMODE`寄存器对一对SOCx进行配置。

偶数编号的SOCx和**它之后**的奇数编号的SOCx(比如SOC0和SOC1)配成一对，一起连接一个使能位(此时为SIMULEN0)。

同步采样的配对规则如下：

1. 首先转换A通道
2. A通道的转换结果存放在偶数编号的`ADCRESULTx`寄存器中，B通道的转换结果则存放在奇数编号的`ADCRESULTx`寄存器中
3. 同步采样时，一般我们只配置偶数SOCx。

参考电压

默认情况选择的是内部参考电压(0 ~ 3.3V)。

使用外部参考电压时，要用VREFHI/VREFLO管脚作为参考电压的引脚。

3. 中断操作

在系统时钟的定时器中，有TINT中断；在GPIO中，有外部中断XINT；在捕获模块中，有捕获中断ECAP_INT；在脉冲宽度调制模块中，有中断EPWM_INT；那么在数模转换模块里，也会有数模转换中断**ADCINT**，并且有九种不同的中断。

当然，由于ADC模块有16组SOC，不可能所有SOC都能作为触发源，我们需要通过寄存器`INTSELxNy`来配置由哪个SOC作为触发源。当转换开始或转换结束时，每个SOC都会发出自己的脉冲`EOCx`，这个就是中断触发源。

此外，ADCINT1和ADCINT2信号可以配置成产生一个SOCx触发事件，这对连续转换来说非常有用。

二、代码讲解

```
ADC_GDF= ADC_GDF *0.98 + ADC_GD*0.1611 * 0.02/10;
ADC_FKF = ADC_FKF *0.98 + ADC_FK*0.1611 * 0.02/10;
EK=ADC_GDF-ADC_FKF;
KP=1;
KI=1;
UK=UK_1+KP*(EK-EK_1)+KI*EK;
if(UK<0)
    UK=0;
else if(UK>1)
    UK=1;
EPwm4Regs.CMPA.half.CMPA=UK*12000;
UK_1=UK;
EK_1=EK;
```

程序中使用这一段代码进行PI控制。

华中科技大学　自动控制系

2022.5.1