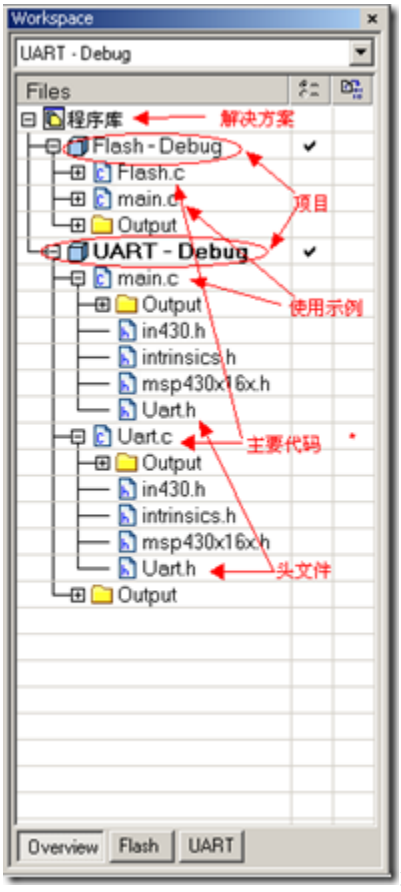
MSP430 程序库<一>综述

转眼已经大三就要结束了，我的大学生活即将结束；由于本人对软件比较感兴趣，毕业之后也许就远离的我的专业(电子信息科学与技术)了；我在大学期间也参加了电子设计竞赛等，在竞赛中我主要负责单片机程序的编写，所以对 msp430 系列的单片机比较熟悉；在这个系列的文章里，我主要介绍我对 430 单片机的理解，整理之前写下的程序，产生一个具有一定通用性的 430 程序库。

我与 msp430 最初的接触来自机械工业出版社出版的《MSP430 系列单片机系统工程设计与实践》这本书；我开始参加电子设计竞赛是在大二的暑假，放假之前听说竞赛用 MSP430F169 的单片机，然后就去图书馆找有关 430 单片机的书籍了，有关这款单片机的书不多，很幸运的是我借到了这本书；我写 430 单片机的程序风格很大程度上受到了此书的影响。

程序库的组织方式：程序库解决方案包含多个项目，每个项目是针对一个单元（如：uart 异步串行口）的程序库和使用示例，如异步串行口的程序库，下图中 UART 项目，Uart.c 是主要的程序库源代码，Uart.h 是对应头文件，使用时需包含此头文件，main.c 是使用示例代码。



程序库使用时只需.c 文件和对应的.h 文件即可。

文件组织方式：程序库的 c 文件和 h 文件一一对应，c 文件至少包含两个头文件，其中一个是 430 的头文件，以使用单片机的硬件资源，另一个是其对应的头文件；如 Uart.c 开头即为

```
#include <msp430x16x.h>
#include "Uart.h"
```

为防止重复包含头文件中均有#define 语句如 Uart.h 开头和结尾：

```
#ifndef __UART_H
#define __UART_H

#endif /* __UART_H */
```

程序库使用方式：第一步，先把 c 文件和 h 文件拷到工程文件夹；然后把 c 文件添加到项目中　在左侧 workspace 中右击项目，选 Add—>Add Files，选择刚添加的 c 文件；如图：

最后在要调用库函数的程序文件中包含拷进来的头文件；之后，就可以正常调用程序库中的函数（H 文件中声明的，需要的话，可以自行添加）。

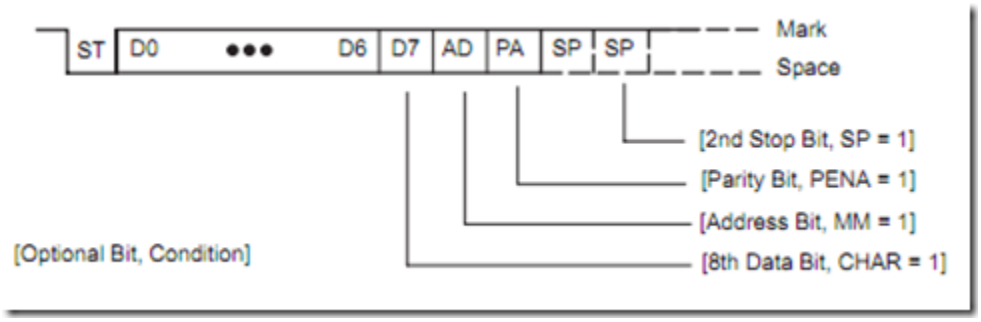
程序库目前打算先从异步串行口写起，多谢网友们的支持了啊。

MSP430 程序库<二>UART 异步串口

串行通信接口是处理器与其他设备进行数据通信最常用的方式之一。我的这个程序库是针对 MSP430f14 系列和 MSP430f16 系列的，我常用的单片机是这两款：msp430f149，msp430f169。这两款单片机中均有两个增强型串行通信接口，都可以进行同步或是异步通信，甚至 169 的模块 USART0 还能进行进行 I2C 协议通信。在这里，我们只讨论异步串行通信。

- 硬件介绍：

MSP 单片机的 USART 模块可以配置成 SPI（同步通信）模式或 UART（异步通信）模式，这里只讨论 UART 方式。UART 数据传输格式如下：



起始位，数据位由高到低 7/8 位，地址位 0/1 位，奇偶校验位 奇偶或无，停止位 1/2 位。数据位位数、地址位、奇偶校验位、停止位均可由单片机内部寄存器控制；这两款单片机都有两个 USART 模块，有两套独立的寄存器组；以下寄存器命中出现 x 代表 0 或是 1,0 代表对应 0 模块的寄存器，1 代表对应 1 模块的寄存器；其中，与串口模式设置相关的控制位都位于 UxCTL 寄存器，与接收相关的控制位都位于 UxRCTL 寄存器，与发送相关的控制位都位于 UxTCTL 寄存器；波特率设置用 UxBR0、UxBR1、UxMCTL 三个寄存器；接收与发送有独立的缓存 UxRXBUF、UxTXBUF，并具有独立的移位寄存器和独立的中断；中断允许控制位位于 IE1/2 寄存器，中断标志位位于 IFG1/2 寄存器。

波特率设置：430 的波特率设置用三个寄存器实现，
UxBR0：波特率发生器分频系数低 8 位。
UxBR1：波特率发生器分频系数高 8 位。
UxMCTL：波特率发生器分频系数的小数部分实现。
设置波特率时，首先要选择合适的时钟源：USART 模块可以设置的时钟源有 UCLK 引脚、ACLK、SMCLK；对于较低的波特率（9600 以下），可选 ACLK 作为时钟源，这样，在 LPM3（低功耗 3）模式下，串口仍能正常发送接收数据；另外，由于串口接收过程有一个三取二判决逻辑，这至少需要三个时钟周期，因此分频系数必须大于 3；波特率高于 9600 时，将不能使用 ACLK 作为时钟源，要调为频率较高的 SMCLK 作为时钟源；另外还可以外部输入 UCLK 时钟。分频系数计算公式如下：

$$N = \frac{BRCLK}{baud\ rate}$$

小数分频是 MSP430 单片机的串口特色之一，UxMCTL 寄存器的作用就是控制小数的分频，控制方法如下：对应位是 1，则分频系数加一，0 则分频系数减一；小数分频器会自动依次取出每一位来调整分频系数。其计算方法：可以先计算小数部分一的个数，然后把 1 均匀的放入 UxMCTL 的 8 位中，这样计算比较简单，分频系数的小数部分乘以 8 即得到 1 的位数，查表得到对应的 UxMCTL 值；另外一种通过计算每一位的错误率，交互计算，直到得到最小错误率的 UxMCTL 值，这种方法比较复杂，但得到的小数分频误差更小，这种方法也是 TI 给的计算方法，详细参考 UserGuide。

另外，有关寄存器，以及其他单片机硬件有关知识请参考德州仪器提供的用户指南和数据手册等资料。

- 程序实现：

- 宏定义：是程序具有更好的移植性。

对模块的寄存器进行宏定义，把 0/1 换成 x，使用时，只需更改宏定义即可更改程序是使用哪个模块；这样程序就具有了比较好的移植性。

```
/***********************宏定义************************/
#define UxCTL   U0CTL
#define UxRCTL  U0RCTL
#define UXTCTL  U0TCTL

#define UxBR0   U0BR0
#define UxBR1   U0BR1
#define UxMCTL  U0MCTL

#define UxRXBUF U0RXBUF
#define UxTXBUF U0TXBUF

#define UxME    U0ME
#define UxIE    U0IE
#define UxIFG   U0IFG

#define UTXEx   UTXE0
#define URXEx   URXE0

#define URXIEx  URXIE0
#define UTXIEx  UTXIE0

#define UARTON  P3SEL |= 0X30             // P3.4,5 = USART0 TXD/RXD
/*****************************************/
```

程序改为 UART1 时，只需把宏定义中的 0 改为 1 UARTON 改为对应端口的即可

- 异步串口初始化（UartInit）：完成波特率，停止位以及其他相关的设置。

串口初始化，首先是波特率寄存器值的计算和设置：本程序选用第二种：通过运算，选取误差最小的寄存器所需值进行设置。

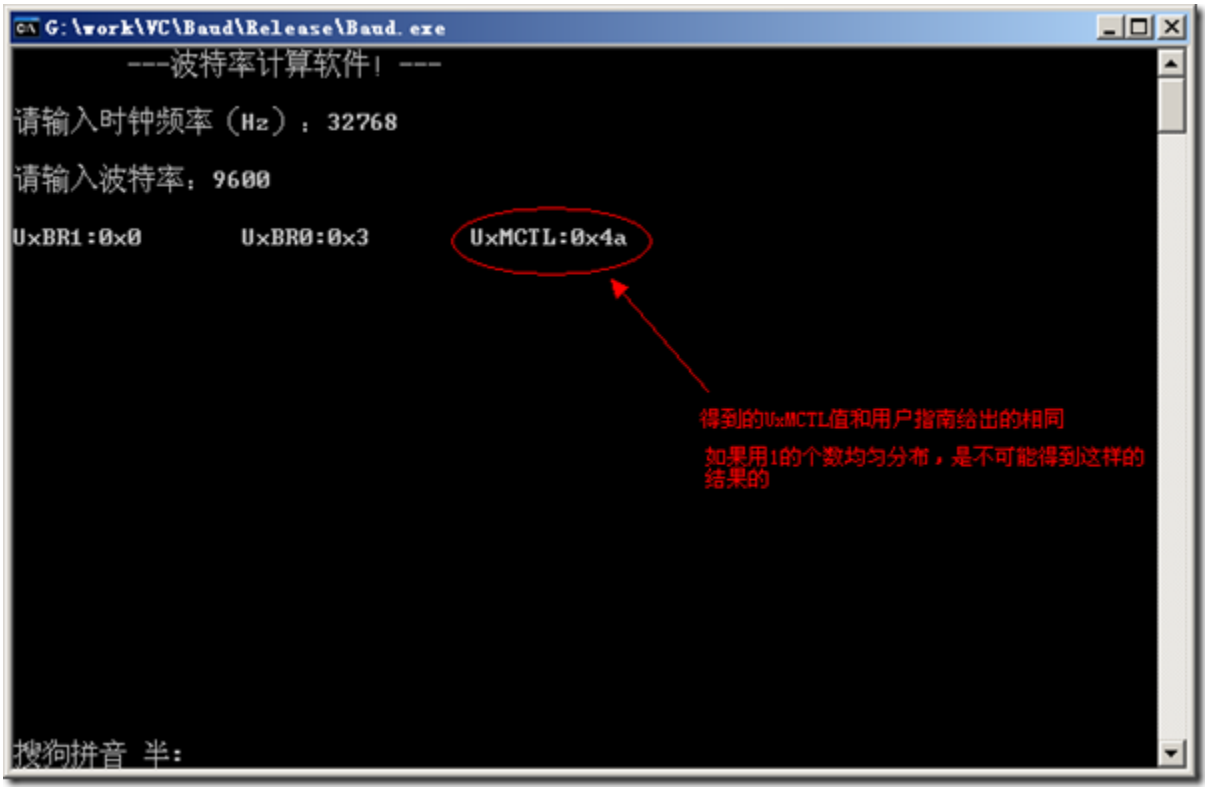
波特率寄存器值根据所选时钟频率和所需波特率值进行设置，计算方法：从 m0（UxMCTL 最低位）开始计算，根据这一位的误差（0 或 1 时）误差较小的 bit 值，直到计算完成。

为了更好的写这个程序，我先用 C 语言写了一个简单的波特率计算软件，为了让设置波特率的函数能够在单片机程序中复用,程序用宏定义模拟的 MSP430 单片机的波特率寄存器。完整程序如下：

```
#include<stdio.h>
#include<math.h>

//函数声明
void SetBaudRateRegisters(long clk,int baud);

/************************宏定义************************/
#define UxBR1    a[0]
#define UxBR0    a[1]
#define UxMCTL   a[2]

unsigned char a[3];                          //数组模拟寄存器
void main()
{
    long clk;                                //时钟
    long baud;                                //波特率
    printf("\t---波特率计算软件！ ---\n");
    printf("\n 请输入时钟频率（Hz）： ");
    scanf("%ld",&clk);
    printf("\n 请输入波特率： ");
    scanf("%ld",&baud);
    getchar();                               //读取多余回车符

    SetBaudRateRegisters(clk,baud);          //设置寄存器值

    //显示寄存器值

printf("\nUxBR1:0x%x\tUxBR0:0x%x\tUxMCTL:0x%x\n",UxBR1,UxBR0,UxMCTL);

    getchar();
}

/*************************************************************
* 名    称：SetBaudRateRegisters
* 功    能：根据时钟 波特率设置对应寄存器
* 入口参数：
*           clk:        所选时钟频率（如：32768）
            baud        波特率       (300~115200)
* 出口参数：无
```

```
* 范    例: SetBaudRateRegisters(32768,9600) //用时钟频率 32768 产生
9600 的波特率
**************************************************/
void SetBaudRateRegisters(long clk,long baud)
{
    int n = clk / baud;      //整数波特率
    char mSum = 0;             //Σ mi
    int txEr0;               //对应位为 0 时错误率
    int txEr1;               //对应位为 1 时错误率
    char i = 0;              //循环计数

    UxBR1 = n >> 8;          //高 8 位
    UxBR0 = n & 0xff;        //低 8 位
    UxMCTL = 0;

    //循环 比较错误率大小 设置 UxMCTL
    for(;i < 8;i++)
    {
        txEr0 = 100*baud*((i+1)*n+mSum)/clk-100*(i+1);
        txEr1 = 100*baud*((i+1)*n+mSum+1)/clk-100*(i+1);
        if(abs(txEr1) < abs(txEr0))
        {
            mSum++;
            UxMCTL |= (1<<i);
        }
    }
}
```

程序可以使用任何的 C 语言编译器编译运行，可供网友们复用此程序。我使用 vs2010 编译运行的，运行结果如下：



运行效果很好，和官方给出的值一样，但是也不全都是这样，4800 的波特率（时钟：32768）时就不一样，可能是我计算式只是用了发送时的误差计算，没有用接收误差，计算结果稍有出入，如果有兴趣，网友可以自行添加接收误差，判断；应该就和官方给出的数值完全一样了。

初始化函数：初始化函数完成串口时钟源选择，波特率初始化，奇偶校验，数据位，停止位，以及其他相关设置。

时钟源选择：根据波特率选取时钟源，波特率大于 9600，选 1M 的 SMCLK 时钟（需要初始化时钟系统对应函数参考使用示例），小于 9600，选 ACLK（32768）以使功耗降低（低功耗 3 仍能正常收发数据）

```
UxTCTL &=~ (SSEL0+SSEL1);          //清除之前的时钟设置
if(baud<=9600)                     //brclk 为时钟源频率
{
  UxTCTL |= SSEL0;                 //ACLK，降低功耗
  brclk = 32768;                   //波特率发生器时钟频率=ACLK(32768)
}
else
{
  UxTCTL |= SSEL1;                 //SMCLK，保证速度
  brclk = 1000000;                 //波特率发生器时钟频率=SMCLK(1MHz)
}
```

波特率设置：直接调用之前实现的设置寄存器函数即可，当波特率在正常范围外时，返回 0。

```
//------------------------设置波特率-------------------------
if(baud < 300||baud > 115200)    //波特率超出范围
{
    return 0;
}
SetBaudRateRegisters();          //设置波特率寄存器
```

奇偶校验、数据位位数、停止位数设置：比较简单，直接根据参数值设置对应寄存器即可。

```
//------------------------设置校验位-------------------------
switch(parity)
{
    case 'n':case'N': UxCTL &=~ PENA;                 break; //无校验
    case 'p':case'P': UxCTL |= PENA + PEV;            break; //偶校验
    case 'o':case'O': UxCTL |= PENA; UxCTL &=~ PEV; break; //奇校验
    default :         return(0);                              //参数错误
}

//------------------------设置数据位-------------------------
switch(dataBits)
{
    case 7:case'7': UxCTL &=~ CHAR; break;       //7 位数据
    case 8:case'8': UxCTL |= CHAR;  break;       //8 位数据
    default :       return(0);                   //参数错误
}
//------------------------设置停止位-------------------------
switch(stopBits)
{
    case 1:case'1': UxCTL &= ~SPB;  break;       //1 位停止位
    case 2:case'2': UxCTL |= SPB;   break;       //2 位停止位
    default :       return(0);                   //参数错误
}
```

其他：包括串口收发使能，串口接收和发送中断设置，第二功能打开等。

```
    UARTON;                     //端口使能
    UxME |= UTXEx + URXEx;      //发送 接收使能

    UCTL0 &= ~SWRST;            // Initialize USART state machine

    UxIE |= URXIEx + UTXIEx;    // Enable USART0 RX interrupt
```

到此，MSP430 异步串行口的初始化工作全部完成，如果需要其他的方式，只需对应设置寄存器即可。

- 写字符（UartxWriteChar）：向 UARTx 模块写（发送）一个字符。

写字符：向串口写入一个字符，通过串口向终端发送一个字符。

```
void UartWriteChar(char c)
{
  while (TxFlag==0) UartLpm();  // 等待上一字节发完，并休眠
  TxFlag=0;                     //
  UxTXBUF=c;
}
```

这个函数根据程序标志 TxFlag 判断上一字符是否发送完成，此标志位将在发送中断中被置位，表示本字符发送完成。发送中断程序如下：

```
#pragma vector=UARTxTX_VECTOR
__interrupt void UartTx ()
{
  TxFlag=1;
  __low_power_mode_off_on_exit();
}
```

发送字符时，先等待上一字符发送完成，然后把字符放入发送缓冲区，待发送完成，中断置标志位，指示发送完成。

- 读取字符（UartxReadChar）：从 UARTx 模块读取（获取）一个字符。

读取字符和写字符类似：调用读取函数后，等待标志位，接收到字符后，读出来。

```
char UartReadChar()
{
  while (RxFlag==0) UartLpm(); // 收到一字节?
  RxFlag=0;
  return(UxRXBUF);
}
```

同样，RxFlag 指示收到一个字符，并且在中断中被置位。中断程序如下：

```
#pragma vector=UARTxRX_VECTOR
__interrupt void UartRx()
{
  RxFlag=1;
  /*在这里添加用户中断服务程序代码，如将数据压入接收缓冲等*/
  __low_power_mode_off_on_exit();
}
```

读取函数将阻塞，如果收不到字符，CPU 将一直处于低功耗状态。

- 写字符串（UartxWriteStr）：向 UARTx 模块写（发送）一个字符串。

写字符串只需调用写字符函数即可，比较简单，程序如下：

```
void UartWriteStr(char *s)
{
    while(*s)
    {
        UartWriteChar(*s++);
    }
}
```

这样，即可调用这个函数通过串口发送字符串。

- 头文件：头文件把要调用的函数声明放进去，需要使用函数时只需包含此文件，不需要再进行函数声明。

头文件内容如下：

```
#ifndef __UART_H
#define __UART_H

char UartInit(long baud,char parity,char dataBits,char stopBits);
void UartWriteChar(char c);
void UartWriteStr(char *s);
char UartReadChar();

#endif /* __UART_H */
```

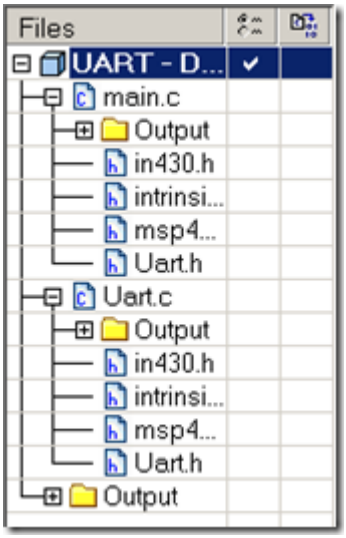
其中#ifndef 等预编译用来防止重复包含。

- 程序调用示例：

要调用这个函数库，首先要包含 Uart.h 头文件；把 Uart.h 拷到对应文件夹中，然后在要调用程序的源程序文件中添加文件包含：

```
#include "msp430x16x.h"    //430 寄存器头文件
#include "Uart.h"          //串口通讯程序库头文件
```

然后，在项目中加入 Uart.c，把 Uart.c 拷入项目文件夹中，在项目中添加文件，加入后文件结构大致如下图：



如果要用 9600 以上的波特率，需要把 SMCLK 设为 1M，我的程序调用了以下的这个函数，把 MCLK 设为 8MHz，SMCLK 设为 1MHz：

```
void ClkInit()
{
    char i;
    BCSCTL1 &= ~XT2OFF;                 //打开 XT2 振荡器
    IFG1&=~OFIFG;                       //清除振荡错误标志
    while((IFG1&OFIFG)!=0)
    {
        for(i=0;i<0xff;i++);
        IFG1&=~OFIFG;                   //清除振荡错误标志
    }
    BCSCTL2 |= SELM_2+SELS+DIVS_3;   //MCLK 为 8MHz，SMCLK 为 1MHz
}
```

调用示例程序如下：

```
ClkInit();

UartInit(38400,'n',8,1);
 //串口初始化,设置成 38400bps,无校验,8 位数据,1 位停止
_EINT();
UartWriteStr(str);
UartWriteChar(0x0d);     //发送"换行"(\r)"
UartWriteChar(0x0a);     //发送"回车"(\n)"

UartWriteStr("下面测试串口收发函数\r\n");

while(1)                          //串口测试
```

```
{
    chr=UartReadChar();       //收 1 字节
    UartWriteChar(chr);       //将收到的数据返回
}
```

如果是 9600 以下的波特率，可以不调用时钟系统初始化函数，否则必须调用这个函数，或者用其他的方法把 SMCLK 频率设为 1MHz；初始化完成之后，还要开中断（因为 Uart 函数库用到了中断）；然后才能正常的使用这些函数。

这样，这个程序库就完成了，欢迎大家下载使用。

另外，网上还有一个用预编译的方法计算波特率的，个人认为比较好，没有选用它的原因是：它只能在编译时确定寄存器内容，无法再运行的时候进行设置。
http://blog.21ic.com/user1/1453/archives/2009/62696.html 还有一个用网页计算波特率的，可以上网时计算波特率很方便：
http://www.838dz.com/calculator/1805.html

附件：程序库，其他相关

MSP430 程序库<三>12864 液晶程序库

液晶是单片机系统最常用的显示设备之一，这个程序库是在 MSP430F169、MSP1430F149 单片机上测试通过的，可以放心使用；液晶选用的是金鹏的：OCMJ4X8C 型号的 12864 液晶；控制液晶用的是并行方式，三个控制口是 P3.0、P3.1、P3.2 三个 IO 口，数据用的是 P5 数据 IO 口。

- 硬件介绍：

430 的数字 IO 口：

MSP430F149、MSP430F169 均有 P1-P6 每个 8 位 共 48 个 IO 口；有大量的 IO 口可供使用，所以对液晶控制可以选用 8 位平行数据方式；430 的每个 IO 口都是双向 IO 口，通过寄存器控制其数据传输方向，很方便实用；有关 msp430 单片机的 IO 口介绍可以参考德州仪器提供的用户指南和数据手册等资料。

液晶 OCMJ4X8C：

此模块可以显示字母、数字符号、中文字型及图形，具有绘图及文字画面混合显示功能。提供三种控制接口，分别是 8 位微处理器接口，4 位微处理器接口及串行接口(OCMJ4X16A/B

无串行接口）。所有的功能，包含显示 RAM，字型产生器，都包含在一个芯片里面，只要一个最小的微处理系统，就可以方便操作模块。

这款液晶内置2M-位中文字型ROM (CGROM) 总共提供8192个中文字型(16x16点阵)，16K 位半宽字型 ROM(HCGROM) 总共提供 126 个符号字型(16x8 点阵)，64x16 位字型产生 RAM(CGRAM)，另外绘图显示画面提供一个 64x256 点的绘图区域（GDRAM），可以和文字画面混和显示。

OCMJ4X8C 的引脚说明：

引脚	名称	方向	说明
1	VSS	-	GND(0V)
2	VDD	-	Supply Voltage For Logic(+5V)
3	NC	-	Supply Voltage For LCD(悬空)
4	RS(CS)	I	H:Data L:Instruction Code
5	R/W(STD)	I	H:Read L:Write
6	E(SCLK)	I	Enable Signal,高电平有效
7	DB0	I/O	数据 0
8	DB1	I/O	数据 1
9	DB2	I/O	数据 2
10	DB3	I/O	数据 3
11	DB4	I/O	数据 4
12	DB5	I/O	数据 5
13	DB6	I/O	数据 6
14	DB7	I/O	数据 7
15	PSB	I	H:Parallel Mode L:Serial Mode
16	NC	-	空脚
17	/RST	I	Reset Signal,低电平有效
18	NC	-	空脚
19	LEDA	-	背光源正极（+5V）
20	LEDK	-	背光源负极（0V）

引脚连接方式：PSB、RST 接高电平（3.3v）；RS 接 P3.0；R/W 接 P3.1；E 接 P3.2；DB0-DB7 接 P5 口，电源接 3.3v（包括背光） GND 接地（包括背光）。

汉字图形显示步骤：

1、显示资料 RAM（DDRAM）

显示数据 RAM 提供 64x2 个字节的空间，最多可以控制 4 行 16 字（64 个字）的中文字型显示，当输入显示资料 RAM 时，可以分别显示 CGROM，HCGROM 与 CGRAM 的字型；本系列模块可以显示三种字型，分别是半宽的 HCGROM 字型、CGRAM 字型及中文 CGROM 字型，三种字型的选择，由在 DDRAM 中写入的编码选择，在 0000H～0006H

的定字型，02H～7FH 的编码中将选择半编码中将选择 CGRAM 的自宽英数字的字型，至于 A1 以上的编码将自动的结合下一个字节，组成两个字节的编码达成中文字型的编码。

BIG5（A140～D75F） GB(A1A0～F7FF)，详细各种字型编码如下：

1）．显示半宽字型：将 8 位资料写入 DDRAM 中，范围为 02H～7FH 的编码。

2）. 显示 CGRAM 字型:将 16 位资料写入 DDRAM 中,总共有 0000H,0002H,0004H,0006H 四种编码。

3）．显示中文字形：将 16 位资料写入 DDRAM 中，范围为 A140H～D75FH 的编码(BIG5) ， A1A0H～F7FFH 的编码(GB)。将 16 位资料写入 DDRAM 方式为透过连写入两个字节的资料来完成，先写入高字节（D15～D8）再写入低字节（D7～D0）。

2、绘图 RAM（GDRAM）

绘图显示 RAM 提供 64x32 个字节的记忆空间(由扩充指令设定绘图 RAM 地址)，最多可以控制 256x64 点的二维绘图缓冲空间,在更改绘图 RAM 时,由扩充指令设定 GDRAM 地址先设垂直地址再设水平地址(连续写入两个字节的数据来完成垂直与水平的坐标地址)，再写入两个 8 位的资料到绘图 RAM,而地址计数器（AC）会自动加一,整个写入绘图 RAM 的步骤如下：

1）．先将垂直的字节坐标（Y）写入绘图 RAM 地址。

2）．再将水平的字节坐标（X）写入绘图 RAM 地址。

3）．将 D15～D8 写入到 RAM 中(写入第一个 Bytes)。

4）．将 D7～D0 写入到 RAM 中(写入第二个 Bytes)。

液晶屏显示地址：

	X 坐标							
Line1	80H	81H	82H	83H	84H	85H	86H	87H
Line2	90H	91H	92H	93H	94H	95H	96H	97H
Line3	88H	89H	8AH	8BH	8CH	8DH	8EH	8FH
Line4	98H	99H	9AH	9BH	9CH	9DH	9EH	9FH

有关液晶其他的或详细的介绍，请参考 12864 液晶的资料。

1． 程序实现：

- 判忙： 等待液晶模块空闲。

液晶模块要求：当模块在接受指令前，微处理顺必须先确认模块内部处于非忙碌状态，即读取 BF 标志时 BF 需为 0，方可接受新的指令；如果在送出一个令前并不检查 BF 标志，那幺在前一个指令和这个指令中间必须延迟一段较长的时间，即是等待前一个指令确实执行完成；在这里，我选用等待忙标志结束。程序如下：

```
void WaitForEnable()
{
    char busy;

    CLR_RS;
    SET_RW;

    DATA_DIR_IN;

    do                          //判忙
    {
        SET_EN;
        _NOP();
        busy = DATA_IN;
        CLR_EN;
    }
    while(busy & 0x80);

    DATA_DIR_OUT;
}
```

这样，每次向液晶写命令或数据时，只需先调用此函数即可，该函数将会阻塞，直到忙标志变回 0（内部空闲，可以接受命令）。

- 写入数据： 向模块内部 RAM 写入数据。

写入数据到 DDRAM 即可显示到液晶，写入函数：

```
void LcdWriteData(char data)
{
    WaitForEnable();

    SET_RS;
    CLR_RW;

    DATA_OUT = data;     //写数据

    SET_EN;
    _NOP();
```

```
    CLR_EN;
}
```

同样，调用这个函数也可以向其他 RAM 写入数据，完成相应操作。

- 写入命令： 向模块写入命令。

写入命令可以通过液晶的指令集，控制液晶完成相应的功能。程序如下：

```
void LcdWriteComm(char cmd)
{
    WaitForEnable();     //检测忙信号?

    CLR_RS;
    CLR_RW;

    DATA_OUT = cmd;     //写命令

    SET_EN;
    _NOP();
    CLR_EN;
}
```

如果 cmd 是 0x80-0x9F，则是向液晶写入地址；地址表参见前面硬件介绍部分。

- 写入字符串： 写入字符串，以显示。

写入字符串即是多次调用写入数据，把字符串写入液晶以供显示。程序如下：

```
void LcdWriteString(char addr,char *str)
{
    LcdWriteComm(addr);
    while(*str!='\0')
    {
        LcdWriteData(*str);
        str++;
    }
}
```

这是向某个地址写入字符串，液晶显示到相应位置。这个函数有个要求，就是字符串是中文字符串；如果不是，每一处的英文必须两个相连，否则将显示乱码，如果只有一个英文字符，可以加入空格；如：**LcdWriteString(0x90,"1 abcd 你好啊");** 1+空格+abcd+汉字中 1 只有一个字符，加空格，ab，cd 两个，直接显示到一个汉字的位置。

- 液晶初始化：液晶必须初始化之后才能正常使用。

初始化就是一系列命令，完成液晶状态的初始工作，以使液晶可供正常使用。程序如下：

```
void LcdInit()
{
    CTRL_DIR_OUT;

    DelayNms(500);
    LcdWriteComm(0x30);      //基本指令集
    LcdWriteComm(0x01);      //清屏，地址 00H
    LcdWriteComm(0x06);      //光标的移动方向
    LcdWriteComm(0x0c);      //开显示，关游标
}
```

在运行过这个函数之后，液晶方能正常的显示；在调用液晶显示函数前，必须先调用这个函数。

2. 程序实现就先到这儿，还可以加入显示图片等功能；要收拾东西回去了，其他功能暂不实现了，以后需要的时候再加入。
3. 使用示例：

这个程序的使用方式和串口程序库的使用方式一样，把 C 文件加入工程；H 文件包含进要调用的程序源文件中即可。

```
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    LcdInit();
    LcdWriteString(0x90,"1 abcd 你好啊");
}
```

这个函数运行后，将在第二行显示 1 abcd 你好啊 字符串，如果把 1 后面的空格去掉，中文部分将是乱码。ClkInit(); 这个函数和前面一个里面调用的一样，把主系统时钟设为 8MHz，SMCLK 设为 1MHz。 有关详细内容参见程序库，mian.c。

到此，液晶的驱动基本完成，其他功能之后再添加了。如果有不好或不对的地方，欢迎大家提出，谢谢啦。

附：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<四>printf 和 scanf 函数移植

printf 和 scanf 函数是 C 语言中最常用的输入出函数，从学习 C 语言开始，就开始使用这两个函数，然而当写用 C 语言写单片机程序时却不能使用这两个函数，总觉得单片机的 C 语言和一般的 C 语言差别很大，写起来不大方便；其实，单片机的 C 语言也是标准 C 语言上扩展或是改动的，都支持格式化输入输出函数（printf 和 scanf）；事实上，printf，scanf 只负责格式化输入输出的字符，至于从哪儿输入，输出到哪儿，他们分别依靠 getchar 和 putchar 函数，只要实现单片机上的 getchar 函数和 putchar 函数，即可正常使用 printf 函数和 scanf 函数，这可以给我们单片机的信息交互带来很多方便。下面我们就来实现他们的移置。

- **硬件介绍：**

硬件部分只需字符型输入输出设备：scanf 从输入字符型设备读取字符，printf 输出到字符型输出设备。在这里，我选用的字符型输入设备是超级终端，通过串口与单片机连接，输入字符；输出设备是超级终端和 12864 的液晶。scanf 从串口读入字符，printf 输出字符到串口和液晶。

scanf 还可以从按键读取信息，可以参考移置方法自行移置。

- **程序实现：**
- printf

单片机在调用 printf 时，printf 是负责将数据解析成 ASCII 码流，通过调用 putchar 函数依次将字符发出。如果在 putchar 内编写从串口发送一字节数据，则 printf 的结果将从单片机串口发送出；如果 putchar 是向液晶写字符，让液晶显示一个字符，则 printf 的结果将显示在液晶上。本程序实现 putchar 同时向串口和液晶同时发送一个字符（液晶是显示一个字符）。

putchar 函数如下：

```
int putchar(int ch)
{
    putchar2Com(ch);
    putchar2Lcd(ch);
    return (ch);
}
```

程序先向串口发送一个字符，然后像向晶发送字符。

其中：putchar2Com，向串口发送一个字符，代码如下：

```
int putchar2Com(int ch)
{
    if (ch =='\n')              //  '\n'(回车)扩展成 '\n''\r' (回车+换行)
    {
        UartWriteChar('\r') ;    //0x0d 换行
    }
    UartWriteChar(ch);          //从串口发出数据
    return (ch);
}
```

代码仅仅调用向串口写字符的函数 UartWriteChar(ch)（详见 Uart.c，在<二>中有介绍），当要输出换行时，需先输出'\n'将光标移至本行首位置，还需要'\r'（换行）才能将光标置于下一行起始位置，即将'\n'扩展为'\r','\n'两个字节依次发出。

purchar2Lcd 函数比较复杂，因为我所使用的 12864 液晶是中文字库的液晶，每行 8 个地址，可以显示 8 个中文字符或 16 个英文字符，而 putchar 只发出一个字节，需要判断每个地址的前半字还是后半字（因为每个字可以显示中文，如果中文的两个字节在相邻的两个地址上，将不会显示，或是显示乱码）。

上代码：

```
int putchar2Lcd(int ch)
{
    char addr,dat;

    if (ch == '\n')             //  '\n'(回车),换行
    {
        ChangeNextRow();
    }
    else
    {
        addr = LcdReadAddr();
        if(ch < 0x80)
        {
            LcdWriteData(ch);
        }
        else
        {
            //写入一个空字符，根据地址判断是否为前半字
            LcdWriteData(0x20);
             if(addr == LcdReadAddr())    //前半字 从新写入 ch 字符
            {
```

```
                LcdWriteComm(addr);
                LcdWriteData(ch);
            }
            else
            {
                LcdWriteComm(addr);
                dat = LcdReadData();
                if(dat < 0x80)              //前一个字符是英文字符
                {
                 LcdWriteData(0x20);                    //空格
              }
                LcdWriteData(ch);
            }
        }
    }
    if((addr != LcdReadAddr()) &&     //写入的是行最后位的后半字则换行
       (addr==0x87 || addr==0x97 || addr==0x8F || addr==0x9F))
    {
        ChangeNextRow();
    }
    return (ch);
}
```

这个函数首先判断换行；然后处理其他一般字符，如果是英文字符，不用考虑前后半字，只需正常写入液晶即可；如果是中文字符，在判断是否是前半字，前半字则直接写入，后半字则判断之前写入的前半字是否是中文，是则直接写入，不是则把英文字符移入后半字，然后写入；最后判断是否到行尾，是则换行。

程序更新为：更新日期：20110821 18:51
目的是修复原来，行尾前半字为英文，再输入中文会显示乱码。

```
int putchar2Lcd(int ch)
{
    char addr,dat;
    char changeRowFlag = 0;

    if (ch == '\n')          //  '\n'(回车),换行
    {
        ChangeNextRow();
        changeRowFlag = 1;
    }
    else if (ch == '\b')     // '\b' (退格)
    {
        BackSpace();
    }
```

```
    else
    {
        addr = LcdReadAddr();
        if(ch < 0x80)
        {
            LcdWriteData(ch);
        }
        else
        {
            //写入一个空字符，根据地址判断是否为前半字
            LcdWriteData(0x20);
            if(addr == LcdReadAddr())    //前半字 从新写入 ch 字符
            {
                LcdWriteComm(addr);
                LcdWriteData(ch);
            }
            else
            {
                LcdWriteComm(addr);
                dat = LcdReadData();
                if(dat < 0x80)            //前一个字符是英文字符
                {
                    LcdWriteData(0x20);                    //空格
                }
                if((addr != LcdReadAddr()) &&//写入的是行最后位的后半
字则换行
                   (addr==0x87 || addr==0x97 || addr==0x8F ||
addr==0x9F))
                {
                    ChangeNextRow();
                    changeRowFlag = 1;
                }
                LcdWriteData(ch);
            }
        }
    }
    if((addr != LcdReadAddr()) &&//写入的是行最后位的后半字则换行,且未
换过行
       (changeRowFlag == 0) &&
       (addr==0x87 || addr==0x97 || addr==0x8F || addr==0x9F))
    {
        ChangeNextRow();
    }
    return (ch);
```

```
}
```

前后半字判断方法如下：读液晶地址，向液晶写入一个空格，再读地址，两地址相同则是前半字，不同则是后半字。读地址函数在 Lcd12864.c 中，新加入函数，代码如下：

```
char LcdReadAddr()
{
    char ch;

    WaitForEnable();

    CLR_RS;
    SET_RW;

    DATA_DIR_IN;

    SET_EN;
    _NOP();

    ch = DATA_IN;     //读数据
    CLR_EN;
    DATA_DIR_OUT;

    return (ch|0x80);
}
```

这个是读地址，ch|0x80 是因为写入液晶地址首位应为 1.。

液晶中新加入两个函数，一个是上边的读地址，另外一个是读数据；作用是读取液晶当前地址处的数据，从而判断之前半字是否是中文。代码如下：

```
char LcdReadData()
{
    char ch;

    WaitForEnable();

    SET_RS;
    SET_RW;

    DATA_DIR_IN;

    SET_EN;
    _NOP();
```

```
    ch = DATA_IN;     //读数据
    CLR_EN;
    DATA_DIR_OUT;

    return ch;
}
```

另外 putchar 还调用了换行——ChangeNextRow 函数，完成液晶输出换至下一行。

代码如下：

```
void ChangeNextRow()
{
    char addr;

    addr = LcdReadAddr();        //当前地址
    if(addr <= 0x88)
    {
        LcdWriteComm(0x90);
    }
    else if(addr <= 0x90)
    {
        LcdWriteComm(0x98);
    }
    else if(addr <= 0x98)
    {
        LcdWriteComm(0x88);
    }
    else
    {
        AddNewline();            //添加行，同时向上滚动
        LcdWriteComm(0x98);
    }
}
```

读取当前地址，判断在哪一行，然后写入下一行首地址；如果是最后一行，则所有安徽那个向上移，写入最后一行首地址。

AddNewLine 函数完成所有行向上滚动一行，然后地址定位至最后一行。

代码如下：

```
void AddNewline()
{
    char str[17];
```

```
    str[16] = 0;

    //第二行 移至第一行
    LcdWriteComm(0x90);
    LcdReadData();                  //空读取
    for(int i = 0;i<16;i++)
    {
        str[i] = LcdReadData();
    }
    LcdWriteString(0x80,str);

    //第三行 移至第二行
    LcdWriteComm(0x88);
    LcdReadData();
    for(int i = 0;i<16;i++)
    {
        str[i] = LcdReadData();
    }
    LcdWriteString(0x90,str);

    //第四行 移至第三行
    LcdWriteComm(0x98);
    LcdReadData();
    for(int i = 0;i<16;i++)
    {
        str[i] = LcdReadData();
    }
    LcdWriteString(0x88,str);

    //第四行 空白
    LcdWriteString(0x98,"                ");     //十六个空格
}
```

读出下一行数据，写入上一行，最后一行写入空格即可。

到此 putchar 函数全部完成，printf 移植的程序部分完成，使用方法详见使用示例。

- scanf

scanf 和 printf 类似，其只负责格式化输入的字符，字符来源是从 getchar 函数获取；同样，在使用 scanf 函数之前，要针对字符输入源自行编写 getchar 函数

最简 getchar：

```
int getchar()
{
    return (putchar(UartReadChar()));
}
```

这是最简单的 getchar 函数，直接调用读取字符函数，输出并返回。

但是人的输入过程会偶尔犯错误的，为了支持退格键等，需要开辟一个缓存区。

详细代码如下：

```
#define LINE_LENGTH 80    //行缓冲区大小，决定每行最多输入的字符数

/*标准终端设备中，特殊 ASCII 码定义，请勿修改*/
#define InBACKSP 0x08           //ASCII  <--  (退格键)
#define InDELETE 0x7F           //ASCII <DEL> (DEL 键)
#define InEOL '\r'              //ASCII <CR>  (回车键)
#define InSKIP '\3'             //ASCII control-C
#define InEOF '\x1A'            //ASCII control-Z

#define OutDELETE "\x8 \x8"     //VT100 backspace and clear
#define OutSKIP "^C\n"          //^C and new line
#define OutEOF "^Z"             //^Z and return EOF
int getchar()
{
    static char inBuffer[LINE_LENGTH + 2];      //Where to put chars
    static char ptr;                            //Pointer in buffer
    char c;

    while(1)
    {
        if(inBuffer[ptr])                       //如果缓冲区有字符
            return (inBuffer[ptr++]);           //则逐个返回字符
        ptr = 0;                        //直到发送完毕，缓冲区指针归零
        while(1)                        //缓冲区没有字符，则等待字符输入
        {
            c = UartReadChar();                 //等待接收一个字符
            if(c == InEOF && !ptr)              //==EOF==  Ctrl+Z
            {                               //只有在未入其他字符时才有效
                printf(OutEOF);                 //终端显示 EOF 符
                return EOF;                     //返回 EOF (-1)
            }
            if(c==InDELETE || c==InBACKSP)      //==退格或删除键==
            {
```

```
                if(ptr)                             //缓冲区有值
                {
                    ptr--;                          //从缓冲区移除一个字符
                    printf(OutDELETE);              //同时显示也删掉一个字符
                }
            }
            else if(c == InSKIP)                    //==取消键 Ctrl+C ==
            {
                printf(OutSKIP);                    //终端显示跳至下一行
                ptr = LINE_LENGTH + 1;              //==0 结束符==
                break;
            }
            else if(c == InEOL)                     //== '\r' 回车==
            {
                putchar(inBuffer[ptr++] = '\n');//终端换行
                inBuffer[ptr] = 0;                  //末尾添加结束符(NULL)
                ptr = 0;                            //指针清空
                break;
            }
            else if(ptr < LINE_LENGTH)              //== 正常字符 ==
            {
                if(c >= ' ')                        //删除 0x20 以下字符
                {
                    //存入缓冲区
                    putchar(inBuffer[ptr++] = c);
                }
            }
            else                                    //缓冲区已满
            {
                putchar('\7');              //== 0x07 蜂鸣符，PC 回响一声
            }
        }
    }
}
```

注释已经很详细了，这里不再详细解释。

scanf 的移植程序部分已经完成，如果需要从键盘读入字符，可以仿照上述函数写 getchar 函数。具体使用和设置见使用示例。

另外，iar 的安装文件夹下，430 文件夹下有一个 src 文件夹，lib/clib 文件夹下（我的具体文件夹是：D:\Program Files\IAR Systems\Embedded Workbench 6.0 Evaluation\430\src\lib\clib\getchar.c），有一个 getchar.c 文件，这是 getchar 的函数，内容如下：

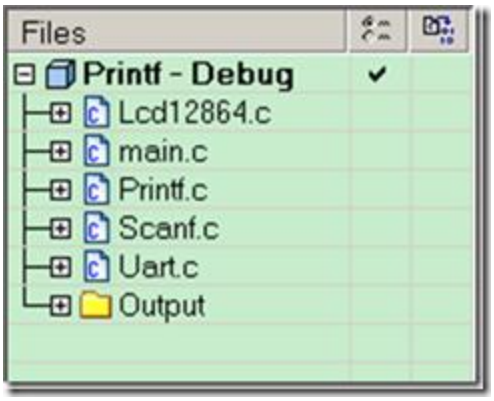
```c
#include "stdio.h"

extern char _low_level_get(void);       /* Read one char from I/O */
                                        /* Should be supplied by user */

static void put_message(char *s)
{
  while (*s)
    putchar(*s++);
}


#define LINE_LENGTH 80          /* Change if you need */

#define In_DELETE 0x7F          /* ASCII <DEL> */
#define In_EOL '\r'             /* ASCII <CR> */
#define In_SKIP '\3'            /* ASCII control-C */
#define In_EOF '\x1A'           /* ASCII control-Z */

#define Out_DELETE "\x8 \x8"    /* VT100 backspace and clear */
#define Out_SKIP "^C\n"         /* ^C and new line */
#define Out_EOF "^Z"            /* ^Z and return EOF */

int getchar(void)
{
  static char io_buffer[LINE_LENGTH + 2];     /* Where to put chars */
  static int ptr;                              /* Pointer in buffer */
  char c;

  for (;;)
  {
    if (io_buffer[ptr])
      return (io_buffer[ptr++]);
    ptr = 0;
    for (;;)
    {
      if ((c = _low_level_get()) == In_EOF && !ptr)
      {
        put_message(Out_EOF);
        return EOF;
      }
      if (c == In_DELETE)
      {
        if (ptr)
```

```
        {
          ptr--;
          put_message(Out_DELETE);
        }
      }
      else if (c == In_SKIP)
      {
        put_message(Out_SKIP);
        ptr = LINE_LENGTH + 1;  /* Where there always is a zero... */
        break;
      }
      else if (c == In_EOL)
      {
        putchar(io_buffer[ptr++] = '\n');
        io_buffer[ptr] = 0;
        ptr = 0;
        break;
      }
      else if (ptr < LINE_LENGTH)
      {
        if (c >= ' ')
        {
          putchar(io_buffer[ptr++] = c);
        }
      }
      else
      {
        putchar('\7');
      }
    }
  }
}
```

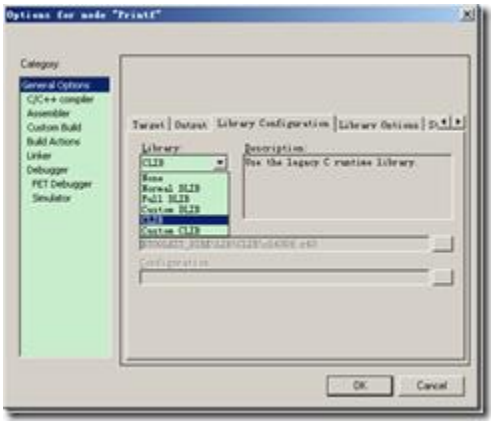
_low_level_get(void); 这个函数需用户定义，不过这个 getchar 函数不支持退格键，可以更改以支持；_low_level_get(void);这个函数可以直接调用 UartReadChar();这个函数，使用时，把 getchar.c 加入项目，同时在项目中添加_low_level_get(void);函数，函数体只有一句：return UartReadChar();即可。

程序调用示例：

程序使用方式，项目中添加 printf.c 文件和 scanf.c 文件（用 printf 函数则加 printf.c 文件，用 scanf 函数就添加 scanf.c 文件），在要使用函数的地方包含 stdio.h（编译器自带库——标准输入输出库）



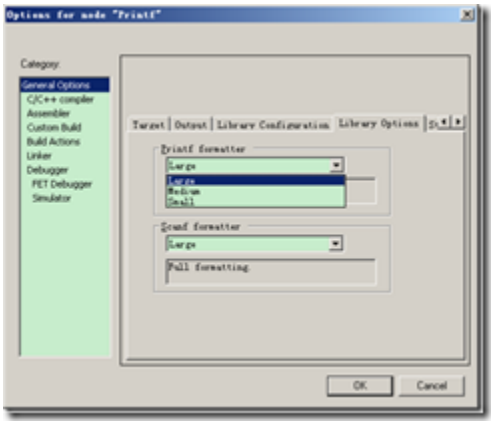
还要设置使用库和 printf 的大小：



如果不进行这项设置，使用 scanf 时将报错：

Error[e27]: Entry "getchar" in module Scanf (G:\work\程序库\Printf\Debug\Obj\Scanf.r43) redefined in module ?getchar (D:\Program Files\IAR Systems\Embedded；用的是 C 语言，这里选择 CLIB。

然后设置库选项：



这里选择大尺寸，目的是支持所有的格式，因为所用单片机有 64kb 的程序存储空间，足够使用，如果程序存储空间不够大，推荐选择中尺寸或小尺寸。大尺寸 printf 占用空间 4.8kb、scanf ：2.3kb，中尺寸 printf：2.5kb、scanf：1.6kb，小尺寸 printf：1.6kb。实际使用时根据需要进行选择。

同时要加入 Lcd12864 的使用（c 文件，h 文件（要调用 lcd12864 的初始化函数））Uart 和液晶一样要调用初始化函数。

```
#include <msp430x16x.h>
#include <stdio.h>
#include "Uart.h"
#include "Lcd12864.h"
```

头文件包含。

```
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    LcdInit();
    UartInit(38400,'n',8,1);//串口初始化,设置成 38400bps,无校验,8 位数据,1 位停止
    //int a;
    _EINT();
    //scanf("%d",&a);
    //printf("刘中原%d\n",a);
    printf("刘中原%f\n",23.6);
    printf("刘中原%1.2f\n",23.6);
}
```

使用时，先调用液晶和串口的初始化函数，然后开中断；就可以正常的调用 scanf 和 printf 函数了。

至此，printf 和 scanf 的移植全部完成，使用这两个函数将给单片机的输入输出带来极大方便。另外，Lcd12864 的液晶使用是 4 行显示，空间较小，可能需要定位至具体位置，以使界面看起来更合理，为此，在 Printf 中再添加一个定位函数（GotoXY）：

```
void GotoXY(char x,char y)
{
    char addr;

    if(y==0)
    {
        addr = 0x80 + x / 2;
```

```
    }
    else if(y==1)
    {
        addr = 0x90 + x / 2;
    }
    else if(y==2)
    {
        addr = 0x88 + x / 2;
    }
    else
    {
        addr = 0x98 + x / 2;
    }
    LcdWriteComm(addr);
    if(x % 2)                       //是奇数，后移一位（写入空格）
    {
        LcdWriteData(0x20);
    }
}
```

这样就方便了液晶程序的编写。

又加入一个函数，在 printf.c 里，目的是支持退格键，内容如下：

```
void BackSpace()
{
    char addr,dat;

    addr = LcdReadAddr();         //当前地址
    LcdWriteData(0x20);     //写入一个空字符，根据地址判断是否为前半字
    if(addr == LcdReadAddr())    //前半字
    {
        if(addr == 0x80)
            return;
        else if(addr == 0x90)
            addr = 0x87;
        else if(addr == 0x88)
            addr = 0x97;
        else if(addr == 0x98)
            addr = 0x8F;
        else
            addr = addr - 1;

        LcdWriteComm(addr);
```

```
        LcdReadData();          //空读取
        dat = LcdReadData();
        LcdWriteComm(addr);
        if(dat < 0x80)
            LcdWriteData(dat);
    }
    else
    {
        LcdWriteComm(addr);
    }
}
```

退格完成功能：仅仅地址向前退一格，详细见源程序。

printf 和 scanf 移植全部完成，欢迎大家使用；有什么不足之处，欢迎提出意见或是建议；多谢大家支持啦。

附件：程序库

MSP430 程序库<五>SPI 同步串行通信

SPI 总线系统是一种同步串行外设接口；是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。 许多的芯片都用这种协议通信：EEPROM、Flash、实时时钟、AD 转换器、数字信号处理器等：MSP430 的 USART 模块不仅能够实现异步模式（见：MSP430 程序库<二>UART 异步串口），而且支持同步串行通信（即 SPI 模式）；其 SPI 支持 3 线、4 线操作，支持主机模式和从机模式，字符长度可以 7 位或 8 位等。由于要用 AD7708 芯片完成 AD 采样，AD7708 是通过 SPI 与其它设备通信的；本程序比较简化，只完成了主机模式的初始化。

- **硬件介绍：**

SPI：SPI 是 Motorola 首先在其 MC68HCXX 系列处理器上定义的，它是一种同步的高速串行通信协议，有关 SPI 协议的详细内容，参考：SPI_互动百科。

MSP430 对 SPI 的支持：当 msp430USART 模块控制器 UxCTL 的位 SYNC 置位时，USART 模块工作于同步模式，对于 149 即工作于 SPI 模式，若是 169，USART0 可以支持 I2C，可以通过另一控制位 I2C 控制，I2C 位 0 则工作于 SPI。在 SPI 模式下，允许单片机以确定的速率发送和接收 7 位或 8 位数据。

同步通信与异步通信类似；同步通信和异步通信寄存器资源一致，具体寄存器的不同位之间的功能存在差异；具体寄存器内容参见 TI 提供的用户指南。

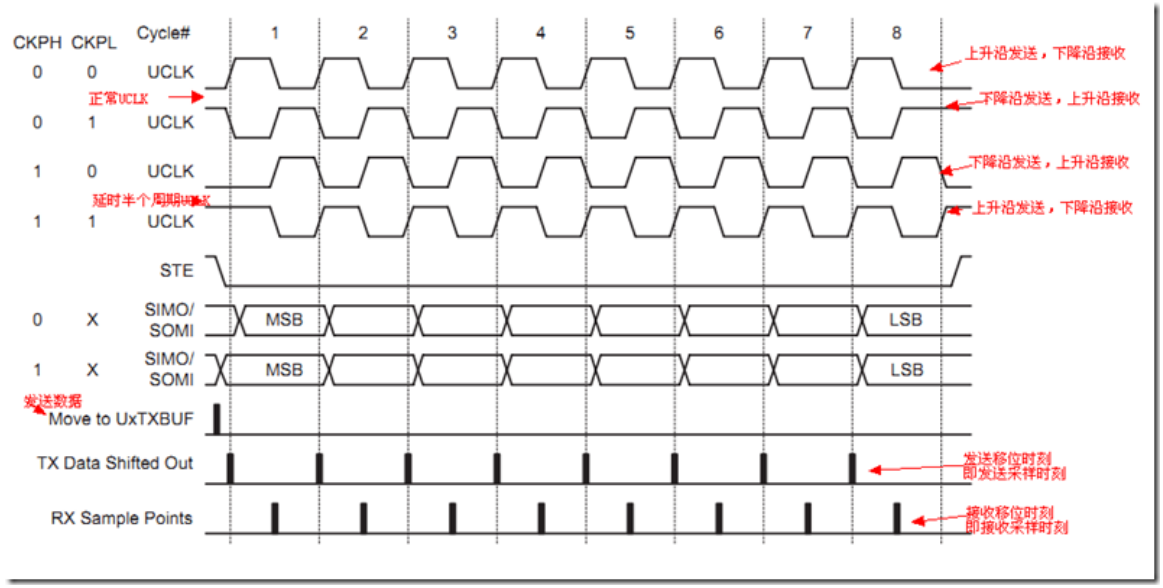
USART 模块的 SPI 操作可以是 3 线和 4 线，其信号如下：
SIMO：从进主出，主机模式下，数据输出；从机模式下，数据输入。
SOMI：从出主进，主机模式下，数据输入；从机模式下，数据输出。
UCLK：USART SPI 模式时钟，信号有主机输出，从机输入。
STE：从机模式发送接收允许控制脚，用于 4 线模式，控制多主从系统中多个从机，避免发生冲突。具体方式如下(图截自 用户指南)：

❑ STE	Slave transmit enable. Used in 4-pin mode to allow multiple masters on a single bus. Not used in 3-pin mode. 4-Pin master mode: When STE is high, SIMO and UCLK operate normally. When STE is low, SIMO and UCLK are set to the input direction. 4-pin slave mode: When STE is high, RX/TX operation of the slave is disabled and SOMI is forced to the input direction. When STE is low, RX/TX operation of the slave is enabled and SOMI operates normally.

四线主机模式：STE 为高电平，SIMO 和 UCLK 操作正常；STE 为低电平，SIMO 和 UCLK 被置为输入方向，主机控制权让出。
四线从机模式：STE 为高电平，从机的发送和接收无效，且把 SOMI 置为输入方向；STE 为低电平，发送接收正常，SOMI 也为正常输出。

USART 模块串行时钟极性和相位设置：

USART 的时钟 UCLK 的极性和相位由位于 UxTCTL 寄存器的 CKPH 和 CKPL 位控制，具体如下图：在程序中，我分别称之为，时钟模式 0、时钟模式 1、时钟模式 2、时钟模式 3。



USART 的波特率产生，SPI 不同于异步通信：异步通信由 UxBR1\UxBR0\UxMCTL 三个寄存器控制，以产生标准频率；而同步模式，主从设备用同一个时钟，不再需要产生标准时钟，故而不再用 UxMCTL 寄存器，设其值为 0.

其他的，与异步通信基本一致，这里不再细说。具体参考用户指南。

- **程序实现：**

程序和异步通信方式类似：首先是初始化函数，然后是读取数据、写入数据函数。此程序采用和我之前的 UART 程序库类似的结构，写入数据后进入低功耗等待中断，判断标志位进行写入数据和读取数据。

这里函数只实现 430 的主机模式，如需从机模式可以仿照我的程序，进行简化实现。

1. 由于，我即将使用的 SPI 设备（AD7708）不是字符型设备，这里不再实现写入字符串函数，也不再移植 printf 和 scanf 函数，如若需要可以自己添加，printf 和 scanf 的移植参考：MSP430 程序库<四>printf 和 scanf 函数移植

初始化函数：SpiMasterInit，实现主机模式的初始化工作，函数内容如下：

```
char SpiMasterInit(long baud,char dataBits,char mode,char clkMode)
{
    long int brclk;                 //波特率发生器时钟频率

    UxCTL |= SWRST;                 //初始

    //反馈选择位，为 1，发送的数被自己接收，用于测试，正常使用时注释掉
    //UxCTL |= LISTEN;

    UxCTL |= SYNC + MM;             //SPI 主机模式

    //时钟源设置
    UxTCTL &=~ (SSEL0+SSEL1);       //清除之前的时钟设置
    if(baud<=16364)                 //
    {
      UxTCTL |= SSEL0;              //ACLK，降低功耗
      brclk = 32768;                //波特率发生器时钟频率=ACLK(32768)
    }
    else
    {
      UxTCTL |= SSEL1;              //SMCLK，保证速度
      brclk = 1000000;              //波特率发生器时钟频率=SMCLK(1MHz)
    }

    //-----------------------设置波特率-----------------------
```

```
if(baud < 300||baud > 115200)    //波特率超出范围
{
    return 0;
}
//设置波特率寄存器
int fen = brclk / baud;          //分频系数
if(fen<2)return (0);             //分频系数必须大于 2
else
{
    UxBR0 = fen / 256;
    UxBR1 = fen % 256;
}

//----------------------设置数据位-----------------------
switch(dataBits)
{
    case 7:case'7': UxCTL &=~ CHAR; break;       //7 位数据
    case 8:case'8': UxCTL |= CHAR;  break;       //8 位数据
    default :       return(0);                   //参数错误
}
//----------------------设置模式-----------------------
switch(mode)
{
    case 3:case'3': UxTCTL |= STC;  USPI3ON;    break;  //三线模式
    case 4:case'4': UxTCTL &=~ STC; USPI4ON;    break;  //四线模式
    default :       return(0);                          //参数错误
}

//----------------------设置 UCLK 模式-----------------------
switch(clkMode)
{
    case 0:case'0': UxTCTL &=~ CKPH;UxTCTL &=~ CKPL;break;//模式 0
    case 1:case'1': UxTCTL &=~ CKPH;UxTCTL |= CKPL;break; //模式 1
    case 2:case'2': UxTCTL |= CKPH;UxTCTL &=~ CKPL;break; //模式 2
    case 3:case'3': UxTCTL |= CKPH;UxTCTL |= CKPL;break;  //模式 3
    default :       return(0);                          //参数错误
}

UxME |= USPIEx;              //模块使能

UCTL0 &= ~SWRST;             // Initialize USART state machine

UxIE |= URXIEx + UTXIEx;     // Enable USART0 RX interrupt
```

```
    return(1);                    //设置成功
}
```

程序注释已经比较详细，这里不再细说；如果要改为从机模式，把时钟设置和波特率设置去掉应该就可以了。

发送函数和接收函数：

```
void SpiWriteDat(char c)
{
    while (TxFlag==0) SpiLpm();  // 等待上一字节发完，并休眠
    TxFlag=0;                     //
    UxTXBUF=c;
}
char SpiReadDat()
{
    while (RxFlag==0) SpiLpm(); // 收到一字节?
    RxFlag=0;
    return(UxRXBUF);
}
```

发送和接收函数和异步通信里面的几乎一样，如果标志位为 0，则等待改变为 1，然后写入或读出；标志位在中断函数里被更改；中断函数如下：

```
#pragma vector=USARTxRX_VECTOR
__interrupt void UartRx()
{
    RxFlag=1;
    __low_power_mode_off_on_exit();
}
#pragma vector=USARTxTX_VECTOR
__interrupt void UartTx ()
{
    TxFlag=1;
    __low_power_mode_off_on_exit();
}
```

中断里面仅仅置标志位后，就退出低功耗；退出后即写入或者读取数据。

读取或写入函数调用的 SpiLpm 函数：

```
void SpiLpm()
{
    if(UxTCTL&SSEL0) LPM3;//若以 ACLK 作时钟,进入 LPM3 休眠(仅打开 ACLK)
    else     LPM0;  //若以 SMCLK 作时钟，进入 LPM0 休眠(不关闭 SMCLK)
```

}

根据不同情况进入低功耗，如果单片机其他地方不允许进入低功耗，可以更改这个函数。

程序部分就这么多了。需要的函数在头文件里面声明，方便使用。

2. 使用示例：

程序使用方式和之前的程序库相同，加入 c 文件，包含 h 文件，调用初始化函数后即可掉用程序库中的函数。

```
#include "msp430x16x.h"    //430 寄存器头文件
#include "Spi.h"           //串口通讯程序库头文件

void main()
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;

    ClkInit();
   // 主机模式,波特率25000,8位数据位,三线模式,时钟模式0(具体见spi.c)
    SpiMasterInit(25000,8,3,0);
    _EINT();


    while(1)                    //串口测试
    {
        SpiWriteDat(0X20);
        char a = SpiReadDat();
    }
}
```

这里只是一个简单的使用示例，详细的使用，将会在下一篇给出；将会使用今天的程序库，完成 SPI 的通信部分。

3. 注意事项：

SPI 是全双工通信，每次写入（发送）8 位/7 位数据的同时，430 的 SPI 主模块都会在发送后半个时钟周期读取采样的 0/1 信号，存入接收缓冲寄存器，所以，每次的写入，均有数据读取，但不一定是从设备发送回来的，这个地方在使用 430 主机模式的时候必须注意，很容易出错（我也是在调试 AD7708 的时候才注意到这个地方的）；SPI 的函数已经添加 SpiWriteData 函数，这个函数会在发送的同时返回发送完成半个时钟周期后的接收到的数据，方便使用；不建议使用前面的发送和读取函数，很容易出错；建议使用刚添加的这个函数，程序库已经更新，可以重新下载。函数 SpiWriteData：

```
char SpiWriteData(char c)
{
    SpiWriteDat(c);
    return SpiReadDat();
}
```

发送后读取即可，程序比较简单。

新的示例程序：

```
void main()
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;

    ClkInit();
   // 主机模式,波特率25000,8位数据位,三线模式,时钟模式0(具体见spi.c)
    SpiMasterInit(25000,8,3,0);
    _EINT();


    while(1)                    //串口测试
    {
        SpiWriteData(0X20);     //只写入
        char a = SpiWriteData(0xff);    //只读取
    }
}
```

详细示例，参参考下一篇：MSP430 程序库<六>通过 SPI 操作 AD7708

SPI 的通信部分完成，有什么不足，欢迎讨论。谢啦！

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<六>通过 SPI 操作 AD7708

AD7708 是 16 位的 Σ-Δ 型 AD 转换芯片，在低频应用中，AD7708 可以作为单电源供电的完整前端。AD7708 内部含有一个 PGA（可编程增益放大器），可以完成对信号的放大；PGA 范围是 2^0-2^8 八档可编程，当取参考电压 2.5v 时可以测量量程 20mv 到 2.56v 的电

压。AD7708 和 AD7718 引脚功能完全一样，只是位数 AD7718 是 24 位的，若用 7718，只需改动少部分的程序，16 位的部分改成 24 位即可。AD7708 是通过 SPI 接口通信的；程序使用前一篇实现的 SPI 程序与 AD 芯片通信，可以作为 SPI 详细的使用示例。

- **硬件介绍：**

硬件主要是 MSP430 的 SPI 接口和 AD7708 芯片的说用说明。

msp430 的 SPI 接口：支持主机模式和从机模式，且始终的极性和相位可调，在于 AD 转换芯片通信的时候，需要极性一致。有关 msp430 的 SPI 的详细介绍，参考：MSP430 程序库<五>SPI 同步串行通信。

AD7718 的外部引脚有 28 个。按性质主要分为模拟、数字两个部分。模拟部分引脚有模拟输入、参考电压输入和模拟电源三类。模拟输入引脚可以配置为 8 通道或 10 通道的伪差分输入，他们共同参考 AINCOM 端。

数字部分引脚有 SPI 接口、数据就绪、通用 I/O 口和数字电源四类。SPI 接口的 4 根标准信号线分别是片选信号 CS 、串行时钟输入 SCLK、串行数据输入 DIN 和串行数据输出 DOUT。当 AD7718 接在 SPI 总线上时是从器件，从引脚 CS 输入低电平信号使能 AD7718。数据就绪 RDY 是一个低电平有效的输出引脚。当所选通道数据寄存器中有有效数据时，输出低电平信号；数据被读出后，输出高电平。AD7718 的通用 I/O 口是 2 个一位口 P1 和 P2。它们既可配置成输入也可配置成输出，单片机通过 SPI 口读写 AD7718 片内相关寄存器实现对 P1 和 P2 的操作。它们扩展了单片机的 I/O 接口能力。

AD7718 的模拟电源和数字电源是分别供电的，都既可以采用+3V 供电，也可以采用+5V 供电。但必须一致，要么都用+3V，要么都用+5V。

AD7708 和 AD7718 是通过一组片内寄存器控制和配置的。这些寄存器的第一个是通信寄存器，它是用来控制转换器的所有操作。这些部件的所有通信必须先写通信寄存器指定要执行的下一个操作。上电或复位后，设备默认等待写通信寄存器。 STATUS 寄存器包含转换器的操作条件的有关信息。STATUS 寄存器是只读寄存器。模式寄存器用于配置转换模式，校准，斩波（chop）启用/禁用，参考电压选择，通道配置和伪差分 AINCOM 模拟输入操作时的缓冲或无缓冲。模式寄存器是一个读/写寄存器。 ADC 控制寄存器是一个读/写寄存器，用来选择活动的通道和编码输入范围和双极性/单极性操作。I/O 控制寄存器是一个读/写寄存器，用于配置了 2 个 I/O 端口的操作。滤波寄存器是一个读/写寄存器，用于编码转换器的数据更新率。 ADC 数据寄存器是一个只读寄存器，它包含在所选通道上的一个数据转换的结果。 ADC 的失调寄存器读/写寄存器包含偏移校准数据。有五个偏移寄存器，每个全差分输入通道之一。当配置为伪差分输入模式下的通道共用偏移寄存器。 ADC 增益寄存器是读/写寄存器，包含增益校准数据。有 5 个 ADC 增益寄存器，每个全差分输入通道之一。当配置为伪差分输入模式通道共享增益寄存器。该 ADC 包含工厂使用的测试寄存器，用户应不改变这些寄存器的操作条件。 ID 寄存器是一个只读寄存器，用于硅识别目的。

我用的硬件连线方式：430 的 P3.0 接 AD7708 的 CS 端，P3.1-P3.2 接对应的 AD 芯片的 SPI 口；RDY 信号没有接；所以，程序使用的是查询方式，等待 STATUS 寄存器的 RDY 位指示转换完成。

有关 AD7708 的详细信息可以参考它的 datasheet；另外我对数据手册的寄存器部分和程序流程的部分进行了翻译，如果需要，可以在本博客底部的**附件**中下载。

- **程序实现：**

首先是对 AD7708 的读写寄存器函数，AD7708 的每次操作都以写通信寄存器开始，通过这一步，指示下一步将进行什么操作；有关寄存器每一位的意义，参考附件(博客结尾)中的 AD7708-寄存器

写入寄存器：

```
void AD7708WriteRegister(char addr,long dat)
{
    SpiWriteData(addr);       //写通信寄存器，通知下个操作：写 addr 寄存器
    if(IsLong[addr])          //如果是 16 位寄存器, 7718 则 24 位若移植要改 if 内语句
    {
        SpiWriteData(dat>>8);
    }
    SpiWriteData(0xFF&dat);         //写入低位数据
}
```

寄存器地址，可以查阅 datasheet 或我翻译的那部分；IsLong 字符数组指示对应的寄存器是 8 位还是 16 位的：

```
char IsLong[16] = {0,0,0,0,1,1,1,0,0,0,0,0,1,1,0,0};
```

读取寄存器：

```
long AD7708ReadRegister(char addr)
{
    char h = 0,l = 0;             //高低字节数据
    SpiWriteData(0x40|addr);     //写通信寄存器，通知下个操作：读 addr 寄存器
    if(IsLong[addr])
    {
        h = SpiWriteData(0xFF);
    }
    l = SpiWriteData(0xFF);
    return ((unsigned int)h<<8)|l;
}
```

SPI 解释：430 是 SPI 主机模块，当发送的时候，同时，另外一个时钟沿采样接收，所以，每次发送完成后的半个周期，均可得到读出的数据；所以 SpiWriteData 函数写入的同时返回同时收到的字符。发送 0xFF 即是为提供读取即将到来的数据提供时钟，详细可以参考上一篇的注意事项部分（刚更新的）。读取结果数据：

```
long AD7708ReadResultData()
{
    while((AD7708ReadRegister(0x00)&0x80)==0); //等待转
换完成
    return AD7708ReadRegister(0x04);
}
```

等待 STATUS 的 RDY 位变高(AD 数据转换更新完成)，读取 data 寄存器的内容。

校准：校准的过程在 datasheet 中有详细的流程图；可以参考 datasheet 或者附件中的 AD7708-寄存器，这个子函数只完成一个通道的校准，通道地址有参数输入，方便调用：

```
void AD7708Cal(char channel)
{
    adccon = (adccon&0x0f)|(channel<<4);
    mode = (mode&0xf8)|0x04;                    //内部 0 校准
    AD7708WriteRegister(0x02,adccon);          //ADC 控制寄
存器，channel 通道
    AD7708WriteRegister(0x01,mode);             //模式寄存器
    while((AD7708ReadRegister(0x01)&0x07)!=0x01);    //
等待校准完成

    mode = (mode&0xf8)|0x05;              //内部 满标度校准
    AD7708WriteRegister(0x01,mode);             //模式寄存器
    while((AD7708ReadRegister(0x01)&0x07)!=0x01);    //
等待校准完成
}
```

adccon 是程序记录的前一次输入的 ADCCON 寄存器的内容，mode 是程序记录的上一次输入的 MODE 寄存器的内容，因为串口读取需要时间，为了获取更快的速度，程序记录了这两个变量，以供使用。通道地址参考 datasheet，或附件中的文档。

初始化：

```
void AD7708Init(char chop)
{
    P3DIR|=BIT0;
    P3OUT&=~BIT0;      //CS 选中
//主机模式,115200,8 位数据位,三线模式,时钟模式 1(具体见 spi.c)
    SpiMasterInit(115200,8,3,1); //时钟不是准确的 115200
    _EINT();  //开中断，spi 读写程序要需要中断

    char filter;
    adccon = 0x0f;
    if(chop == 0)
    {
        filter = 0x03; //滤波寄存器设为最小值，可以更改
```

```
        mode = 0x91; //斩波禁止，10 通道，无缓冲，空闲模式
    }
    else
    {
        filter = 0x0D; //滤波寄存器设为最小值，可以更改
        mode = 0x11; //斩波启用，10 通道，无缓冲，空闲模式
    }

    AD7708WriteRegister(0x07,0x00); //IO 寄存器，不用==
    AD7708WriteRegister(0x03,filter); //滤波寄存器
    AD7708WriteRegister(0x02,0x0F); //ADC 控制寄存器，0 通
道，单极性
    AD7708WriteRegister(0x01,mode);      //模式寄存器
    if(chop == 0)
        for(int i = 0; i<5;i++)
        {
//校准，因只有 5 个失调寄存器，多的就会覆盖之前的只校准 5 个即可
            AD7708Cal(5);
        }

    _DINT();
}
```

初始化制引入了斩波这一个参数，其他的均使用固定的参数：10 通道伪差分、单极性、无缓冲、滤波寄存器设为斩波或禁止斩波时候的最快速度，需要的话可以自行修改。SPI 初始化之后开中断，目的是向 AD 写内容以初始化 AD。初始化完成后关中断，为了让程序库的初始化后一致，但调用这个函数后，需要开中断，才能正常使用 AD 采样的其它函数。

采样启动：本程序只支持了单词采样的开始，若需要连续模式的，可以自行实现（比较容易实现：只需更改寄存器的值即可）：

```
void AD7708Start(char channel)
{
    adccon = (adccon&0x0f)|(channel<<4);
    mode = (mode&0xf8)|0x02;
    AD7708WriteRegister(0x02,adccon);
    AD7708WriteRegister(0x01,mode);
}
```

1. 根据之前一次的控制寄存器和模式寄存器的设置，更改现在需要的值，写入相应寄存器即可。到此，程序部分完成，需要扩展，可以自行添加。

2. **使用说明：**

使用时，只需加入 AD7708.c，文件包含 AD7708.h，然后就可以正常使用本程序提供的函数；具体可以参考示例工程和其中的 main.c 文件。

```
long a;
```

```
void main()
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    AD7708Init(0);  //禁止斩波 1 时启用斩波

    _EINT();  //开中断，程序需要用到 SPI 的中断；冲突，可以更改 SPI 函数

    while(1)                    //串口测试
    {
        AD7708Start(0);
        a = AD7708ReadResultData();//读取 AD 采样后的结果
        //电压计算方法：a*1.024*2.5(参考电压)/65535
        a = AD7708ReadRegister(0); //去状态值，此处函数不需要用

    }
}
```

AD7708 的程序库（简化，其他需求可以自行添加：有读写寄存器的函数之后，添加其他功能比较简单）已经完成，有什么不足之处欢迎大家讨论；谢啦。

附件：程序库 AD7708-寄存器

MSP430 程序库<七>按键

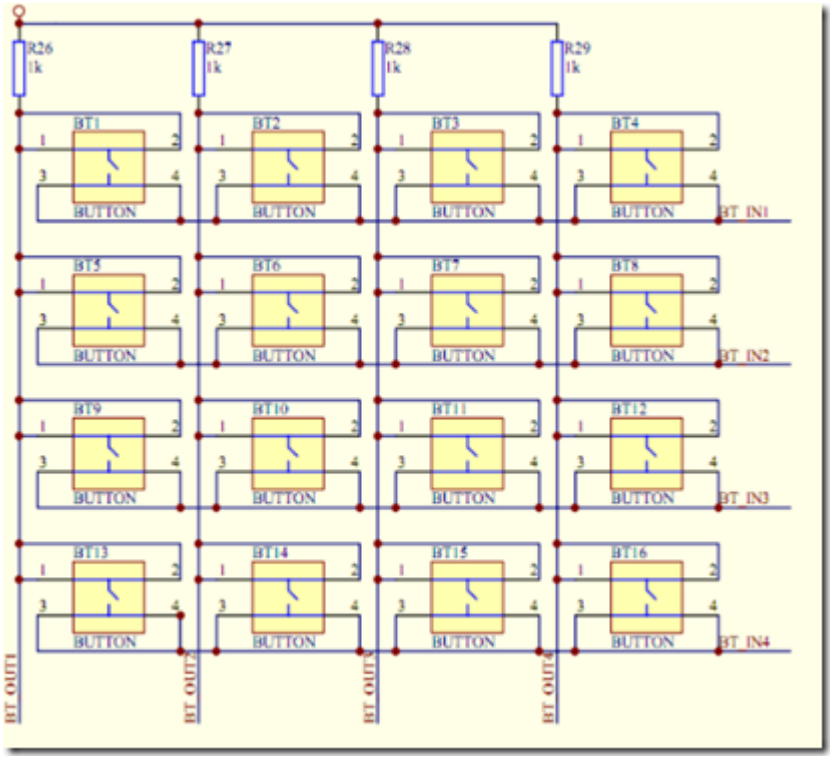
按键是单片机系统最常用的输入设备之一；几乎是只要需要交互输入，就必须有键盘。这篇博客实现了一个通用的键盘程序，只要提供一个读取键值的函数(底层键值)，程序将完成消抖、存入队列等一些列处理。同时本程序提供最常用的 4*4 矩阵键盘的程序，和 4 个按键的程序。

1. 硬件介绍：

本文主要实现了一个键盘的通用框架，可以很方便的改为不同的键盘函数，这里实现了两种按键 4 个单独按键和 4*4 行列扫描的键盘。

4 个按键的是这样的：四个按键分别一端接地，另一端接上拉电阻后输入单片机的 P1.0-P1.3 口；这样，按键按下时，单片机接到低电平，松开时单片机输入信号有上拉电阻固定为高电平。

4*4 的按键：行输入信号配有桑拉电阻，无按键时默认电平高电平；列扫描信号线直接接到按键列线；读键时，列扫描信号由单片机给出低电平信号(按列逐列扫描)，读取行信号，从而判断具体是哪个按键；电路图大概如下：



图中，IN 是键盘的列扫描线，OUT 是键盘的输出的行信号线。扫描是也可以按行扫描，这时 IN 是行扫描线，OUT 的按键输出的列信号线。我的程序是按列扫描的（行列扫描原理一样，只是行列进行了交换）。

这里，同时实现了 4*4 按键的 scanf 函数的移植，同时，加入了之前实现的液晶的 printf 函数的移植，搭建了一个可以交互输入输出的完整的一个系统；液晶的 printf 又加入了函数，实现了退格；可以在输入错误数字的时候退格重新输入。

2. **程序实现：**

先说一下程序的结构，程序实现了一个循环队列，用来存放已按下的键值，可以保存最新的四个按键，可以防止按键丢失；程序使用的是中断的方式进行按键，每 16ms(用的是看门狗的间隔中断)读一次按键，进行判断键值是否有效，有效则放入队列，等待读取。

循环队列的实现:用数组实现，为判断队满，数组的最后一个元素不用于存储键码值：

```
/**********************宏定义************************/
#define KeySize      4              //键码值队列
#define Length       KeySize+1    //队列数组元素个数
/*****************************************************/

/**********************键值队列**********************/
//可 KeySize(Length-1)个键码循环队列占用一个元素空间
char Key[Length];
```

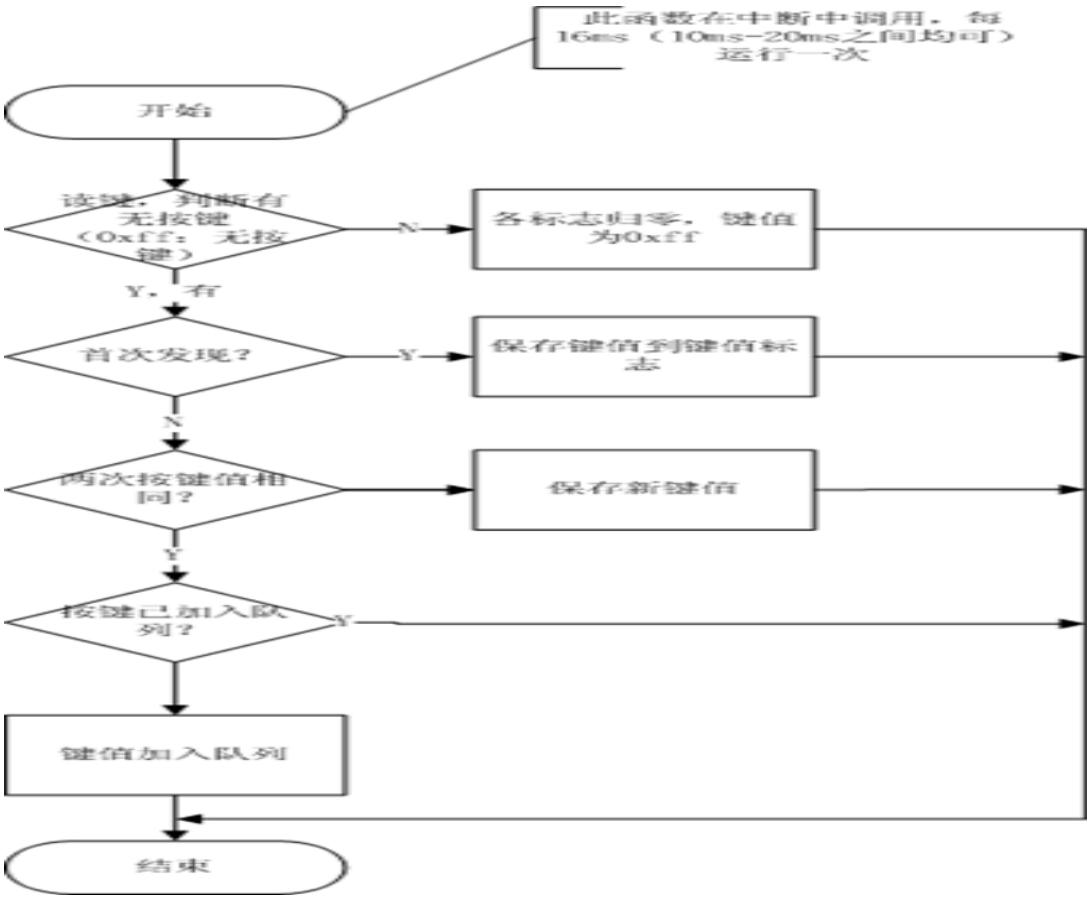
入队函数：入队时，队满则出队一个，以保存最新的四个按键。

```
void AddKeyCode(char keyCode)
{
    if((rear+1)%Length==front)        //队满
    {
        front=(front+1)%Length;       //出队一个
    }
    Key[rear] = keyCode;
    rear=(rear+1)%Length;
}
```

出队函数：出队函数即是读取按键的函数，以供其他需要的地方调用。

```
char ReadKey()
{
    char temp;
    //if(rear==front) return '\0';      //无按键
    while(rear==front);
    temp = Key[front];
    front=(front+1)%Length;
    return temp;
}
```

KeyProcess:这个函数即是键盘处理函数,需要被每 10ms-20ms 的时间调用一次的函数，在这里把它放入了看门狗定时器 16ms 的中断中;函数流程图和函数内容如下：



```
void KeyProcess()
{
    static char keyValue = 0xff;      //按键标识，键值
    static char addedFlag = 0;        //加入队列标志
    char keyVal = GetKey();
    if(keyVal==0xff)                  //无按键
    {
        keyValue = 0xff;
        addedFlag = 0;
        return;
    }
    if(keyValue==0xff)                //之前状态无按键
    {
        keyValue = keyVal;
        return;
    }
    if(keyValue!=keyVal)              //和前次按键不同
    {
        keyValue = keyVal;            //保存新按键值
        return;
    }
    if(addedFlag==1)                  //已加入队列
    {
        return;
    }
    addedFlag = 1;
    AddKeyCode(KeyCode[keyVal]);
}
```

这个函数完成按键的判断，并和上次的比较，从而判断是否是有效按键，再根据是否已经入队保存，去判断是否要保存，入队列保存按键。

这个函数需要每 10ms-20ms 中断运行一次：

```
#pragma vector=WDT_VECTOR
__interrupt void WDT_ISR()
{
    KeyProcess();
}
```

这是 430 看门狗的间隔定时中断，设置的是每 16ms 中断一次：

```
WDTCTL=WDT_ADLY_16;    //看门狗内部定时器模式 16ms
    IE1 |= WDTIE;          //允许看门狗中断
```

KeyProcess 里调用了 GetKey 函数，这个函数需要用户提供，以满足特殊的按键需求，这里提供了两个实例：4 个按键和 4*4 矩阵键盘。

4 个按键的 getkey 函数：

```
char GetKey()
{
    if((P1IN&0X0F)==0x0E)
    {
        return 0;
    }
    if((P1IN&0X0F)==0x0D)
    {
        return 1;
    }
    if((P1IN&0X0F)==0x0B)
    {
        return 2;
    }
    if((P1IN&0X0F)==0x07)
    {
        return 3;
    }
    return 0xff;
}
```

这里根据每个按键，输出按键原始键值，没有按键则输出 0xff；当自己提供 getkey 函数时，也需要这样，无按键时返回 0xff 把对应原始键值翻译成所需键码，用数组 KeyCode：

```
char KeyCode[] = "0123";     /*4 个按键时*/
```

这里把它转化成 ASCII 码输出，需要的话可以自行更改。

4*4 矩阵键盘：getkey：

```
char GetKey()
{
    P1DIR |= 0XF0;                      //高四位输出
    for(int i=0;i<4;i++)
    {
        P1OUT = 0XEF << i;
        for(int j=0;j<4;j++)
        {
            if((P1IN&(0x01<<j))==0)
            {
                return (i+4*j);
            }
        }
    }
    return 0xff;
```

```
}
```

这里是按列扫描，可以随意改成其他扫描方式，只要获取原始键值即可，无按键是须返回 0xff。KeyCode，翻译成 ASCII 码：

```
char KeyCode[] = "0123456789ABCDEF"
```

到这里，正常的键盘程序结束，调用时只需加入 Key.c，包含 Key.h 即可使用，先调用 KeyInit 后，就可以正常的读键了。这里不再细说。

scanf 移植：scanf 移植时，需要的是 ASCII 码字符型设备，利用 ASCII 码输入数据还必须要有回车键，只有这样，才能用 scanf 输入数据，这里为了输入数据错误时，可以退格修改，按键还有一个退格键。

键盘结构：

1	2	3	退格
4	5	6	保留
7	8	9	保留
保留	0	保留	回车

保留键用字符'\0',回车'\n'退格'\b'

所以：KeyCode：

```
char KeyCode[] = "123\b456\000789\0\0000\0\r"; /*
4*4,scanf 移植*/
```

在字符串里，\0 后面是数字时，必须用'\000'否则，c 语言编译器认为\0 和后面的数字组合为一个字符。

scanf 的移植，需要实现 getchar 函数，这里和之前的 getchar 函数类似，把它放到了 Getchar.c 文件里，内容如下：

```
#include <stdio.h>
#include "Key.h"

#define LINE_LENGTH 20            //行缓冲区大小，决定每行最多输入的字符数

/*标准终端设备中，特殊 ASCII 码定义，请勿修改*/
#define InBACKSP 0x08              //ASCII  <--  (退格键)
#define InDELETE 0x7F              //ASCII <DEL> (DEL 键)
#define InEOL '\r'                //ASCII <CR>  (回车键)
#define InLF '\n'                  //ASCII <LF>  (回车)
#define InSKIP '\3'               //ASCII control-C
#define InEOF '\x1A'               //ASCII control-Z

#define OutDELETE "\x8 \x8"       //VT100 backspace and clear
#define OutSKIP "^C\n"            //^C and new line
```

```
#define OutEOF "^Z"              //^Z and return EOF

int getchar()
{
    static char inBuffer[LINE_LENGTH + 2];       //Where to put chars
    static char ptr;                            //Pointer in buffer
    char c;

    while(1)
    {
        if(inBuffer[ptr])                       //如果缓冲区有字符
            return (inBuffer[ptr++]);           //则逐个返回字符
        ptr = 0;                                //直到发送完毕，缓冲区指针归零
        while(1)                                //缓冲区没有字符,则等待字符输入
        {
            c = ReadKey();                      //等待接收一个字符==移植时关键
            if(c == InEOF && !ptr)              //==EOF==  Ctrl+Z
            {                                   //只有在未入其他字符时才有效
                printf(OutEOF);                 //终端显示 EOF 符
                return EOF;                     //返回 EOF（-1）
            }
            if(c==InDELETE || c==InBACKSP)      //==退格或删除键==
            {
                if(ptr)                         //缓冲区有值
                {
                    ptr--;                      //从缓冲区移除一个字符
                    printf(OutDELETE);          //同时显示也删掉一个字符
                }
            }
            else if(c == InSKIP)                //==取消键 Ctrl+C ==
            {
                printf(OutSKIP);                //终端显示跳至下一行
                ptr = LINE_LENGTH + 1;          //==0 结束符==
                break;
            }
            else if(c == InEOL||c == InLF)      //== '\r' 回车=='\n'回车
            {
                putchar(inBuffer[ptr++] = '\n');//终端换行
                inBuffer[ptr] = 0;              //末尾添加结束符（NULL）
                ptr = 0;                        //指针清空
                break;
            }
            else if(ptr < LINE_LENGTH)          //== 正常字符 ==
            {
```

```
                if(c >= ' ')                    //删除 0x20 以下字符
                {
                    //存入缓冲区
                    putchar(inBuffer[ptr++] = c);
                }
            }
            else                                //缓冲区已满
            {
                putchar('\7');                  //== 0x07 蜂鸣符，PC 回响一声
            }
        }
    }
}
```

这里是支持退格等键的详细函数。

如果不需要支持退格，可以简化为：

```
int getchar()
{
    return ReadKey();
}
```

- 要实现 scanf 调用，还需要设置，详细设置参考：MSP430 程序库<四>printf 和 scanf 函数移植；需要把库设置为 CLIB；在 Option-general option-library configuration 里面。

这样，键盘的 scanf 移植完成，需要使用时，只需加入对 stdio.h 文件的包含，然后完成键盘的初始化即可。

- **使用示例：**

这里，示例实现的是键盘和液晶的简单交互；键盘输入数据，液晶正常显示；就像 c 语言调试时键盘和屏幕一样；当然没有那个丰富啦。

液晶的部分，用的是原来实现的程序，在这里，为了支持输入错误时退格，对原来的 printf 函数加入了退格支持。

项目中接入液晶的 c 程序文件和 printf 的程序文件（Lcd12864.c、Printf.c），加入 Lcd12864.h 的文件包含；初始化液晶后，就可用 printf 向液晶输出要显示的内容了。

键盘：加入 Key.c，包含 Key.h，加入 Getchar.c，程序中初始化键盘；然后设置所用的 lib 为 CLIB。之后就可以用键盘和液晶完成和 430 单片机简单的交互了。

详细参考示例工程和 main.c。

```
#include <msp430x16x.h>
#include <stdio.h>
#include "Lcd12864.h"
#include "Key.h"

long a;
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    LcdInit();
    KeyInit();
    _EINT();
    while(1)
    {
        printf("请输入数字：");
        scanf("%ld",&a);
        printf("输入的数字是：%ld",a);
        _NOP();
    }
}
```

这样，就可以用键盘向单片机输入数据，同时利用液晶可以很容易的知道数据输入的是否有问题。

键盘的程序库就到这里，有什么不足，欢迎讨论。

附件：程序库

MSP430 程序库<八>DAC12 的使用

MSP430 带有的 DAC12 模块，可以将运算处理的结果转换为模拟量，以便操作被控制对象的工作过程。DA 是在控制操作过程中常用的器件之一;MSP430 有些系列中含有 DAC12 模块，给需要使用 DA 的方案提供了许多方便。这里实现较为简单的 DAC 的驱动，方便以后使用。

1. **硬件介绍：**

MSP430x14x 系列不含 DAC12 模块，所以本文的实现只能用于 16 系列等含有 DAC12 模块的单片机中。

MSP430F169 单片机的 DAC12 模块有 2 个 DAC 通道，这两个通道在操作上是完全平等的。并且可以用 DAC12GRP 控制位将多个 DAC12 通道组合起来，实现同步更新，硬件还能确保同步更新独立于任何中断或者 NMI 事件。

这个 DAC12 模块有以下特点：8 位或 12 位分辨率可调、可编程时间对能量的损耗、可选内部或外部参考源、支持二进制原码和补码输入、具有自校验功能、可以多路 DAC 同步更新、还可以用 DMA 等。

这里实现的是较为简化的版本，需要可以自己添加或改写功能，如：初始化函数内部调用自校验的函数，可以在每一次初始化时候均自校验。

DAC12 每个模块只有两个寄存器：控制寄存和数据寄存器，控制寄存器用来初始化和设置模块的使用，数据寄存器用来存放要输出的电压数字量。169 的 DAC 的寄存器如下：

DAC12_0 控制寄存器　DAC12_0CTL
DAC12_0 数据寄存器　DAC12_0DAT
DAC12_1 控制寄存器　DAC12_1CTL
DAC12_1 数据寄存器　DAC12_1DAT

控制寄存器每一位的功能如下：

DAC12REFx：选择 DAC12 的参考源
0，1 Vref+
2，3 Veref+
DAC12RES：选择 DAC12 分辨率
0 12 位分辨率
1 8 分辨率
DAC12LSELx：锁存器触发源选择
当 DAC12 锁存器得到触发之后，能够将锁存器中的数据传送到 DAC12 的内核。
当 DAC12LSELx=0 的时候，DAC 数据更新不受 DAC12ENC 的影响。
0 DAC12_XDAT 执行写操作将触发（不考虑 DAC12ENC 的状态）
1 DAC12_XDAT 执行写操作将触发（考虑 DAC12ENC 的状态）
2 Timer_A3.OUT1 的上升沿
3 Timer_B7.OUT2 的上升沿
DAC12CALON：DAC12 校验操作控制
置位后启动校验操作，校验完成后自动被复位。校验操作可以校正偏移误差。
0 没有启动校验操作
1 启动校验操作
DAC12IR：DAC12 输入范围
设定输入参考电压和输出的关系
0 DAC12 的满量程为参考电压的 3 倍（不操作 AVcc）
1 DAC12 的满量程为参考电压
DAC12AMPx：DAC12 运算放大器设置
0 输入缓冲器关闭，输出缓冲器关闭，高阻
1 输入缓冲器关闭，输出缓冲器关闭，0V

2 输入缓冲器低速低电流，输出缓冲器低速低电流
3 输入缓冲器低速低电流，输出缓冲器中速中电流
4 输入缓冲器低速低电流，输出缓冲器高速高电流
5 输入缓冲器中速中电流，输出缓冲器中速中电流
6 输入缓冲器中速中电流，输出缓冲器高速高电流
7 输入缓冲器高速高电流，输出缓冲器高速高电流
DAC12DF：DAC12 的数据格式
0 二进制
1 二进制补码
DAC12IE：DAC12 的中断允许
0 禁止中断
1 允许中断
DAC12IFG：DAC12 的中断标志位
0 没有中断请求
1 有中断请求
DAC12ENC：DAC12 转换控制位
DAC12LSEL>0 的时候，DAC12ENC 才有效。
0 DAC12 停止
1 DAC12 转换
DAC12GRP：DAC12 组合控制位
0 没有组合
1 组合

详细的有关 DAC12 的资料可以参考 TI 提供的用户指南。

2. **程序实现：**

DAC12 模块的程序比较简单，因为每组只有一个寄存器用来控制；本程序实现的功能如下：DAC 模块初始化，完成两个 DAC 模块的初始化，可以根据参数判断要是、初始化的是哪个模块或两个都初始化，或是两个一组同时更新；用参数传递 DAC12AMPx 的值，方便设置，程序中注释很详细，如果不理解，可以直接设 AMPx 为 5 或 0x05；校准函数，完成 DAC12 模块的自校准，也是通过参数传递要校准的模块；电压输出函数，同样这个也是用参数传递要输出的模块。

初始化：

```
/*****************************************
* 函数名称：DAC12Init
* 功    能：DAC12 用到的相关资源初始化
* 参    数：
*           module 模块   0:使用模块 DAC12_0
*                         1:使用模块 DAC12_1
*                         2:使用模块 DAC12_0/1
*                         3:使用模块 DAC12_0/1 共同更新
*           DAC12AMPx：DAC 运算放大器设置：
*               0 输入缓冲器关闭，输出缓冲器关闭，高阻
*               1 输入缓冲器关闭，输出缓冲器关闭，0V
```

```
*                2 输入缓冲器低速/电流，输出缓冲器低速/电流
*                3 输入缓冲器低速/电流，输出缓冲器中速/电流
*                4 输入缓冲器低速/电流，输出缓冲器高速/电流
*                5 输入缓冲器中速/电流，输出缓冲器中速/电流
*                6 输入缓冲器中速/电流，输出缓冲器高速/电流
*                7 输入缓冲器高速/电流，输出缓冲器高速/电流
* 返 回 值：char，设置成功返回 1，参数错误返回 0
* 说    明：其他默认为：12 位方案、写入即更新输出，module 模
*           块为 3 时，两个都写入更新；DAC12 的满量程为参考电
*           压；内部 2.5v 参考电压：需要 AD 设置参考源打开 2.5.
*********************************************/
char DAC12Init(char module,char DAC12AMPx)
{
    if(DAC12AMPx>7)
    {
        return(0);
    }
    //---------------设置模块---------------
    switch(module)
    {
        case 0:case'0': DAC12_0Init(DAC12AMPx); break;
//模块 0
        case 1:case'1': DAC12_1Init(DAC12AMPx); break;
//模块 1
        case 2:case'2':
            DAC12_0Init(DAC12AMPx);
            DAC12_1Init(DAC12AMPx);
            break;                  //模块 0、1
        case 3:case'3':
            DAC12_0Init(DAC12AMPx);
            DAC12_1Init(DAC12AMPx);
            DAC12_0CTL |= DAC12GRP;
            break;  //无校验
        default :          return(0);        //参数错误
    }
    return (1);
}
```

这里参数无效返回 0，设置完成返回 1，不过要注意的是在使用 DAC 之前，必须开启内部参考源（在 ADC 模块里面，具体可以参考使用示例）。

DAC12_0Init 和 DAC12_1Init 函数内容一样，只不过控制寄存器分别是 DAC12_0CTL 和 DAC12_0CTL，这里只给出 DAC12_0Init 的函数，另一个参考源程序：

```
void DAC12_0Init(char DAC12AMPx)
{
    // Internal ref gain 1
    DAC12_0CTL = DAC12SREF_0 + DAC12IR;
    DAC12_0CTL |= DAC12LSEL_1 + (DAC12AMPx << 5);
    DAC12_0CTL |= DAC12ENC;
}
```

这个函数仅仅完成控制寄存器的设置。选内部参考源，输出满量程是参考电压的 1 倍，更新方式：写入即更新，如果 group 设置，则两个都写入才更新。

校准函数：完成 DAC12 模块自校准，

```
void DAC12Cal(char module)
{
    switch(module)
    {
        case 0:case'0':
            DAC12_0CTL |= DAC12CALON;                   // 启动效验 DAC
            while((DAC12_0CTL & DAC12CALON) != 0);   // 等待效验完成
            break;                                      //模块 0
        case 1:case'1':
            DAC12_1CTL |= DAC12CALON;                   // 启动效验 DAC
            while((DAC12_1CTL & DAC12CALON) != 0);   // 等待效验完成
            break;                                      //模块 1
        case 2:case'2':
        case 3:case'3':
            DAC12_0CTL |= DAC12CALON;                   // 启动效验 DAC
            while((DAC12_0CTL & DAC12CALON) != 0);   // 等待效验完成
            DAC12_1CTL |= DAC12CALON;                   // 启动效验 DAC
            while((DAC12_1CTL & DAC12CALON) != 0);   // 等待效验完成
            break;                                      //模块 0、1
        default :       return;                     //参数错误
    }
}
```

参数含义和前初始化的函数相同，为了使用函数时一致。

输出函数：

```
void DAC12Out(char module,int out)
{
    switch(module)
    {
```

```
        case 0:case'0': DAC12_0DAT=out; break;        //模块 0
        case 1:case'1': DAC12_1DAT=out; break;        //模块 1
        case 2:case'2':
        case 3:case'3': DAC12_0DAT=out;DAC12_1DAT=out;break;//模块 0、1
        default :       return;             //参数错误
    }
}
```

1. 输出函数的参数也和初始化的 module 参数含义相同，这个函数比较简单，只是按照要输出的值赋给 DAT 寄存器。

   DAC12 的程序库就这么多，DAC12 还可以严格按时间更新数据，以输出一定频率的波形，可以设置为 TA out1 上升沿更新数据，或 TB out2 上升沿更新。另外还可以和 DMA 共同使用，完成更复杂的功能；这里均没有实现，需要的话可以根据寄存器功能来实现。

   程序部分就到这了。

2. **使用示例：**

   这里的使用方式依然和之前的程序一样，加入 C 文件，包含 H 文件即可，另外，使用本程序，必须开启内部 AD 参考源，为 DAC12 模块提供参考电压。

```
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    DAC12Init(3,5);                    //初始化
    DAC12Cal(2);                       //校准
    ADC12CTL0 = REF2_5V + REFON;      //开启内部参考源
2.5v 必须有；以供 DA 使用
    DAC12Out(2,0x666);
}
```

ADC12CTL0 = REF2_5V + REFON;这句即是开启参考电压 2.5v 以供 DA 使用。

DAC12 模块的程序库就到这里了，有什么不对的地方欢迎拍砖。

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<九>数码管显示

数码管也是单片机系统最常用的输出设备之一(还有液晶、发光二极管等)。七段(这里用的是 8 段，有小数点)数码管可以完成显示 0-9 数字和一部分的英文字符如：A、b。本文实现的程序完成显示数字和可显示的英文字符；同时完成数码管显示的 printf 函数的移植，以支持 printf 的格式化字符等好用的特点（我用的数码管 8 个排为一排，方便数字等的显示）。

- **硬件介绍：**

这里所用到的硬件资源包括 8 个数码管、和 msp430 单片机的两个 8 位 IO 口(这里用的是 P3 和 P5 口，如有改变，可以通过宏定义更改)。

数码管是 8 个共阴的数码管，a-h 8 段通过一个 200Ω 的电阻接到 430 单片机的 P5 口。共阴端是由单片机的 P3 口控制，单片机的一位 IO 通过一个三极管接到数码管的共阴端，以完成位选。

单片机的 P3 口时数码管的位选口，某位为高则选中；P5 口时段选口；要数码管显示时，通过 P3 位选，选中某个数码管亮，P5 段选选择 8 段（a-h）中的那些亮，从而控制某一位显示数字或字符。

要同时显示多个数码管，就要动态扫描；动态扫描时，本程序选用的是由看门狗的中断扫描显示：每 1.9ms 显示其中的一位，动态扫描显示每一位，从而让数码管看起来是同时亮的。

- **程序实现：**

数码管显示首先要有一个数码管显示的断码表（完成数字和字符到数码管段值的表），程序中采用了《MSP430 系列单片机系统工程设计与实践》这本书推荐的方式实现的这个数码表：先用宏定义定义每段对应的单片机要输出的段值，然后再实现是个表，当硬件改变时，只需更改前面的每段的段值定义即可，改动的地方少了很多，代码如下：

```
/*宏定义，数码管 a-h 各段对应的比特，更换硬件只用改动以下 8 行*/
#define a       0x01                                //   AAAA
#define b       0x02                                // F      B
#define c       0x04                                // F      B
#define d       0x08                                //   GGGG
#define e       0x10                                // E      C
#define f       0x20                                // E      C
#define g       0x40                                //   DDDD   HH
#define h       0x80          //小数点
/*用宏定义自动生成段码表，很好的写法，值得学习*/
/*更换硬件无需重写段码表*/
const char Tab[] = {
    a + b + c + d + e + f,                  // Displays "0"
    b + c,                                  // Displays "1"
    a + b + d + e + g,                      // Displays "2"
    a + b + c + d + g,                      // Displays "3"
    b + c + f + g,                          // Displays "4"
```

```
    a + c + d + f +g,                   // Displays "5"
    a + c + d + e + f + g,              // Displays "6"
    a + b + c,                          // Displays "7"
    a + b + c + d + e + f + g,          // Displays "8"
    a + b + c + d + f + g,              // Displays "9"
    a + b + c + e + f + g,              // Displays "A"
    c + d + e + f + g,                  // Displays "B"
    a + d + e + f,                      // Displays "C"
    b + c + d + e + g,                  // Displays "D"
    a + d + e + f + g,                  // Displays "E"
    a + e + f + g,                      // Displays "F"
    a + c + d + e + f,                  // Displays "G"
    b + c + e + f + g,                  // Displays "H"
    e + f,                              // Displays "I"
    b + c + d + e,                      // Displays "J"
    b + d + e + f + g,                  // Displays "K"
    d + e + f,                          // Displays "L"
    a + c + e + g,                      // Displays "M"
    a + b + c + e + f,                  // Displays "N"
    c + e + g,                          // Displays "n"
    c + d + e + g,                      // Displays "o"
    a + b + c + d + e + f,              // Displays "O"
    a + b + e + f + g,                  // Displays "P"
    a + b + c + f + g,                  // Displays "Q"
    e + g,                              // Displays "r"
    a + c + d + f +g,                   // Displays "S"
    d + e + f + g,                      // Displays "t"
    a + e + f ,                         // Displays "T"
    b + c + d + e + f,                  // Displays "U"
    c + d + e,                          // Displays "v"
    b + d + f + g,                      // Displays "W"
    b + c + d + f + g,                  // Displays "Y"
    a + b + d + e + g,                  // Displays "Z"
    g,                                  // Displays "-"
    h,                                  // Displays "."
    0                                   // Displays " "
};
#undef a
#undef b
#undef c
#undef d
#undef e
#undef f
#undef g
```

0-9 的位置对应显示 0-9，之后的是 A 开始往后显示，为了方便访问这个表格，定义了 AA 等一系列的常量，方便访问这个表。

```
#define AA   10
#define BB   AA+1
#define CC   BB+1
#define DD   CC+1
#define EE   DD+1
#define FF   EE+1
#define GG   FF+1
#define HH   GG+1
#define II   HH+1
#define JJ   II+1
#define KK   JJ+1
#define LL   KK+1
#define mm   LL+1
#define NN   mm+1
#define nn   NN+1
#define oo   nn+1
#define OO   oo+1
#define PP   OO+1
#define QQ   PP+1
#define rr   QQ+1
#define SS   rr+1
#define tt   SS+1
#define TT   tt+1
#define UU   TT+1
#define VV   UU+1
#define WW   VV+1
#define YY   WW+1
#define ZZ   YY+1
#define NEG ZZ+1     /*   -   */  //负号
#define DOT NEG+1    /*   .   */  //小数点
#define SP   DOT+1   /* 空白 */  //空格
```

A 从 10 开始访问这个表格，如果要显示 A 只需这样用 Tab[AA]，即可得到需要的段值，AA-空格的宏定义放在 H 文件里，方便其他文件访问（当要调用显示函数的时候需要 AA 等宏定义）。为什么是 AA 而不是 A 呢？主要原因是单字母的有几个已经在单片机 430 的头文件里定义了，为了访问的时候一致，就都用两个字母的了。

为了动态扫描，这里定义了一个全局数组(数码管的程序可以访问)Nixie[8]在这个里面的 8 个 char 对应 8 个数码管要显示的段值。初始值是 8 个数码管都不显示：

```
char Nixie[8] = "\0\0\0\0\0\0\0\0";      //初始状态 不显示
```

动态扫描时，函数每 1.9ms(设的看门狗定时中断)调用一次显示函数，每次显示一位(为了让中断占用更少的时间，这样中断里只需赋值即可)。函数如下：

```
void Display()
{
    static char i = 0;        //记录扫描显示到哪位
    CTRL_OUT = 1<<i;
    DATA_OUT = Nixie[i];
    i++;
    if(i>7)
        i = 0;
}
```

这个函数供中断调用，i 用来保存要显示哪一位。CTRL_OUT 、DATA_OUT 是宏定义的位选和段选口。中断程序如下：

```
#pragma vector=WDT_VECTOR
__interrupt void WDT_ISR()
{
    Display();
}
```

中断只调用了一个函数，这样很方便换其他中断来定时。

中断是必须初始设置的，还有 IO 口，要设为输出方向 ，初始化函数完成数码管用到的单片机资源的初始工作：

```
void NixiettubeInit()
{
    WDTCTL = WDT_ADLY_1_9;  //看门狗内部定时器模式
16ms
    IE1 |= WDTIE;              //允许看门狗中断
    CTRL_DIR_OUT;
    DATA_DIR_OUT;
}
```

首先，设置中断并允许中断；然后设置位选和段选所用的端口为输出方向。CTRL_DIR_OUT; DATA_DIR_OUT; 和刚才用到的两个 OUT 的宏定义如下：

```
#define DATA_DIR_OUT     P5DIR|=0XFF
#define CTRL_DIR_OUT     P3DIR|=0XFF
#define DATA_OUT         P5OUT
#define CTRL_OUT         P3OUT
```

这样处理之后，要显示数字就很简单了:只需把要显示的数字或字符的段码值放入 Nixie[8] 数组对应的位置即可，如显示韩输入下：

```
void NixiettubeDisplayChar(char ch,char addr)
{
    if(ch == DOT)        //小数点,不需单独占一位
    {
        Nixie[addr] |= Tab[ch];
```

```
    }
    else
    {
        Nixie[addr] = Tab[ch];
    }
}
```

如果是小数点，放入对应位置的 h 段即可，其他直接覆盖。

插入字符函数：在最右端插入数字或字符.

```
void NixiettubeInsertChar(char ch)
{
    if(ch == DOT)         ////小数点,不需单独占一位
    {
        Nixie[0] |= Tab[ch];
        return;
    }
    for(int i = 7;i > 0;i--)
        Nixie[i] = Nixie[i - 1];       //已显示字符左移一位
    Nixie[0] = Tab[ch];
}
```

这个也是先判断小数点，小数点直接放到 h 段，其他的，则要已显示的左移再覆盖最右一位，源程序的注释很详细，可具体才、可以下载附件的程序库。

数码管清除函数，这个函数把数码管全部显示去掉，即把缓存数组内每项都置为 0：

```
void NixiettubeClear()
{
    for(int i = 0;i < 8;i++)
        Nixie[i] = Tab[SP];       //显示空格
}
```

程序比较简单，这里就不多解释了。

数码管的程序就这么多了，所有函数都列出来了。下面开始介绍 printf 的移植，具体过程不再详细说了。这里主要介绍所需程序。

单片机 printf 使用需要用户提供底层驱动-putchar 函数，printf 完成格式化等一系列活动后调用 putchar 输出字符流。只要实现 putchar，包含 stdio.h 文件，就可以使用 printf 函数。移植的数码管的 putchar 函数如下：

```
#include <stdio.h>
#include "ctype.h"          /*isdigit 函数需要该头文件*/
#include "Nixietube.h"

int putchar(int ch)
{
    //'\f'表示走纸翻页，相当于清除显示
```

```
    if(ch=='\n'||ch=='\r')
        NixiettubeClear();

    //数字和对应 ASCII 字母之间差 0x30    '1'=0x31 '2'=0x32...
    //isdigit 也是 C 语言标准函数
    if(isdigit(ch))
        NixiettubeInsertChar(ch-0x30);  //若字符是数字则显示
数字
    else                //否则，不是数字，是字母
        {
        switch(ch)     //根据字母选择程序分支
        {
            case 'A': case 'a':
NixiettubeInsertChar(AA);break; //字符 A
            case 'B': case 'b':
NixiettubeInsertChar(BB);break; //字符 B
            case 'C': case 'c':
NixiettubeInsertChar(CC);break; //...
            case 'D': case 'd':
NixiettubeInsertChar(DD);break;
            case 'E': case 'e':
NixiettubeInsertChar(EE);break;
            case 'F': case 'f': NixiettubeInsertChar(FF);break;
            case 'G': case 'g':
NixiettubeInsertChar(GG);break;
            case 'H': case 'h':
NixiettubeInsertChar(HH);break;
            case 'I': case 'i': NixiettubeInsertChar(II);break;
            case 'J': case 'j': NixiettubeInsertChar(JJ);break;
            case 'K': case 'k':
NixiettubeInsertChar(KK);break;
            case 'L': case 'l': NixiettubeInsertChar(LL);break;
            case 'M': case 'm':
NixiettubeInsertChar(mm);break;
            case 'N':
NixiettubeInsertChar(NN);break;
            case 'n':
NixiettubeInsertChar(nn);break;
            case 'O':
NixiettubeInsertChar(OO);break;
            case 'o':
NixiettubeInsertChar(oo);break;
            case 'P': case 'p':
NixiettubeInsertChar(PP);break;
```

```
            case 'Q': case 'q':
NixiettubeInsertChar(QQ);break;
            case 'R': case 'r': NixiettubeInsertChar(rr);break;
            case 'S': case 's':
NixiettubeInsertChar(SS);break;
            case 'T': case 't': NixiettubeInsertChar(tt);break;
            case 'U': case 'v':
NixiettubeInsertChar(UU);break;
            case 'V': case 'u':
NixiettubeInsertChar(VV);break;
            case 'W': case 'w':
NixiettubeInsertChar(WW);break;
            case 'Y': case 'y':
NixiettubeInsertChar(YY);break; //...
            case 'Z': case 'z':
NixiettubeInsertChar(ZZ);break; //字符 Z
            case '-':
NixiettubeInsertChar(NEG);break;//字符-
            case '.':
NixiettubeInsertChar(DOT);break;//小数点，直接显示在右下角
            case ' ':
NixiettubeInsertChar(SP);break; //空格
            default :
NixiettubeInsertChar(SP);break;//显示不出来的字母用空格替代
        }
    }
    return(ch);  //返回显示的字符(putchar 函数标准格式要求返
回显示字符)
}
```

头文件必须包含 stdio.h，这样告诉编译器 printf 调用时，用这里的 putchar 函数。然后判断字符，分类进行显示，不能显示的空一格。

数码管的程序就完成了，如果需要可以自己添加改写函数，如：当和键盘共同使用时，如果键盘移植了 scanf 函数，并且支持退格；可以改写函数-让数码管的 putchar 支持退格操作。或者用的是我的键盘程序，需要 10 多 ms 调用一次键盘处理函数，这样可以和这个数码管扫描公用一个中断：

```
void Display()
{
    static char i = 0;        //记录扫描显示到哪位
    CTRL_OUT = 1<<i;
    DATA_OUT = Nixie[i];
    i++;
    if(i>7)
```

```
    {
        i = 0;
        KeyProcess();
    }
}
```

这样改写，然后把键盘的中断去掉(别忘了 key.h 包含和加入 KeyProcess(); 的声明；如果程序中有两个指向同一个中断时，会编译错误)；这样就可以键盘、和数码管共同使用了。

1. **使用示例**：

使用方法还是和之前一样，工程中加入 Nixietube.c 文件，然后在要调用的地方加入 Nixietube.h 的包含；如 puchr 函数，和示例工程的 main.c

main.c 调用的方式如下：

```
#include <msp430x16x.h>
#include <stdio.h>
#include "Nixietube.h"

void ClkInit()
{
    char i;
    BCSCTL1 &= ~XT2OFF;                 //打开 XT2 振荡器
    IFG1&=~OFIFG;                       //清除振荡错误标志
    while((IFG1&OFIFG)!=0)
    {
        for(i=0;i<0xff;i++);
        IFG1&=~OFIFG;                   //清除振荡错误标志
    }
    BCSCTL2 |= SELM_2+SELS+DIVS_3;   //MCLK 为 8MHz，
SMCLK 为 1MHz
}

void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    NixiettubeInit();
    _EINT();
    //while(1)
    {
        NixiettubeDisplayChar(AA,5);
        NixiettubeDisplayChar(DOT,5);
        NixiettubeInsertChar(2);
```

```
        NixiettubeInsertChar(DOT);
        NixiettubeInsertChar(2);
        printf("%1.2f",1.2);
    }
}
```

包含 msp430 的头文件，以便使用 430 单片机的先关资源；加入 stdio.h 以使用 printf 函数；加入 Nixietube.h 使用数码管的相关程序。

还要注意，为了数码管正常显示，必须打开总中断，以使数码管动态扫描显示。另外，本程序单步调试看不到数码管正常显示，因为没有扫描。只有全速运行才可以看到数码管的显示情况。

数码管的程序就到这里，不足之处，欢迎讨论提出建议。

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<十>ADC12 模块

msp430 内部含有 ADC12 模块，可以完成 12 位的模数转换，当对精度或其他指标要求不高时，可以选用 430 单片机内部的 ADC12 完成模数转换工作。这里主要实现了一个比较通用的 ADC12 模块初始化程序，具体的数据存储和处理需要自己在中断处理函数中添加。

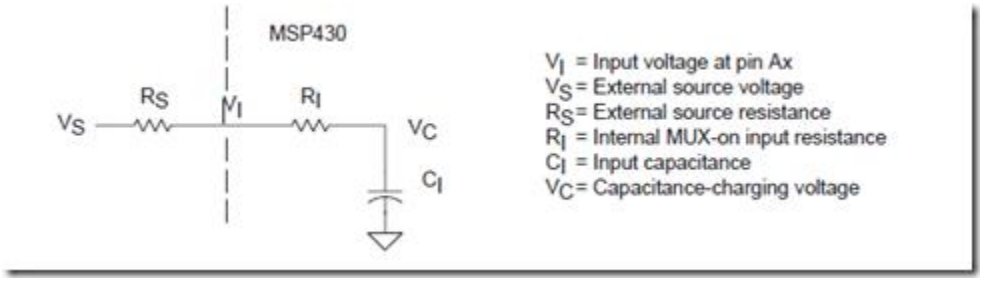
1. **硬件介绍：**

   msp430 单片机内的 ADC12 模块的特点如下：12 位转换精度，1 位非线形误差，1 位非线形积分误差；多种时钟源给 ADC12 模块，切本身自带时钟发生器；内置温度传感器；TimerA/TimerB 硬件触发器；8 路外部通道和 4 路内部通道；内置参考电压源和 6 种参考电压组合；4 种模式的模数转换；16bit 的转换缓存；ADC12 关闭支持超低功耗；采用速度快，最高 200Kbps；自动扫描和 DMA 使能。430 内部的 ADC12 功能还是蛮强大的，可以有定时器触发模数转换开始，还可以和内部的 DMA 模块共同使用，完成高速的采样转储等高级功能。

   这个 AD 的转化公式如下，可以根据它计算采样的模拟电压值：

$$N_{ADC} = 4095 \times \frac{Vin - V_{R-}}{V_{R+} - V_{R-}}$$

   使用 AD 是还要注意采样时间，430 单片机的模数 ADC12 模块的等效模拟电压输入电路如下：



其中 V_S 是信号源电压，R_S 是信号源内阻，V_I 在 Ax(ADC12 模块模拟输入端)上的电压，R_I 单片机内多路开关等效电阻，V_C 是保持电容上的电压(ADC12 模块采样的电压)，C_I 是电容的值。需要根据这些值计算采样时间：

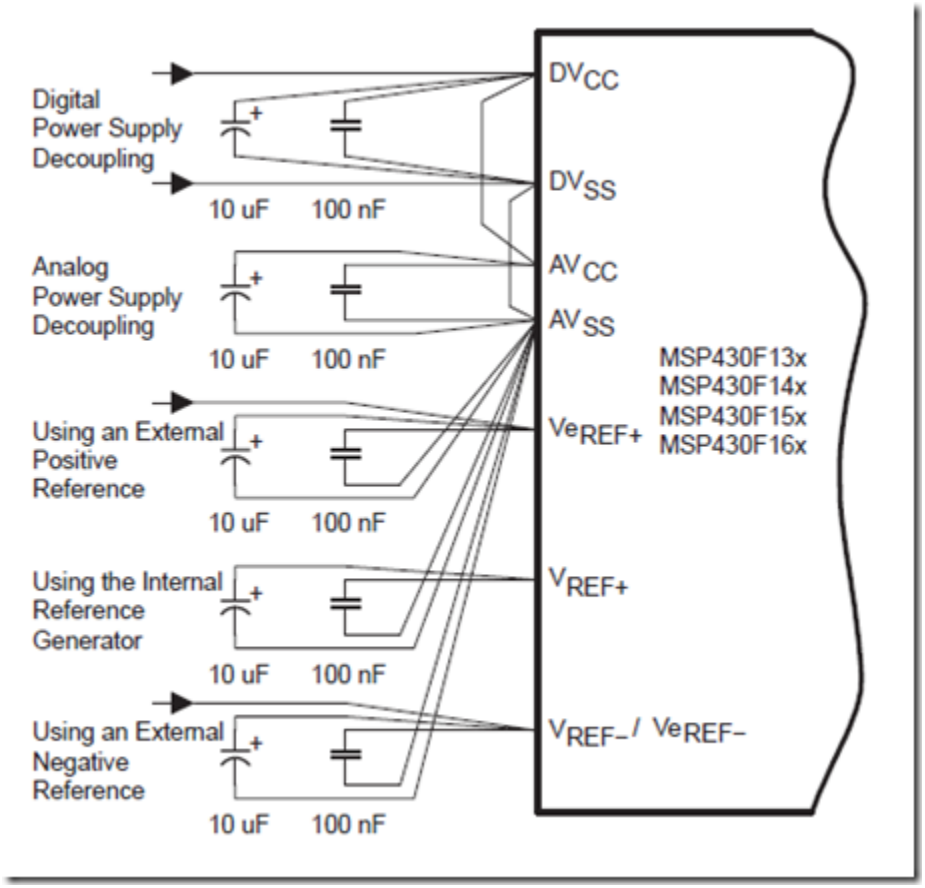
$$t_{sample} > (R_S + R_I) \times \ln(2^{13}) \times C_I + 800ns$$

代入单片机上的参数后公式如下：

$$t_{sample} > (R_S + 2k\Omega) \times 9.011 \times 40pF + 800ns$$

我的程序中采样时间设的是 4us，可以算出如果用我的程序(不更改采样时间)的话，最大信号源内阻可以是 6.8k，当信号源内阻更大时，可以自己按要求设采样时间(在程序的初始化函数内的寄存器设置部分)。

还有，ADC 模数转换时要求参考电压等很稳定，为了达到这个要求，德州仪器要求这部分的电路如下：



即：所有参考源和电源均并联一组 0.1uF 和 10uF 的电容。

硬件部分就说这么多了；如果需要更详细的说明，参考用户指南。

2. **程序实现：**

程序主要实现的是一个比较通用的初始化程序，内容如下：

```
char ADC12Init(char n,char channels[],char rep)
{
    if(n>15)
        return 0;
    //SHT0_0
    ADC12CTL0 = ADC12ON + MSC + SHT0_0 + REFON +
REF2_5V;// 开启 ad,参考电压 2.5v
    ADC12CTL1 = SHP + ADC12SSEL_3;
//Use sampling timer, SMCLK

    for(int i = 0;i < n;i++)
    {
        if(channels[i] >= 0x80)
            return 0;
        *(char*)(ADC12MCTL0_ + i) = channels[i];      //每个
MCTL 设置
    }
```

```
    *(char*)(ADC12MCTL0_ + n - 1) |= EOS;               //序
列结束

    if(rep != 0)                                        //多次转换
    {
        ADC12CTL1 |= CONSEQ_3;
    }
    else
    {
        ADC12CTL1 |= CONSEQ_1;
    }

    ADC12IE = 1<<(n-1);                                   //
Enable ADC12IFG.n-1
    return 1;
}
```

程序先判断 n 通道总数是否超过了可用的个数，超过则返回零然后设置 ADC12CTL0 和 ADC12CTL1 中不需要特殊设置的部分，然后在设置通道模式(根据 rep 参数的值)；for 循环设置的是每个存储寄存器的设置 ADC12MCTLx ；*(char*)(ADC12MCTL0_ + n - 1) |= EOS; //序列结束 这句加入序列结束标志；最后设置中断寄存器并返回成功设置标志。其中比较特殊的是 ADC12MCTL0_，这个是 430 提供的头文件中定义的 ADC12MCTL0 的地址值，以其为指针首址操作 ADCMCTLx 寄存器，从而利用循环设置寄存器的内容，大量减少了代码行数。

参数 channels[]是每个存储寄存器的设置(除 EOS 位之外的)，含义如下：

```
channels[]:对应通道设置，高四位，参考源选择；
低四位，通道选择。具体如下：
SREFx Bits
6-4
Select reference
000 VR+ = AVCC and VR. = AVSS
001 VR+ = VREF+ and VR. = AVSS
010 VR+ = VeREF+ and VR. = AVSS
011 VR+ = VeREF+ and VR. = AVSS
100 VR+ = AVCC and VR. = VREF./ VeREF.
101 VR+ = VREF+ and VR. = VREF./ VeREF.
110 VR+ = VeREF+ and VR. = VREF./ VeREF.
111 VR+ = VeREF+ and VR. = VREF./ VeREF.
INCHx Bits
3-0
Input channel select
0000 A0
0001 A1
```

```
0010 A2
0011 A3
0100 A4
0101 A5
0110 A6
0111 A7
1000 VeREF+
1001 VREF./VeREF.
1010 Temperature sensor
1011 (AVCC – AVSS) / 2
1100 (AVCC – AVSS) / 2
1101 (AVCC – AVSS) / 2
1110 (AVCC – AVSS) / 2
1111 (AVCC – AVSS) / 2
```

这是从用户指南里复制来的，每一位和 ADC12MCTLx 的意义相同(去掉 EOS 位)，所以可用宏定义来制定这个参数，如：

```
char channels[3];
channels[0] = SREF_1+INCH_0;
channels[1] = SREF_1+INCH_1;
channels[2] = SREF_1+INCH_2;
ADC12Init(3,channels,1);
```

这是 3 个通道 A0-A2 采样，多次采样。

启动转换函数：

```
void ADC12Start()
{
    ADC12CTL0 |= ENC;
    ADC12CTL0 |= ADC12SC;
}
```

ADC 初始化完成后，调用此函数开始 AD 转换，转换完成后(一个序列通道，如：刚才的 0-2)，程序自动进入 AD 中断，用户需要在这里为自己的函数添加处理逻辑；这里只存储了转化的结果：

```
#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR (void)
{
    static int i;
    results[0][i] = ADC12MEM0;                  // Move
results, IFG is cleared
    results[1][i] = ADC12MEM1;                  // Move
results, IFG is cleared
    results[2][i] = ADC12MEM2;                  // Move
results, IFG is cleared
```

```
    i++;
    if(i>31)                                    //多次转换时 转
换次数
    {
        //多次重复采样时，在这里方处理函数
        ADC12CTL0 &=~ ENC;                      //停止转
换
        i=0;
    }

}
```

该程序实现的是多次 A0-A2 32 次转换，把结果存入 results 数组。单次时，仅仅采样一次(A0-A2)可用自己更改处理函数。

程序部分就完成了，调用时注意要自己实现处理逻辑或存储逻辑。

1. **使用示例：**

本程序使用方式还是加入 C 文件，包含 H 文件；不过和之前的程序不同的是要自己实现中断处理逻辑。

使用示例参见程序库中的 ADC12.

```
#include <msp430x16x.h>
#include "ADC12.h"
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    char channels[3];
    channels[0] = SREF_1+INCH_0;
    channels[1] = SREF_1+INCH_1;
    channels[2] = SREF_1+INCH_2;
    ADC12Init(3,channels,1);
    _EINT();
    ADC12Start();
    LPM0;
}
```

这里实现的是 3 通道多次转换，参考电压都是内部参考电压。自己实现的处理逻辑参见前面的程序实现的最后一部分。

ADC12 模块部分就到这里了，有什么不足之处，欢迎提出建议、讨论。

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<十一>定时器 TA 的 PWM 输出

定时器是单片机常用的其本设备，用来产生精确计时或是其他功能；msp430 的定时器不仅可以完成精确定时，还能产生 PWM 波形输出，和捕获时刻值(上升沿或是下降沿到来的时候)。这里完成一个比较通用的 PWM 波形产生程序。

1．硬件介绍：

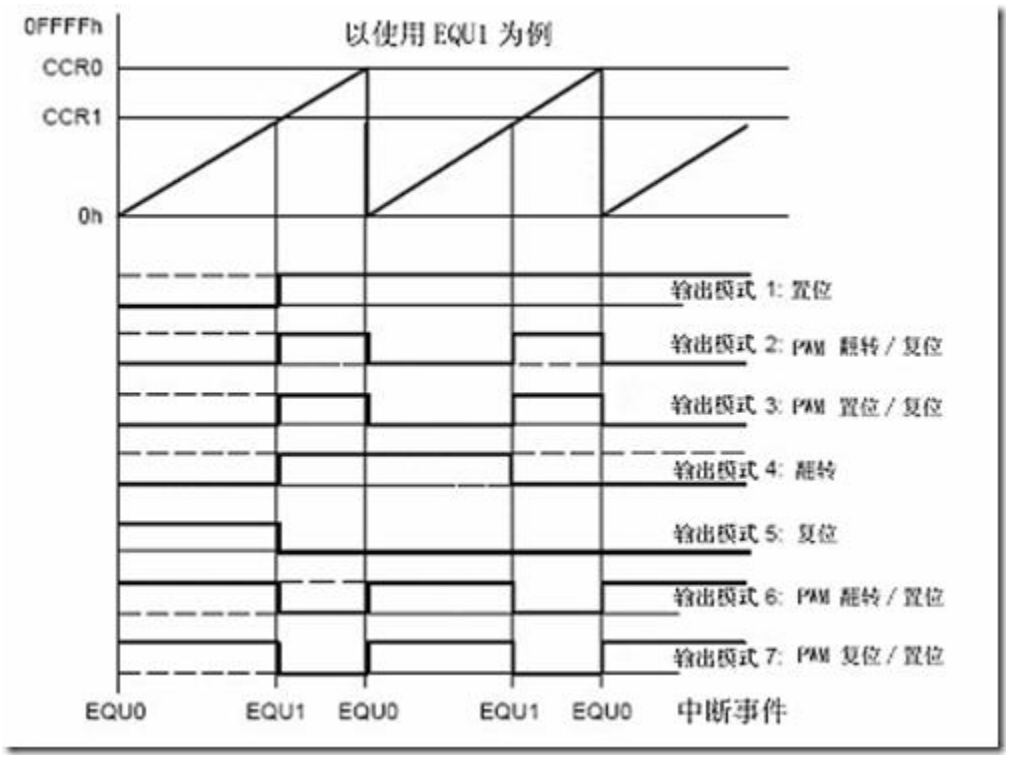
MSP430 系列单片机的 TimerA 结构复杂，功能强大，适合应用于工业控制，如数字化电机控制，电表和手持式仪表的理想配置。它给开发人员提供了较多灵活的选择余地。当 PWM 不需要修改占空比和时间时，TimerA 能自动输出 PWM，而不需利用中断维持 PWM 输出。

MSP430F16x 和 MSP430F14x 单片机内部均含有两个定时器，TA 和 TB；TA 有三个模块，CCR0-CCR2；TB 含有 CCR0-CCR67 个模块；其中 CCR0 模块不能完整的输出 PWM 波形(只有三种输出模式可用);TA 可以输出完整的 2 路 PWM 波形；TB 可以输出 6 路完整的 PWM 波形。

定时器的 PWM 输出有有 8 种模式：

输出模式 0　输出模式：输出信号 OUTx 由每个捕获/比较模块的控制寄存器 CCTLx 中的 OUTx 位定义，并在写入该寄存器后立即更新。最终位 OUTx 直通。
输出模式 1 置位模式：输出信号在 TAR 等于 CCRx 时置位，并保持置位到定时器复位或选择另一种输出模式为止。
输出模式 2 PWM 翻转/复位模式：输出在 TAR 的值等于 CCRx 时翻转，当 TAR 的值等于 CCR0 时复位。
输出模式 3 PWM 置位/复位模式：输出在 TAR 的值等于 CCRx 时置位，当 TAR 的值等于 CCR0 时复位。
输出模式 4 翻转模式：输出电平在 TAR 的值等于 CCRx 时翻转，输出周期是定时器周期的 2 倍。
输出模式 5 复位模式：输出在 TAR 的值等于 CCRx 时复位，并保持低电平直到选择另一种输出模式。
输出模式 6PWM 翻转/置位模式：输出电平在 TAR 的值等于 CCRx 时翻转，当 TAR 值等于 CCR0 时置位。
输出模式 7PWM 复位/置位模式：输出电平在 TAR 的值等于 CCRx 时复位，当 TAR 的值等于 CCR0 时置位。

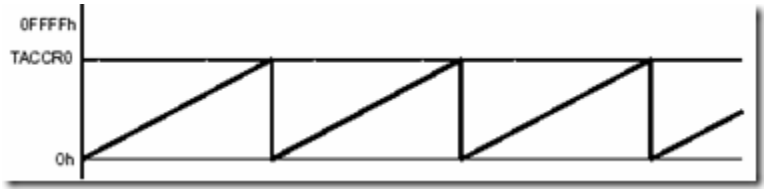
下图是增计数模式下的输出波形(本程序使用的是增模式 3 和 7)：



计数模式：

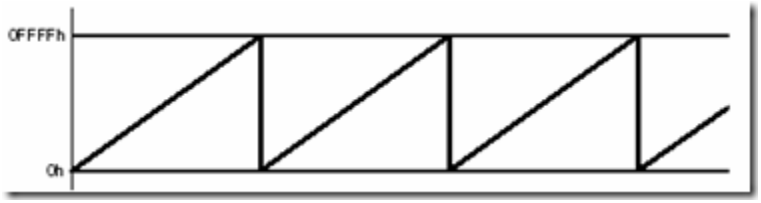
增计数模式

捕获/比较寄存器 CCR0 用作 Timer_A 增计数模式的周期寄存器，因为 CCR0 为 16 位寄存器，所以该模式适用于定时周期小于 65 536 的连续计数情况。计数器 TAR 可以增计数到 CCR0 的值，当计数值与 CCR0 的值相等(或定时器值大于 CCR0 的值)时，定时器复位并从 0 开始重新计数。



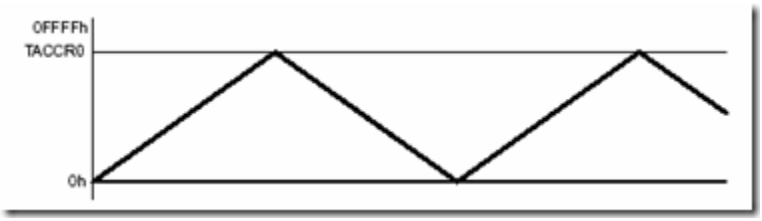
连续计数模式

在需要 65 536 个时钟周期的定时应用场合常用连续计数模式。定时器从当前值计数到 0FFFFH 后，又从 0 开始重新计数



增/减计数模式

需要对称波形的情况经常可以使用增/减计数模式，该模式下，定时器先增计数到 CCR0 的值，然后反向减计数到 0。计数周期仍由 CCR0 定义，它是 CCR0 计数器数值的 2 倍。



TA 定时器有比较、捕获两种工作方式；比较可以产生 PWM 波形等，捕获可以精确的测量时间；这里用的是比较输出。

硬件介绍就这么多了，其他的可以参考 msp430x1xx_family_users_guide(用户指南)。

2. **程序实现：**

本程序是直接从 msp430f42x 移植的，只改动了端口就能正常使用了。由此，430 的模块在不同的系列中是通用的，有关寄存器是一样的；只是也许外部端口不太一样。

程序初始化部分：完成 TA 相关寄存器的初始化。

```
char TAPwmInit(char Clk,char Div,char Mode1,char Mode2)
{
    TACTL = 0;                      //清除以前设置
    TACTL |= MC_1;                   //定时器 TA 设为增计数模式
    switch(Clk)                     //选择时钟源
    {
        case 'A': case 'a':   TACTL|=TASSEL_1; break;
//ACLK
        case 'S': case 's':   TACTL|=TASSEL_2; break;
//SMCLK
        case 'E':              TACTL|=TASSEL_0; break;     //
外部输入(TACLK)
        case 'e':              TACTL|=TASSEL_3; break;     //
外部输入(TACLK 取反)
        default :   return(0);                              //参数
有误
    }
    switch(Div)                     //选择分频系数
    {
        case 1:    TACTL|=ID_0; break;    //1
        case 2:    TACTL|=ID_1; break;    //2
        case 4:    TACTL|=ID_2; break;    //4
        case 8:    TACTL|=ID_3; break;    //8
        default :   return(0);             //参数有误
    }
    switch(Mode1)                   //设置 PWM 通道 1 的输出模式。
    {
        case 'P':case 'p':           //如果设置为高电平模式
```

```
            TACCTL1 = OUTMOD_7;       //高电平 PWM 输出
            P1SEL |= BIT2;            //从 P1.2 输出 (不同型号
单片机可能不一样)
            P1DIR |= BIT2;            //从 P1.2 输出 (不同型号
单片机可能不一样)
            break;
        case 'N':case 'n':            //如果设置为低电平模式
            TACCTL1 = OUTMOD_3;       //低电平 PWM 输出
            P1SEL |= BIT2;            //从 P1.2 输出 (不同型号
单片机可能不一样)
            P1DIR |= BIT2;            //从 P1.2 输出 (不同型号
单片机可能不一样)
            break;
        case '0':case 0:              //如果设置为禁用
            P1SEL &= ~BIT2;           //P1.2 恢复为普通 IO 口
            break;
        default :   return(0);        //参数有误
    }
    switch(Mode2)                     //设置 PWM 通道 1 的输出
模式。
    {
        case 'P':case 'p':            //如果设置为高电平模式
            TACCTL2 =OUTMOD_7;        //高电平 PWM 输出
            P1SEL |= BIT3;            //从 P1.3 输出 (不同型号
单片机可能不一样)
            P1DIR |= BIT3;            //从 P1.3 输出 (不同型号
单片机可能不一样)
            break;
        case 'N':case 'n':            //如果设置为低电平模式
            TACCTL2 =OUTMOD_3;        //低电平 PWM 输出
            P1SEL |= BIT3;            //从 P1.3 输出 (不同型号
单片机可能不一样)
            P1DIR |= BIT3;            //从 P1.3 输出 (不同型号
单片机可能不一样)
            break;
        case '0':case 0:              //如果设置为禁用
            P1SEL &= ~BIT3;           //P1.3 恢复为普通 IO 口
            break;
        default :   return(0);        //参数有误
    }
    return(1);
}
```

主要是设置 TACTL 寄存器，让 TA 工作于增模式，设置时钟源和分频；CCTLx 设置对应的输出模式；并且打开相应端口的第二功能。

设置周期函数：设置 PWM 波形的周期，单位是多少个 TACLK 周期。

```
void TAPwmSetPeriod(unsigned int Period)
{
    TACCR0 = Period;
}
```

工作于增模式时，TA 计数到 TACCR0,设 CCR0 就完成了周期的设置。

设置占空比：设置 TA 的 PWM 输出的有效电平的时间。

```
void TAPwmSetDuty(char Channel,unsigned int Duty)
{
    switch(Channel)
    {
        case 1: TACCR1=Duty; break;
        case 2: TACCR2=Duty; break;
    }
}
```

根据参数分别设置每一路的参数。设置占空比，用千分比设置：

```
* 入口参数：Channel：当前设置的通道号  1/2
            Percent：PWM 有效时间的千分比 (0~1000)
* 出口参数：无
* 说    明：1000=100.0%  500=50.0% ，依次类推
* 范    例：TAPwmSetPermill(1,300)设置 PWM 通道 1 方波的占空比为 30.0%
            TAPwmSetPermill(2,825)设置 PWM 通道 2 方波的占空比为 82.5%
            */
void TAPwmSetPermill(char Channel,unsigned int Percent)
{
    unsigned long int Period;
    unsigned int Duty;
    Period = TACCR0;
    Duty = Period * Percent / 1000;
    TAPwmSetDuty(Channel,Duty);
}
```

1. 这个函数用千分比来设置 PWM 输出的有效时间。方便程序的使用。

   有关定时器，TI 提供的大量的例程，这些历程都很简洁、清晰。需要其他功能可以自己根据例程编写对应的程序。程序实现就这么多了，下面说下本程序的使用方法。

2. **使用示例：**

   使用方式：依然是在工程中加入 c 文件；文件包含 h 头文件；然后就可以正常使用本函数了。详细参考示例工程和 main.c。

main 主要程序如下：

```
#include "msp430x16x.h"       //430 寄存器头文件
#include "TAPwm.h"            //TA PWM 输出程序库头文件

void main()
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();

    TAPwmInit('A',1,'P','P');    //将定时器 TA 初始化成为 PWM 发生器
                    //时钟源=ACLK ； 无分频；  通道 1 和通道 2 均设为高电平模式。
    TAPwmSetPeriod(500);          //通道 1/2 的 PWM 方波周期均设为 500 个时钟周期
    TAPwmSetDuty(1,200);          //1 通道  有效 200 个时钟周期
    TAPwmSetPermill(2,200);       //2 通道  20.0%

    LPM0;
}
```

本程序调用程序库，产生两路 PWM 波形。

TA 的 PWM 输出就到这儿了，如果需要更多路的 PWM 波，可以使用 TB，他可以产生 6 路完整的 PWM 波形；可以参考本程序编写 TB 的波形输出程序。有什么不足之处，欢迎评论，讨论。

附件：程序库

MSP430 程序库<十二>SVS(电源电压监控器)模块

电源电压监控对于单片机来说，也是经常要用的模块。当需要稳定的工业级产品时，经常要对电源电压监控，以保证单片机系统工作于正常环境或范围中。MSP430F16x 提供了一个现成的电源电压监控器模块 SVS，方便检测电源电压或者是外部电压，可以设置为电压过低时复位 或置标志位。本程序即完成 SVS 的设置使用的程序库(msp430f14x 没有此模块)。

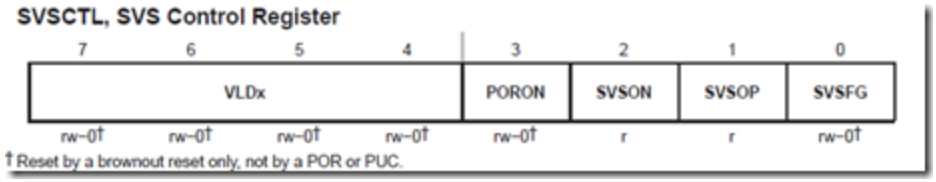
硬件介绍：

MSP430 单片机含有的 SVS 模块可以很方便的监控电源电压或外部电压。

电源电压监控器（SVS）是用于监控 AVCC 电源电压或外部电压。 SVS 的可配置当电源电压或外部电压下降到低于用户选择的电压级别时设置一个标志，或产生 POR 复位。

SVS 模块有以下特点：可以监控 AVCC 电压；可选择产生复位信号；可软件设置 SVS 比较器输出信号；低电压标志可以被锁定或被用户程序访问；有 14 个可供选择的电压门限；可以监控外部输入电压。SVS 模块可以很方便的监控电源电压或系统的其他电压，可以产生复位信号或是置标志位。

SVS 模块仅有一个 8 位的寄存器，使用十分方便。寄存器 SVSCTL：



高四位 VLDx 用来设置监控电源电压的门限、关闭 SVS 或者选择监控外部输入电压。具体含义如下：

0000 SVS is off　　0001 1.9 V 检测 AVCC 是否低于 1.9v,以下类似

0010 2.1 V　　0011 2.2 V
0100 2.3 V　　0101 2.4 V
0110 2.5 V　　0111 2.65 V
1000 2.8 V　　1001 2.9 V
1010 3.05　　1011 3.2 V
1100 3.35 V　　1101 3.5 V
1110 3.7 V
1111 检测由 SVSIN 引脚输入的电压是否低于 1.2 V.

1. 当高四位是 0 时，SVS 模块是关闭的；1-14 分别是对电源电压监控的 14 个门限电压；15 时，监控外部电压，门限电压是 1.2v。

PORON 位设置是否启动电压低于门限时，单片机复位：1 复位 0 置标志位 SVSFG

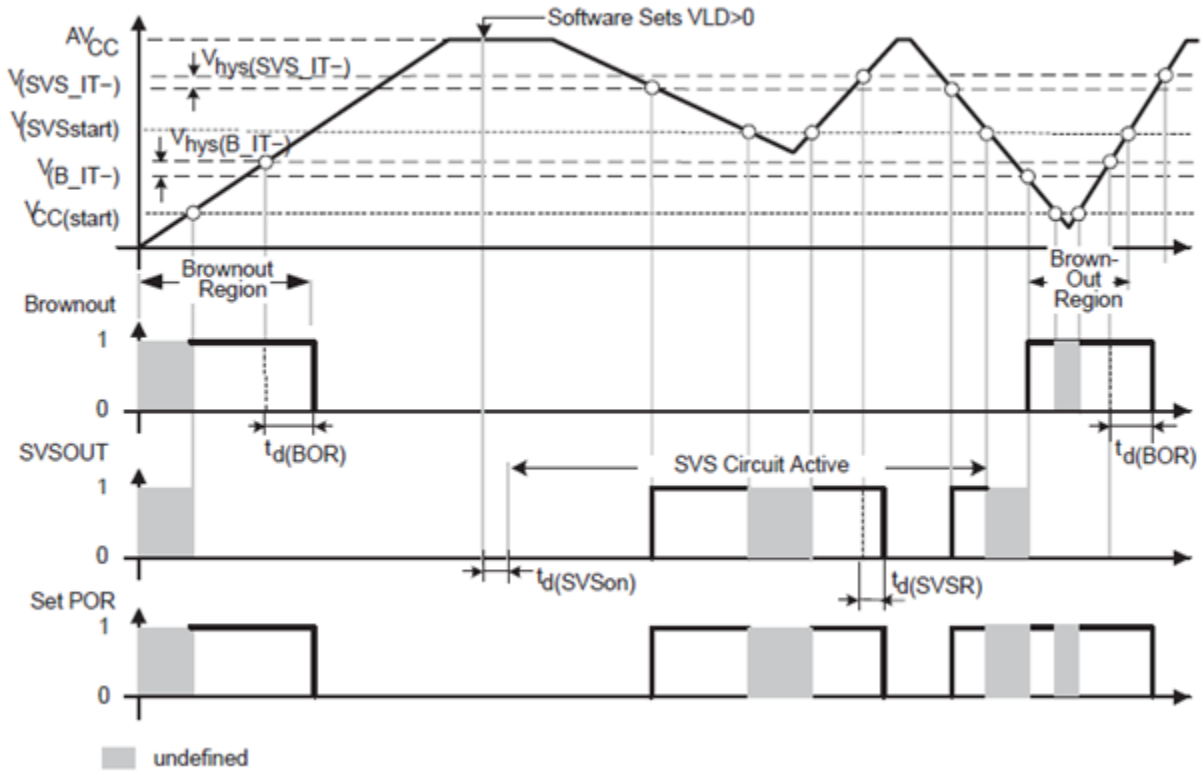
SVSON 位，这位和其他模块的 ON 位不太一样，SVSON 位仅仅指示当前 SVS 模块是否打开，而不是用来开关模块的。

SVSOP 位，这位是设置 SVS 内部比较器输出值：0 输出低电平 1 输出高电平。

SVSFG 位，标志位 指示是否检测到低电压 仅 PORON 为 0 时有效 出现低电压后置 1；改为不会自动清零，必须软件清零。

另外，SVS 模块值得一提的是：SVS 门限电压已经设置回差带：每个 SVS 的水平已经滞后 AVCC，接近临界值时，以减少小型电源电压的变化的敏感性。 SVS 的操作和 SVS /掉电互操作如图：

Figure 6−3. Operating Levels for SVS and Brownout/Reset Circuit



如图：为防止电压在门限附近变动时，SVS 过于敏感，每个门限附近都有回差带。这样 SVS 模块用起来更好用。

2. **程序实现：**

程序主要是对 SVS 模块寄存器 SVSCTL 的设置和检测。首先是设置 SVS 函数：

```
void SVSSetup(char voltageLevel,char reset)
{
    SVSCTL = voltageLevel << 4;
    /*if(voltageLevel == 0x15)            //外部输入 打开对应功能口
    {
        P6SEL |= BIT7;                  //不需要，当用SVSIN 时，自动从此脚检测
    }*/
    if(reset <= 1)
    {
        SVSCTL |= reset << 3;
    }
}
```

voltageLevel：这个参数和寄存器 SVSCTL 的高四位 VLDx 意思完全一样，程序仅仅是把它移动到高四位赋值给寄存器 SVSCTL，reset 参数对应 PORON 位，也是直接赋值给对应位完成设置。

检测是否有低于门限电压的情况出现：

```
char SvsFlg()
{
    return (SVSCTL&SVSFG);
}
```

这个函数更简单，仅仅把标志位 SVSFG 的值返回，以便用户判断是否出现了低于门限的情况出现。

标志位清零：

```
/*****************************
* 名      称：ClearSvs
* 功      能：电源电压监控器的过低标志
* 入口参数：sync：同步 1：阻塞运行直到该函数电压恢复正常 0:不阻塞,清除即返回
* 出口参数：无
* 说      明：若传入参数为0 不阻塞 则如果电压没有恢复到正常范围则标志会立即被
                单片机重新置位(1)
*****************************/
void ClearSvs(char sync)
{
    if(!sync)
    {
        SVSCTL &=~ SVSFG;
        return;
    }
    while(SVSCTL&SVSFG)
        SVSCTL &=~ SVSFG;    //清除标志 直到电压正常
}
```

- 由于 SVSFG 标志位不会在处理后自动被清除，所以必须软件清零。这个函数有两种工作方式，同步阻塞等待，直到电压恢复正常后才返回和清零后即返回。

程序实现比较简单，但能够完成 SVS 的功能。下面介绍如何使用本程序库。

- **使用示例：**

使用程序库的方式还是和以前一样：工程中加入 SVS.c 文件，源文件中加入对 SVS.h 的文件包含。

main.c 主要内容如下：

```
#include <msp430x16x.h>    //430 寄存器头文件
#include <stdio.h>
#include "Lcd12864.h"
#include "SVS.h"

/*************************************************
*****************************
* 名      称：main 主程序
* 功      能：设置串口，输出信息，从串口读计算机键盘输入数据，测
试串口收发
* 入口参数：无
* 出口参数：无
* 说      明：复位测试时 每次电压调低再调正常 液晶显示的数据加 1
            不复位时 每次调低 输出一个电压过低。
*************************************************
*****************************/
void main()
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    LcdInit();

    /*//========= 电压过低时复位测试==============
    __no_init char ff;        //复位不初始化

    SVSSetup(0x0A,1);         //检测电源电压 3.05v 低于
3.05v 时单片机复位
    ff++;                      //此变量 每次复位加 1
    printf("%d",ff);          // 电压调低（<3.05v）再调高，显示
变量将加 1
    */
    SVSSetup(0x0A,0);         //测电源电压 3.05v 低于 3.05v
时单片机 不复位
                              //0x0A 改为 0x0f 则对 P6.7 电压监
控 检测是否低于 1.2v
    while(1)
    {
        if(SvsFlg())
            printf("电压过低");
        //SVSFG 位必须 软件清零，如果电压没有回到 3.05 以上，
        //位的值立即被单片机置为 1
```

```
            ClearSvs(1);            //清除标志 直到恢复正常电压
        }
    }
```

本程序使用 12864 液晶来显示电压过低的情况：复位时，设置一个__no_init 变量，每次复位加 1，可以看到电压调低后，显示数字被加 1.不复位置，显示电压过低。这里使用的是 12864 的底层驱动和 printf 函数移植，比之前做了稍微更改，这些在注释中说明的已经很详细了，这里不在细说。

SVS 模块就到这里了，有什么不足，欢迎拍砖。

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<十三>硬件乘法器使用

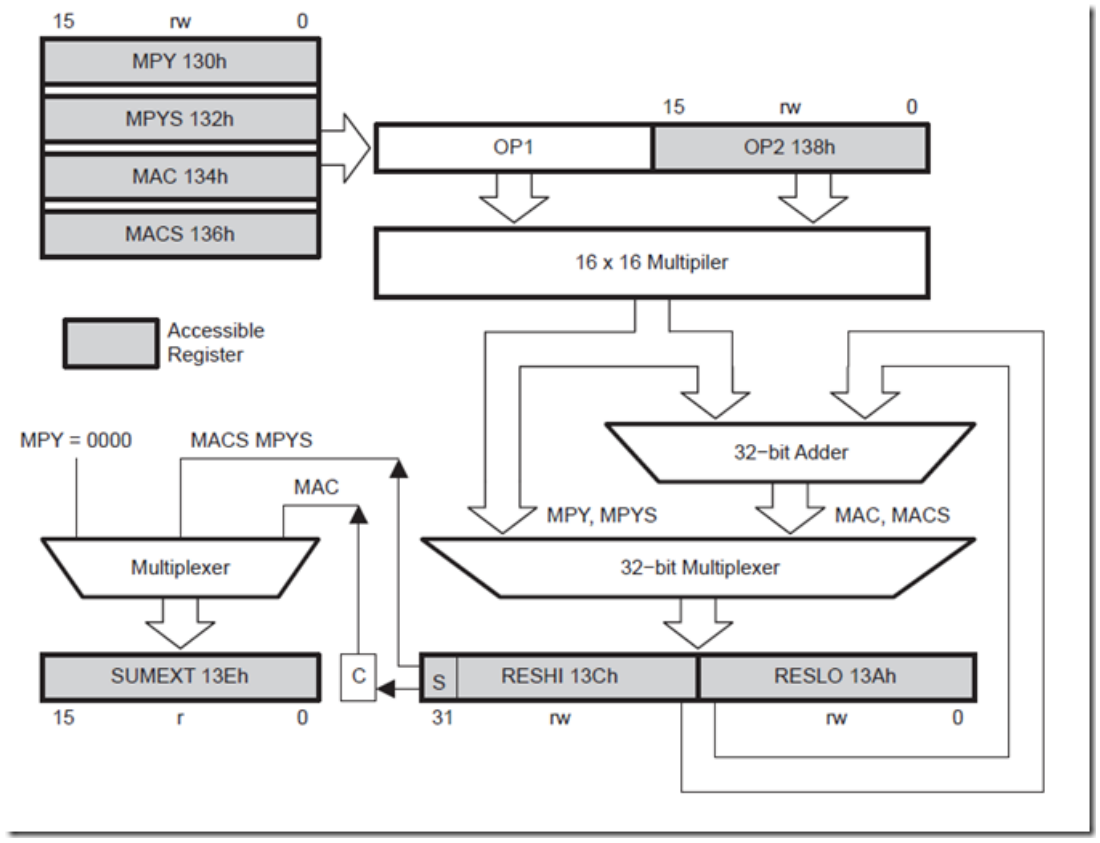
硬件乘法器不占用 CPU 周期，有硬件实现，速度比软件实现的乘法速度快很多。msp430f14x、msp430f16x 中都含有硬件乘法器模块，方便用户需要速度的时候使用。

硬件介绍：

在 MSP430 系列单片机中，硬件乘法器是外围模块，而不是 CPU 内核的一部分；所以它的活动与否与 CPU 的活动与否无关，它的寄存器和其他的外围寄存器一样通过 CPU 指令读写。

硬件乘法器模块支持一下功能：无符号乘法、有符号乘法、无符号乘加、有符号乘加；可以支持 16*16 16*8 8*16 8*8bits 的乘法。

硬件乘法器的模块框图如下：



硬件乘法器模块的四种操作类型(无符号乘法、有符号乘法、无符号乘加、有符号乘加)是由写入的第一个操作数的位置决定的。这个模块有两个操作数寄存器：OP1 和 OP2、三个结果寄存器 RESLO, RESHI, 和 SUMEXT。RESLO 寄存器存储结果的低字(低 16 位)；RESHI 寄存器存储结果的高字(高 16 位)；SUMEXT 寄存器存储结果的有关信息。结果在 3 个时钟周期后即可完成；写入 OP2 后的下一条指令即可读取结果，有一种情况例外：用间接寻址方式访问结果。用间接寻址方式访问结果时，读取结果之前需要有一条 NOP 指令。

操作数 OP1 有四个地址(MPY:0130h MPYS:0132h MAC:0134h MACS:0136h)，这四个寄存器用来选择乘法的操作模式。写入第一个操作数寄存器决定用哪种操作：无符号 用符号等，但是不启动相乘操作；写入第二个操作数寄存器启动相乘的操作。计算完成后结果存入寄存器 RESLO,RESHI, 和 SUMEXT。

操作数 1 的四个地址对应的操作：

OP1 Address	Register Name	Operation
0130h	MPY	Unsigned multiply（无符号乘法）
0132h	MPYS	Signed multiply（有符号乘法）
0134h	MAC	Unsigned multiply accumulate（无符号乘加）
0136h	MACS	Signed multiply accumulate（有符号乘加）

四种操作模式下高位结果寄存器的内容如下：

```
Mode          RESHI Contents
MPY           Upper 16-bits of the result
MPYS          The MSB is the sign of the result. The remaining
bits are the upper
              15-bits of the result. Two's complement notation
is usedfor the result.
MAC           Upper 16-bits of the result
MACS          Upper 16-bits of the result. Two's complement
notation is used for the result.
```

四种操作模式 SUMEXT 寄存器的内容：

```
Mode          SUMEXT
MPY           SUMEXT is always 0000h
MPYS          SUMEXT contains the extended sign of the result
              00000h Result was positive or zero
              0FFFFh Result was negative
MAC           SUMEXT contains the carry of the result
              0000h No carry for result
              0001h Result has a carry
MACS          SUMEXT contains the extended sign of the result
              00000h Result was positive or zero
              0FFFFh Result was negative
```

连续乘法运算时，如果操作数 1 不需改变就可以运算，则可以不需要重新写入和以保存内容相同的数；但 OP2 必须重新写入以启动乘法运算。

MACS Underflow and Overflow（MACS 时的上溢和下溢）：硬件乘法器不检测有符号乘加时运算结果的上溢出和下溢出。结果的正数范围：0 到 7FFF FFFFh；负数范围：0FFFF FFFFh 到 8000 0000h。下溢出是两个负数的和结果寄存器得到的是正数，上溢出是两个正数的和结果寄存器得到的是负数。SUMEXT 寄存器存储有结果的符号，可以根据它判断是否溢出（0000h 负数和 则上溢 0FFFFh 正数和 则下溢）。使用时 程序必须合适的检测、处理 MACS 的溢出情况。

程序示例(用户指南上给出的汇编示例)：

所有乘数模式的例子如下。所有的 8x8 模式使用的寄存器的绝对地址，因为汇编器将不允许 B 访问到字寄存器时使用标准定义的文件标签。

```
; 16x16 Unsigned Multiply
MOV #01234h,&MPY ; Load first operand
MOV #05678h,&OP2 ; Load second operand
; ... ; Process results
; 8x8 Unsigned Multiply. Absolute addressing.
MOV.B #012h,&0130h ; Load first operand
MOV.B #034h,&0138h ; Load 2nd operand
; ... ; Process results
; 16x16 Signed Multiply
```

```
MOV #01234h,&MPYS ; Load first operand
MOV #05678h,&OP2 ; Load 2nd operand
; ... ; Process results
; 8x8 Signed Multiply. Absolute addressing.
MOV.B #012h,&0132h ; Load first operand
SXT &MPYS ; Sign extend first operand
MOV.B #034h,&0138h ; Load 2nd operand
SXT &OP2 ; Sign extend 2nd operand
; (triggers 2nd multiplication)
; ... ; Process results
; 16x16 Unsigned Multiply Accumulate
MOV #01234h,&MAC ; Load first operand
MOV #05678h,&OP2 ; Load 2nd operand
; ... ; Process results
; 8x8 Unsigned Multiply Accumulate. Absolute addressing
MOV.B #012h,&0134h ; Load first operand
MOV.B #034h,&0138h ; Load 2nd operand
; ... ; Process results
; 16x16 Signed Multiply Accumulate
MOV #01234h,&MACS ; Load first operand
MOV #05678h,&OP2 ; Load 2nd operand
; ... ; Process results
; 8x8 Signed Multiply Accumulate. Absolute addressing
MOV.B #012h,&0136h ; Load first operand
SXT &MACS ; Sign extend first operand
MOV.B #034h,R5 ; Temp. location for 2nd operand
SXT R5 ; Sign extend 2nd operand
MOV R5,&OP2 ; Load 2nd operand
; ... ; Process results
```

上面的程序虽然和标准的汇编差异比较大，但是有一定汇编基础的人还是很容易就能够看懂。这里的程序给出了多种方式写入操作数寄存器。

间接寻址结果寄存器时，在写入 OP2 操作数启动乘法后，至少需要一个指令的延迟后才能访问结果寄存器 RESLO 等；直接寻址时可以写入 OP2 后，下一条指令即可读取结果。示例程序（汇编）：

```
; Access multiplier results with indirect addressing
MOV #RESLO,R5 ; RESLO address in R5 for indirect
MOV &OPER1,&MPY ; Load 1st operand
MOV &OPER2,&OP2 ; Load 2nd operand
NOP ; Need one cycle 写入两个操作数 乘法运算开始后 需要一个NOP
MOV @R5+,&xxx ; Move RESLO
MOV @R5,&xxx ; Move RESHI
```

如果在写入 OP1 和写入 OP2 之间产生了中断，中断响应后，源操作数的计算模式丢失；运算结果不确定。为了避免这种情况的发生，在写入操作数时禁止中断或在中断响应函数中不使用硬件乘法器。如：

```
; Disable interrupts before using the hardware multiplier
DINT ; Disable interrupts
NOP ; Required for DINT
MOV #xxh,&MPY ; Load 1st operand
MOV #xxh,&OP2 ; Load 2nd operand
EINT ; Interrupts may be enable before
; Process results
```

硬件部分就说这么多了，有什么不大明白的可以参考用户指南。

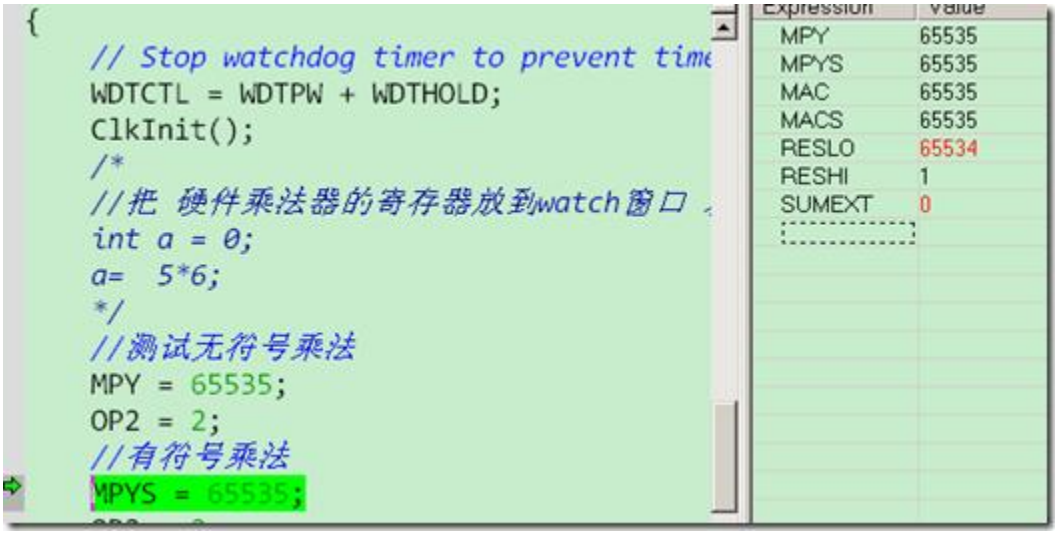
使用示例：

我的程序仅仅是用 C 语言演示硬件乘法器的使用。程序主要内容如下：

```
#include <msp430x16x.h>
/*****************************************
* 名      称：main 主程序
* 功      能：硬件乘法器程序库使用演示
* 入口参数：无
* 出口参数：无
*****************************************/
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();

    /*把 硬件乘法器的寄存器放到 watch 窗口 观察是否变化
    int a = 0;
    a=  5*6;
    */
    //测试无符号乘法
    MPY = 65535;
    OP2 = 2;
    //有符号乘法
    MPYS = 65535;
    OP2 = 2;
    //无符号乘加
    MAC = 65535;
    OP2 = 2;
    //有符号乘加
    MACS = 65535;
    OP2 = 2;
```

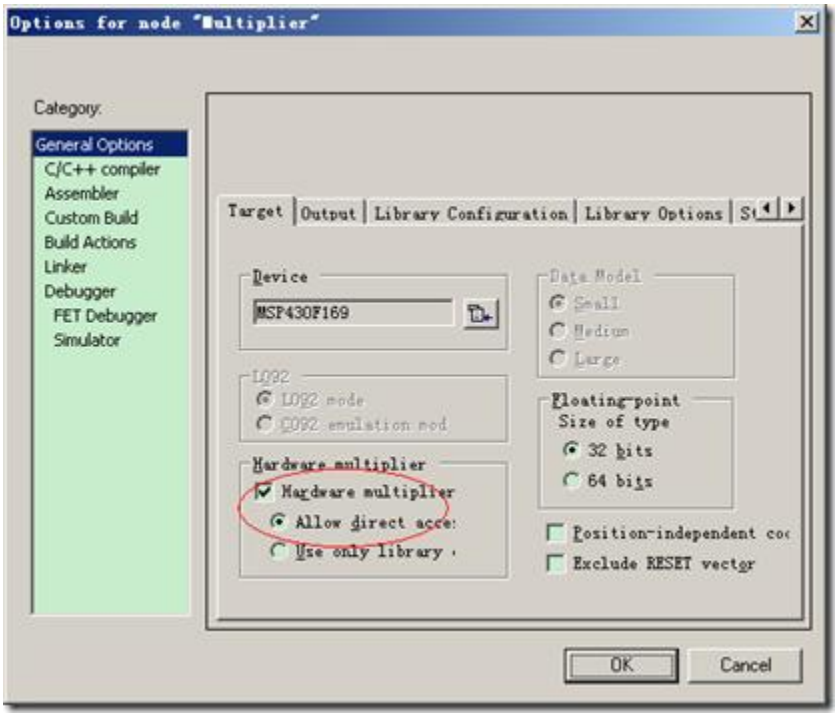
```
    LPM0;
}
```

程序演示了 4 中乘法模式：使用时单步调试，观察硬件乘法器的有关寄存器。如：



硬件乘法器运算速度很快，只需 3 个时钟周期；这里 IAR 单步调试时，OP2 赋值结束，在 watch 窗口马上就可以看到运算结果。其他三种模式类似。

注释掉的这部分是我用来检测 IAR 编译程序是否使用硬件乘法器进行测试。默认情况下，乘法应该是用硬件乘法器运算的。默认的设置如下：



硬件乘法器是选中的，这时应该是使用硬件乘法器的，但是我的调试结果显示它没有使用硬件乘法器，截图下：

运行后乘法器相关位没有对应变化，如果使用的话，应该变化。

硬件乘法器不选中时，寄存器也没有相应变化，从这看，IAR 没有使用硬件乘法器；也许程序没有优化太多或是 debug 版本不使用硬件乘法器。

如果需要直接使用硬件乘法器，有必要时把设置的硬件乘法器去掉，以防冲突。

下面是直接使用硬件乘法器的一个实例：

```
#include "msp430x16x.h"
unsigned int Result[7];
unsigned char Data1[7];
unsigned char Data2[7];
void main(void)
{
    unsigned char i;
    WDTCTL = WDTPW + WDTHOLD; // 关看门狗
    for(i=0; i<7; i++)
    {
        Data1[i] = 10 * i; // 对两数组赋值
        Data2[i] = 25 * i;
    }
    for(i=0; i<7; i++)
    {
        MPY = Data1[i];
        OP2 = Data2[i];
        _NOP(); // 延迟
        _NOP();
        _NOP();
        Result[i] = RESLO; // 保存结果，由于是 8×8 型，所以未用到 RESHI；
    }
}
```

这个程序用无符号乘法运算，结果存入结果数组中。值的注意的是程序中的 3 个 NOP，这里 NOP 不需要，根据头文件推测，IAR 编译器应该使用的是直接寻址方式，可以不要。如果不太放心，一个 NOP 即可，即便用的是间接寻址，一个 NOP 的延迟已经足够。

硬件乘法器一般不会像上面的程序那么使用，如果这样就太浪费了；还不如直接用 *操作符来的简便；硬件乘法器主要用来对时间要求苛刻的情况。如：用 430 进行数字滤波，快速傅里叶变换等。ti 有一篇应用笔记介绍的就是用 msp430f169 实现数字滤波方案。

硬件乘法器就到这里了，希望对大家有所帮助。有什么不足之处，欢迎拍砖讨论。

附件：程序库

MSP430 程序库<十四>DMA 程序库

直接存储器存取(DMADirect Memory Access)方式是用硬件实现存储器与存储器之间或存储器与 I\O 设备之间直接进行高速数据传送，不需要 CPU 的干预。这种方式通常用来传送数据块。MSP430f16x 系列单片机内部含有 DMA 模块，而且几乎内部所有外设都可以触发 DMA 开始存取数据。这里实现了这个模块的程序通用的函数库，方便使用。

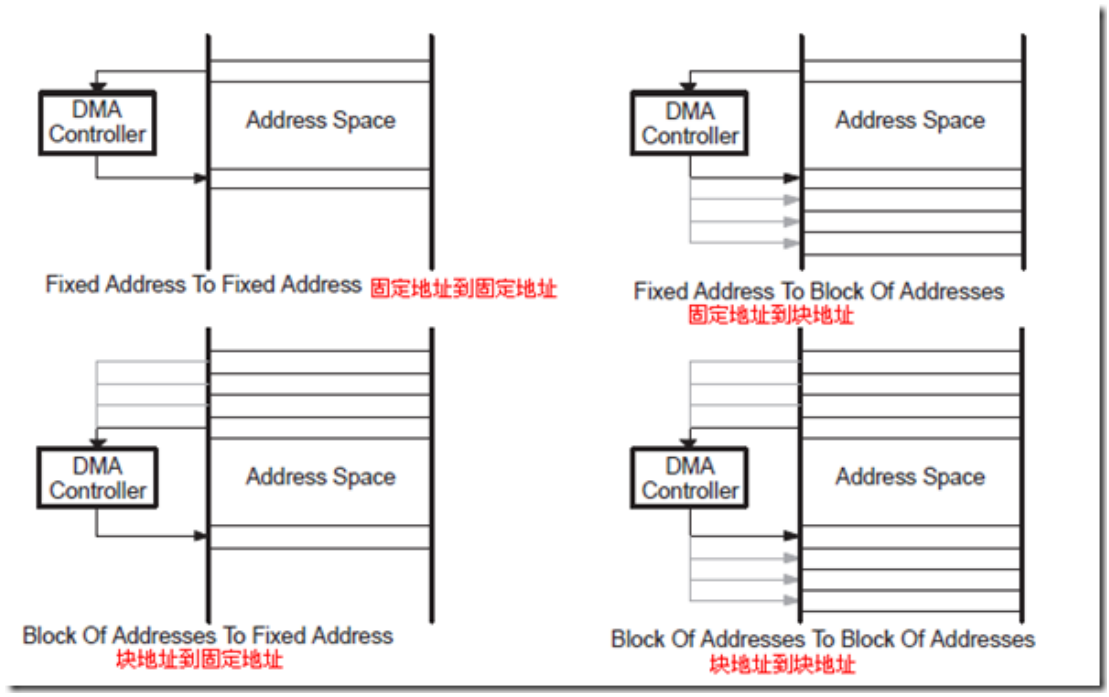
- **硬件介绍：**

MSP430F15X/16X 系列单片机具有 DMA 控制器，从而能够为数据高速传输提供保证。例如，通过 DMA 控制器可以直接将 ADC 转换存贮器的内容传到 RAM 单元。

MSP430 系列单片机扩展的 DMA 具有来之所有外设的触发器，不需要 CPU 的干预即可提供先进的可配置的数据传输能力，从而加速了基于 MCU 的信号处理进程，DMA 传输的触发来源对 CPU 来说是完全透明的，DMA 控制器可在内存与外部及外部硬件之间进行精确的传输控制。DMA 消除了数据传输延迟时间以及各种开销，从而可以解放 16 为 RISC CPU，以便其将更多的时间用于处理数据，而非执行正在处理的任务。

MSP430F16x 系列单片机的 DMA 模块有以下特点：数据传送不需要 CPU 介入，完全由 DMA 控制器自行管理。在整个地址空间范围内传输数据，块方式传输可达 65536 字节；能够提高片内外设数据吞吐能力，实现高速传输，每个字或者字节的传输仅需要 2 个 MCLK；减少系统功耗，即使在片内外设进行数据输入或输出时，CPU 也可以处于超低功耗模式而不需唤醒；字节和字数据可以混合传送：DMA 传输可以是字节到字节、字到字、字节到字或者字到字节。当字到字节传输时，只有字中较低字节能够传输，当从字节到字传输时，传输到字的低字节，高字节被自动清零；四种传输寻址模式：固定地址到固定地址、固定地址到块地址、块地址到固定地址以及块地址到块地址；触发方式灵活：边沿或者电平触发。单个、块或突发块传输模式：每次触发 DMA 操作，可以根据需要传输不同规模的数据

DMA 的四种寻址模式如下图所示：



DMA 控制器模块：3 个独立的传输通道：通道 0、通道 1 和通道 2。每个通道都有源地址寄存器、目的地址寄存器、传送数据长度寄存器和控制寄存器。每个通道的触发请求可以分别允许和禁止；可配置的通道优先权：优先权裁决模块，传输通道的优先级可以调整，对同时有触发请求的通道进行优先级裁决，确定哪个通道的优先级最高。MSP430 的 DMA 控制器可以采用固定优先级，还可以采用循环优先级。程序命令控制模块，每个 DMA 通道开始传输之前，CPU 要编程给定相关的命令和模式控制，以决定 DMA 通道传输的类型；可配置的传送触发器：触发源选择模块，DMAREQ（软件触发）、Timer_ACCR2 输出、Timer_BCCR2 输出、I2C 数据接收准备好、I2C 数据发送准备好、USART 接收发送数据、DAC12 模块 DAC12IFG、ADC12 模块的 ADC12IFGx、DMAxIFG、DMAE0 外部触发源。并且还具有触发源扩充能力。

DMA 有六种传输模式：单字或者单字节传输；块传输；突发块传输；重复单字或者单字节传输；重复块传输；重复突发块传输。前三个，传输完成后 DMAEN 自动复位；再次传输时需要重新置位 DMAEN 位以使能 DMA 通道。后三个为重复模式，一次传输完成后，DMAEN 不复位；再次出发时，可以再次启动数据传输。六种传输模式通过 DMADTx 寄存器设置：

DMADTx	Transfer Mode	Description
000	Single transfer	Each transfer requires a trigger. DMAEN is automatically cleared when DMAxSZ transfers have been made.
001	Block transfer	A complete block is transferred with one trigger.

		DMAEN is automatically cleared at the end of the block transfer.
010, 011	Burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN is automatically cleared at the end of the burst-block transfer.
100	Repeated single transfer	Each transfer requires a trigger. DMAEN remains enabled.
101	Repeated block transfer	A complete block is transferred with one trigger. DMAEN remains enabled.
110, 111	Repeated burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN remains enabled.

单字或者单字节传输：DMA 通道被定义为单字或者单字节传输模式，每个字或者字节的传输都要触发信号触发。设置 DMADTx=0 就定义了单字或者单字节传输模式，规定的传输完毕后 DMAEN 位自动清除，如果需要再次传输，必须重新置位 DMAEN。如果设置 DMADTx＝4 为重复单字或者单字节传输模式，DMAEN 位一直保持置位，每次触发伴随一次传输。DMAxSZ 寄存器保存传输的单元个数，如果该寄存器为 0，则没有传输。传输之前 DMAxSZ 寄存器的值写入到一个临时的寄存器中，每次操作之后 DMAxSZ 做减操作。当 DMAxSZ 减为零的时候，它所对应的临时寄存器将原来的值重新置入 DMAxSZ，同时相应的 DMAIFG 标志置位。

块传输模式：在块传输模式，每次触发可以传输一个数据块。设置 DMADTx=1 为块传输模式，每个数据块传输完毕，DMAEN 位自动清除，在触发传输下一个数据块之前，该位要被重新置位。在传输某个数据块期间，其他的传输请求将被忽略。设置 DMADTx=5 为重复块传输模式，某个数据块传输完毕，DMAEN 位仍然保持置位，之后，新的触发可以引起又一次数据块传送。DMAxSZ 寄存器保存数据块所包含的单元个数。DMASRCINCR 和 DMADSTINCR 反映在数据块传输过程中的目的地址和源地址的变化情况。在块传输或者重复块传输过程中，DMAxSA，DMAxDA，DMAxSZ 寄存器的值写入到对应的临时寄存器中，DMAxSA，DMAxDA 寄存器所对应的临时值在块传输过程中增加或者减少，而 DMAxSZ 在块传输过程中减计数，始终反映当前数据块还有多少单元没有传输完毕，当 DMAxSZ 减为 0，它所对应的临时寄存器将原来的值重新置入 DMAxSZ，同时相应的 DMAIFG 被置位。在块传输过程中，CPU 暂停工作，不参与数据的传输。数据块需要 2×MCLK×DMAxSZ 个时钟周期。当每个数据块传输完毕，CPU 按照暂停前的状态重新开始执行。

突发块传输模式：这个和块传输模式类似，只不过每传输 4 个字或字节，DMA 释放内部总线，CPU 运行 2 个 MCLK 周期；在传输过程中 CPU 有 20%的执行时间，而块传输需要等 DMA 完全传送完之后，CPU 方能运行。

DMA 触发源：每个通道的触发源有 DMAxTSELx 位进行控制的，这些位必须在 DMAEN 位为 0 是进行设置，否则可能出现不可预料的 DMA 触发。

DMAxTSELx	Operation
0000	DMAREQ bit (software trigger)
0001	TACCR2 CCIFG bit
0010	TBCCR2 CCIFG bit
0011	URXIFG0 (UART/SPI mode), USART0 data received (I2C mode)
0100	UTXIFG0 (UART/SPI mode), USART0 transmit ready (I2C mode)
0101	DAC12_0CTL DAC12IFG bit
0110	ADC12 ADC12IFGx bit
0111	TACCR0 CCIFG bit
1000	TBCCR0 CCIFG bit
1001	URXIFG1 bit
1010	UTXIFG1 bit
1011	Multiplier ready
1100	No action
1101	No action
1110	DMA0IFG bit triggers DMA channel 1 DMA1IFG bit triggers DMA channel 2 DMA2IFG bit triggers DMA channel 0
1111	External trigger DMAE0

另外，单片机的中断程序不影响 DMA 的传输，当 DMA 传输过程中，单片机不响应中外部 NMI 中断(必须 DMA 的控制位 ENNMI 位为 1 时响应 NMI 中断，否则不予处理)外的所有中断；必须等待 DMA 数据传送结束之后才运行系统的中断处理程序。

DMA 的中断：数据传送过程中，DMAxSZ 寄存器值减为 0 时，DMA 置位 DMAIFG，DMA 的中断和 DAC12 模块共享中断向量，使用中断时需要软件判断具体是那个中断。中断响应后 DMAIFG 不会自动复位，使用时必须软件清零 DMAIFG 位。

DMA 的寄存器如下：

Register	Short Form	Register Type	Address	Initial State
DMA control 0	DMACTL0	Read/write	0122h	Reset with POR
DMA control 1	DMACTL1	Read/write	0124h	Reset with POR
DMA channel 0 control	DMA0CTL	Read/write	01E0h	Reset with POR
DMA channel 0 source address	DMA0SA	Read/write	01E2h	Unchanged
DMA channel 0 destination address	DMA0DA	Read/write	01E4h	Unchanged

```
DMA channel 0 transfer size          DMA0SZ
Read/write       01E6h       Unchanged
DMA channel 1 control                DMA1CTL
Read/write       01E8h       Reset with POR
DMA channel 1 source address         DMA1SA
Read/write       01EAh       Unchanged
DMA channel 1 destination address    DMA1DA
Read/write       01ECh       Unchanged
DMA channel 1 transfer size          DMA1SZ
Read/write       01EEh       Unchanged
DMA channel 2 control                DMA2CTL
Read/write       01F0h       Reset with POR
DMA channel 2 source address         DMA2SA
Read/write       01F2h       Unchanged
DMA channel 2 destination address    DMA2DA
Read/write       01F4h       Unchanged
DMA channel 2 transfer size          DMA2SZ
Read/write       01F6h       Unchanged
```

有关每个寄存器的详细内容参考 ti 提供的用户指南。

- **程序实现：**

DMA 的使用主要是 DMA 寄存器的初始设置，设置完成后，DMA 接到触发信号即可自动传输数据。

设置函数如下：

```
void DMAInit(char channel,char trigger,char transMode,char
srcMode,char dstMode,
             unsigned int src,unsigned int dst,unsigned int
size)
{
    unsigned int *DMAxCTL,*DMAxSA,*DMAxDA,*DMAxSZ;

    DMACTL0 = trigger << (channel << 2);
    DMACTL1 = 0x04;           //DMA 收到触发请求时，等待当前
指令执行完成后

    switch (channel)          //选择当前设置哪个 DMA 通道
    {
        case 0:
            DMAxCTL = (unsigned int *)&DMA0CTL;
            DMAxSA = (unsigned int *)&DMA0SA;
            DMAxDA = (unsigned int *)&DMA0DA;
            DMAxSZ = (unsigned int *)&DMA0SZ;
```

```
            break;                                          //指
针 = 0 通道控制
        case 1:
            DMAxCTL = (unsigned int *)&DMA1CTL;
            DMAxSA = (unsigned int *)&DMA1SA;
            DMAxDA = (unsigned int *)&DMA1DA;
            DMAxSZ = (unsigned int *)&DMA1SZ;
            break;                                          //指
针 = 1 通道控制
        case 2:
            DMAxCTL = (unsigned int *)&DMA2CTL;
            DMAxSA = (unsigned int *)&DMA2SA;
            DMAxDA = (unsigned int *)&DMA2DA;
            DMAxSZ = (unsigned int *)&DMA2SZ;
            break;                                          //指
针 = 2 通道控制
    }

    switch (transMode)        //设置 DMA 通道的传输模式
    {
        case 'S': *DMAxCTL = DMADT_0;    break;
//单次传输
        case 's': *DMAxCTL = DMADT_4;    break;
//重复单次传输
        case 'B': *DMAxCTL = DMADT_1;    break;
//块传输
        case 'b': *DMAxCTL = DMADT_5;    break;
//重复块传输
        case 'I': *DMAxCTL = DMADT_2;    break;
//突发块传输 交错
        case 'i': *DMAxCTL = DMADT_6;    break;            //
重复突发块传输 交错
    }

    *DMAxCTL |= (srcMode & 0x04) << 2;
//源 字或字节
    *DMAxCTL |= (srcMode & 0x03) << 8;
//源 地址改变方式

    *DMAxCTL |= (dstMode & 0x04) << 3;
//目的 字或字节
    *DMAxCTL |= (dstMode & 0x03) << 10;
//目的 地址改变方式
```

```
    *DMAxSA = src;
    *DMAxDA = dst;
    *DMAxSZ = size;

    *DMAxCTL |= DMAEN;
//DMA 使能
}
```

函数比较麻烦，函数内容按参数设置每个寄存器。DMACTL0 = trigger << (channel << 2)；这个是设置对应 channel 通道的的参考源，不大明白的可以看下 DMACTL0 的寄存器内容；switch (channel)语句则根据通道设置对应指针指向的寄存器；然后对应设置参数即可。

当设置成非重复模式时，需要重新置位 DMAEN，本程序就函数 DMAReEnable 实现：

```
void DMAReEnable(char channel)
{
    switch (channel)          //使能对应通道
    {
        case 0: DMA0CTL |= DMAEN;    break;       //0 通道
        case 1: DMA1CTL |= DMAEN;    break;       //1 通道
        case 2: DMA2CTL |= DMAEN;    break;       //2 通道
    }
}
```

这个函数比较简单，只是根据传入参数设置对应通道的 DMAEN 位。

当设置为软件触发时，需要软件启动 DMA 程序如下：

```
void DMAStart(char channel)
{
    switch (channel)          //使能对应通道
    {
        case 0: DMA0CTL |= DMAREQ;   break;       //0 通道
        case 1: DMA1CTL |= DMAREQ;   break;       //1 通道
        case 2: DMA2CTL |= DMAREQ;   break;       //2 通道
    }
}
```

这个和上个函数类似：仅仅设置一个控制位，函数很简单，不再解释啦。

程序实现就这么多了，有关详细内容可以下载附件里的程序库，程序的注释很详细。

- **使用示例：**

使用这个程序时，步骤和原来的相同：工程中加入 DMA.c 文件，然后源文件中包含 DMA.h 头文件即可。

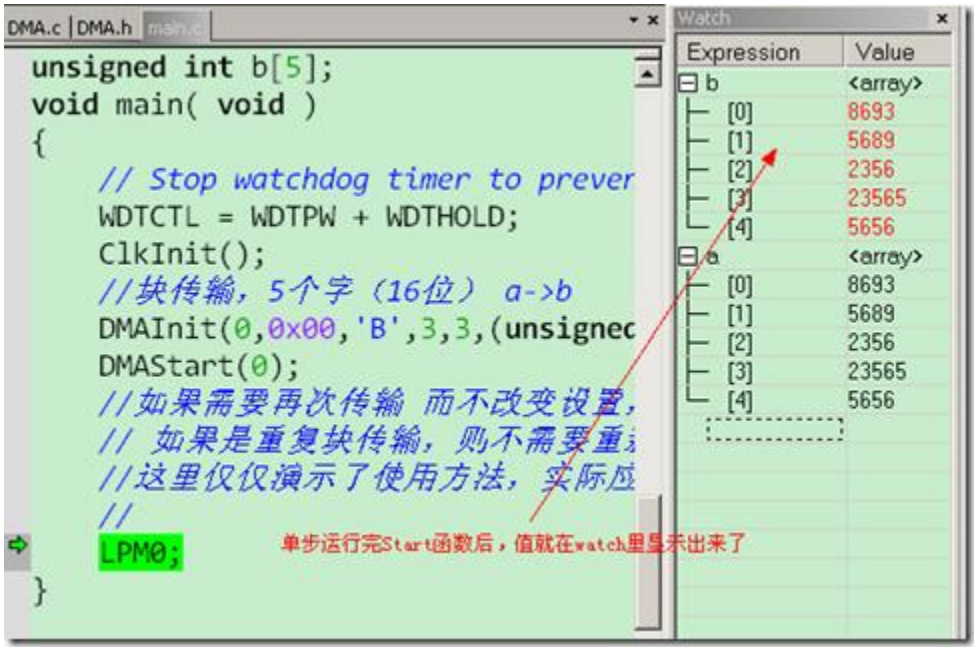
示例程序主要如下：

```
#include <msp430x16x.h>
```

```
#include "DMA.h"

unsigned int a[5] = {8693,5689,2356,23565,5656};
unsigned int b[5];
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    //块传输，5 个字（16 位） a->b
    DMAInit(0,0x00,'B',3,3,(unsigned int)a,(unsigned
int)b,5);
    DMAStart(0);
    //如果需要再次传输 而不改变设置，只需调用 DMAReEnable 再
次启动传输即可
    // 如果是重复块传输，则不需要重新使能 DMAReEnable 直接启
动即可
    //这里仅仅演示了使用方法，实际应用中，应根据需要选择适当的
触发源。
    //
    LPM0;
}
```

示例程序完成功能很简单，仅仅把一个数组的值赋给另外一个数组。数组地址即是数组名强制转换为所需类型(无符号 16 位)，传入函数初始化设置。这里为了简便，设置为软件启动。

运行效果如下：



单步运行完启动DMA传输后，结果即出来了；说明DMA传输数据的速度是很快的。

DMA 可以用于对速度要求比较高的程序中。例如：DMA 配合硬件乘法器和 ADC12 模块，可以很容易的实现比较高频率的数字滤波方案。

附件：程序库

MSP430 程序库<十五>Flash 控制器

一般，在单片机中的 Flash 存储器用于存放程序代码，属于只读型存储器。而在 MSP430 些列的单片机中，都可以通过内置的 Flash 控制器擦除或改写任何一段的内容。另外，msp430 的单片机内部还专门留有一段 Flash 区域(information memory)，用于存放掉电后需要永久保存的数据。利用 430 内部的 Flash 控制器，可以完成较大容量的数据记录、用户设置参数在掉电后的保存等功能。

1．**硬件介绍：**

要对 Flash 读写，首先要了解 MSP430 的存储器组织。430 单片机的存储器组织结构采用冯诺依曼结构，RAM 和 ROM 统一编址在同一寻址空间中，没有代码空间和数据空间之分。

一般 430 的单片机都统一编址在 0-64k 地址范围中，只有少数高端的型号才能突破 64k（如：FG461x 系列）。绝大多数的 msp430 单片机都编址在 64kB 范围内。地址的大概编码方式如下：

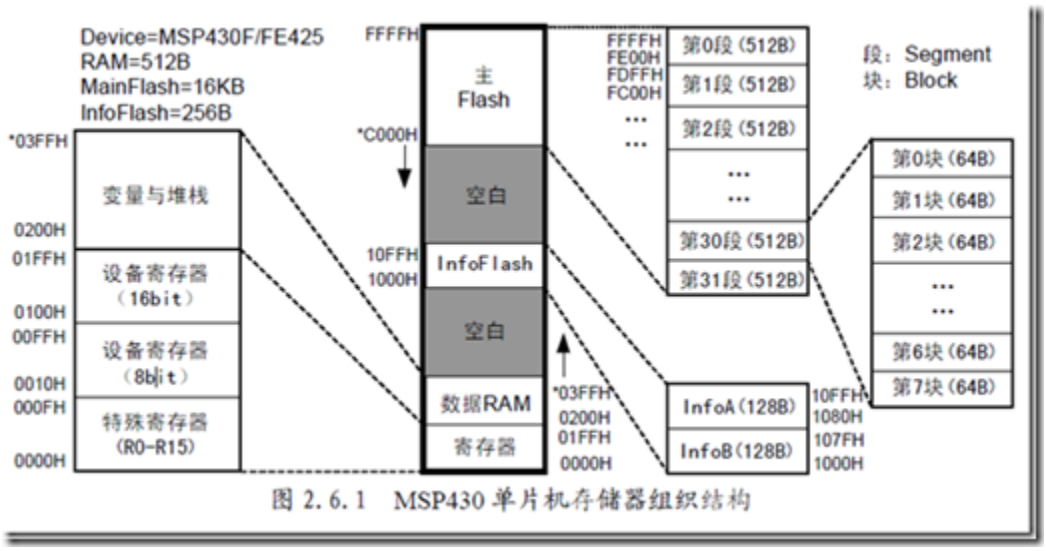


图 2.6.1　MSP430 单片机存储器组织结构

这是 msp430f425 的存储器分配图，其他在 64k 范围内的存储器的单片机编址方式与此类似：低 256B 是寄存器区，然后是 RAM；空白；1000H 到 10FFH 是信息 Flash 区；大于 1100H-0FFFFH 是主存储器区(从 0FFFFH 开始往低地址有单片机的主 Flash，多余的部分空白)。

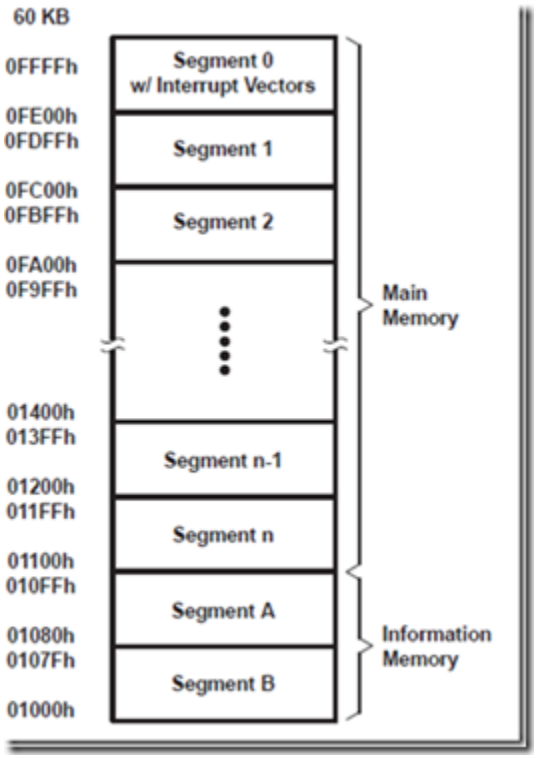
MSP430F14x 的 Flash 分布：

		MSP430F133	MSP430F135	MSP430F147 MSP430F1471	MSP430F148 MSP430F1481	MSP430F149 MSP430F1491
Memory Main: interrupt vector Main: code memory	Size Flash Flash	8KB 0FFFFh – 0FFE0h 0FFFFh – 0E000h	16KB 0FFFFh – 0FFE0h 0FFFFh – 0C000h	32KB 0FFFFh – 0FFE0h 0FFFFh – 08000h	48KB 0FFFFh – 0FFE0h 0FFFFh – 04000h	60KB 0FFFFh – 0FFE0h 0FFFFh – 01100h
Information memory	Size Flash	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h
Boot memory	Size ROM	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h
RAM	Size	256 Byte 02FFh – 0200h	512 Byte 03FFh – 0200h	1KB 05FFh – 0200h	2KB 09FFh – 0200h	2KB 09FFh – 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h

MSP430F16x 的 Flash 分布：

		MSP430F167	MSP430F168	MSP430F169
Memory Main: interrupt vector Main: code memory	Size Flash Flash	32KB 0FFFFh – 0FFE0h 0FFFFh – 08000h	48KB 0FFFFh – 0FFE0h 0FFFFh – 04000h	60KB 0FFFFh – 0FFE0h 0FFFFh – 01100h
Information memory	Size Flash	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h
Boot memory	Size ROM	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h
RAM	Size	1KB 05FFh – 0200h	2KB 09FFh – 0200h	2KB 09FFh – 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h

主 Flash 部分和信息 Flash 部分如下(60kB Flash 对应的单片机，如 msp430f149、msp430f149)：

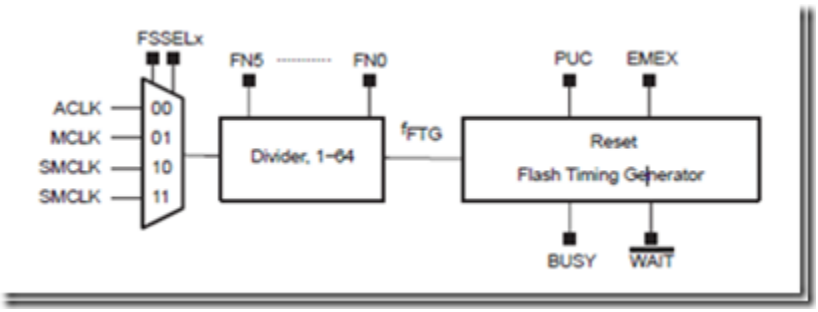


主 Flash 分为以 512B 为段的单位，0 段是单片机中断向量等程序入口地址，使用时不要擦除此段或改写此段，若要擦除或是改写，请先保存内容到 RAM 或其他段；主 Flash 各段内容均要避免写入或擦除，以免造成不可预料的后果。

信息 Flash 分为两段：段 A 和段 B，每段 128B；可以保存用户自己的内容(主 Flash 也可以但是要避免与程序代码区冲突)；这里就把信息 Flash 的两段称为 InfoA(1080H-10FFh)和 InfoB(1000H-10FFH)。

Flash 的操作包括：字或字节写入；块写入；段擦除；主 Flash 擦除；全部擦除。任何的 Flash 操作都可以从 Flash 或从 RAM 中运行。

Flash 操作时需要时序发生器，Flash 控制器内部含有时序发生器用以产生所需的 Flash 时钟，Flash 时钟的范围必须在 257kHz 到 476kHz 之间。时序发生器的框图如下：

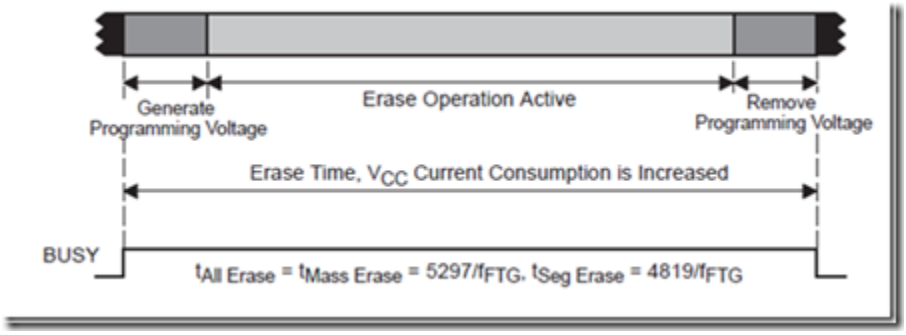


时序发生器可以选择 ACLK、MCLK、SMCLK 作为时钟源，通过分频获得所需的 257kHz 到 476kHz 之间的 Flash 操作时钟。如果时钟频率不再这个范围内，将会产生不可预料的结果。

擦除：擦除之后，存储器中的 bit 都变为 1；Flash 中的每一位都可以通过编程写入有 1 到 0，但是要想由 0 变为 1，必须通过擦除周期。擦除的最小单位是段。有三种擦除模式：

MERAS	ERASE	Erase Mode
0	1	Segment erase
1	0	Mass erase (all main memory segments)
1	1	Erase all flash memory (main and information .segments)

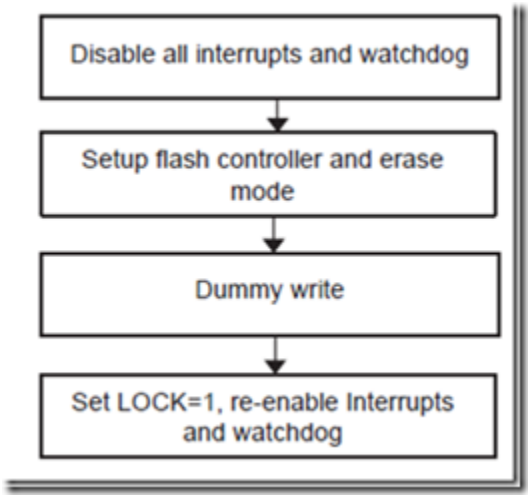
可以通过 MERAS、ERASE 位来设置擦除的模式：段擦除，主 Flash 擦除，全部擦除。对要擦除段内的一个地址空写入启动擦出周期：空写入可以启动时序发生器和擦除操作。空写入后 BUSY 位立即变高直到擦除周期结束，这一位变为低(0)。BUSY, MERAS 和 ERASE 位在擦除周期结束后会自动复位。擦除周期的时间和要擦出的 Flash 大小无关，每次擦除的时间对于 MSP430F1xx 系系列单片机来说，所需时间是一样的。擦除的时序如下：



当空写入到的地址不在要擦除的段地址范围内的时候，空写入无效，直接被忽略。在擦除周期内，应该关中断，直到擦除完成，重新开中断，擦除期间的中断已经置标志位，开中断后立即响应。

从 Flash 中启动的擦除操作：擦除操作可以从 Flash 中启动或是从 RAM 中启动。当操作是从 Flash 中启动的时候，Flash 控制器控制了操作时序，CPU 运行被暂停直到擦除结束。擦除周期结束后，CPU 继续执行，从空写入之后的指令开始运行。当从 Flash 中启动擦除操作时，可以擦除即将运行的程序所在的段，如果擦除了即将运行的程序所在的 Flash 段时，擦除结束后，CPU 的运行不可预料。

从 Flash 启动时擦除周期如下：

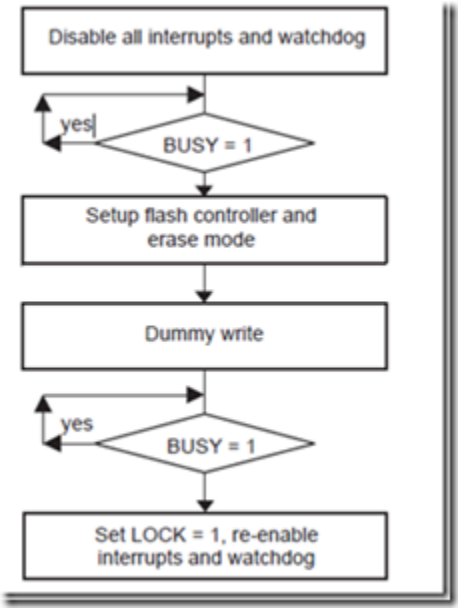


用户指南里面的示例汇编程序如下：

```
; Segment Erase from flash. 514 kHz < SMCLK < 952 kHz
; Assumes ACCVIE = NMIIE = OFIE = 0.
MOV #WDTPW+WDTHOLD,&WDTCTL ; Disable WDT
DINT //; Disable interrupts
MOV #FWKEY+FSSEL1+FN0,&FCTL2 ; SMCLK/2
MOV #FWKEY,&FCTL3 ; Clear LOCK
MOV #FWKEY+ERASE,&FCTL1 ; Enable segment erase
CLR &0FC10h ; Dummy write, erase S1
MOV #FWKEY+LOCK,&FCTL3 ; Done, set LOCK
... ; Re-enable WDT?
EINT ; Enable interrupts
```

从 RAM 中启动擦除操作：任意擦除周期都可以从 RAM 启动，这时 CPU 不再暂停而是继续从 RAM 中运行接下来的程序。CPU 可以访问任何 Flash 地址之前，必须检查 BUSY 位以确定擦除周期结束。如果 BUSY = 1 访问 Flash，这是一个访问冲突，这时 ACCVIFG 将被设置，而擦除的结果将是不可预测的的。

从 RAM 中启动擦除操作时，过程如下：



要在擦除之前确认没有访问 Flash，然后擦除完成之前不允许访问 Flash。

```
; Segment Erase from RAM. 514 kHz < SMCLK < 952 kHz
; Assumes ACCVIE = NMIIE = OFIE = 0.
MOV #WDTPW+WDTHOLD,&WDTCTL   ; Disable WDT
DINT                         ; Disable interrupts
L1 BIT #BUSY,&FCTL3          ; Test BUSY
JNZ L1                       ; Loop while busy
MOV #FWKEY+FSSEL1+FN0,&FCTL2; SMCLK/2
MOV #FWKEY,&FCTL3            ; Clear LOCK
MOV #FWKEY+ERASE,&FCTL1      ; Enable erase
CLR &0FC10h                  ; Dummy write, erase S1
L2 BIT #BUSY,&FCTL3          ; Test BUSY
JNZ L2                       ; Loop while busy
MOV #FWKEY+LOCK,&FCTL3       ; Done, set LOCK
...                          ; Re-enable WDT?
EINT                         ; Enable interrupts
```

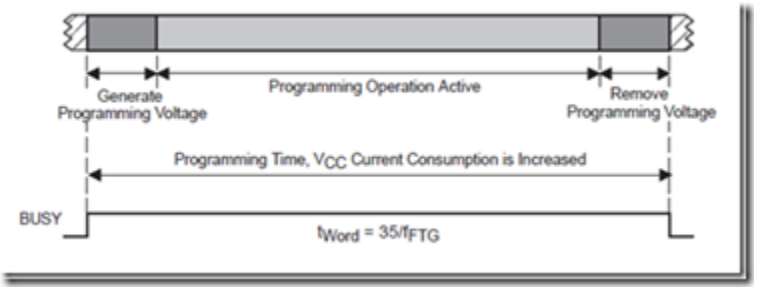
写 Flash 操作：写入的模式由 WRT 和 BLKWRT 位来确定：

BLKWRT	WRT	Write Mode
0	1	Byte/word write
1	1	Block write

这两种模式中块写入大约是字或字节写操作时的两倍快，因为在块写入完成之前，变成电压一直维持直到块写入完成。同一个位置不能在擦除周期之前写入两次或以上，否则将发生数据损坏。写操作时，BUSY 位被置 1，写入完成后，BUSY 被自动清零。如果写操作是从 RAM 发起的，在 BUSY=1 时，程序不能访问 Flash，否则会发生访问冲突，置位 ACCVIFG，Flash 写入操作不可以预料。

字或字节写入：字或字节写入可以从 Flash 内部发起，也可以从 RAM 中发起。如果是从 Flash 中启动的写操作，时序将由 Flash 控制，在写入完成之前 CPU 运行将被暂停。写入完成后 CPU 将继续运行。

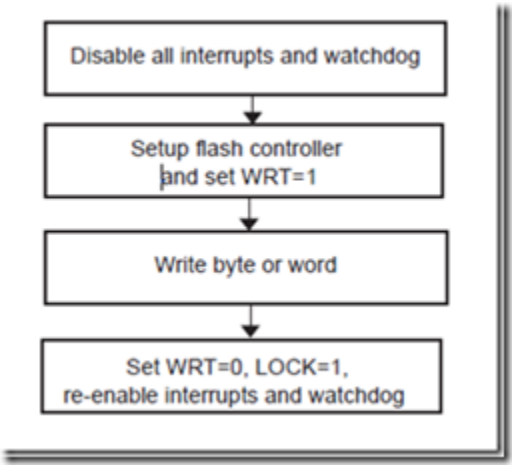
操作时序如下：



若是从 RAM 中启动写 Flash，程序将继续从 RAM 中运行。CPU 再次访问 Flash 之前必须确认 BUSY 位已经清零，否则会发生访问冲突，置位 ACCVIFG，写入的结果将不可预料。

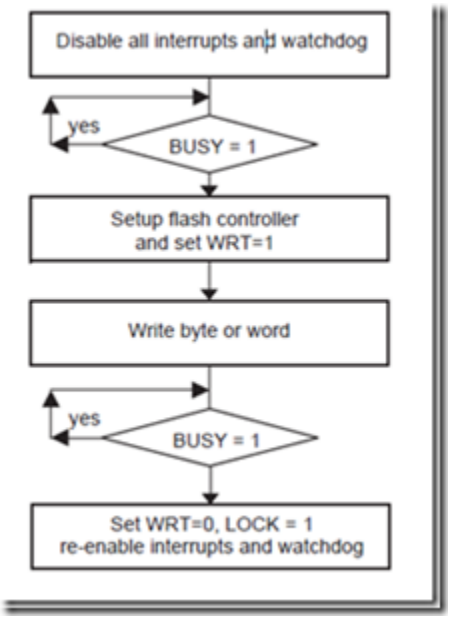
字或字节写入模式下，内部产生的编程电压时适用于完整的 64 个字节块的写入
In byte/word mode, the internally-generated programming voltage is applied
to the complete 64-byte block, each time a byte or word is written, for 32 of the
35 fFTG cycles. With each byte or word write, the amount of time the block is
subjected to the programming voltage accumulates. The cumulative
programming time, tCPT, must not be exceeded for any block. If the cumulative
programming time is met, the block must be erased before performing any
further writes to any address within the block.

从 Flash 发起写字节或字时：



```
; Byte/word write from flash. 514 kHz < SMCLK < 952 kHz
; Assumes 0FF1Eh is already erased
; Assumes ACCVIE = NMIIE = OFIE = 0.
MOV #WDTPW+WDTHOLD,&WDTCTL        ; Disable WDT
DINT ; Disable interrupts
MOV #FWKEY+FSSEL1+FN0,&FCTL2      ; SMCLK/2
MOV #FWKEY,&FCTL3                 ; Clear LOCK
MOV #FWKEY+WRT,&FCTL1             ; Enable write
MOV #0123h,&0FF1Eh                ; 0123h -> 0FF1Eh
MOV #FWKEY,&FCTL1                 ; Done. Clear WRT
MOV #FWKEY+LOCK,&FCTL3            ; Set LOCK
...                               ; Re-enable WDT?
EINT                              ; Enable interrupts
```

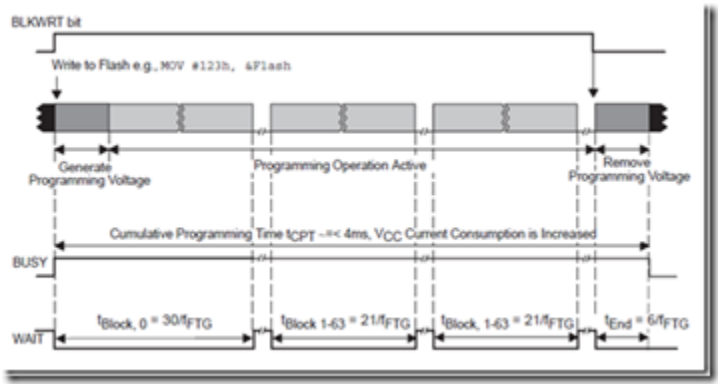
从 RAM 中启动写入操作时：



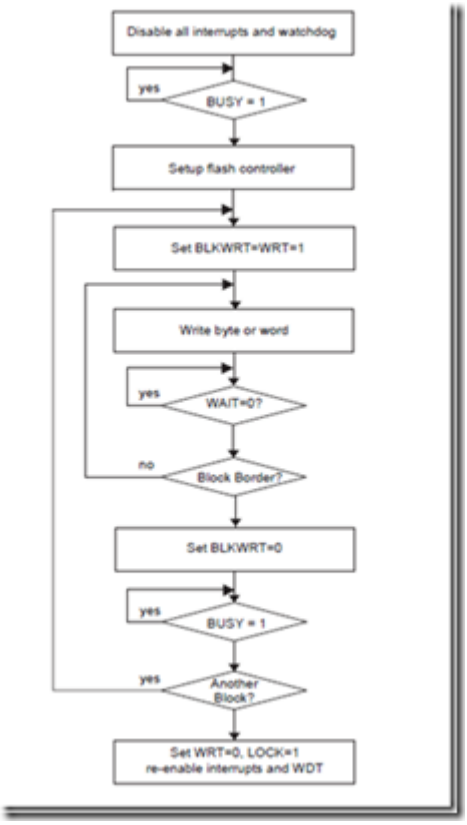

```
; Byte/word write from RAM. 514 kHz < SMCLK < 952 kHz
; Assumes 0FF1Eh is already erased
; Assumes ACCVIE = NMIIE = OFIE = 0.
MOV #WDTPW+WDTHOLD,&WDTCTL        ; Disable WDT
DINT                              ; Disable interrupts
L1 BIT #BUSY,&FCTL3               ; Test BUSY
JNZ L1                            ; Loop while busy
MOV #FWKEY+FSSEL1+FN0,&FCTL2      ; SMCLK/2
MOV #FWKEY,&FCTL3                 ; Clear LOCK
MOV #FWKEY+WRT,&FCTL1             ; Enable write
MOV #0123h,&0FF1Eh                ; 0123h -> 0FF1Eh
L2 BIT #BUSY,&FCTL3               ; Test BUSY
JNZ L2                            ; Loop while busy
MOV #FWKEY,&FCTL1                 ; Clear WRT
```

```
MOV #FWKEY+LOCK,&FCTL3              ; Set LOCK
...                                 ; Re-enable WDT?
EINT                                ; Enable interrupts
```

块写入：当需要写入连续的多个字或字节时，块写入能够能够提高 Flash 访问速度。块写入时，内部产生的编程电压一直存在在 64 个字节的块写入期间。块写入不能有 Flash 存储器内启动，必须从 RAM 中发起块写入操作。块写入期间，BUSY 位被置位。在写入每个字节或字时必须检测 WAIT 位。下一个字或字节能够被写入时，WAIT 位被置位。



块写入的过程如下：




```
; Write one block starting at 0F000h.
; Must be executed from RAM, Assumes Flash is already
    erased.
```

```
; 514 kHz < SMCLK < 952 kHz
; Assumes ACCVIE = NMIIE = OFIE = 0.
MOV #32,R5                          ; Use as write counter
MOV #0F000h,R6                      ; Write pointer
MOV #WDTPW+WDTHOLD,&WDTCTL          ; Disable WDT
DINT                                ; Disable interrupts
L1 BIT #BUSY,&FCTL3                 ; Test BUSY
JNZ L1 ; Loop while busy
MOV #FWKEY+FSSEL1+FN0,&FCTL2        ; SMCLK/2
MOV #FWKEY,&FCTL3                   ; Clear LOCK
MOV #FWKEY+BLKWRT+WRT,&FCTL1        ; Enable block
   write
L2 MOV Write_Value,0(R6)            ; Write location
L3 BIT #WAIT,&FCTL3                 ; Test WAIT
JZ L3                               ; Loop while WAIT=0
INCD R6                             ; Point to next word
DEC R5                              ; Decrement write counter
JNZ L2                              ; End of block?
MOV #FWKEY,&FCTL1                   ; Clear WRT,BLKWRT
L4 BIT #BUSY,&FCTL3                 ; Test BUSY
JNZ L4                              ; Loop while busy
MOV #FWKEY+LOCK,&FCTL3              ; Set LOCK
...                                 ; Re-enable WDT if needed
EINT                                ; Enable interrupts
```

当任何写或擦除操作是从 RAM 启动，而 BUSY = 1，CPU 不能读取或写入或从任何 Flash 位置。否则，发生访问冲突，ACCVIFG 设置，结果是不可预知的。此外，如果闪存写入让 WRT= 0，ACCVIFG 中断标志设置，Flash 不受影响。

如果写入或擦除操作时从 Flash 启动的，CPU 访问下一条指令时(从 Flash 读取指令)，Flash 控制器返回 03FFFH 给 CPU；03FFFH 是指令 JMP PC，这让 CPU 一直循环直到 Flash 操作完成。Flash 写入或擦除操作完成后，允许 CPU 继续访问接下来的指令。

当 BUSY=1 时，Flash 访问时：

Flash Operation	Flash Access	WAIT	Result
Any erase, or Byte/word write	Read	0	ACCVIFG = 0. 03FFFh is the value read
	Write	0	ACCVIFG = 1. Write is ignored
	Instruction fetch	0	ACCVIFG = 0. CPU fetches 03FFFh. This is the JMP PC instruction.
Block write	Any	0	ACCVIFG = 1, LOCK = 1
	Read	1	ACCVIFG = 0, 03FFFh is the value read
	Write	1	ACCVIFG = 0, Flash is written
	Instruction fetch	1	ACCVIFG = 1, LOCK = 1

在开始 Flash 操作之前，需要停止所有的中断源。如果在 Flash 操作期间有中断响应，读中断服务程序的地址时，将收到 03FFFH 作为中断服务程序的地址。如果 BUSY=1；CPU 将一直执行难 IMP PC 指令；Flash 操作完成后，将从 03FFFH 执行中断服务程序而不是正确的中断程序的地址。

停止写入或擦除：任何写入和擦除操作都可以在正常完成之前，通过设置紧急退出位 EMEX 退出操作。设置 EMEX 时，立即停止当前活动的操作，停止 Flash 控制器；所有的 Flash 操作停止，Flash 返回可读模式，FCTL1 的所有位复位；操作的结果不可预料。

设置和访问 Flash 控制器：FCTLx 是 16 位的、密码保护的、可读写的寄存器。写入这些寄存器都必须在高位包含密码 0A5H，如果写入的不是 0A5H，将会引起复位。读寄存器时高位读出的是 96H。

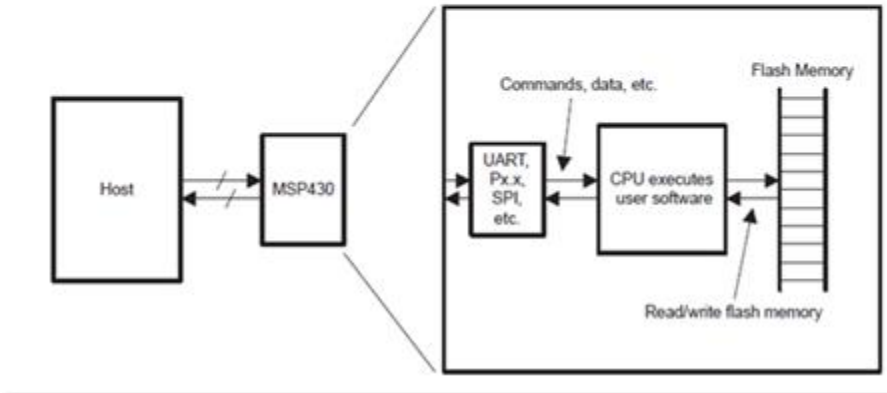
在擦除或写入字或字节时写 FCTL1 寄存器将会引起访问冲突，置位 ACCVIFG.块写入时，WAIT=1 时可以写 FCTL1 寄存器，当 WAIT=0 时写 FCTL1 寄存器是访问冲突，置位 ACCVIFG。BUSY=1 时，所有写入 FCTL2 寄存器都是访问冲突。BUSY=1 时，所有的 FCTLx 都可以读操作，不会引起访问冲突。

Flash 的中断：Flash 控制器有两个中断源：KEYV，和 ACCVIFG。ACCVIFG 在访问冲突的时候被置位。当 ACCVIE 在 Flash 操作完成后被重新使能后 ACCVIFG 会引起中断请求。ACCIFG 和 NMI 同样的中断向量，所以这个中断不需要 GIE 位允许即可产生中断请求。必须通过软件检测 ACCVIFG 位，以确定发生了访问冲突；ACCVIFG 位必须软件复位。KEYV 是关键值错误当写 Flash 的寄存器时没有写正确的高位密码时被置位，这是会立刻引起 PUC 信号复位整个硬件。

编程 Flash 的硬件：编程 430 的 Flash 内容有三种选择，通过 JTAG、通过 BSL 和用户定制。用户定制即是通过单片机的程序访问自己的 Flash。

The ability of the MSP430 CPU to write to its own flash memory allows for in-system and external custom programming solutions as shown in Figure 5–12. The user can choose to provide data to the MSP430 through any means available (UART, SPI, etc.). User-developed software can receive the data and program the flash memory. Since this type of solution is developed by the user, it can be completely customized to fit the application needs for programming, erasing, or updating the flash memory.

Figure 5–12. User-Developed Programming Solution



Flash 的寄存器列表如下：

```
Register                                    Short Form
    Register Type     Address       Initial State
Flash memory control register 1       FCTL1
    Read/write        0128h         09600h with PUC
Flash memory control register 2       FCTL2
    Read/write        012Ah         09642h with PUC
Flash memory control register 3       FCTL3
    Read/write        012Ch         09618h with PUC
Interrupt Enable 1                    IE1
    Read/write        000h          Reset with PUC
```

Flash 的硬件部分就介绍这么多了，有什么不大懂的地方请参考 TI 提供的用户指南。

2. **程序实现：**

首先设置 Flash 的时钟，初始化 Flash 控制器：

```
void FlashInit()
{
    FCTL2 = FWKEY + FSSEL_2 + FN1; // 默认 SMCLK/3
   =333KHz
}
```

这个函数仅仅设置了时钟。

擦除函数：

```
void FlashErase(unsigned int Addr)
{
    char *FlashPtr;
    FlashPtr = (char *)Addr;
    FCTL1 = FWKEY + ERASE; // Set Erase bit
    FCTL3 = FWKEY;      // Clear Lock bit
    DINT;
    *FlashPtr = 0; // Dummy write to erase Flash segment B
    WaitForEnable();                //Busy
    EINT;
    FCTL1 = FWKEY;                  // Lock
    FCTL3 = FWKEY + LOCK;   // Set Lock bit
}
```

这个和上面给出的流程一样，参数是要被擦除的段的首地址。WaitForEnable 函数等等待 BUSY 标志变回零即操作完成。

```
void WaitForEnable()
{
    while((FCTL3 & BUSY) == BUSY); //Busy
}
```

写入字节：

```
void FlashWriteChar(unsigned int addr,char Data)
{
    char *FlashPtr = (char *)addr; // Segment A pointer
    FCTL1 = FWKEY + WRT;// Set WRT bit for write operation
    FCTL3 = FWKEY;                // Clear Lock bit
    DINT;
    *FlashPtr = Data;      // Save Data
    WaitForEnable();        //Busy
    EINT;
    FCTL1 = FWKEY;                  // Clear WRT bit
    FCTL3 = FWKEY + LOCK;     // Set LOCK bit
}
```

写入字：

```
void FlashWriteWord(unsigned int addr,unsigned int Data)
{
    unsigned int *FlashPtr = (unsigned int *)addr;
    FCTL1 = FWKEY + WRT; // Set WRT bit for write
   operation
    FCTL3 = FWKEY;          // Clear Lock bit
    DINT;
    *FlashPtr = Data;          // Save Data
    WaitForEnable();             //Busy
    EINT;
    FCTL1 = FWKEY;               // Clear WRT bit
    FCTL3 = FWKEY + LOCK;      // Set LOCK bit
}
```

写入字或字节两个函数差别仅仅是指针类型不同。

读取字或字节：

```
char FlashReadChar(unsigned int Addr)
{
    char Data;
    char *FlashPtr = (char *) Addr;
    Data = *FlashPtr;
    return(Data);
}

unsigned int FlashReadWord(unsigned int Addr)
{
    unsigned int Data;
    unsigned int *FlashPtr = (unsigned int *)Addr;
```

```
    Data = *FlashPtr;
    return(Data);
}
```

这两个函数的差别也是仅仅指针类型不同。

这些函数和前面硬件介绍部分的程序流程相同，这里不再详细说明。

3. **使用示例：**

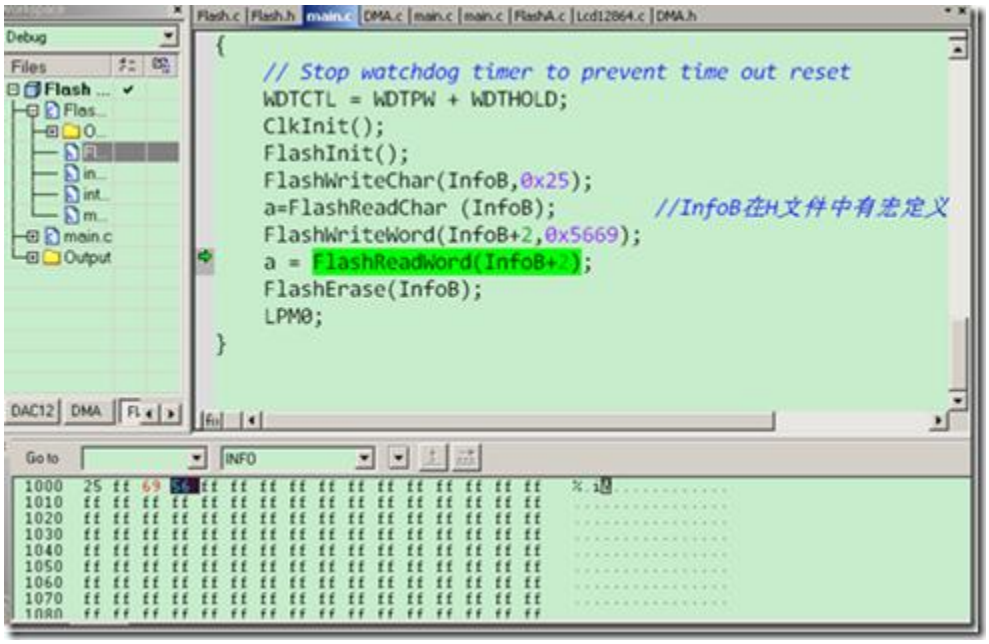
使用方法和之前的一样，工程中加入 C 文件，源代码文件中文件包含 H 文件，即可使用，具体参考示例项目：

演示主要程序主要如下：

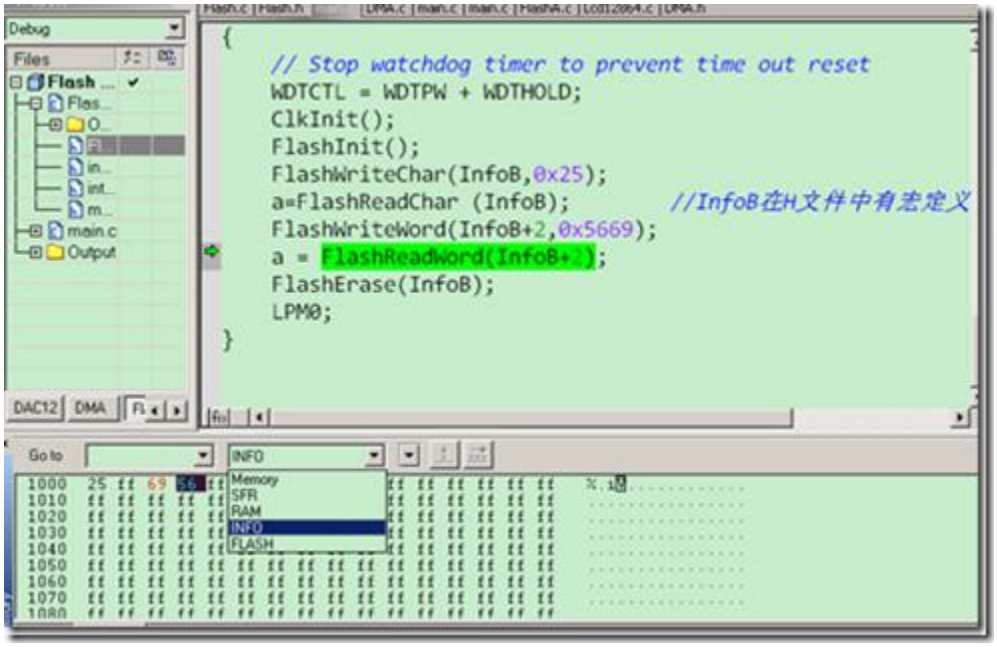
```
#include <msp430x16x.h>
#include "Flash.h"

int a;
void main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    ClkInit();
    FlashInit();
    FlashWriteChar(InfoB,0x25);
    a=FlashReadChar (InfoB);      //InfoB 在 H 文件中有宏定义
    FlashWriteWord(InfoB+2,0x5669);
    a = FlashReadWord(InfoB+2);
    FlashErase(InfoB);
    LPM0;
}
```

这里向 InfoB(1000h)首地址开始写数据，先写一个字节 再写入一个字（注意写入字时，必须是偶数地址，奇数地址会写在这个地址所在的前一个偶数地址），读出，然后擦除。这里的程序都是在 Flash 中运行的，没有演示 RAM 中运行的程序。如果在 RAM 运行程序，则需要先把程序从 Flash 中移到 RAM 中，然后跳转到 RAM 中运行。

调试截图如下：



调试时，view-Memory 菜单，调出存储器窗口；观察 Flash 内容。



这里写入的是 Info Flash 部分，观察这部分的结果，和写入的结果同。

Flash 程序控制器的程序就到这儿了。Flash 可以用于存储长期保存的数据。有什么不足之处，欢迎讨论。

附件：程序库

作者：**给我一杯酒**

出处：http://Engin.cnblogs.com/

MSP430 程序库<十六>总结

全国电子设计竞赛结束已经有一段时间了，这个程序库也就到这儿了，程序库已经包含 msp430f14x msp430f16x 系列的单片机大多数的片内资源。目录如下：

这个程序库的程序和有关资料可以用 svn check-out 地址为：svn://www.oksvn.com/msp430，这里；也可以从程序库下载。

程序库的内容就到这里了，多谢广大网友的支持啦；有什么不足之处，欢迎拍砖。