MSP430 系列超低功耗单片机系统

设计与实践

The Practice of MSP430 Ultra-Low Power System Design

编　写：　谢　楷

西安电子科技大学　　测控技术与仪器教研中心

目　录

前　言

70 年代末，处理器和相关外设(RAM/ROM 等)被集成在了一个芯片上，“单芯片计算机”诞生了，这就是后来被广泛应用的“单片机”。它将布满整个电路板的数字芯片全部集成在了一个芯片上，从而结束了“单板机”的历史。

在 90 年代中期以前，大规模数字电路和精密模拟电路集成在一个芯片内被认为是不可能的事情。但随着半导体技术的发展，越来越多的模拟电路能够和处理器集成在同一块芯片上。其中包括许多高性能的模拟设备，如 16 位 ADC，12 位 DAC 等。这些技术进步诞生了混合型微处理器，使得单片机不仅能够处理数字信号，还能够采集、处理、输出各种模拟量，甚至在内部通过软件配置运放参数，用软件直接处理模拟信号。在一个芯片上可以构成完整的测控系统。这意味着单片机系统正向“单芯片系统”(SOC)方向发展。

MSP430 系列单片机是 TI（德州仪器）公司近年来推出的一系列优秀的混合型微处理器，它不仅具有 16 位高效的处理器系统，还具有丰富的、功能强大的外设，其中包括许多高性能模拟外设。TI 公司每年都会推出数款新型的单片机，从而不断扩充 MSP430 家族系列的成员，在大部分热门产品应用中都可以单芯片完成设计。更可贵的是它能够以极低的功耗运行，因而被广泛应用在电池供电的手持设备上。即使在某些不需要低功耗的场合，MSP430 单片机仍然可以作为一款高性能单片机使用。

MSP430 单片机 2001 年进入中国以来，迅速地得以广泛应用。但目前实用参考资料较少，基本以翻译《User Guide》为蓝本。且很少有低功耗系统设计与编程方面的指导书籍。事实上， 430 的低功耗性能大部分靠软件来实现，本书将用两章的篇幅讲解 430 单片机软件设计的基本思想。

笔者以 MSP430FE425 单片机为核心，为读者设计了一款学习板(MAGIC-430)。本书中所有范例均可在学习板上验证。本书光盘内附带的软件包括 MSP430F4XX 系列单片机有所有资源的使用范例、模块化库程序以及实用程序范例，所有程序均为 100%注释的商业级代码。特别是附带的模块化程序库，读者可以在不需要了解底层硬件和寄存器的情况下，通过调用模块库内的函数来快速完成设计任务。

笔者假定读者均有 C 语言入门基础，曾经学习或接触过一款单片机（如 51 单片机）。对于购买了学习板的读者，本书可以作为实践指导用。对于一般读者，本书也可作为提高设计能力的参考书使用。

第一章　MSP430 单片机入门基础

本章将从零开始，介绍 MSP430 单片机开发的步骤、开发软件使用方法、编程语言、编译环境的基本设置、以及软件工程基础知识。本章末尾特别加入了代码风格、可移植性与软件管理方面的基本知识。国内教材、参考书对这方面内容介绍较少，而本书将这些内容放在第一章的目的在于让读者了解：养成良好的编程习惯和软件思想才是好的开端。

1．1　初识 430 单片机

在开始学习 430 单片机之前，先来做一个实验，直观的感受一下 430 单片机的“超低功耗”称号。准备一个柠檬(橙子、苹果、猕猴桃等酸性水果均可)；找几个废锌锰干电池，拆下外壳的锌皮并剪开，压平；再准备三把铜钥匙（或铜币）。

将水果切成 3 份，在果肉两侧分别贴上锌皮和铜钥匙，构成原电池：铜是正极，锌皮是负极，水果的酸性成分构成电解质。实测每节水果电池电压约 0.8V 左右。三节电池电池串联达到 2.4V 即可为 430 单片机系统供电。

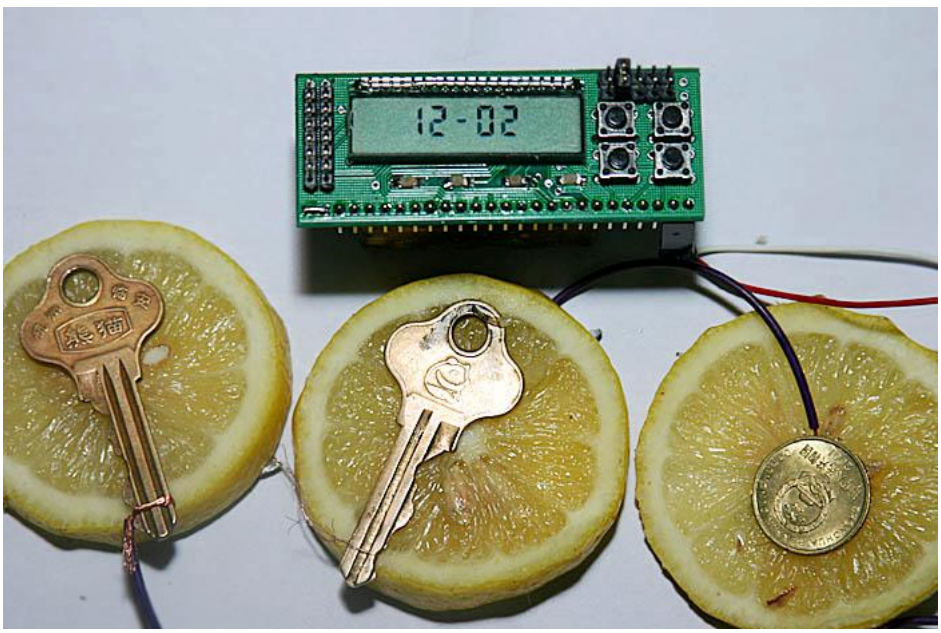


图 1.1.1　用水果电池供电的某 430 单片机系统

在 430 单片机上运行一个电子表程序。只要水果不腐烂，能运行一个月以上，直到水果干枯。我们知道原电池只能提供极其微弱的电流（实测短路电流不到 100uA）；而在上述 430 单片机系统中，运行一个电子表程序的 CPU 工作耗电不到 2uA，加上液晶耗电约 3uA，因此总电流不超过 5uA。对于一个水果电池来说，提供的电能绰绰有余。

再举个更直观的例子说明 5uA 耗电的概念：一节普通的 CR2032 纽扣电池，能够为这

个 430 单片机运行电子表程序提供 5 年的电力。

在 MAGIC430 学习板上，用到的单片机型号是 MSP430FE425。我们再以该单片机为例，来了解一下 430 单片机丰富的外设和强大功能。

MSP430FE425 单片机内部资源：

- 16 位 RISC 指令集处理器，速度可达 8 百万条指令/秒
- 512 字节 RAM（数据） + 16K 字节 Flash 存储器（代码）
- 2 段 128 字节的 InfoFlash，可作掉电存储器使用
- 内置 Flash 控制器，所有程序未用 Flash 均可作数据存储用
- 内置时钟管理单元(FLL+)，可软件设置 CPU 时钟倍频。
- 三个独立的 16 位高精度 ADC
- 内置可编程增益放大器(1~32 倍)
- 内置温度传感器
- 内置 1.2V 基准源和输出缓冲器
- 内置 128 段 LCD 驱动器，可直接驱动段码液晶板
- 1 个增强型 UART 串口，自带波特率发生器，支持 7/8 位数据、支持奇偶校验、支持 9 位地址模式、带自动桢判别功能、可配置成 SPI 模式…
- 内置看门狗，也可配置成定时器使用
- BasicTimer 定时器，方便产生 $1/2^n$ 秒定时
- 16 位 TA 定时器，带 3 路捕获和 2 路 PWM 发生器
- 自带 BOR 检测电路，能自动躲避上电瞬间的毛刺并产生可靠的复位信号。
- 14 个双向 IO 口，每个 IO 口均可作为中断源。
- 自带仿真调试功能和调试接口，支持 2 个断点。
- 内置电能计量模块，可无需 CPU 干预的情况下自动完成交流电压、电流有效值、功率、有功电能等测量和计量工作(和 ADC 不能同时用)

从上面的资源列表中，我们看到 430 单片机的特点之一就是资源极其丰富。不同的模块组合，就构成不同的单片机型号。当然，模块越多，容量越大的单片机价格越高。在单片机选型的时候，可根据实际项目的需要，选择最合适的型号。即使有些资源浪费不用，成本也比自己用分立芯片搭建系统要低得多！比如上述芯片的官方报价 4.9 美元(折合人民币 35 元)，而要自行搭建这么庞大的系统，成本将超过 300 元。

430 系列单片机详细的型号列表、资源列表和参考价格可以查阅每年的《430 单片机选型手册》（登陆 TI 官方网站 www.ti.com 下载）。每年都会有新的型号或新的系列推出，也会有新的模块被开发出来，从而为 430 系列单片机家族不断注入新的生命力。

1.2 430 单片机开发软件和工具入门

430 开发软件国内普及的种类不多，主要有 IAR 公司的 Embedded WorkBench for MSP430（以下简称 EW430）和 AQ430 两大类。目前 IAR 的用户居多，本书所有的例程

也以 IAR EW430 作为开发环境。IAR EW430 软件提供了工程管理、程序编辑、代码下载、调试等所有功能，还提供了一个针对 430 处理器的编译器（ICC430 编译器）。整个开发过程所需的全部功能由一套软件全部提供，这类软件被称为“集成开发环境”，简称 IDE（Intergrated Develop Enviroment）。而且 IAR EmbeddedWorkBench 系列开发软件涵盖了目前大部分主流的 8/16/32 位处理器系统，软件界面和操作方法保持不变。只要学会其中一种，就可以很顺利地过渡到另一种处新理器的开发工作(如 IAR EmbeddedWorkBench for ARM 等)。

读者可以登陆 IAR 的官方网站(www.iar.com)下载各种开发工具的试用版，一般都提供 2 种试用版本：4K 限制版和 1 个月限制版。前者可以永久免费使用，但编译生成的机器码不允许超出 4K，否则不予编译。后者无代码大小限制，但只能试用 1 个月。

IAR EW430 安装运行后界面如下，和大部分的开发工具一样，它也采用了 VC 风格的界面。位于主窗体左侧的是工程管理窗，负责工程内文件的添、删除、管理、浏览等。右侧是编辑窗，在该窗口内编写程序。

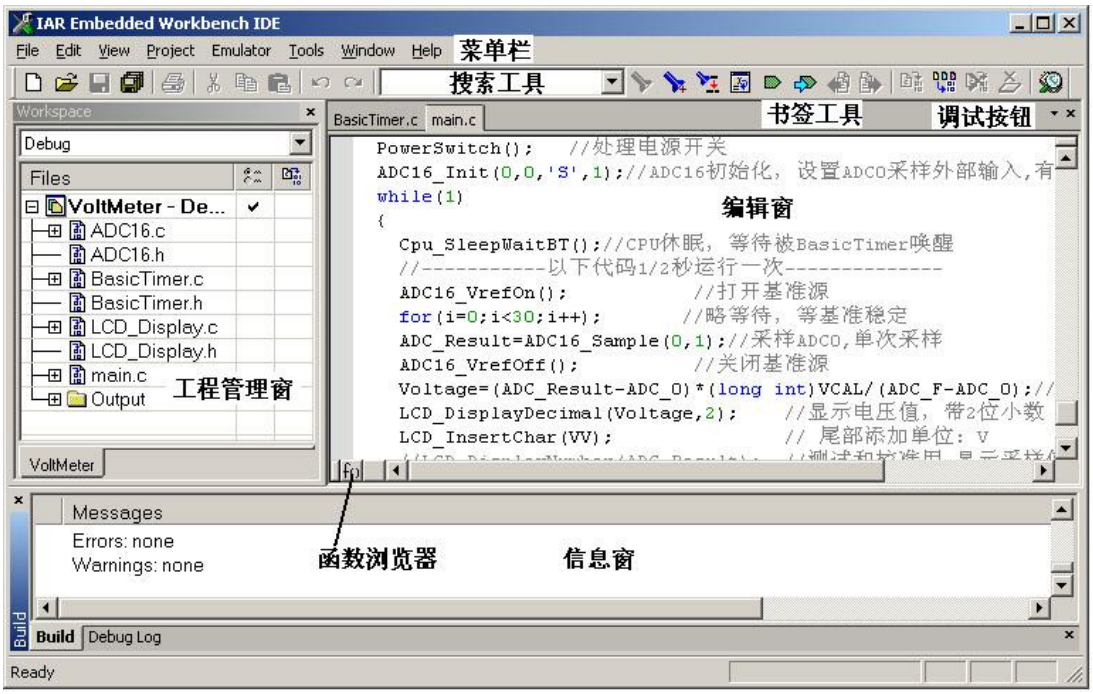


图 **1.2.1** IAR EW430 编辑界面

位于主窗口下方的是信息窗，编译过程、最终代码大小、RAM 开销情况、程序错误提示等内容会显示在该窗口内。另外软件界面中还有一些实用的小工具，如搜索和替换工具、书签工具、函数快捷浏览器等。

在 IAR 的软件中，不支持对单个 C 语言文件进行编译。所有的程序都必须建立工程后才能编译。考虑到一个设计中可能包含多个 CPU 或多个版本的程序， EW430 中还引入了工作空间（Workspace）的概念。每个 Workspace 对应着一个设计，每个设计中允许添加若干个工程(Project)，每个 Project 下，才是若干个源文件。因此，对于我们简单应用来说，即使只写一个文件，也要建立工作空间和工程。

参照下面的步骤，我们开始尝试为 430 单片机写第一个程序：

1. 在 File 菜单里面选择 File->New->Workspace，建立一个空工作空间
2. 在 Project 菜单里面选择 Project->Creat New Project，在当前空间建立一个新工程。新建过程中，首先会弹出语言选择窗，选择该工程的编程语言。EW430 支持 430 汇编、C 语言和 C++三种编程语言。本书的程序均采用 C 语言编写。选择 C->main，将自动生成 main.c 并添加进工程。如果不需要自动生成第一个文件，可以选择 empty project，以后自行新建并添加文件。下一步将提示选择工程路径并输入工程名（例如 first.ewp）。并且最好新建一个文件夹，将工程保存在文件夹内以方便管理。
3. 在 File 菜单里选择 File->Save Workspace，提示选择路径并输入文件名(例如 first.eww)，保存当前工作空间。
4. 在工程管理窗内会看到新建的名为 first 的工程，若在第 2 步选择了“C->main”，则工程内已经包含了 main.c，双击文件，在编辑窗打开后即可编辑。若在第 2 步选择了“empty project”，则需新建一个文件，保存成 xxx.c 文件，在工程名上右键->Add files 手动添加该文件进入工程。
5. 配置工程属性。在工程管理器内工程名上击右键->Options… 弹出一个属性配置菜单。该菜单中有大量的可设置选项，用于配置工程。在后面将随着程序的深入逐步介绍部分配置选项的用法。这里大部分采用默认设置，只用改动 2 个选项：首先要指定 CPU 型号，不同处理器的 Flash 存储空间不同，这关系到生成代码的定位。在 General Options 项内选择 Device->MSP430x4xxFamily->MSP430FE425。其次是设置仿真调试器驱动，在 Debugger 项内的 Setup 页 Driver 框内选择 FET Debugger。

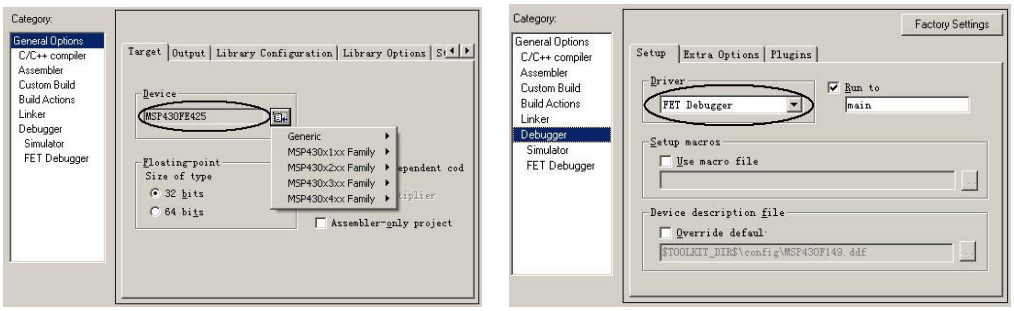


图 1.2.2 工程属性配置

若读者没有目标板，可以使用默认设置（Debugger 设为 Simulator），即软件仿真模式，可以模拟运行代码，能调试纯软件的程序，如某些数据处理算法。但不能从实际硬件上反映出运行结果。

6. 连接 FET Debugger。FET Debugger 的全称是“Flash 仿真工具”（Flash Emulation Tools）。在 430 单片机内部，自带了下载程序的接口(JTAG 接口)。通过单片机的 4 根专用引脚与 PC 机的并口(打印机接口)相连即可向单片机内烧写程序。除了下载程序外，通过该接口还能查看和更改 CPU 的各种寄存器与 RAM 内容、能够暂停 CPU 运行，从而能够实现仿真和调试功能。由于 430 采用 3V 逻辑电平而 PC 机并口采用 5V 逻辑，FET Debugger 实际上是一个电平转换器，在 PC 机并口和 430 单片机的 JTAG 接口之间起数据桥梁作用。由于每次调试都要写单片机的 Flash 代码存储器，调试完毕单片机就

可脱机工作，因此被称为 Flash Emulation Tools。除此之外，FET Debugger 还能从计算机并口上窃取 10mA 左右的电流，提供给目标系统用。在大部分应用中，足够 430 单片机系统使用，在调试过程中不需自备电源为目标板供电。

7. 在编辑窗口输入第一个 430 单片机程序：

```
/* 该程序让 P2.0 口上的 LED 闪烁 */
#include "msp430x42x.h" /*430 单片机寄存器头文件*/
void main( void )       //主程序
{
 int i;
 WDTCTL=WDTPW+WDTHOLD;  //停止看门狗
 FLL_CTL0 |=  XCAP18PF; // 设置晶振匹配电容 18pF 左右
 P2DIR|=0x01;           //P2.0 设为输出
 while(1)               //永远循环,单片机程序不能结束
 {
   for(i=0;i<20000;i++);//延迟
   P2OUT^=0x01;         //P2.0 取反(闪烁)
 }
}
```

8. 在 Project 菜单点击 Rebuild All ,开始编译。随后信息窗将提示编译结果：

```
70 bytes of CODE  memory
80 bytes of DATA  memory (+ 4 absolute )
2 bytes of CONST memory
Total number of errors: 0
Total number of warnings: 0
```

上述信息说明，代码编译后，生成的机器码占用了 70 字节 Flash ROM 空间，80 字节的 RAM。没有错误也没有警告。若提示出错，则根据错误提示在编辑窗修改程序后重新编译。

编译通过之后，我们开始在目标板上运行程序。首先要将编译生成的机器码下载到单片机的片内 Flash 存储器中。点击主界面上的“开始调试”按钮  ，弹出下载进程框。

图 1.2.3　下载进程框

若遇到“找不到设备”的提示，应检查目标板电源是否正常、是否有其他程序占用了并口、在 BIOS 里将并口设为 ECP 方式、连接线是否有断线或短路？若遇到“数据写入错误”的提示，多半是电源电压不足 2.7V 造成的。Flash 存储器写入时要求电源电压在 2.7V 以上。若计算机没有并口，则只能使用 USB 仿真机，但价格比并口 FET-Debugger 高。

下载过程结束后，EW430 软件自动进入调试(Debug)状态，界面也会随之改变：

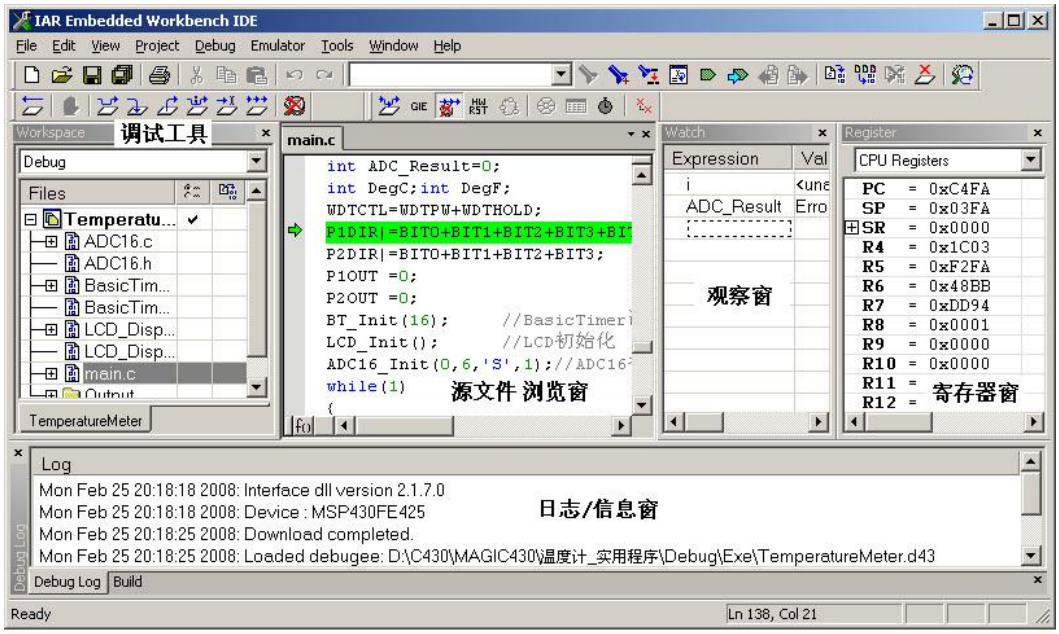


图 1.2.3 EW430 调试界面

点击“  ”按钮全速执行，P2.0 口的 LED 开始闪烁。第一个程序运行成功！

然而实际中，几乎不会有一次成功的程序。编译过程中只能检查出程序中的语法错误，不能检查出功能错误、逻辑错误、表达式写错、下标溢出等人为错误。因此在 EW430 进入调试界面后，主要功能不再是编写程序，而是检查和排除各种错误。

调试界面中提供了许多强大的调试工具，供调试、控制和跟踪程序流程：

1.  **全速执行**(Go)：按该按钮后，程序全速执行。

2.  **暂停程序**(Break)：只要程序处于运行状态，按该按钮可以强制暂停程序。

3.  **程序复位**(Reset)：按该按钮可以将程序复位，并暂停在第一句。

4.  **单步执行**(Step Over)：每按一次该按钮，程序执行一条语句后暂停。如果遇到函数，则执行完该函数后暂停。

5.  **跟踪执行**(Step In)：每按一次该按钮，程序执行一条语句后暂停。如果遇到函数，则跳入该函数后暂停在第一句。

6.  **跳出函数**(Step Out)：若当前程序处于某函数，按该按钮后，执行到函数返回前最后一条语句处自动暂停。

7.  **运行至光标处**(Run To)：点击该按钮后，程序全速运行，直到运行到光标所在语句自动暂停。

8.  **断点**(Toggle Break Point)：断点是一种常用调试手段，类似于陷阱。点击该按钮后，光标所在行会变红，提示该行已被设置成一个断点(Break Point)。程序无论因为何种原因（单步、全速）运行到该行，都会自动被暂停。再按一次断点按钮，可取消该断点。

9.  **重新下载**(Make & Download)：在发现错误之后，可以直接在编辑窗口修改源代码，然后点击该按钮，重新编译并自动下载。

10.  **结束调试**(Stop Debug)：按该按钮退出调试模式，回到编辑模式。

除了调试程序流程之外，程序员在排错过程中还需要查看各种变量和寄存器的值，以确定程序运行中间结果是否正确。在 EW430 调试状态下，View 菜单里面提供了功能丰富的查看功能：

1. 在线查看变量：只要程序处于暂停状态，将鼠标停在源代码任何一个变量上 2 秒钟不动，就会自动显示该变量的值。这是一种简便快捷的查看方法，但每次只能察看一个变量，并且不能更改变量值。
2. 通过菜单 View->Watch 打开观察窗。这是最常用的功能之一。在 Expression 栏内输入变量名或表达式，在 Value 栏可以看到变量或表达式的值。通过观察窗可以同时察看多个变量的值，且能够在 Value 栏直接输入数据，更改变量值。用鼠标右键还可切换数据显示格式。
3. 通过菜单 View->Register 打开寄存器窗。可以查看单片机内部各个模块的控制寄存器。430 单片机内部的寄存器较多，寄存器窗内已经按照模块分类，方便察看。类似于观察窗，只要可写的寄存器，都可以直接输入新数值而改变寄存器值。
4. 通过菜单 View->Disassembly 打开反汇编窗。它将 C 语言生成的机器码重新翻译回汇编语言，供有经验的程序员调试用。由于每条 C 语句对应一条以上的汇编语言，打开反汇编窗口后，单步执行和跟踪执行每次执行一条汇编语句而不再是一条 C 语句。
5. 通过菜单 View->CallStack 打开调用关系窗。可查看程序执行到当前位置所经历的函数路径
6. 通过菜单 View->Stack 打开堆栈窗。可以看到当前堆栈空间使用情况。
7. 通过菜单 View->Memory 打开内存窗。可以看到内存中数据存放情况。由于 430 单片机属于冯诺伊曼结构，数据空间和程序空间统一编址。所以内存窗也可查看 Flash 内代码或数据情况。

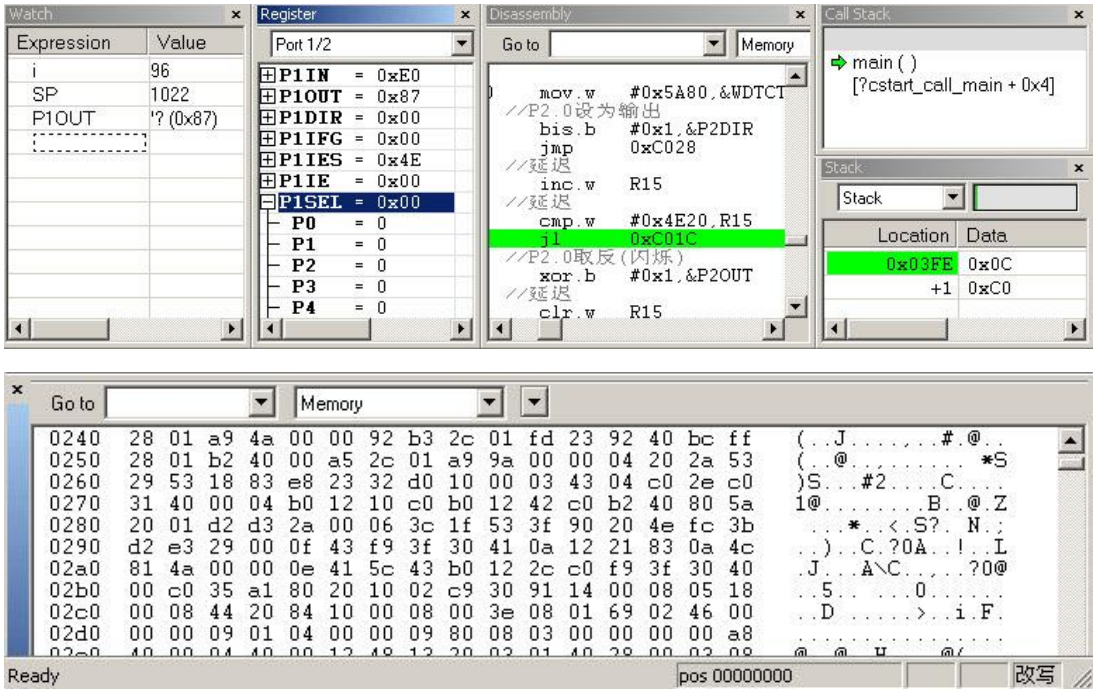


图 1.2.4 各种观察窗口

通过上述调试和查看功能的巧妙组合，能够创造出强大的排错能力。对于初学者来说，对复杂问题排错可能像瞎子摸象，没有目的地乱找，比较吃力。换成经验丰富的程序员，也许两三步就能发现错误。排错方法是一个需要不断练习和积累经验的过程。对于新手，以下的几个基本方法是普遍适用的：

1. **包围法**：将程序划为若干段，打开观察窗，监视可疑变量或中间结果，用“断点”或“运行到光标处”工具检验各段程序运行的结果是否正确，逐步缩小问题范围，最后可以用单步、跟踪工具找到问题。
2. **极限法**：对于某些偶尔出现或周期出现的问题，很可能是某些变量处于溢出边缘，或者>=和>混淆，下标数值差 1 之类的习惯性错误。利用变量观察窗，改变变量值，尽可能取极限情况，试验个个函数是否工作正常，逐步缩小错误范围最终找到问题。
3. **陷阱法**：当怀疑系统偶尔出现某种不应出现的状态时，或者怀疑某变量偶尔出现了不应出现的值时，可以用一个 if 语句判断该状态的出现，在后面跟一条空操作语句 _NOP();并在空语句处设置断点。在出现这种状态时，被断点捕捉，此后可以打开观察窗，察看各变量，看哪些可疑，分析错误来源。
4. **穷举法**：当怀疑某个函数有可能在某些特殊的输入情况下产生错误结果，可以用一个 for 循环对所有可能输入进行穷举，再用 if 语句设置错误陷阱，看哪些输入会造成错误，然后用观察窗的变量赋值功能专门产生错误输入情况，最后用跟踪工具找错误来源。
5. **对比法**：当自己写的程序无法正常执行，恰巧手头有可参考的代码；或者以前写的程序正常，现在写的却突然不能用了；写了一段新程序后，前面已经调试通过的代码却突然失灵。遇到类似情况可以分别运行两个程序，通过 Register 窗口或 Watch

窗查看并记录下相关寄存器和变量的值，二者对比，找到设置错误的变量或寄存器，再跟踪出错原因，最终排除错误。

下面通过一个实例来示范排错的过程。下面这段代码“突然失灵”：

```
#include "msp430x42x.h" /*430 单片机寄存器头文件*/
void main( void )        //主程序
{
 int i;
 WDTCTL=WDTPW+WDTHOLD;   //停止看门狗
 FLL_CTL0 |=  XCAP18PF; // 设置晶振匹配电容 18pF 左右
 P2DIR|=0x01;           //P2.0 设为输出
 while(1)               //永远循环,单片机程序不能结束
 {
   for(i=0;i<20000;i++)   //延迟
   P2OUT^=0x01;           //P2.0 取反(闪烁)
 }
}
```

运行上述程序，本应闪烁的 LED 却一直亮。下面开始排查错误：

1. 在 for 循环处执行“运行到光标处”。用 Register 窗查看 P2DIR 寄存器=0x01，赋值正确。
2. 在 P2DIR|=0x01;句设断点，全速运行，程序没有停止，说明看门狗没有复位程序。
3. 暂停，在 P2OUT^=0x01;句设断点，全速运行，会停下来，再运行几次，发现到每运行一次，LED 都会亮灭交替一次。证明硬件正常。
4. 为什么单步运行 LED 正常闪烁，全速运行 LED 却不会闪烁？上面已经将问题缩小包围至 for 循环那一句，单步跟踪一下，发现 for 语句只执行 1 次循环就执行 P2OUT^=0x01;而不是预计的 20000 次。用 Watch 窗察看 i 变量，发现每次 LED 取反后才执行 i++，而不是循环 20000 次完毕后才执行 P2OUT^=0x01;

至此，我们发现了问题：for(i=0;i<20000;i++) 这句忘记打分号！c 语言将下一行的程序连在 for 循环之后，相当于 for(i=0;i<20000;i++) P2OUT^=0x01;造成闪烁速度极快，使人眼根本看不到闪烁。

1．3　MSP430 单片机 C 语言基础

MSP430 系列单片机的 CPU 属于 RISC（精简指令集）型处理器，只有 27 条正交汇编指令。在 CPU 系统中，增加指令意味着增加电路，最后导致硅片面积增加，限制速度提升。RISC 结构处理器设计思想之一就是只要用若干指令组合能完成的功能，决不增加一条专用指令。例如乘法能够用加法和移位运算组合实现：（其中<<是二进制左移符号）

$$12\times45=12\times(32+8+4+1)=12\times(1<<5+1<<3+1<<2+1)$$
$$=12<<5+12<<3+12<<2+12$$

那么 RISC 处理器的指令集中就会省去乘法指令。在保证功能的情况下，由于减少了电路、降低硅片面积，RISC 处理器不仅价格低廉，而且速度快。但这也意味着用汇编语

言写程序是非常困难的。因为即使简单的功能也可能需要很多条指令才能完成。RISC 处理器基本上是为高级语言所设计的，因为正交指令系统很大程度上降低了编译器的设计难度，利于产生高效紧凑的代码。事实上，目前 430 单片机的 C 编译器的表现非常优秀，如果没有几年的手工汇编经验，很难写出比 C 编译器更高效的代码。对于初学者完全可以在不深入了解汇编指令系统的情况下直接开始 C 语言开发。

任何一款处理器的 C 语言，都是在标准 C 语言（ANSI-C）上增加对相应处理器的特殊操作而构成。此外，C 语言写的源程序属于文本文件，不能直接被目标 CPU 运行。需要经过编译器才能生成机器码。同一份 C 语言源程序，经过不同的编译器编译和链接之后，就能生成不同处理器的机器码。

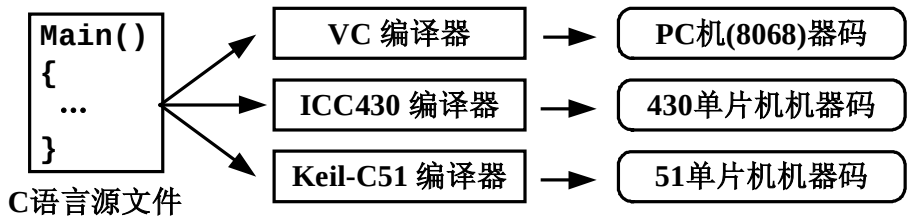


图 1.3.1 C 语言、编译器和机器码之间的关系

如果我们在编程过程中注意尽量消除不同 CPU 硬件上的差异，或者将硬件差异集中到某个很小的局部，那么这个程序通过很简单的修改就能编译成另一 CPU 的机器码，在其他处理器上运行。这就是所谓的“代码移植”，是嵌入式软件设计最重要的思想之一。如果读者在一开始就树立可移植性的概念，养成好的编程习惯，对减少重复劳动、加快开发速度有巨大帮助。

430 单片机的 C 语言（以下简称 C430）语法和标准 C 语言差异很小，读者可以参考任何一本 C 语言教材，在此不再赘述。下面仅列出 430 单片机 C 语言和标准 C 语言的差异，以及和 CPU 相关的特殊语句。

Ⅰ 变量

在 C 语言的国际标准中，对各变量字节数没有做严格限定，因此不同编译器可能会略有差别。C430 中还增加了 8 字节的 long long int 型变量，能够对十进制 20 位整数进行运算。下面列出常见变量类型：

表 1.3.1 C430 中变量类型

变量类型	字节数	值域	备注
char	1	-128~+128 或 0~255	Compile Option 选项 设置 char 变量值域
usigned char		0~255	
int	2	-32768~32767	
unsigned int		0~65535	
long	4	-21474836478~2147483647	
unsigned long		0~4294967295	
long long	8	-92233720368547758078 ~9223372036854775807	

unsigned long long		0~18446744073709551615	
float	4	$-1.175494351\times10^{-38}$ $\sim3.402823466\times10^{+38}$	
double	4 或 8	$2.2250738585072014\times10^{-308}$ $\sim1.7976931348623158\times10^{+308}$	Genaral Option 选项 设置浮点指针长度

为了和某些其他处理器的软件开发人员编程习惯兼容，EW430 允许改变某些变量的特性。在工程名上右键打开 Option 菜单，在 GeneralOption 项可以设置浮点指针长度，这决定了 double 变量的字节数，默认是 32bit，即 4 字节（和 8051 等 8 位处理器兼容）。在 Compile Option 里可以设置 char 是否等效为 unsigned char （为了和某些早期的 C 语言兼容）。建议读者可以将浮点指针设为 64 位，char 等效为 signed char。

在变量定义表达式中增加某些关键字可以给变量赋予某些特殊性质：

const ：定义常量。在 C430 语言中，const 关键字定义的常量实际上被放在了 ROM 中。可以用 const 关键字定义显示表之类的常数数组。

static ：相当于本地全局变量，只能在函数内使用，可以避免全局变量混乱。

volatile ：定义“挥发性”变量。编译器将认为该变量的值会随时改变，对该变量的任何操作都不会被优化过程删除。

no_init 或__no_init：定义无需初始化的变量。C 语言 main 函数开始运行之前，都会将所有 RAM 清零(全部变量都清零)。若某变量被定义为无需初始化，在初始化过程中不会被清零。

范例：

```
const unsigned char Table[7]={1,2,3,4,5,6,7}; // 在 ROM 中定义一个表格
static int a;       //定义一个 int 型静态变量 a
volatile int b;     //定义 int 型变量 b，不要被编译器优化
__no_init int c;    //定义 int 型变量 c，程序开始不对它初始化
```

数学运算

C430 的数学运算符与标准 C 语言完全一致。在此了解一下 430 单片机的数据运算特点和编程原则，对于初学者仍然是很有帮助的。

1. **尽可能避免浮点运算。**浮点数数值范围大，不易溢出，精度较高。对于习惯写 PC 程序的设计者来说，遇到数值计算会优先考虑浮点数。但对于单片机，浮点数的运算速度很慢，RAM 开销也大；在低功耗应用中 CPU 运算时间直接关系到平均功耗。因此在编程初期就要养成尽量避免使用浮点数的习惯。
2. **防止定点数溢出。**定点数运算首先要防止数据溢出。例如：

```
long int x;
int a;
x=a*1000;
```

虽然 x 是 long 变量，但 a 和常数 1000 都是 int 型，相乘结果仍然是 int 型。在 a>65 的情况下，结果就会溢出。程序应该修改为：

```
x=a*(long)1000;  或 x=(long)a*1000;
```

若遇到多个变量相乘，更需要细心检查。所以，在测试每一段软件的时候，一定要取边界条件做极限测试。

3. **小数的处理。**遇到需要保留小数的运算，可以采用浮点数，但是软件开销较大。用定点数也可以处理小数。原理就是先扩大，再运算。例如，我们需要计算温度并保留 1 位小数，假设温度计算公式是：`Deg_C=ADC*1.32/1.25-273`

   为了让小数 1.32 能被定点运算，先扩大 100 倍变成 132，当然，除数 1.25 也要随之扩大 100 倍，公式变为：`Deg_C=(long)ADC*132/125-273` 。

   这样运算结果只能保留到整数，为了让结果保留 1 位小数，需要人为将所有数值都扩大 10 倍，得到最终计算公式 `Deg_C=(long)ADC*1320/125-2730` 。假设温度应该是 `23.4` 度，上述公式的运算结果将是 `234`。在显示的时候，将小数点添加在倒数第 2 位上，即可显示 `23.4`。用定点数处理小数，如需要保留 `N` 位小数，就要将数值扩大 10^N 倍。注意防止溢出，且要记住每个数值所扩大的倍数，在程序中应添加注释。

4. **尽量减少乘除法。**430 单片机没有乘法/除法指令，乘除操作会被编译器转换成移位和加法来实现。如果乘除的数值刚好是 2 的幂，那么可以用移位直接替代乘除法，运算速度会提高很多。例如对 16 次采样数据求平均：

```
for(i=0;i<16;i++) Sum+=ADC_Value[i];  //求和
Aver=Sum/16;     //这一句的运算较慢
```

   对于除 16 写成如下形式，运行速度会提高很多：

```
Aver=Sum>>4;     // 除以 16
```

   若将编译器优化级别设置得比较高，在遇到乘除 2 的幂表达式时，编译器会自动用移位替代除法（编译器很聪明），从而加快执行速度。

I 位操作

位操作指令大部分存在于早期速度不高的 CISC 处理器上(以 8051 为代表)，以提高执行效率，弥补 CPU 运算速度的不足。目前几乎所有的 RISC 型处理器都取消了位操作指令，430 单片机也不例外。在 430 的 C 语言中，也不支持位变量，因为位操作完全可以由变量与掩模位(mask bits)之间的逻辑操作来实现。

例如将 P2.0 置高、将 P2.1 置低，将 P2.2 取反，我们可以写成：

```
P2OUT |=  0x01;      //P2.0 置高
P2OUT &= ~0x02;      //P2.1 置低
P2OUT ^=  0x04;      //P2.2 取反
```

在寄存器头文件中，已经将 BIT0~BIT7 定义成 0x01~0x80，上述程序也可以写成：

```
P2OUT |=  BIT0;      //P2.0 置高
P2OUT &= ~BIT1;      //P2.1 置低
P2OUT ^=  BIT2;      //P2.2 取反
```

对于多位可以同时操作，例如将 P1.1 、P1.2、P1.3、 P1.4 全部置高/低可以写成：

```
P1OUT |=  BIT1+BIT2+BIT3+BIT4;      //P1.1/2/3/4 全置高
P1OUT &=~(BIT1+BIT2+BIT3+BIT4);     //P1.1/2/3/4 全置低   注意括号!
```

实际上，这条语句相当于

```
P1OUT |=  0x1e;      //P1.1/2/3/4 全置高
```

对于读操作，也可以通过寄存器与掩模位(mask bits)之间的“与”操作来实现。例如有通过 P1.5、P1.6 口控制位于 P2.0 口的 LED。下面代码示范读取 P1.5 和 1.6 的值：

```
char Key;
if((P1IN & BIT5)==0)    P2OUT|=BIT0;   //若 P1.5 为低，则 P2.0 口的 LED 亮
```

```
if( P1IN & BIT5)        P2OUT|=BIT0;   //若 P1.5 为高，则 P2.0 口的 LED 亮
if( P1IN & (BIT5+BIT6)) P2OUT|=BIT0;//若 P1.5 和 P1.6 任一为高，则点亮 LED
if((P1IN & (BIT5+BIT6)) != (BIT5+BIT6)) P2OUT|=BIT0;
                                        //若 P1.5 和 P1.6 任一为低，则点亮 LED
if(P1IN & BIT5)  Key=1;
else             Key=0;                  //读取 P1.5 状态赋给变量 Key。
```

另外还有一种流行的位操作写法，用(1<<x)来替代 BITx 宏定义：

```
P2OUT |=  (1<<0);   //P2.0 置高
P2OUT &= ~(1<<1);   //P2.1 置低
P2OUT ^=  (1<<2);   //P2.2 取反
if((P1IN & (1<<5))==0)  P2OUT|=(1<<0); //若 P1.5 为低，则 P2.0 口的 LED 亮
```

这种写法的好处是使用纯粹的 C 语言表达式实现，不依赖于 msp430 的头文件中 BITx 的宏定义，无需改动即可移植到任何其他单片机上，但可读性较差。

I 寄存器操作

430 单片机内部各模块的配置、操作全部通过寄存器进行。430 单片机内部有数百个寄存器，上千个控制位。通过这些寄存器可以配置各个模块的工作方式、状态、连接关系等参数。第二章将逐个模块讲解寄存器功能配置。这里仅从 C 语言语法上讨论寄存器的设置方法。

在 430 单片机中，寄存器实际上是位于 RAM 低端的一些存储单元。对于初学者来说，可以先不考虑寄存器的绝对地址，只要在文件开头包含相应的头文件，即可像操作变量一样操作寄存器。由于本书的范例使用的单片机型号是 MSP430FE425，所以在每个 C 文件开头都包含了“msp430x42x .h”头文件。头文件内的寄存器和标志位名称与《User Guide》内列出的名称完全一致。

寄存器读写操作中会遇到大量的位操作。例如需要允许串口收发中断，查《User Guide》找到串口收、发中断控制位名称分别是 URXIE0 和 UTXIE0，位于 IE1 寄存器的最高 2 位，需要将这两位置高。参考位操作方法，写成：

```
IE1 |=  BIT6 + BIT7;    // URXIE0 和 UTXIE0 置高,打开串口收发中断
```

这种写法不能直观看到被设置的标志位名称，为了解决这个问题，在头文件中已经将各个标志位都作了宏定义。打开 msp430x42x .h 文件，我们能找到如下的宏定义：

```
#define URXIE0           (0x40)
#define UTXIE0           (0x80)
```

因此我们可以把程序改写成更容易读的形式：

```
IE1 |=  URXIE0+ UTXIE0;    // URXIE0 和 UTXIE0 置高,打开串口收发中断
```

读者今后在读程序的过程中，会大量见到这种写法。

但这种写法也有致命缺点：C 语言的语法检查不能检测出赋值错误。假设读者在编程时将寄存器和控制位张冠李戴：

```
IE2 |=  URXIE0+ UTXIE0;      // 寄存器错误
P1OUT |=  URXIE0+ UTXIE0;    // 寄存器错误，实际将 P1.6/7 置高
```

URXIE0 和 UTXIE0 两个控制位在 IE1 寄存器内，却将它赋给了 IE2 寄存器，或者我们故意制造更离谱一些的错误，赋给 P1 口。这下情况下，编译器仍会为是合法语句，不会报错。

为了避免上述问题，EW430 提供了另一类头文件，叫做 IO 头文件。它将各个寄存器都做成了结构体，可以模仿出传统位变量的操作语法。上例中，若将头文件替换成“io430x42x .h”，那么可以用下面的语句来赋值：

```
#include "io430x42x.h"    /*替换掉"msp430x42x.h"*/
P2OUT_bit.P2OUT_0 =1;     //P2.0 置高
P2OUT_bit.P2OUT_1 =0;     //P2.1 置低
P2OUT_bit.P2OUT_2 ^=1;    //P2.2 取反
IE1_bit.URXIE0=1;         //IE1 的 URXIE0 位置 1
IE1_bit.UTXIE0=1;         //IE1 的 UTXIE0 位置 1
```

用这个头文件，编译器可以检测出赋值错误。假设代码写成：

```
IE2_bit.URXIE0=1;         // 寄存器错误
```

编译器会发现 IE2_bit.URXIE0 未定义，从而报错。

如果读者以前已经习惯于 8051 单片机之类的有位操作的单片机系统，可以使用 IO 头文件，若读者已经习惯了 RISC 处理器，可使用普通头文件。实际上使用 IO 头文件更加安全，但目前厂商提供的参考范例程序全部使用普通头文件，更易阅读。

在每个头文件中，不仅包括了寄存器的定义、标志位的定义，还包括了一些组合宏定义。例如我们在 BTCTL 寄存器的下方会看到这样的宏定义：

```
#define BT_ADLY_16        (BTDIV)                /* 16ms       */
#define BT_ADLY_32        (BTDIV+BTIP0)          /* 32ms       */
#define BT_ADLY_64        (BTDIV+BTIP1)          /* 64ms       */
#define BT_ADLY_125       (BTDIV+BTIP1+BTIP0)    /* 125ms      */
... ...
```

EW430 的头文件中，已经将一些常用的寄存器值的组合做成了宏定义，一般都会有详细的注释来说明这些组合的含义。例如我们可以直接使用下面语句：

```
BTCTL = BT_ADLY_125 ;          // BT 定时器设为 1/8 秒(125ms)中断一次
```

读者在编写 430 寄存器操作代码的时候，要习惯性的搜索一下头文件，看看头文件内有没有提供现成的组合宏定义，减少工作量，降低出错可能。

另外要注意，用“|=”来赋值的好处是不影响该寄存器内的其他比特，但一定要保证被赋值的若干比特在赋值之前一定要为 0，否则可能会导致错误的结果。例如下面的语句本意是利用三个 IO 口控制一个循环流水灯，但是赋值语句（2）执行后并不改变 BIT1，将会导致之后三个灯全亮，且永远不熄灭：

```
while(1)
{
  P1OUT |= BIT1 + BIT2 ;          // P1.1 和 P1.2 输出高电平（1）
  Delay(1000);                    // 延迟 1 秒
  P1OUT |= BIT2 + BIT3 ;          // P1.2 和 P1.3 输出高电平（2）
  Delay(1000);                    // 延迟 1 秒
  P1OUT |= BIT3 + BIT1 ;          // P1.3 和 P1.1 输出高电平（3）
  Delay(1000);                    // 延迟 1 秒
 }
```

正确的程序是在每句赋值前增加将三个比特全部清零的语句：

```
while(1)
{
  P1OUT &= ~(BIT1 + BIT2 +BIT3);       // 清除 IO 输出三个比特
```

```
    P1OUT |= BIT1 + BIT2 ;              // P1.1 和 P1.2 输出高电平（1）
    Delay(1000);                        // 延迟 1 秒
    P1OUT &= ~(BIT1 + BIT2 +BIT3);      // 清除 IO 输出三个比特
    P1OUT |= BIT2 + BIT3 ;              // P1.2 和 P1.3 输出高电平（2）
    Delay(1000);                        // 延迟 1 秒
    P1OUT &= ~(BIT1 + BIT2 +BIT3);      // 清除 IO 输出三个比特
    P1OUT |= BIT3 + BIT1 ;              // P1.3 和 P1.1 输出高电平（3）
    Delay(1000);                        // 延迟 1 秒
  }
```

在控制寄存器中，大部分在上电复位时都会被自动清零。它们在初始化的时候可以用“|=”号赋值。但在以后需要更改设置时再用“|=赋值”，就会出现问题，特别时使用快捷宏定义时更容易出错。例如设置 PWM 控制器的输出模式时，需要设置三个控制位 OUTMODE0/1/2。在头文件中提供了 OUTMODE_0~ OUTMODE_7 的快捷宏定义可以使用，分别对应三个控制位的 8 种组合。假设先利用快捷宏定义将通道一的输出模式设为模式 3，一段时间后后，再将其改为模式 6：

```
    TACCTL1 |= OUTMODE_3 ;              //  通道 1 设为输出模式 3
    ...                                 //  运行一段时间
    TACCTL1 |= OUTMODE_6 ;              //  通道 1 改为输出模式 6 （出错）
```

实际上，上面的程序等价于

```
    TACCTL1 |= OUTMODE0 + OUTMODE1;     //  3 = BIT0 + BIT1
    ...                                 //  运行一段时间
    TACCTL1 |= OUTMODE1 + OUTMODE2;     //  6 = BIT1 + BIT2
```

最后的结果是 OUTMODE0、1、2 三个控制位都被置 1，因此实际上设为了模式 7 而不是模式 6。所以，在使用“|=”符更改设置时，必须先将原有设置位清零：

```
    TACCTL1 |= OUTMODE_3 ;              //  通道 1 设为输出模式 3
    ...                                 //  运行一段时间
    TACCTL1 &=~ OUTMODE_7;              //  清除 3 个比特的原有设置
    TACCTL1 |=  OUTMODE_6;              //  通道 1 改为输出模式 6 （正确）
```

类似的情况还出现在上电未被初始化的寄存器，如 BTCTL 寄存器，第一次若用“|=”赋值也可能出错，且具有一定的随机性，例如经常因为 BTHOLD 位上电为 1 导致 LCD 无法显示。这种情况可以先清零再赋值，也可用等号赋值来避免上述问题。

用等号赋值时也要注意，他会改变寄存器内的其他标志位，所以必须一次对所有控制位全部赋值。

I 中 断

在下一章我们将会了解到 430 单片机的低功耗主要靠休眠来实现。能够将 CPU 从休眠状态唤醒的条件只有发生中断或复位。因此低功耗和中断之间的关系密不可分。430 单片机所有的大部分功能模块均能够在不需 CPU 干预的情况下独立工作，且都能能引发中断。430 软件的基本结构之一就是先向某模块发出指令（如 ADC 开始转换），然后 CPU 休眠，等待模块操作完毕产生中断唤醒 CPU 继续下面的任务，从而将 CPU 运行时间降到最少。

430 单片机具有中断向量表，位于 ROM 最高段 512 字节中(0xFE00~0xFFFF)，事先需

要将中断服务程序入口地址装入中断向量表内。中断发生后，如果当前该中断被允许，CPU会自动将当前程序地址和 CPU 状态寄存器 SR 压入堆栈。然后跳转到中断服务程序入口。中断服务程序内可能会改变的寄存器都要通过软件压入堆栈，之后才能开始执行中断服务程序。退出中断前需要通过软件从堆栈恢复寄存器值，最后 CPU 自动恢复 SR 寄存器，跳转到中断发生前的地址继续执行。有趣的是，430 单片机的 SR 寄存器保存着低功耗休眠标志位。假设中断发生前是休眠的，中断返回后 CPU 将仍然是休眠状态。如果希望唤醒CPU，需要通过一些软件手段在退出中断前修改堆栈内 SR 的值。

事实上，在 EW430 中，上述所有复杂的操作过程都可以交给 C 编译器来完成，读者只需要专注于编写中断服务程序。定义中断服务程序的方法非常很简单，例如：

```
#pragma vector = BASICTIMER_VECTOR    //BasicTimer 中断向量
__interrupt void BT_ISR(void)         //声明一个中断服务程序，名为 BT_ISR()
{
  ... ...                             //在这里写中断服务程序
}
```

在中断函数前加__interrupt 关键字（注意有 2 个_ ），告诉编译器这个函数是中断程序。然后在函数前一行写#pragma vector = XXXX_VECTOR 指明中断源，决定该函数为哪个中断服务。可以简单的理解为只要 XXXX 中断发生，程序就会立即执行该函。

因为不同型号 430 单片机含有的模块种类不一样，中断资源也不同。具体可以参考头文件中 Interrupt Vectors 段的定义。以 x42x 单片机头文件的定义为例：

```
/************************************************************
* Interrupt Vectors (offset from 0xFFE0)  中断向量(从 0xFFE0 开始)
************************************************************/

#define BASICTIMER_VECTOR  (0 * 2u)  /* 0xFFE0 基础定时器中断向量 */
#define PORT2_VECTOR       (1 * 2u)  /* 0xFFE2  P2 口中断向量 */
#define PORT1_VECTOR       (4 * 2u)  /* 0xFFE8 P1 口中断向量向量*/
#define TIMERA1_VECTOR     (5 * 2u)   /* 0xFFEA TA 定时器 CCR1/2 中断向量 */
#define TIMERA0_VECTOR     (6 * 2u)  /* 0xFFEC TA 定时器 CCR0 中断向量*/
#define USART0TX_VECTOR    (8 * 2u)  /* 0xFFF0 串口发完一字节中断向量 */
#define USART0RX_VECTOR    (9 * 2u)  /* 0xFFF2 串口收到一字节中断向量*/
#define WDT_VECTOR         (10 * 2u) /* 0xFFF4 看门狗定时器溢出中断向量 */
#define SD16_VECTOR        (12 * 2u) /* 0xFFF8 ADC 中断向量 */
#define NMI_VECTOR         (14 * 2u) /* 0xFFFC NMI 中断向量*/
#define RESET_VECTOR       (15 * 2u) /* 0xFFFE 复位入口向量 */
```

上述宏定义声明了 MSP430x42x 系列单片机的中断源。例如我们需要写一个名为 ADC_ISR 的中断服务程序，为 ADC 中断服务，用于读取转换结果。从上面查到 ADC 的中断向量已经定义为 SD16_VECTOR，那么该中断函数可以写为：

```
#pragma vector = SD16_VECTOR      //16 位 ADC 中断源
__interrupt void ADC_ISR (void)   //声明一个中断服务程序，名为 ADC_ISR()
{
  ... ...                         //在这里写读取 ADC 结果的代码
}
```

前文已经分析过，假设中断发生前 CPU 是休眠的，中断返回后 CPU 将仍然是休眠状态。这非常适合编写并发程序（全部执行都由中断完成，主程序只有休眠一句）。如果希

望唤醒 CPU，需在退出中断前修改堆栈内 SR 的值，清除掉休眠标志。在 EW430 中，针对这一特殊操作提供了一个修改堆栈内 SR 的函数：__low_power_mode_off_on_exit();只要在退出中断之前调用该函数，能够修改被压入堆栈的 SR，从而在退出时唤醒 CPU。

```
#pragma vector = BASICTIMER_VECTOR    //BasicTimer 中断源
__interrupt void BT_ISR(void)         //声明一个中断服务程序，名为 BT_ISR()
{
  ... ...                             //在这里写中断服务程序
  __low_power_mode_off_on_exit();     //退出中断时唤醒 CPU。注意开头两个'_'
}
```

由于 430 单片机的中断源数量很多，比如 P1、P2 口每个 IO 都能产生中断，3 个 ADC 采样结束以及遇到错误都会产生中断，为了便于管理，430 的中断管理机制将同类的中断合并成一个中断源，再由中断标志位去判断详细的中断源。例如 P1 口任何一个 IO 中断发生时，程序都会通过中断向量表 PORT1_VECTOR 位置的入口跳到 P1 口中断服务程序中，在中断程序中再去判断是哪一个 IO 发生了中断。下例示范让 P1.5、P1.6 口发生的中断执行不同功能。

```
#pragma vector = PORT1_VECTOR      //P1 口中断源
__interrupt void P1_ISR(void)       //声明一个中断服务程序，名为 P1_ISR()
{
  if(P1IFG & BIT5)                 //判断 P1 中断标志第 5 位(P1.5)
    {
      ... ...                      //在这里写 P1.5 中断服务程序
    }
  if(P1IFG & BIT6)                 //判断 P1 中断标志第 6 位(P1.6)
    {
      ... ...                      //在这里写 P1.6 中断服务程序
    }
   P1IFG=0;                        //清除 P1 所有中断标志位
}
```

I 内部函数

标准 C 语言具有普遍适用性，但每种 CPU 都有其独特之处，很可能对某 CPU 来说一个简单操作，用标准 C 语言表达出来却很复杂。比如上面修改 SR 的例子，因为 C 不能直接操作堆栈，只能用指针操作来进行，将会低效而晦涩。为了解决类似问题，编译器一般会提供一些针对目标 CPU 的特殊函数，以及经汇编高度优化的常用函数。这些函数被称为内部函数(Intrinsic Functions)。上面的__low_power_mode_off_on_exit();就是 ICC430 编译器提供的一个内部函数。

常用的一些内部函数有：

```
__low_power_mode_0(); 或 LPM0;  //进入低功耗模式 0
__low_power_mode_1(); 或 LPM1;  //进入低功耗模式 1
__low_power_mode_2(); 或 LPM2;  //进入低功耗模式 2
__low_power_mode_3(); 或 LPM3;  //进入低功耗模式 3
__low_power_mode_off_on_exit(); //退出时唤醒 CPU
__delay_cycles(long int cycles); //靠 CPU 空操作延迟 cycle 个时钟周期
__enable_interrupt();  或_EINT();     //打开总中断开关
```

```
__disable_interrupt(); 或_DINT();     //关闭总中断开关
__no_operation();      或_NOP();      //空操作
__swap_bytes(x); 或 _SWAP_BYTES(x); //高低字节交换，返回整形值
__bcd_add_short(unsigned int,unsigned int);  //整形 bcd 加法，返回整型
__bcd_add_long (unsigned long, unsigned long);//长整形 bcd 加法，返回长整型
__bcd_add_long_long(unsigned long long, unsigned long long);
                                        //长长整形 bcd 加法，返回 long long 型
```

还有一些较少使用的内部函数，请参考“intrinsc.h”和“in430.h”头文件。这 2 个文件是默认包含在工程内的，程序中不用包含任何头文件，可以直接使用内部函数。

l 库函数

除了编译器提供针对 CPU 独有的函数之外，C 语言作为一种通用平台，也应该提供一些实用的函数。C 语言国际标准规定了每种 C 编译器都必须提供格式化输入/输出、字符串操作，数据转换、数学运算等标准函数。这些函数以库文件形式提供。和内部函数相反，这些函数都是与硬件完全无关的，换句话说，不管任何处理器的编译器，都会提供库函数。EW430 提供了 100 个库函数。其中包括读者已经很熟悉的 printf/scanf 函数、数学运算函数、字符串函数等。库函数里面提供的函数都是非常经典的，且经过高度优化。如果熟练掌握大部分库函数，能节省很多开发时间。这 100 个库函数的详细使用方法、需要包含的头文件可参阅《IAR C LIBRAR FUNCTIONS Reference Guide》。这篇文档位于 IAR EW430 安装目录\430\doc\clib.pdf。

部分库函数需要简单移植后才能使用，比如 scanf 和 printf 函数，本身只负责格式化输入/输出功能，具体从何设备上获取字符，字符输出到何设备上，需要由用户提供输入/输出函数决定。第三章将有一个范例说明如何利用 printf/scanf 函数做简易人机界面。

1．4 文件管理

受到简短的 C 语言入门程序范例的影响，初学初学者往往将所有的代码都写在一个 C 文件中。这是一种非常坏的习惯。当程序逐渐变长，只要超过 5 个屏幕长度，会发现编辑、查找和调试都将变得非常困难。而且这种代码很难用在别的项目中，除非仔细地理顺函数关系，然后寻找并复制每一段函数。

合理的方法是将一个大程序划分为若干个小的 C 文件。在单片机程序中，最常用的划分方法是按硬件功能划分，将每个功能模块做成独立的 C 文件。例如一个项目中会用到 BT 定时器、16 位 AD 转换器、LCD 显示、键盘，可以划分为 4 个文件：BasicTimer.C、ADC16.C、LCD.C、Key.C。属于每个功能模块的函数写在相应的文件中，然后在相应头文件中声明对外引申的函数与全局变量。

做好文件划分和管理之后，每个文件都不会很长，如果需要修改或调试某个函数，打开相应模块的 C 文件，很容易找到。打开相应的头文件还可查看函数列表。这些代码还能被重复使用。假设另一项目也用到 LCD 显示器，只要把 LCD.C 和 LCD.h 文件复制并添加到新工程内即可调用各种 LCD 显示函数，避免了重复劳动。

这里用一个简单的例子说明 C 文件与头文件的关系：

假设我们编写一个数据处理功能模块 DataProcess.C，包含两个功能函数：数据求和与数据求平均值。新建 DataProcess.C 并添加进工程，为两个功能写代码：

```
int Sum(int a,int b,int c)          //3 个数据求和函数
{
  int y;
  y=a+b+c;
  return(y);
}
float Average(int a,int b,int c)    //3 个数据求均值函数
{
  float y;
  y=a+b+c;
  return(y/3);
}
```

假设在 main.c 文件内的某函数需要调用 DataProcess.C 内的两个函数，需要在 main.c 开头用 extern 关键字声明外部函数，告诉编译器这两个函数位于其他文件。

```
extern int Sum(int a,int b,int c);            //声明 Sum 是外部函数
extern int float Average(int a,int b,int c);  //声明 Average 是外部函数
```

为了避免重复劳动，可以建立 DataProcess.h 头文件，将上述两句写入头文件内。在 main.c 文件开头处包含`"DataProcess.h"`，相当于作了函数声明。

仿照上面的方法，在编写程序时，为每个功能模块都写一个 C 文件和一个同名的头文件。在 C 文件内写代码；在头文件内写对外引申的函数声明。若在 File_A 文件中需要调用 File_B 文件内的函数，只需在 File_A 文件的开头添加 `#include "File_B.h"`即可。在工程管理器中，还提供了文件夹功能。当文件数目增多的时候，可以通过建立文件夹来分类管理文件。在工程名上右键->Add->Group 新建文件夹，然后用鼠标拖动文件放入文件夹。将头文件和源文件分别放在 Header 和 Source 文件夹是一种常用的文件夹管理方法，使文件列表长度减少了一半。当文件继续增多时，可以按照功能层次或类别来建立文件夹，还可建立子文件夹，将文件归类管理。

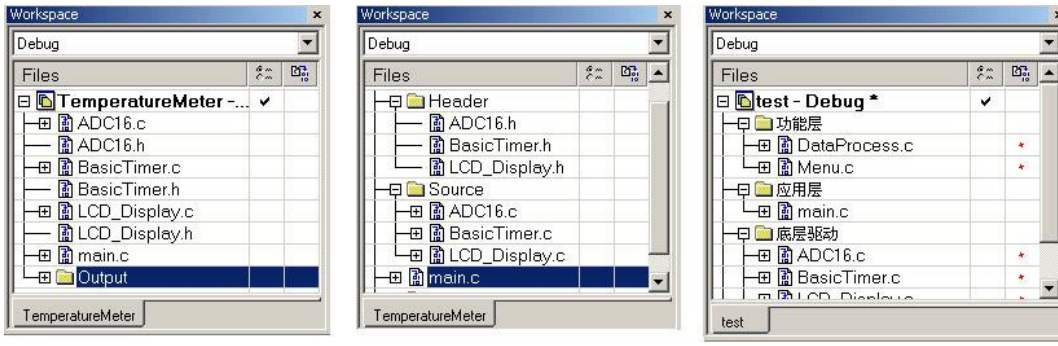


图 1.4.1　用文件夹功能管理文件

文件划分和管理不仅方便了阅读和调试，还可以为每个文件独立设置属性，如优化级别等参数，这些设置都保存在工程文件中。目前流行的开发工具都具有工程管理器，已经不再像 TC 那样支持独立 C 文件的编译。所以，建议初学者在开始就要养成文件管理的习惯，即使对于简单的程序，也要按模块划分和管理文件，对以后提高工作效率有很大帮助。

在编写程序的时候，也要注意所写的函数尽量能被重复使用。

通过各种项目积累下来的一些通用文件，可以作为程序库使用，对于一个对开发团队来说，积累的程序库是一笔宝贵的财富。一个优秀的程序库，应该屏蔽掉底层所有复杂的特征，对外呈现简洁的接口；并且具有通用性和可移植性。本书光盘里附带了MSP430FE42X 单片机所有内部资源的程序库，供读者参考或直接用于项目开发，也可移植到别的处理器上。

1.5 代码优化

在编译过程中，同一段 C 语言代码可以有不同的编译结果。某些编译结果速度特别快，但可能占用 ROM 比较多，某些编译结果可能占用 ROM 少，但可能执行速度慢。例如对于同一条 C 语言语句：

```
unsigned char Array[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

若在编译过程中，用 15 条 MOV 指令来实现赋值，则运行速度最快，但代码较长。若结合循环指令来实现，代码能够明显变短，但运行速度相对会变慢。

为了让程序员能够对编译过程有所控制，编译器留有一个优化选项，让程序员选择编译生成代码的优化倾向以及优化程度。在工程管理器内工程名上右键打开 Option 菜单，C/C++ Compler 项选择 Optimization 页，可以看到如下的优化菜单。

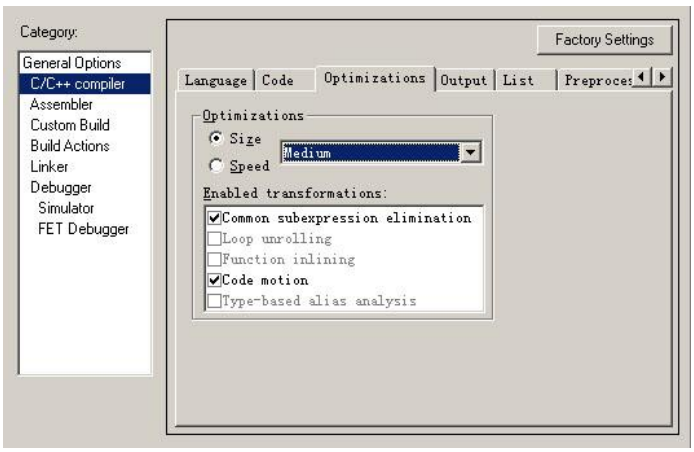


图 1.5.1 优化选项菜单

用户可以指定优化倾向是速度(Speed 选项)或者代码大小(Size 选项)。若选择优化速度，编译器在生成机器码时会尽量让代码速度执行的更快，以提高执行性能。若选择优化代码大小，则编译器在生成机器码时会尽量让代码长度更短，以便在同样容量的存储器内装入更多代码，降低成本。

右侧的选择框用于指定优化级别，有高（High）、中（Medium）、低（Low）、无（None）四档优化级别。优化级别越高，机器码生成得越紧凑。但优化级别较高时，编译器在保证运行结果正确的前提下会自作主张修改代码。比如会删除部分没有用到的代码、将程序中若干处都频繁出现的相同语句合并、移动代码位置、将次数较少的 for 循环展开等。优化级别越高，编译器对代码进行的修改就越多。在下方“Enabled transformations”提示框里

会告知当前优化级别下，编译器会对代码进行哪些修改。

优化级别越高，能生成越小、越快、越紧凑的代码，但调试越困难。优化级别较高时，可能会出现某些语句根本不执行的情况（被优化器删除或挪动了），或者发现某些运算的中间结果完全是错误的（中间过程被编译器省略，或者与其他运算合并了），从而给测试和排错工作带来很大困难。建议在程序调试完毕之前，先用低级优化或关闭优化功能。程序发布前再选择高优化级别。

若读者在调试过程中，即使将优化级别设成“无”，仍然有部分变量被优化过程改变，可尝试在定义变量时加 volatile 关键字。

1.6 风 格

程序不仅要被计算机读，还要给程序员读。一个风格清爽而严谨的程序更容易被读懂，更容易被修改和排错。良好的编程风格和正确的习惯还有助于保持思维清晰，写出正确无误的代码。特别是一个开发团队共同工作时，保持一致的编程风格尤其重要。

目前单片机开发人员对编程风格问题重视度还不够。事实上，每个初学者在项目初期都会因为不良编程习惯浪费大量时间，因此若能在开始写程序时就重视编程风格问题，对顺利渡过提高阶段有很大帮助。篇幅所限，本节仅浅述编程风格几个最基本原则。

l 变量命名规则

变量名尽量使用具有说明性的名称，避免使用 a,b,c,x,y,z 等无意义字符。使用范围大的变量，如全局变量，更应该有一个说明性的名称。变量名尽量使用名词，长度控制在 1~4 个单词最佳。若名称包含多个单词，每个单词首字母大写以便区分单词，例如：

```
int InputVoltage;   //输入电压
int Temperature;    //温度
```

当单词间必须出现空格才好理解的时候，可以用下划线’_’替代空格：

```
int Degree_C;       //摄氏度
int Degree_F;       //华氏度
```

当单词较长的时候，可以适当简写：

```
int NumOfInputChr; //输入字符数
int InputChrCnt;    //输入字符数
int Deg_F;          //华氏度
```

一旦约定某方式简写，以后必须保持风格统一。

若多个模块都可能出现某个变量，可以按“模块名_变量名”的方式命名：

```
char  ADC_Status;       // ADC 的状态
char  BT_IntervalFlag;  // BasicTimer 定时到达标志
int   UART_RxCharCnt;   // 串口收到的字符总数
```

对于约定俗成的变量，如用变量 i、j 做为循环变量，p、q 作为指针，s、t 表示字符串等，不要改动。这里采用更长的变量名反而不符合习惯。

l 函数命名规则

和变量一样，函数名称也应具有说明性。函数名应使用动词或动作性名字，后面可以

跟名词说明操作对象。按照“模块名_功能名”的方式命名：

```
unsigned int  ADC16_Sample();   //16 位 ADC 采样
char          LCD_Init();       //LCD 初始化
char          RTC_GetVal();     //获取实时钟的数据
void          PWM_SetPeriod();   //设置 PWM 周期
void          PWM_SetDuty();    //设置 PWM 占空比
void        Flash_WriteChar();  //向 Flash 写入一字节数据
char        UART_GetChar();     //从串口读取一字节数据
char        Key_GetKey();       //从键盘读取一次按键
char        TouchPad_GetKey();  //从触摸板读取一次按键
```

每个单词首字母大写，便于阅读，专有名词或缩略词(ADC/LCD 等)全部大写。遇到太长的单词也可以在不影响阅读的情况下适当简写，例如用 Tx 替代单词 Transmit，Rx 替代单词 Receive，Num 替代单词 Number，Cnt 替换 Count，数字 2（Two）替代单词 To 等。和变量一样，一旦约定某种简写方式，以后必须保持风格统一。对于所有模块都通用的函数，如求均值函数，可以不写模块名。

对于返回值是布尔类型值的函数（真或假），名称应清楚反映返回值情况，例如编写某函数检查串口发送缓冲区是否填满：

```
char UART_CheckTxBuff();    //不恰当的函数名
char UART_IsTxBuffFull();   //意义明确的函数名
```

第一个函数字面意思是“检查发送缓冲区”，若返回真(1)表示什么意思不明确。

第二个函数字面意思是“发送缓冲区满了么？”，若返回真(1)有明确的意思：满了。

表达式

表达式应该尽量自然、简洁、无歧义。写代码的时候，要杜绝各类“技巧”。我们的目标是要写最清晰的代码，而不是最巧妙的代码。下面 2 个表达式所表达的条件是等价的，第一个逻辑拐了个弯，难以理解，写成第二种表达方式就清晰许多。

```
if(!(RxCharNum<20)||(!(RxCharNum>=16)) //晦涩的表达式
if((RxCharNum>=20)||(RxCharNum<16)) //清晰的表达式
```

再看下面的表达式想要干什么？

```
SubKey=SubKey>>(Bits-(Bits/8)*8);      //难懂的表达式
```

最内层表达式把 Bits 除以 8 再乘以 8，实际上相当于把最低三位清零。再从 Bits 原值减去这个结果，实际上相当于取 Bits 的最低 3 位。最后用这三位确定 SubKey 的移位次数。实际上与下面简洁的表达式等价：

```
SubKey=SubKey >> (Bits & 0x07);  //清晰的表达式
SubKey >>= (Bits & 0x07);        //清晰的表达式
SubKey >>= (Bits%8);             //清晰的表达式
```

一个好的表达式应该能够用英语朗读出来。可以作为检验表达式好坏的依据。例如：

```
if(UART_IsTxBuffFull())  UART_ClearTxBuff();
else                     UART_PutChar(0x55);
```

上面的表达式可以被朗读出来：“如果串口发送缓冲区满了，就清空发送缓冲区，否则从串口发送字符 0x55”，可读性很好。

表达式不仅要清晰，还要消除歧义。若初学者对 i++ 和++i 顺序比较含糊，完全可

以将表达式拆开避免歧义。若对运算符优先级问题没有把握，可以用括号消除可能出现的歧义。

风格一致性

对于书写格式，向来争议很大，例如括号配对就有 2 种流行的写法：

```
for(i=0;i<100;i++){
   for(j=0;j<200;j++){      //括号配对风格 1
   ...
   }
}

while(a==b){
   if(c==d){                //括号配对风格 1
   ...
   }
   else{
   ...
   }
}
```

另一种写法是

```
for(i=0;i<100;i++)
{                           //括号配对风格 2
   for(j=0;j<200;j++)
     {
      ...
     }
}
while(a==b)
{
   if(c==d)
     {                      //括号配对风格 2
       ...
     }
   else
     {
      ...
     }
}
```

除了书写风格，命名方法也有很多种标准，且经常可以看到哪种格式更好的讨论。事实上，最好的格式、最好的命名方法是和惯用风格保持一致。如果加入一个开发团队，团队目前所使用的风格就是最好的格式；如果改写别人写的程序，保持原程序的风格就是最好的风格。总之，程序的一致性比本人的习惯更重要。如果初学者还没有形成自己的风格，那么可以参考官方提供的范例程序，或者与平台供应商的代码保持风格一致。

注释

注释是帮助程序读者的一种手段，但是如果注释只是代码的重复，将会变得毫无意义。若注释与代码矛盾，反倒会帮倒忙。最好的注释是简洁明了的点明程序的突出特征，或者阐明思路，或提供宏观的功能解释，或者指出特殊之处，以帮助别人理解程序。比较同一

段代码的两种注释方法：

```
 for(i=6;i>DOT;i--)              //从第 6 位(最高位)到小数点之间依次递减
  {
    if (DispBuff[i]==0) DispBuff[i]=' '; //如果该位数值是 0，则替换成空格
    else break;                          //如果不是，则跳出循环
  }
```

另一种注释方法

```
 for(i=6;i>DOT;i--)
  {
    if (DispBuff[i]==0) DispBuff[i]=' '; //消隐显示数据小数点前的无效 0
    else break;
  }
```

第 1 种注释每行都有注释，但读者仍然不明白这段程序功能或目的是什么。因为每个注释无非是代码的解释和重复。第 2 种注释虽然只有 1 行，但简明扼要的说明了这个 for 循环的功能，帮助读者理解程序。写注释的时候要从读程序的思路来考虑而不要以设计者的角度思考。

对于每个函数，特别是底层函数，都要注释。在函数前面注释该函数的名称、参数、参数值域、返回值、设计思路、功能、注意事项等。如果参与团队开发，还应写若干典型应用范例，供他人参考。商业化的代码中，注释比程序长的情况很常见。摘录光盘附带的显示库程序 LCD.C 文件中的一个函数为例：

```
/*********************************************************************
* 名    称：LCD_InsertChar()
* 功    能：在 LCD 最右端插入一个字符。
* 入口参数：ch  :插入的字符  可显示字母请参考 LCD_Display.h 中的宏定义
* 出口参数：无
* 说    明：调用该函数后，LCD 所有已显示字符左移一位，新的字符插入在最右端一位。
           该函数可以实现滚屏动画效果，或用于在数据后面显示单位。
* 范    例：LCD_DisplayDecimal(1234,1);
           LCD_InsertChar(PP);
           LCD_InsertChar(FF);  显示结果：123.4PF
*********************************************************************/
void LCD_InsertChar(char ch)
{  char i;
   char *pLCD = (char *)&LCDM1;            //取 LCD 控制器显存地址
   for(i=6;i>=1;i--) pLCD[i]=pLCD[i-1];    //已显示内容全部左移 1 位
   pLCD[0]=LCD_Tab[ch];                    //新字符显示在屏幕最右侧
}
```

在代码维护、调试与排错时，若修改代码，要养成立即修改注释的习惯，否则很容易出现代码与注释不一致的情况，很可能造成难以排查的错误，严重影响工作效率。

I 宏定义

数值没有任何能表达自身含义的可读性，因此对于程序中出现的数值，他们应该有自己的名字。一般可以用宏定义来实现，并且用宏定义处理数据之间的关系。在寄存器操作章节中，我们已经看到头文件中对寄存器地址、标志位等都作了宏定义，用宏对寄存器赋

值后，程序立刻有了可读性。类似的，我们可以用宏定义来对常数数值赋予可读性。

```
#define TXBUFF_SIZE  (128)                      /*发送缓冲大小*/
#define LCM_ROW      (64)                       /*点阵液晶行数*/
#define LCM_CLUMN    (128)                      /*点阵液晶列数*/
#define LCM_BUF_SIZE  (LCD_CLUMN*LCD_ROW/8)   /*点阵液晶缓冲区大小*/
```

将常数值用宏定义之后写出来的代码可读性增强了：

```
unsigned char TxBuff[TXBUFF_SIZE];              //定义发送缓冲区
char IsTxBuffFull()
{
   if(NumOfTxChars>=TXBUFF_SIZE) return(1);    //缓冲区是否满?
   else                          return(0);
}
```

在上例中，一旦需要改变缓冲区大小，只需要修改宏定义即可。宏定义属于字符型替代，因此在使用宏定义的时候要注意防止产生歧义。例如数据全部加括号，以免和程序前后文构成意料之外的运算优先级。宏定义后的注释使用 /**/而不要用// ，以免某些版本的编译器在代码中将宏定义连同注释全部替换造成错误。

使用宏定义还要防止定点计算溢出，例如：

```
#define VOLT_RATE  (1000)                       /*比例系数*/
...
int Voltage;
int InputValue;
...
Voltage=InputValue*VOLT_RATE;                  //可能溢出
```

若将宏定义做如下修改即可避免溢出问题：

```
#define VOLT_RATE  ((long)1000)                 /*比例系数，强整成 long*/
```

宏定义还有许多用途，比如增加程序可移植性，进行软件版本管理等，下一节将详细讨论。

1．7　可移植性

虽然本书的主题是 430 单片机用法与软件设计，但作为今后的嵌入式软件的设计者，读者编写的每个程序都可能会移植到不同的硬件环境或者其他的处理器平台上运行。了解一些移植性的概念和基本知识对今后的开发很有帮助。

例如 LCD 的显示程序可能会被移植到数码管上实现，串口通讯程序可能会被移植到 8051 系列或 ARM 系列的单片机上运行等等。若读者写出的代码移植性强，这个工作将会是愉快而有趣的，否则将是重写大部分代码的枯燥重复劳动。

前文已经引出了移植性的概念，如果我们在编程过程中注意尽量消除不同处理器硬件上或语法上的差异，或者将差异集中到某个很小的局部，那么这个程序通过很简单的修改就能编译成另一 CPU 的机器码，在其他处理器上运行。考虑到读者大部分是初学者，所接触的单片机不多，这里通过 MSP430 系列单片机和 8051 系列单片机作对比，举几个简单的例子帮助读者培养移植性的概念。

消除 CPU 差异

若希望自己写的程序在不同处理器上运行，首先需要了解两者的不同之处。其中包括硬件的不同以及特殊语法的差异。例如 430 单片机没有位操作，51 单片机有，若两者之间的程序需要相互移植，首先需要消灭这个差异。下面示范用宏定义来消除 IO 口位操作的差异：

```
#include "MSP430X42X.h"                 /*430 单片机 */
#define LED_ON   P2OUT |= BIT0          /*LED 亮 */
#define LED_OFF  P2OUT &=~BIT0          /*LED 灭 */
```

以后的程序中，所有控制 LED 的语句均通过这两个宏定义进行，不要再直接操作硬件。一旦需要将这段程序移植到 51 单片机上，只需修改宏定义：

```
#include "reg51.h"          /*51 单片机*/
sbit LED=P2.0;              /*定义 IO 口，(51IO 口低电平才能驱动 LED)*/
#define LED_ON   LED=0      /*LED 亮*/
#define LED_OFF  LED=1      /*LED 灭*/
```

此后整个程序中所有 LED 的控制都无需修改。读者应该养成使用 IO 口之前先定义的习惯。

消除硬件差异

即使同一款单片机，在不同项目中硬件接法仍可能导致程序不通用。以 LCD 或数码管显示为例，假设连接 LCD 或数码管的 IO 口顺序发生改变，结果将是显示数据与实际段码的关系发生改变。这意味着要重写显示段码表。而且很不幸，出于布线方便的考虑，LCD 或数码管的八段连接关系经常被随意调整——反正显示表的调整可以交给软件处理，布线过程怎么方便怎么连。

仔细分析上述问题，我们会发现，字形与实际亮的笔划之间的关系是永远不变的，例如需要显示‘1’，永远是右侧两段亮（b 段和 c 段）。真正改变的只是各个段在一字节显示数据中的比特位。利用这一点，可以利用宏定义自动生成段码表：

```
//-------------------------------------------------------------------
/      *宏定义，数码管 a-g 各段对应的比特，更换硬件只用改动以下 8 行*/
//-------------------------------------------------------------------
#define d      0x01                     //  AAAAA
#define g      0x02                     // F    B
#define b      0x04                     // F    B
#define a      0x08                     //  GGGGG
#define DP     0x10                     // E    C
#define e      0x20                     // E    C
#define f      0x40                     //  DDDDD   DP
#define c      0x80
//-------------------------------------------------------------------
/                *用宏定义自动生成段码表，请勿修改*/
//-------------------------------------------------------------------
const char LCD_Tab[] =                  //LCD 段码表，放在 ROM 中
{
  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"
};
#undef a                            //清除宏定义，以免和变量名冲突
#undef b
#undef c
#undef d
#undef e
#undef f
#undef g
```

宏定义将 a、b、c、d、e、f、g 七段分别定义为 0x01~0x80 ，分别对应各段的比特位。因为显示数字与 a~g 七段之间的关系永远不会改变，可以用加法运算自动生成段码表。例如显示数字 7 要亮 a、b、c 三段，那么段码表中数字 7 的字形就是 a+b+c。具体 a、b、c 三段对应的 IO 口所在比特位是多少，交给宏定义去修改。所以只需改动 a~g 的比特位定义即可应付各种硬件改动。

除了宏定义之外，函数也经常被用来作为消除硬件差异的手段。比如在我们写`Motor_ON()`和`Motor_OFF()`两个函数来控制电动机的启停。某直流电机控制系统中可能只是简单的 IO 口赋高低电平值就能实现控制；在步进电机控制系统中需要控制步进发生定时器的启停；而另一大功率交流系统中需要通过串口发送指令控制变频器缓慢增减转速。无论电动机控制方法如何变化，只需要修改这两个函数，整个上层软件保持不变。

I 软件层次划分

一个完整的软件，往往需要很多功能部件的配合。这将导致大量的函数，以及复杂的函数间调用关系。对于初学者来说，写一个完成设计要求的程序也许并不困难，但是整理清楚函数之间的关系可能是一件复杂的事情。如果从开始写程序就能了解函数分层的思想，可以将函数间的联系复杂性降到最低。调用关系的简单化在某种程度上意味着可移植性的提高。下面以菜单和人机界面程序作为范例说明一些基本概念：

菜单和界面可以看做是单片机和操作者沟通的一种手段，至少包括输入和输出设备。在本书附带的学习板上，段码液晶（LCD）是最常用的输出设备，用于显示数据和提示符。小键盘是最常用的输入设备。菜单肯定要操作底层的设备，但如果我们在菜单内直接读写硬件，程序将会变得冗长且复杂。做个直观的比喻：从菜单到底层硬件之间有很长的“距离”，如果我们直接跨越这么大的距离，将会很困难，如果把这个距离划分成几个小台阶，每一步都走的很轻松。

下面的图示范了一种典型的层次划分结构。我们将菜单软件模块划分成 4 个层次：应用层、功能函数层（模块程序库）、硬件隔离层、硬件驱动层。软件分层的最核心思想是每一层的函数只能调用下一层的函数，决不跨层调用。例如菜单需要显示某些内容，它只

能调用 LCD 功能层的 6 个函数组合来实现，决不允许操作显示缓冲区，更不允许操作硬件。

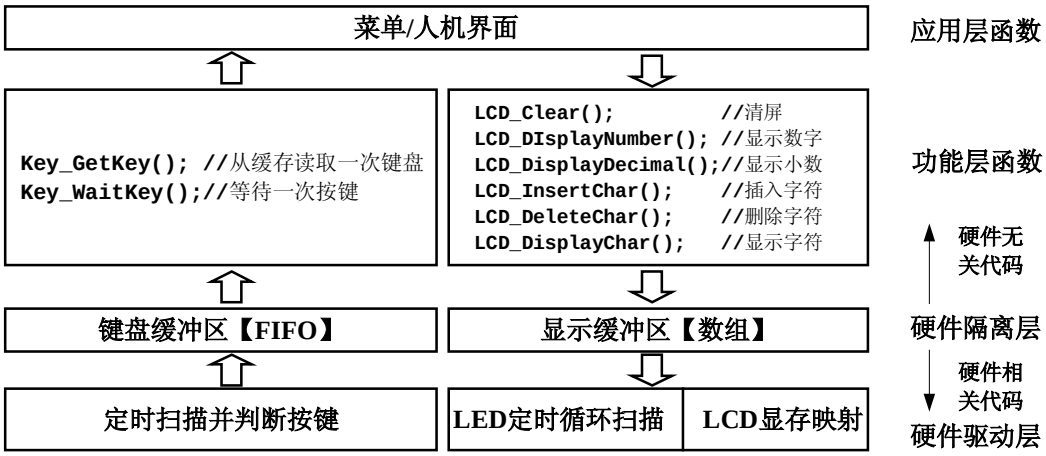


图 1.7.1 人机接口的函数层次划分图

同样的，LCD 功能层的 6 个函数，只能写显示缓冲区，决不能操作硬件。硬件驱动层只负责将显示缓冲区映射到显示设备上，而不管显示缓冲区数据是如何得来的。对于每一层来说，都杜绝了其前后层函数之间的调用。因此有个形象的比喻：每一层都是“不透明的”，遮挡了其前后两层，仅有相邻的层之间是“可见的”。

软件分层之后，条理会变得非常清晰。比如编写 LCD_DisplayNumber(int Num)函数的时候，仅需要考虑怎样将传入的 int 型参数 Num，转换成显示缓冲区内的 LCD 段码值。可以集中精力解决完成该功能所涉及的细节问题，比如负数显示、无效 0 消隐等，不必考虑硬件如何显示、如何被菜单调用等无关问题。

软件分层之后带来的另一好处就是可移植性的提高。每一层都可以被替换，只要保证函数接口一致即可。假设读者在另一个设计中需要将 LCD 替换成动态扫描的数码管，只需要将显示缓冲区内容写入 LCD 的函数删除，替换成定时中断扫描数码管程序。中断内循环扫描程序只负责将显示缓冲区的数据映射到数码管上，程序就被移植到了数码管显示的硬件平台中，且整个上层建筑完全不用做任何改变。

软件分层还能够将大型软件任务分块，团队作业。由项目总管负责划分软件层次和结构，将不同层次的函数功能要求、接口规范等标准发给若干程序员，每个程序员只用负责自己的函数功能与接口正确，最后就能组合成一个大软件。

这里最值得一提的是“硬件隔离层”。硬件隔离层也被称为“硬件抽象层”（Hardware Abstract Layer 简称 HAL）。一般用宏定义、缓冲区等手段，将硬件特征消灭。在硬件隔离层之上，所有的函数都不允许直接操作硬件，不允许体现出任何硬件特征。有了硬件隔离层，之上的任何函数都是硬件无关的，因此任何硬件上的改动都不会影响到整个庞大的上层软件。作为一个嵌入式软件程序员，面对一个新的平台（新的处理器或者新的硬件），首要任务就是隔离硬件。

接 口

接口（Interface）是一个很宽泛的概念。可以广义的理解为两个部件之间的连接。对于每一个函数来说，传入的参数就是输入接口，返回值是输出接口。对于上面软件分层的例子，LCD 模块对外提供了 6 个硬件无关的函数。这 6 个函数可以看作是 LCD 模块对上一层函数的接口。LCD 模块操作的是显示缓冲区，因此缓冲区是 LCD 模块对下一层函数的接口。若干个模块最终构成整个菜单界面，实现人机接口。所以说从微观到宏观，接口无处不在。这一节我们主要讨论如何规划和设计模块对外呈现的接口。

对于规划模块函数库来说，第一个问题是要对外提供哪些服务和访问？例如，LCD 模块要提供哪些显示功能函数（服务），提供哪些函数用于读取状态（访问）。事实上我们可以编写无数种功能各异的显示函数，不过很快就会发现功能过于凌乱，反而难以驾驭。因此，使用方便，具有丰富功能，又不至于过多过滥以至于无法控制是规划模块接口的原则之一。

通过观察各种菜单和界面发现，一般的 LCD 都会遇到显示数字（包括正负数）、显示小数、显示字母（数字之前）、显示单位（数字之后）、删除字符、清屏等功能操作。还可能在低功耗应用中遇到关闭 LCD 以省电等要求。以此为目标，规划 LCD 所需函数，力求少而精。最后规划为如下 9 个函数。其中 6 个硬件无关的函数位于功能层，3 个硬件有关的函数位于硬件驱动层。

```
void LCD_Init();                                    //初始化 LCD (硬件有关)
void LCD_ON();                                     //开启 LCD    (硬件有关)
void LCD_OFF();                                    //关闭 LCD    (硬件有关)
void LCD_DisplayDecimal(int Number, char DOT);//显示小数，小数点后留 DOT 位
void LCD_DisplayNumber (int Number);           //显示整数 Number
void LCD_DisplayChar   (char ch,char Location);//在指定位置显示字符
void LCD_InsertChar(char ch);//从 LCD 右侧插入一个字符，可用于显示单位或做动画
void LCD_DeleteChar(char ch);//从 LCD 右侧删除一个字符，可用于删除操作或做动画
void LCD_Clear(); //清屏
```

这份 LCD 函数清单所具有的功能可以满足大部分菜单和显示要求，但对于同时显示 2 个数字、某一位闪烁等功能仍然不能实现。读者可以尝试自行编写这些函数。

同样的，模块对下层的接口也要简洁。以一个实时钟的例子说明模块对下层接口的设计。实时钟（Real Time Clock，简称 RTC）在系统中的功能是提供当前年、月、日、时、分、秒等时间和日历信息，即使设备关机后仍然不停走时。在传统的设计中需要专用芯片来实现（DS1302/DS12887 等）。而在大部分 430 单片机系统中，一节电池能工作好几年，几乎可以不考虑关机和断电问题，所以 RTC 以可以由软件来实现。运行所需的 CPU 开销折合成功耗约 2uA。

实时钟对上层的接口很明确：设置时间信息和读取时间信息两个函数。下层接口就不那么明显了，在最底层，RTC 需要走时，一般都在定时中断内让时钟秒递增实现走时。如果直接占用一个定时器，在 1 秒一次的中断内写程序当然能完成设计（新手大部分会这么做）。但这样带了移植性问题：不同的系统中可能会用不同的定时器来驱动 RTC，或者需要和某些其他模块复用一个定时器，定时器的定时周期也可能不一样。或者移植到别的处理器上之后，定时器用法完全不一样。这些问题很繁琐，但如果仔细思考，能将差异归类为 3 种：定时器不同，定时周期不同，定时器用法不同（另一处理器）。再深入一些，会

发现一个令人愉快的结论：这些差异是正是硬件隔离层要解决的问题，把 RTC 走时部分的程序做成一个函数就能充当硬件隔离层。

我们编写一个与硬件无关的 RTC_Tick()函数，专门负责走时工作。这就解决了定时器用法的不同和定时器不同两个问题。无论更换处理器，还是更换 RTC 走时的定时器，只需要在对应的定时中断内调用 RTC_Tick()函数即可。

```
void RTC_Tick(int DivSec)
{ char Days;
  DSEC++;
  DISABLE_INT; //关闭中断以免运算到一半时被其他中断内的函数读取时间。
  if(DSEC  >=DivSec)  {SECOND++;DSEC=0;}    //1 秒一次
  if(SECOND>=60)      {MINUTE++;SECOND=0;}  //60 秒一次
  if(MINUTE>=60)      {HOUR++;MINUTE=0;}    //60 分一次
  if(HOUR  >=24)      {DATE++;HOUR=0;}      //24 小时一次
  if(MONTH==2)                              //处理润年 2 月份问题
  {
    if(YEAR%4==0)   Days=29;                //逢 4 年润二月
    else           Days=28;
  }
  else           Days=MONTH_Table[MONTH-1]; //正常月份，查表得到当月天数
  if(DATE  >Days) {MONTH++;DATE=1;}         //一个月一次
  if(MONTH >12)   {YEAR++;MONTH=1;}         //一年一次
  if(YEAR>=100)   {YEAR=0;}                 //100 年一次
  RESTORE_INT;//恢复中断  (关闭和恢复中断都是宏定义，不同处理器的语句不一样)
}
```

还剩下最后一个问题：不同应用中可能定时周期不一样。系统中可能无法专门为 RTC 专门提供一个定时器，可能要和某些定时中断复用，那么应该设计一种方法让 RTC_Tick 函数知道这个定时器的中断频率，才能让秒走的准确。最简单的办法是将定时中断频率通过参数（DivSec）传进函数。例如需要在 5ms 定时中断内走时，则调用 RTC_Tick(200)，在 1/32 秒定时中断内走时，则调用 RTC_Tick(32)。至此，RTC 所有的函数都变成了硬件无关代码，并给出了一个简洁的接口应付各种硬件变化，这是一个令人愉快结果。

1 屏 蔽

在物理学中，“屏蔽”一词的意思是不让干扰电波进出某个区域。软件中借用了“屏蔽”这个概念，指的是不让软件中某个区域的细节（或者说差异）暴露出来干扰编程者的思路。一段优秀的程序，能对所描述的对象高度抽象，让后续编程者看不到繁琐而复杂实现的细节。

先看一个例子。我们把手机比作一个软件模块：任何两款手机都不同，但用户拿到一款新手机后，无需学习，在键盘上输入电话号码再按拨号键就能打电话。因为不同款手机的拨号操作都是一样的，手机的软件屏蔽了拨号功能具体实现的细节，如果追究细节，CDMA 和 GSM 手机的拨号过程之间是差异巨大的。类似的思想，如果我们在编写软件的时候，注意屏蔽细节，保留共性，对可移植性的提高帮助很大。

下面用串口设置函数为例简单说明这一概念。先思考串口需要设置的参数有哪些共

性？哪些是编程者应该看到的，哪些是不应该看到的？通过对比 51、430、ARM、VB、VC 等不同硬件和不同编程软件之间的区别，我们会发现：串口硬件结构千差万别，寄存器操作方法、波特率的产生原理各不相同，但这些差别最终都是为了改变 4 个参数：波特率、校验方式、数据位位数、停止位位数。

我们把相同的部分抽象出来，作为参数传入 UART_Init()函数。例如，我们需要将串口设置为波特率 2400bps，无校验，8 位数据，1 位停止位，我们希望只需简单的调用：

```
UART_Init(2400,'n',8,1);    //设成 2400bps，无校验，8 位数据，1 位停止位
```

该函数对外应该屏蔽掉实现过程中所有的细节。包括寄存器设置、时钟选择、波特率计算等等复杂的过程，全部都应在 UART_Init()函数内完成。只保留具有工程意义的参数作为简洁的对外接口。遵照这个思路，写出 MSP430F42x 系列单片机的串口设置函数供参考：

```
/*************************************************************************
* 名    称：UART_Init()
* 功    能：初始化串口。设置其工作模式及波特率。
* 入口参数：Baud        波特率    (300~115200)
           Parity      奇偶校验位('n'=无校验  'p'=偶校验  'o'=奇校验)
           DatsBits    数据位位数    (7 或 8)
           StopBits    停止位位数    (1 或 2)
* 出口参数：返回值为 1 时表示初化成功，返回 0 表示参数出错，设置失败
* 范    例：UART_Init(9600,'n',8,1) //设成 9600bps，无校验，8 位数据，1 位停止位
          UART_Init(2400,'p',7,2) //设成 2400bps，偶校验，7 位数据，2 位停止位
*************************************************************************/
char UART_Init(long int Baud,char Parity,char DataBits,char StopBits)
{
   unsigned long int BRCLK;      //波特率发生器时钟频率
   int FreqMul,FLLDx,BRDIV,BRMOD; //倍频系数、DCO 倍频、波特率分频系数、分频尾数
   int i;
   char const ModTable[8]={0x00,0x08,0x88,0x2A,0x55,0x6B,0xdd,0xef};
          //分频尾数所对应的调制系数(将 0~7 个"1"均匀分布在一个字节的 8bit 中)
   //-------------设置波特率发生器时钟源，并计算波特率时钟频率--------------
   UTCTL0 &=~(SSEL0+SSEL1);    //清除之前的时钟设置
   if(Baud<=4800)
   {
     UTCTL0 |= SSEL0;          //低于 4800 的波特率，用 ACLK，降低功耗
     BRCLK=F_ACLK;             //获得波特率发生器时钟频率=ACLK
   }
   else
   {
     UTCTL0 |= SSEL1;          //高于 4800 的波特率，用 SMCLK，保证速度
     FreqMul=(SCFQCTL&0x7F)+1; //获得倍频系数
     FLLDx=((SCFI0&0xC0)>>6)+1;//获得 DCO 倍频系数(DCOPLUS 所带来的额外倍频)
     BRCLK=F_ACLK*FreqMul;     //计算波特率发生器时钟频率=ACLK*倍频系数
     if(FLL_CTL0&DCOPLUS) BRCLK*=FLLDx; //若开启了 DCOPLUS,还要计算额外倍频
   }
//------------------------设置波特率-------------------------
  if((Baud<300)||(Baud>115200))  return(0);  //波特率范围 300-115200bps
```

```
  BRDIV=BRCLK/Baud;                       //计算波特率分频系数(整数部分)
  BRMOD=((BRCLK*8)/Baud)%8;               //计算波特率分频尾数(除不尽的余数)
  UBR00 = BRDIV%256;
  UBR10 = BRDIV/256;                      //整数部分系数
  UMCTL0 = ModTable[BRMOD];               //余数部分系数作小数分频
//------------------------设置校验位-------------------------
  switch(Parity)
  {
   case 'n':case'N': U0CTL&=~PENA;    break;          //无校验
   case 'p':case'P': U0CTL|= PENA+PEV ;break;         //偶校验
   case 'o':case'O': U0CTL|= PENA; U0CTL&=~PEV;break; //奇校验
  default : return(0);                                //参数错误
  }
//------------------------设置数据位-------------------------
  switch(DataBits)
  {
   case 7:case'7': U0CTL&=~CHAR; break;    //7 位数据
   case 8:case'8': U0CTL|= CHAR; break;    //8 位数据
  default : return(0);                     //参数错误
  }
//------------------------设置停止位-------------------------
  switch(StopBits)
  {
   case 1:case'1': U0CTL&=~SPB; break;     //1 位停止位
   case 2:case'2': U0CTL|= SPB; break;     //2 位停止位
  default : return(0);                     //参数错误
  }
//-------------------------其他设置-------------------------
  P2SEL |= 0x30;                // P2.4,5 设为串口收发引脚
  ME1 |= UTXE0 + URXE0;         // 打开 TXD/RXD 部分电路的供电
  UCTL0 &= ~SWRST;              // 复位串口硬件
  IE1 |= URXIE0+UTXIE0;         // 允许 RX/TX 中断
  _EINT();                       //总中断允许
  for(i=0;i<4000;i++);          //略延迟，等待波特率分频稳定
  return(1);                    //设置成功
}
```

读者暂时不用去细读这段程序。事实上，调用 UART_Init(2400,'n',8,1) 时，上面“复杂”的函数和下面 10 行“简洁”的程序是等价的：

```
 P2SEL |= 0x30;                    // P2.4,5 设为串口收发引脚
  ME1 |= UTXE0 + URXE0;            //打开 TXD/RXD 部分电路
  UCTL0 |= CHAR;                   // 8-bit character
  UTCTL0 |= SSEL0;                 // 串口时钟源 = ACLK (32.768KHz)
  UBR00 = 0x0D;                    // 分频系数=32768/2400 = 13.65 =13+5/8
  UBR10 = 0x00;                    //
  UMCTL0 = 0x6B;                   // 分频尾数=5/8 (5 个 1 均匀分布在一字节内)
  UCTL0 &= ~SWRST;                 // 复位串口硬件
  IE1 |= URXIE0+ UTXIE0;           //允许 RX/TX 中断
```

```
_EINT();                                    //总中断允许
```

比较上面 2 个串口设置程序，前者程序复杂，但是接口简单。后者程序简单，却没有接口可言：将串口硬件中繁琐而复杂的细节完全甩给了程序员自己。初学者往往会写出“简洁”的版本，代价是每一次开发都要重复劳动，重新规划时钟、计算波特率、设置寄存器，重新查阅寄存器表，更致命的是没法移植到别的应用中去。

如果按照第一个版本来写初始化函数，代价是占用了更多的 ROM 空间、编程时要考虑到所有的情况、需要仔细研究所有相关寄存器的用法、花费更多的时间精力。但换来的是今后的永久“免维护”：在下一个设计中，可以直接重复使用这段代码。

对于像串口这类模式繁多、寄存器关系复杂的硬件设备，移植问题是繁琐而丑陋的，因为涉及到很多硬件寄存器。而且不同的处理器，机理各不相同。8051 用定时器 T1 溢出产生波特率，而 430 用系统时钟小数分频器来产生，大部分 ARM 系统用 16C550 兼容的控制器来产生……。解决的办法是为每一种处理器写串口设置函数时保证接口形式完全一致。利用函数来屏蔽实现细节的差别。

关于移植性的实际问题还有许多，限于篇幅只能讨论到此。深入的知识需要读者在多接触一些其他处理器后才能有所体会。如果读者能够写出移植性很强的代码，只要学会 **1-2** 种单片机，则相当于会用其他任何一款单片机（或处理器）。

1．8　版本管理

如果读者曾经或正在从事产品软件设计开发工作，每年会写数十甚至上百份不同的程序。特别对于嵌入式开发程序员来说，每个应用都是特殊的，更导致软件版本繁多。加上不同的客户可能会要求在产品上增加或删除某些功能，或者系列化产品中功能差异，都会导致类似功能的不同软件版本。当这些类似的软件不断增多，管理就成了很头疼的问题。

例如：用户反映在 A 版本产品中发现一个 bug，经技术部门分析是软件问题，bug 报告被提交到程序设计人员手中，分析后发现是一个并不严重的小问题。但随后的问题远比 bug 更可怕：版本 BCDEFG…的软件都存在同样的 bug，于是要对所有版本的软件都进行 bug 排查，然后每个版本都要再进行功能测试以确定问题都被消除，并且需要修改 bug 后，在不同版本中彻底检查是否会引起新的问题。这项工作有趣么？

如果程序员不希望自己被卷入类似的无穷无尽的代码维护工作中，首先应该将尽量减少软件版本。下面以一个产品为例，示范如何通过版本管理减少软件副本数量。

某公司设计一款手持测温仪，一个系列共有 5 种产品 ABCDE。从 A 到 E，功能逐渐增多，价格也依次递增。该测温仪总共有四家大客户：甲、乙、丙、丁。每家客户面对的购买群体略有差别，对 5 种产品的操作习惯略有不同。比如对于报警功能，甲要求上下限报警，而乙要求双上限报警；丙的应用场合比较特殊，不允许噪声，要求去掉按键声音；丁要求在-20 度低温使用，不能用液晶，只能用数码管。这样 4×5 组合将有 20 种不同的软件。

读者也许会发现，尽管整个系列产品需要 20 种不同的软件，但这些软件大部分是相同的。事实上，我们可以通过条件编译来管理并统一这 20 个软件。

新建一个 Config.h 文件，添加进工程，并在每个 C 文件开头都包含 Config.h 文件。通

过 Config.h 来配置软件的差异。下面摘抄 Config.h 文件的一部分：

```
//------------------------------------------------------------------------
//                          功能配置文件
//------------------------------------------------------------------------
#define ON          1
#define OFF         0
#define NONE        0
#define LEV_2       1
#define HI_LO       2
#define MAX         1
#define AVE         2
#define NORM        0
#define AVE         1
#define LCD         1
#define LED         0
//------------------------------------------------------------------------
//                        以下内容供生产部门修改
//------------------------------------------------------------------------
#define MINORCUT    OFF     /*是否打开小值切除功能                          */
#define RS485       ON      /*是否打开 RS-485 通讯功能                      */
#define DAC         ON      /*是否打开变送功能                              */
#define STORAGE     OFF     /*是否打开数据存储功能                          */
#define ALARM_MODE  NONE    /*报警模式 NONE=无报警 LEV_2=双限 HI_LO=高低限 */
#define OFFSET      ON      /*是否打开偏移补偿功能                          */
#define KEYTONE     ON      /*是否开启按键音                                */
#define PRINTER     ON      /*是否打开打印机功能                            */
#define DATETIME    ON      /*是否打开日历功能                              */
#define SAMPMODE    AVE     /*采样模式 NORM=正常 AVE=平均滤波  MAX=峰值保持 */
#define SCREEN      LCD     /*显示屏驱动 LED=数码管  LCD=液晶               */
```

在软件中，用条件编译宏来删除或添加某几段程序，从而改变软件实际功能。以按键音为例：

```
char Key_GetKey()               //读键盘函数
{  char Key;
   Key=Key_ReadFIFO();          //从键盘缓冲区读一个按键
#if(KEYTONE==ON)              /*-----KEYTONE 功能打开，才编译下面一句------*/
   If(Key!=NOKEY) BEEP(30);     //如果是有效按键，则蜂鸣器响 30ms
#endif                        /*----------------------------------------*/
   retuen(Key);                 //返回键值
}
```

如果 Config.h 里面配置了#define KEYTONE ON 才会编译蜂鸣器鸣响的程序，如果配置#define KEYTONE OFF 则不编译蜂鸣器鸣响程序。这样客户丙的需求得以满足，且不增加软件副本。类似的方法，程序员可以将 **20** 种软件全部统一到一个工程文件中，一旦发现 **bug**，修改一处相当于 **20** 个软件的 **bug** 全部得到修改。在此还要提醒读者：在第一版程序发布之后的任何修改，都要做好日志记录，以备今后查阅。

借用前面的概念，**config.h** 文件为源程序和生产部之间提供了一个接口。假设生产部门需要临时生产一批功能组合比较特殊的产品，不必再转达研发部门，可以直接修改配置文件，即可编译生成所需的代码。**config.h** 文件屏蔽了源程序的复杂性和细节，将研发人员才能进行的复杂的程序修改工作，通过简洁的接口，简化成生产工人也能操作的形式。

本章小结

本章涵盖了从软件入门到软件管理方面的知识，跨度比较大。后半部分内容看似与 **430** 单片机关系不大，但是在动手写程序之前如果对程序风格、移植性、软件管理等概念有所了解的话，能避免走很多弯路。许多初学者对单片机的学习和研究都过分沉湎于硬件，而忽视了单片机也是一款处理器，其灵魂仍然是软件。软件工程学的知识对单片机开发依然能起指导作用。

书中的范例大部分来自作者在研发工作中的经验总结，确切的说，应该是来自无数次失败的教训。如果工程开始的时候，能够注意移植性，下一个项目程序就不用重写全部代码；如果注意风格和注释，**3** 年前写的程序就不会看得一头雾水；如果注意版本管理，就不会被一个小 **bug** 弄得焦头烂额……

每次碰壁，作者都会想，如果项目开始之初有人能够提醒一下该多好。所以，本书将这部分内容放在了第一章，也是希望读者能尽量能少走弯路。

习 题

1. 在某 430 系统上，P1.3、P1.1、P1.4 口分别接了红色、绿色、蓝色三只 LED，均为高电平点亮。编写一个程序，让三色 LED 依次点亮，间隔时间约 1 秒。要求编译无错误无警告，下载到 MAGIC-430 学习板中实际运行。

2. 打开光盘中附带的《简易电子表》程序，编译并下载到 MAGIC-430 学习板中实际运行，测量工作电流。尝试自制水果电池或太阳能电池为它供电。

3. 光盘中附带的《简易电子表》程序，将全部代码写在一个文件中。尝试对这个文件进行划分，分成若干个独立的文件，并添加进工程。要求仍然编译无错误无警告，下载到 MAGIC-430 学习板中实际运行，与原始程序运行效果应该一致。

4. 为光盘中附带的《简易电子表》程序写注释。尽可能注释所有的函数，以及关键语句的功能。

5. 将范例中的串口设置函数移植到 51 单片机上（或读者熟悉的任何一款处理器），保持接口一致。

思考题：查阅资料，找出下列问题答案、思路或进展情况：

1. 如何为水果电池充电？

2. 海水能否用类似水果电池的方法被用来发电？遇到的最大的困难是什么？

3. 空间中的电磁波能量（手机辐射、电台广播等等）能否为 430 单片机供电？设想一种尽量简单的方案并努力实现。

4. 计算机能否用来自动写程序？目前的技术能做到什么程度？尝试寻找并下载一款能为 MSP430 单片机自动写程序的软件。

第二章　MSP430 单片机内部资源

通过第一章的学习，读者掌握了 MSP430 单片机开发软件的使用方法、软件设计原则和基本思想。本章将对 MSP430 单片机内部的各个模块逐个进行讲解和分析，并用工程实例示范各个模块的应用。

学习单片机的最好途径是实践。建议读者对于书上的每个范例要自己动手实践，并尝试用自己的想法写出类似的功能，或尝试实现自己的新想法。为了便于学习，读者可购买或参考本书最后一章自制一块试验板。该试验板选取了一款资源和功能相对丰富的 MSP430FE425 单片机，涵盖了大部分的常用资源，同时保持了较低的价格。本章将以该单片机的内部模块为主，同时也将介绍一些在 MSP430F1xx 单片机中的常用模块。

在开始学习一款 430 单片机（如 MSP430F42x）之前，读者应该事先准备好和这款单片机相关的四份文档资料：

首先是芯片数据手册《MSP430F42x DataSheet》（以下简称 DataSheet 或数据手册），数据手册里给出了该款单片机详细资料，包括引脚排列、总体框图、存储器组织结构、内部模块简介、电气参数指标。

其次是 MSP430F4xx 系列单片机的模块使用指南《MSP430F4xx UserGuide》，（以下简称 UserGuide 或用户指南），这份文档给出了每个功能模块的详细结构图、寄存器列表、控制字列表、寄存器与控制字的功能等详细内容，是最主要的参考文档。

第三是官方提供的程序范例 Code Example for MSP430x4xx。它由数十个短小的程序组成，用最简单语句示范每个模块的不同用法。是重要的软件参考资料。

最后，是 MSP430x42x 头文件，位于 IAREW430 安装目录\430\inc\目录下。头文件也是一份重要参考资料，因为头文件里面有大量的快捷组合宏定义可供使用，在编程时能减少劳动量，避免出错。建议读者先对头文件做好备份，以免文件在阅读过程中因误操作被修改。

在本书附带的光盘里，附带了上述资料。对于 MSP430 系列其他的单片机，读者可以在 TI 公司网站上(www.ti.com)下载相关资料。

2．1　MSP430 单片机选型

MSP430 单片机家族型号繁多，TI 公司用 3 或 4 位数字表示型号，其中第一位数字表示大系列。目前有 4 个大系列：带有液晶驱动器的 MSP430F4xx 系列单片机、不带液晶驱动器的 MSP430F1xx 系列单片机、16MIPS 高速 MSP430F2xx 系列单片机、一次性写入（OTP）型低价 MSP430C 系列。在每个大系列中，又分若干子系列。单片机型号中的第二位数字表示子系列号，一般子系列号越大，所包含的功能模块越多。最后 1 或 2 位数字表示存储容量，数字越大表示 RAM 和 ROM 容量越大。

430 家族中还有针对热门应用而设计的一系列专用单片机。如 MSP430FW4xx 系列水

表专用单片机、MSP430FG4xx 系列医疗仪器专用单片机、MSP430FE4xx 系列电能计量专用单片机等。这些专用单片机都是在同型号的通用单片机上增加专用模块而构成的。例如 FW4xx 系列在 F4xx 系列上增加了 SCAN-IF 无磁流量检测模块；FG4xx 系列在 F4xx 系列上增加了可编程差动放大器；FE4xx 系列在 F4xx 系列上增加了 E-Meter 电能计量模块。针对市场不断涌现的新产品，TI 公司今后仍会不断推出专用单片机。有趣的是，某些专用单片机反而比同系列的通用单片机价位还低。例如近年来，国内数字式电表大量采用 MSP430FE42x 系列电能计量专用单片机，市场供求规律导致 FE42x 的实际价格比 F42x 还要低。这也是本书附带的学习板选用了 MSP430FE425 而非通用型的 MSP430F425 的原因。

读者今后在设计中若用到 MSP430 单片机，第一步应该进行选型工作。选择功能模块最接近项目需求的系列，然后根据程序复杂程度估算存储器和 RAM 空间，决定最终选用的型号。附录里摘录 07 年的选型表供读者参考。型号列表每年都会更新，最新的型号列表可以通过 TI 公司网站下载。

2.2 I/O 口

IO 口是处理器系统对外沟通的最基本部件，从基本的键盘、LED 到复杂的外设芯片等，都是通过 IO 口的输入、输出操作来进行读取或控制的。

MSP430 系列中，不同单片机的 IO 口数量不同。体积最小的 MSP430F20xx 系列只有 10 个 IO，适合在超小型设备中应用；功能最丰富的 MSP430FG46xx 系列多达 80 个 IO 口，足够应付外部设备繁多的复杂应用。在 MSP430FE425 单片机中，共有 14 个 IO 口，属于 IO 口较少的系列。但由于需要大量引脚的设备，如 LCD、多通道模拟量输入等都有专用引脚，不占用 IO 口。因此在大部分设计中 IO 数量还是够用的。

1 IO 口寄存器

和大部分单片机类似，MSP430 单片机也将 8 个 IO 口编为一组。例如 P1.0~P1.7 都属于 P1 口。每组 IO 口都有 4 个控制寄存器，其中 P1 和 P2 口还额外具有 3 个中断寄存器。

表 2.1.1 IO 口寄存器列表。

寄存器名	寄存器功能	读写类型	复位初始值
PxIN	Px 口输入寄存器	只读	无
PxOUT	Px 口输出寄存器	可读可写	保持不变
PxDIR	Px 口方向寄存器	可读可写	0（全部输入）
PxSEL	Px 口第二功能选择	可读可写	0（全部为 IO 口）
PxIE	Px 口中断允许	可读可写	0（全部不允许中断）
PxIES	Px 口中断沿选择	可读可写	保持不变
PxIFG	Px 口中断标志位	可读可写	0（全部未发生中断）

这是本书第一次出现寄存器列表，有必要说明一下 MSP430 单片机的寄存器以及标志位全部是大写的。若出现的小写的“x”，表示该设备不止一个，因此寄存器也不止一个。为了缩短列表长度，不用全部列出，用字母 x 表示序号。例如对于表中的 PxOUT，当 x

取 1、2、3 时，就变成了 P1OUT、P2OUT、P3OUT。

- *PxDIR 寄存器用于设置每一位 IO 口方向： 0=输入 1=输出*

MSP430 单片机的 IO 口属于双向 IO 口，因此在使用 IO 口时首先要用方向选择寄存器来设置每个 IO 口的方向。例如 P1.5、P1.6、P1.7 接有按键，P1.1、P1.3、P1.4 接有 LED，那么 P1.5、P1.6、P1.7 要设为输入，P1.1、P1.3、P1.4 要设为输出：

```
P1DIR|=BIT1+BIT3+BIT4;                  // P1.1、P1.3、P1.4 设为输出
P1DIR &=~ (BIT5+BIT6+BIT7);             // P1.5、P1.6、P1.7 设为输入(可省略)
```

由于 PxDIR 寄存器在复位过程中会被清 0，没有被设置的 IO 口方向均为输入状态，因此第二句可以被省略。

对于所有已经设成输出状态的 IO 口，可以通过 PxOUT 寄存器设置其输出电平；对于所有已经被设成输入状态的 IO 口，可以通过 PxIN 寄存器读回其输入电平。例如读回 P1.5 口上的开关状态，并判断若处于按下状态（低电平）则从 P1.1 口输出高电平点亮 LED：

```
if((P1IN & BIT5)==0)  P1OUT|=BIT1;   //若 P1.5 为低电平则 P1.1 输出高电平
```

- *PxSEL 寄存器用于设置每一位 IO 的功能： 0=普通 IO 口 1=第二功能*

在 MSP430 单片机中，很多内部功能模块也需要和外界进行数据交流，为了不增加芯片引脚数量，大部分都和 IO 口复用管脚。因此大多数 IO 引脚都具有第二功能。通过寄存器 PxSEL 可以指定某些 IO 引脚作为第二功能使用。例如从附录中管脚排布图中查到 MSP430x42x 系列单片机的 P2.4、P2.5 口和串口的 TXD、RXD 公用引脚。若需要将这两个引脚配置为串口收发脚，则须将 P2SEL 的 4、5 位置高：

```
P2SEL |= BIT4 + BIT5;                  // P2.4,5 设为串口收发引脚
```

l IO 口中断

在 MSP430 所有的单片机中，P1 口、P2 口总共 16 个 IO 口均能作引发中断。在 MSP430x42x 系列中，14 个 IO 均属于 P1 或 P2 口，因此每个 IO 都能作为中断源使用。通过下列 2 个寄存器配置 IO 口作为中断使用：

- *PxIE 寄存器用于设置每一位 IO 的中断允许： 0=不允许 1=允许*
- *PxIES 寄存器用于选择每一位 IO 的中断触发沿： 0=上升沿 1=下降沿*

在使用 IO 口中断之前，需要先将 IO 口设为输入状态，并允许该位 IO 的中断，再通过 PxIES 寄存器选择触发方式为上升沿触发或者下降沿触发。例如将 P1.5、P1.6、P1.7 口设为中断源，下降沿触发：

```
P1DIR &=~(BIT5 + BIT6 + BIT7);          // P1.5、P1.6、P1.7 设为输入(可省略)
P1IES |=  BIT5 + BIT6 + BIT7;          // P1.5、P1.6、P1.7 设为下降沿中断
P1IE  |=  BIT5 + BIT6 + BIT7;          // 允许 P1.5、P1.6、P1.7 中断
EINT();                                // 总中断允许
```

- *PxIFG 寄存器是 IO 中断标志寄存器：0=中断条件不成立 1=中断条件曾经成立过*

无论中断是否被允许，也不论是否正在执行中断服务程序，只要对应 IO 满足了中断条件（例如一个下降沿的到来），PxIFG 中的相应位都会立即置 1 并保持，只能通过软件

人工清除。这种机制的目的在于最大可能的保证不会漏掉每一次中断。在 MSP430 系列单片机中，P1 口的 8 个中断和 P2 口 8 个中断各公用了一个中断入口，因此该寄存器另一重要作用在于中断服务程序中用于判断哪一位 IO 产生的中断。下面的中断服务程序示范 P1.5、P1.6、P1.7 发生中断后执行不同的代码：

```
#pragma vector = PORT1_VECTOR       //P1 口中断源
__interrupt void P1_ISR(void)        //声明一个中断服务程序，名为 P1_ISR()
{
  if(P1IFG & BIT5)                  //判断 P1 中断标志第 5 位(P1.5)
    {
      ... ...                       //在这里写 P1.5 中断处理程序
    }
  if(P1IFG & BIT6)                  //判断 P1 中断标志第 6 位(P1.6)
    {
      ... ...                       //在这里写 P1.6 中断处理程序
    }
  if(P1IFG & BIT7)                  //判断 P1 中断标志第 7 位(P1.7)
    {
      ... ...                       //在这里写 P1.7 中断处理程序
    }
   P1IFG=0;                         //清除 P1 所有中断标志位
}
```

注意在退出中断前一定要人工清除中断标志，否则该中断会不停发生。类似的原理，即使 IO 口没有出现中断条件，人工向写 PxIFG 寄存器相应位写“1”，也会引发中断。更改中断沿选择寄存器也相当于跳变，也会引发中断。所以更改 PxIES 寄存器应该在关闭中断后进行，并在打开中断之前及时清除中断标志。

MSP430 单片机大量的 IO 中断非常适合做键盘输入用，但要注意键盘存在机械结构，在闭合或松开的过程中，机械结构的碰撞和反弹会造成信号上数毫秒的“毛刺”。

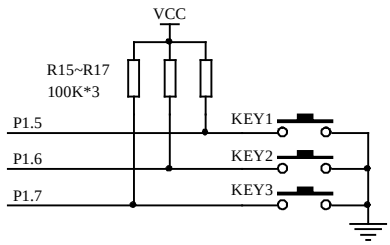


图 2.2.1 用 I0 口作键盘输入

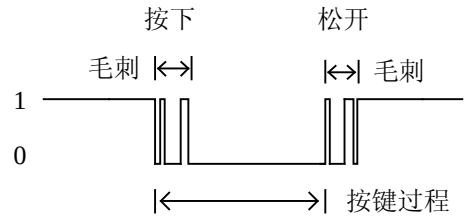


图 2.2.2 按键过程产生的毛刺

在键盘应用中为了防止毛刺造成多次按键的假象，可以在中断中延迟 10~20ms，避开毛刺后再做处理。以图 2.21 中 3 个按键调整 Speed 变量值为例：

```
#pragma vector = PORT1_VECTOR
__interrupt void PORT1_ISR(void)      //P1 口中断服务程序
{ unsigned int i;
  unsigned char PushKey;
  PushKey =P1IFG & (BIT5+BIT6+BIT7); //读取 P1IFG 的 5、6、7 位（哪个键被按下）
  for(i=0;i<1000;i++){};             //略延迟后再做判断
  if((P1IN & PushKey) == PushKey)    //如果按键变高了（松开），则判为毛刺
```

```
    { //上句逻辑上需注意：按键低电平表示按下，而 P1IFG 高电平表示中断发生（键按下）
      P1IFG=0; return;                    //认为按键无效，不作处理直接退出
    }
   if(PushKey & BIT5)                     //若 P1.5 所在按键被按下
     {
       Speed++;                           //执行 P1.5 按键的功能
     }
   if(PushKey & BIT6)                     //若 P1.6 所在按键被按下
     {
       Speed--;                           //执行 P1.6 按键的功能
     }
   if(PushKey & BIT7)                     //若 P1.7 所在按键被按下
     {
       Speed=0;                           //执行 P1.7 按键的功能
     }
   P1IFG=0;                               //退出中断前清除 IO 口中断标志
   return;                                //退出中断程序
}
```

1 IO 口基本应用

IO 口最重要的功能就是输入、输出开关量，以控制其他设备，或者读取其他设备的状态。上节中图 2.21 就是一个 IO 直接读入开关量的例子。对于输出，最简单的用法就是直接用 IO 口驱动如 LED 等设备，但注意在输出状态下，每个 IO 口最大输出或吸收电流为 6mA，所有 IO 电流总和不超过 48mA。若某设备耗电大于 6mA，或者所需工作电压高于 430 的供电电压，可用三极管扩流。

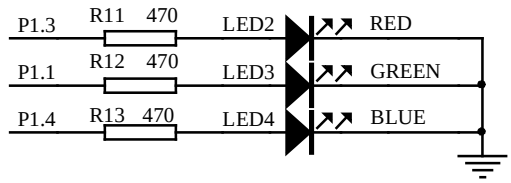


图 2.2.4 用 IO 口直接驱动 LED

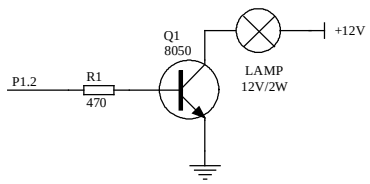


图 2.2.5 用 IO 口驱动 12V 负载

对于工作电压更高或功率更大的被控对象，如 220V 负载的灯泡、加热器、电动机等，可以使用继电器或可控硅进行控制。图 2.2.6 是典型的继电器驱动电路，二极管 D1 防止断开继电器线圈（电感）时产生的反向电动势击穿三极管。

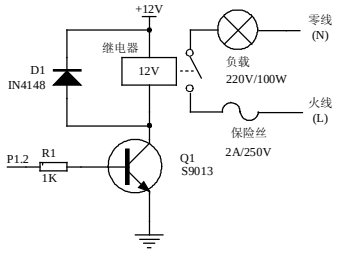


图 2.2.6 用继电器控制功率负载

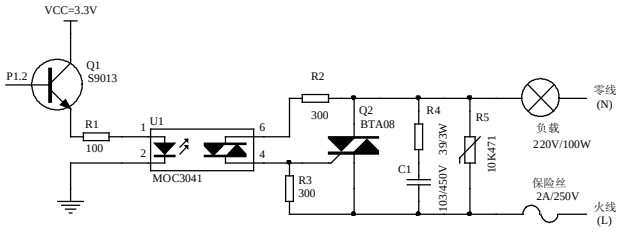


图 2.2.7 用可控硅控制功率负载

因为继电器存在机械结构，其动作寿命有限。对于需频繁启停的场合，应使用固态继电器。图 2.2.7 是一种使用最为广泛的固态继电器电路。IO 口通过光触发器 U1（MOC3041）触发双向可控硅 Q2（BTA08）导通，从而控制后端交流功率负载，并将高压部分与弱信号部分进行了光电隔离（这部分电路也有成品可以选用）。由于 MOC3041 要求 15mA 以上的驱动电流，超出 IO 口最大承受能力（6mA），因此需要用三极管 Q1（9013）构成射随器扩流后才能驱动。R2 用于限制触发电流、R3 用于防止可控硅误导通、R4 和 C1 构成缓冲吸收电路，防止负载断开产生的冲击损坏可控硅，也防止过高的 dv/dt 误触发可控硅。R5 是压敏电阻，一种限压保护器件，对浪涌、雷击等瞬间能量起到限制和吸收作用。10K471 表示直径 10mm，转折电压 470V。固态继电器因为使用半导体器件，抗过载能力较差，所以无论何种功率控制电路，在火线上串联保险丝都是必不可少的。

因为 MSP430 单片机的工作温度宽（-40~85 度），也适用于工业产品。在工业应用中，各种远方送来的开关量都可能会带有强干扰或对地有电压。比如某控制设备的 IO 输入端需要拉长达上百米的导线才能到开关安装处，相当于一根巨大的天线接收各种干扰信号，特别是和动力电缆埋设在同一电缆沟内，干扰更为严重。因此这类 IO 信号不允许直接和单片机控制电路连接。这种情况下使用光藕进行光电隔离是必须的：

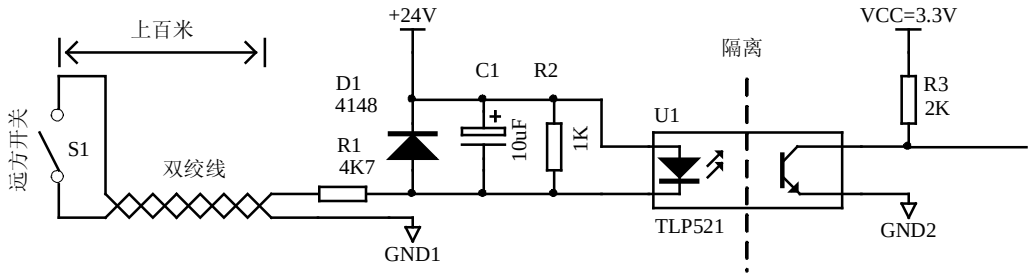


图 2.2.8 用光耦对远方输入开关量进行隔离

上图示例了一种从远方获取慢速开关量的电路。该电路两个电源系统之间没有电气连接从而阻隔干扰进入单片机系统。当远方开关合上时，光耦 U1（TLP521）的发光管导通引起接收管导通，P1.0 读回低电平。当远方开关断开时，光耦截止，P1.0 读回高电平。D1 提供了一个泄放通道，防止远方传入比+24V 还高的电压时反向击穿光耦；R2 为导通电流设置了门限，若遇到远方开关 S1 或线路受潮导致轻微漏电，只要漏电电流在 R2 上压降小于光耦中发光管导通电压（1.1V）就不会触发光耦。C1 从硬件上消除开关的颤动造成的误动作。该电路只能用于速度较慢（百毫秒级）的场合，如按钮、报警装置等。高速长距离传输开关信或数字量号需要差分收发电路，并采用高速光耦（如 6N136/137 等）。

I 线与逻辑

MSP430 单片机的 IO 口属于 CMOS 型：当 IO 处于输入状态时，呈高阻态；当 IO 处于输出状态时，高低电平都具有较强输出能力。若输出高电平的 IO 口和输出低电平的 IO 口直接连接，很可能造成损坏，因此不能像 8051 的 IO 那样实现“线与”功能。但可以通过 IO 方向切换的方法模拟出来。以 P1.0 口为例，硬件上加一个上拉电阻至 VCC，软件中先将 P1OUT 的 BIT0 置 0，再通过软件切换方向来改变输出：

```
#define IO_H   P1DIR &=~ BIT0 /*IO 输出高，实际将方向切成输入，利用上拉电阻输出弱 1*/
```

```
#define IO_L   P1DIR  |= BIT0 /*IO 输出低，实际将方向切成输出，利用 IO 口输出强 0*/
#define IO_R   (PIIN & BIT0)  /*IO 读，将 PIN 寄存器中 P1.0 值读回*/
```

用这种方法模拟出来的 IO 操作中，高电平不是由 IO 直接输出的，而是通过上拉电阻拉高的，因此高电平输出电流很弱；而低电平由 IO 直接输出，驱动能力较强。当高电平遇到低电平的时候，会被拉低。当若干 IO 连在一起的时候，只要有 1 根输出 0，整体就为 0，总输出相当于各 IO 相与。“线与”逻辑在 I2C 总线、多机通讯中都有重要用途。

电平冲突

电平冲突的问题经常发生在数据输入和双向数据交换的应用中，要特别注意。例如下面一些代码都很可能是单片机的毁灭者：

```
//                                                    !! 请勿运行以下代码!!
//----------------------例 1(按键)-----------------------------------
P1DIR |= BIT5+BIT6+BIT7;    // P1.5、P1.6、P1.7 对地接有按键，IO 方向设错
P1OUT |= BIT5+BIT6+BIT7;    // IO 输出高，按下后对地短路，很可能烧坏 IO 口
//-----------------例 2(双向总线)-----------------------------------
LCM_CS_L;           //将某点阵液晶的片选线 CS 拉低（选中）
P1DIR=0xFF;         //P1 作为数据口，设为输出
P1_OUT=WrData;      //P1 输出 1 字节数据
LCM_WR_L;           //将该点阵液晶的写线 WR 拉低（写数据）
LCM_WR_H;           //将该点阵液晶的写线 WR 置高（写数据完成）
...
LCM_RD_L;           //将该点阵液晶的写线 RD 拉低（读数据）/*液晶立即返回数据*/
P1DIR=0x00;         //P1 作为数据口，设为输入.  /*这里设断点，单片机立即损坏!*/
RdData=P1_IN;       //将数据读回存于 RdData 变量
LCM_RD_H;           //将该点阵液晶的写线 RD 拉低（读数据完成）
LCM_CS_H;           //将某点阵液晶的片选线 CS 置高（释放）
```

例 1 示范了 IO 方向设置错误造成的后果：若果某 IO 本应该是输入，而且输入源较强，则很可能因方向设置错误而永久损坏。虽然 MSP430 单片机的 IO 口都具有一定的短路保护能力，但若多个 IO 同时出现超载，或长时间超载仍会导致损坏，或性能不稳定。

例 2 是 P1 口作为双向数据总线的应用。摘录了某点阵液晶模块底层时许程序中的一部分，该程一直运行正常，但某次调试中单步运行过一遍之后，发现单片机被损坏。仔细检查后发现问题：程序中先向液晶写入数据，P1 已经被设为输出模式；而接下来程序读液晶数据时，执行 LCM_RD_L 后，液晶已经输出数据，P1 口置为输入状态晚于液晶输出数据一句，造成 1 微秒时间差。在这 1us 内两个设备的 IO 全部是输出状态，若 0 遇到 1 则相当于短路。全速运行时由于时间极短不足以损坏 IO 口，在单步过程中一旦程序停在这一句，大电流长时间流过这两个设备的 IO 口，就造成了损坏。解决的办法是将这两句顺序调换。实际中有很多双向数据传输的接口应用，一定要注意设备之间数据方向切换配合的问题。在样机设计阶段最保险的方法是在双向 IO 上串联 300 欧左右电阻。

兼容性

为了电池供电应用，MSP430 单片机工作电压较低（1.8~3.6V）。大部分应用中取 3V 左右，因此 IO 逻辑电平属于 3V 逻辑。且 MSP430 单片机任何一个引脚输入电压不能超过

VCC+0.3V，不能低于-0.3V，否则将启动内部泄放电路。泄放电路最大只能吸收 2mA 电流，超过可能会损坏 IO 口。因此 MSP430 单片机的 IO 口和 5V 逻辑器件连接时，必须考虑电平转换问题。可分为几种情况考虑：

1. 5V 逻辑器件输出至 MSP430 单片机的输入口。

这是最简单的一种情况，将 5V 逻辑通过 10K 和 20K 电阻分压后即转换成 3V 逻辑（图 a）。若 5V 逻辑属于弱上拉型（例如和 8051 通讯），可直接连接。利用 430 单片机内部泄放电路将电压箝至 3V。

2. MSP430 单片机输出至 5V 逻辑器件。

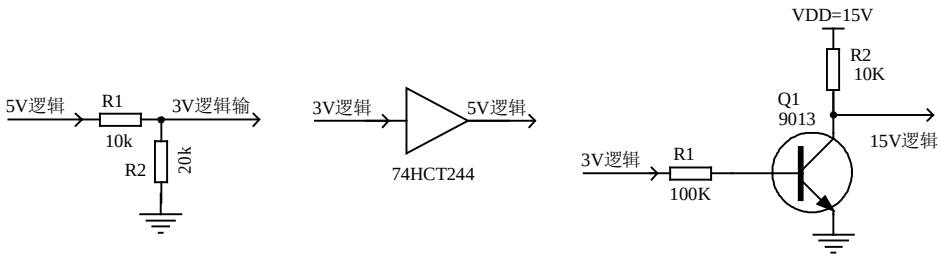
这种情况首先要看接收器件的高电平门限，一般手册都会给出（VIH 值）。某些器件具有 2.5V 以下门限（如大部分液晶控制器）则可直接连接无需额外电路。若接收方门限较高，可在两者之间加一片 74HCT244 缓冲器。74HCT 器件具有 2.2V 固定逻辑门限，在 5V 电源时能够识别 3V 逻辑输入（图 b）。

3. 双向数据传输。

双向数据传输中，不仅要转换电平，还需要切换方向。最好选用电平转换专用芯片（如 74LVCH245）实现。如果对方高电平门限在 2.5V 以下，图 a 的方案也能实现，因为 5V->3V 被分压衰减，3V->5V 没有被衰减。

4. 驱动 5V 以上的逻辑。

例如 MSP430 的 IO 口输出逻辑控制 15V 电源电压供电的模拟开关，可以用三极管反向实现，实现电平转换但逻辑相反。



a.5V 逻辑转为 3V 逻辑　　b.3V 逻辑转为 5V 逻辑　　c. 3V 逻辑转为 15V 逻辑

图 2.2.9 逻辑电平转换电路

无论选用何种转换方案，在 MSP430 系统中出现 3V 和 5V 混合逻辑都是不值得推荐的。不仅破坏了 MSP430 系统简洁的设计原则，还额外增加了功耗、增加了电源管理的难度。在 MSP430 系统设计时，应该尽量全部选用 3V 逻辑的芯片。

和所有的 CMOS 电路一样，MSP430 单片机的 IO 口在输入状态时也呈高阻态。若悬空则等效于天线，可能随附近电场而随机地感应出中间电平（0V 和 VCC 之间的电压）。MSP430 单片机 IO 口内部带有施密特触发器和总线保持器，悬空或输入中间电平不会造成错误或损坏。但会因为输入级 CMOS 门的截止不良额外增加数微安的耗电。所以在超低功耗应用中，每个输入 IO 口都应该有确定电平（0V 或 VCC），对于未用的 IO，可以接地或设为输出状态，以保证电平确定。

MSP430 单片机的 IO 口属于静电敏感电路，尽量不要用手触摸。业余条件下，接触芯

片前可以先触摸接地金属（暖气管、水管等），将静电电荷释放。

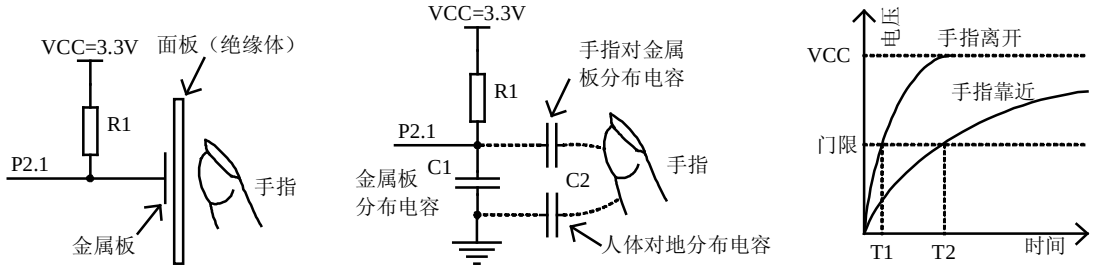
Ⅰ 电容式感应触控

近年来电容式感应触控键被广泛地用于替代传统机械式按键。比如在电磁炉、微波炉等生活家电上隔着玻璃可以操作的按钮，以及近年来 iPod 等音乐播放器上的触摸操作板都是电容式感应触控键的应用。比起传统机械式按键，不仅成本降低，还具有优良的防水、防尘性能，更重要的是这种操作方式不存在机械结构，寿命几乎无限。

在 MSP430 单片机中，每个 IO 口都可以作为电容式感应触控输入使用。电容式感应触控键的结构如下图 a 所示，在一个 IO 口末端接一块金属板（一般是直接做在 PCB 上的一个铜皮块），并在 IO 口接一个上拉电阻。金属板藏在绝缘的玻璃或塑料面板后面。

图 b 是这种结构的等效电路。金属板和系统地之间存在着非常小的分布电容(数皮法)。由于人体相当于一个大面积的导体，当手指靠近这块金属板时，人体和金属板之间构成分布电容，人体与系统地之间也构成分布电容。总的效果是增加了金属板对地之间的分布电容量。人体所带来的电容增往往是固有分布电容的数倍（10~50pF）只要探测到电容量的增加，即可认为“键”被按下。

粗略测电容量最简单的方法是测量电容的充电时间：首先将 IO 口置低，将电容上的电荷完全泄放。然后再将 IO 置为输入方式，此时电容上依然保持 0V，读回的是低电平。此后 VCC 将通过 R1 给分布电容不断充电，IO 低电平时间的长短就能直观反映电容量的大小。如图 c 所示，当探测到充电过程时间变长，则判定手指靠近按键。



a.感应触控键结构　　b.等效电路　　c.识别原理

图 2.2.10 电容式感应触控按键的结构与原理

R1 的取值根据分布电容的大小和绝缘板厚度而定，一般取 1M~10M 欧之间。若相邻两个 IO 口轮流输出高电平，为另一键提供上拉电阻所需的 VCC，则两个触控键之间可以共用一个电阻，如图 2.2.11。

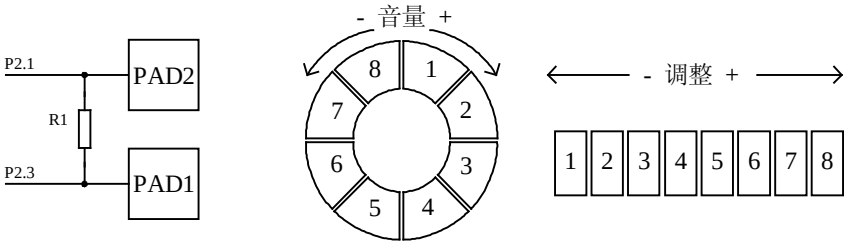


图 2.2.11 两个触控键公用电阻　　图 2.2.12 常见触控键排列方式

若将多个触控键可以排列成一定的形状，不仅可以判别手指位置，还可以计算出手指的运动方向、速度等信息，从而完成某些复杂的操作功能。图 2.2.12 示例了两种最常见的触控键排布方式。圆形排列方式在 iPod 等播放器中最常见，通过手指在圆形触控板上弧线滑动，可进行调整音量等操作。直线型排列可以用于翻页、拖拽进度条、调整数值大小等操作。

以图 2.2.11 电路为例，依照电容测量的过程，为触控键编写读取按键状态的代码：

```
#define PAD1_OUT_L  P2DIR|=BIT3; P2OUT&=~BIT3  /*将 PAD1 置为低电平的宏定义*/
#define PAD2_OUT_H  P2DIR|=BIT1; P2OUT|= BIT1  /*将 PAD2 置为高电平的宏定义*/
#define PAD1_DIR_IN P2DIR&=~BIT3               /*将 PAD1 设为输入状态的宏定义*/
#define PAD1_IN (P2IN&BIT3)                    /*读回 PAD1 电平的宏定义*/
#define LED_RED_ON  P1OUT|=BIT3                /*红灯亮的宏定义*/
#define LED_RED_OFF P1OUT&=~BIT3               /*红灯灭的宏定义*/
/*********************************************************************
* 名    称：TouchPad_GetPad1()
* 功    能：读回触控键 PAD1 的状态
* 入口参数：无
* 出口参数：返回 1 表示有手指按在 PAD1 上，返回 0 表示 PAD1 未被触摸
* 说    明：可靠性不高，请勿直接使用
*********************************************************************/
    char TouchPad_GetPad1(void)
    {   int Count;
        PAD2_OUT_H;                     //PAD2 置高(对充电电阻提供 VCC)
        PAD1_OUT_L;                     //PAD1 置低
        _NOP();                         //略等待，将电荷泄放完
        Count=0;                        //时间计数值清零
        PAD1_DIR_IN;                    //PAD1 置为输入状态
        while(PAD1_IN==0){Count++;}     //在 PAD1 为低的过程中计数值增加
        _NOP(); //调试时可以在这句设断点，查看计数值 Count。
        if(Count>=30) return(1);        //如果低电平时间大于门限时间，则返回 1
        else          return(0);        //否则返回 0
    }
```

读者可以运行光盘上的代码实例《电容感应式触控键原理》来检验效果，该程序在主循环中调用上面的函数，并用返回值控制 LED 的亮灭：

```
        if(TouchPad_GetPad1()==1)  LED_RED_ON;   //如果 PAD1 按下，红灯亮
        else                       LED_RED_OFF;
```

实验会发现虽然可以识别出手指触摸，但是手指在靠近触摸板的过程中 LED 是闪烁的，说明存在一个识别不可靠的区域。这是因为人体在等效于于一个大面积导体的同时，还起到了天线的效果。人体将各种干扰（主要是 50Hz 工频干扰）也通过等效电容加在了 IO 口上，造成一定概率的判断失误。

解决误判问题的办法非常简单，把判别依据扩大到相邻 N 次检测中，并且相邻的测量之间需要有一定的时间间隔。判断逻辑为：若相邻 N 次充电时间全部都大于门限，则认为键被按下；若相邻 N 次全部小于门限，才认为键被松开；若 N 次结果不一致则保持上次判定结果。实验发现 N>=4 即能可靠地识别按键。

照此思路，在程序中增加一个 4 字节 FIFO 队列，用于保存相邻 4 次的测量结果。将

上述函数改名 TouchPad_ScanPad1()，放在定时中断里执行，判别结果保存在全局变量内。而原 TouchPad_ GetPad1()函数只负责读取判别结果，这样就保持了和上例程序兼容：

```
#define PAD1_OUT_L  P2DIR|=BIT3; P2OUT&=~BIT3  /*将 PAD1 置为低电平的宏定义*/
#define PAD2_OUT_H  P2DIR|=BIT1; P2OUT|= BIT1  /*将 PAD2 置为高电平的宏定义*/
#define PAD1_DIR_IN P2DIR&=~BIT3               /*将 PAD1 设为输入状态的宏定义*/
#define PAD1_IN (P2IN&BIT3)                    /*读回 PAD1 电平的宏定义*/
#define PAD1_THRESHOLD     30                 /*PAD1 时间门限值，可能需调整*/
unsigned char PAD1_BUFF[4];               //记录 PAD1 相邻 4 次充电时间的队列
unsigned char PAD1_REG=0;                 //记录 PAD1 状态
/*********************************************************************
* 名    称：TouchPad_ScanPad1()
* 功    能：定时扫描触控键 PAD1 的状态
* 入口参数：无
* 出口参数：无，结果保存在全局变量 PAD1_REG 内
* 说    明：在 1/32 秒~1/256 秒定时中断内调用该函数
*********************************************************************/
void TouchPad_ScanPad1 (void)
{
   int i,Count;
   _BIC_SR(SCG0); //清除 SR 寄存器的 SCG0 控制位，恢复 DC 发生器，得到准确的 MCLK
   PAD2_OUT_H;                    //PAD2 置高(对充电电阻提供 VCC)
   PAD1_OUT_L;                    //PAD1 置低
   _NOP();                        //略等待，将电荷泄放完
   Count=0;                       //充电时间计数值清零
   PAD1_DIR_IN;                   //PAD1 置为输入状态
   while(PAD1_IN==0){Count++;}    //在 PAD1 为低的过程中计数值增加
   PAD1_BUFF[0]=PAD1_BUFF[1];
   PAD1_BUFF[1]=PAD1_BUFF[2];     //模拟 FIFO,前 3 次计数值依次移位
   PAD1_BUFF[2]=PAD1_BUFF[3];
   PAD1_BUFF[3]=Count;            //本次低电平时间计数值进入 FIFO
   for(i=0;i<4;i++)
     {
       if(PAD1_BUFF[i]<PAD1_THRESHOLD)  break; //任意一次小于门限则跳出循环
     }                                         //若提前跳出循环，i 将小于 4
   if(i==4) PAD1_REG=1;       // i 等于 4 说明 4 次全部大于门限，认为键按下
   for(i=0;i<4;i++)
     {
       if(PAD1_BUFF[i]>=PAD1_THRESHOLD) break;//任意一次大于门限则跳出循环
     }                                        //若提前跳出循环，i 将小于 4
   if(i==4) PAD1_REG=0;       // i 等于 4 说明 4 次全部小于门限，认为键松开
}

/*********************************************************************
* 名    称：TouchPad_GetPad1()
* 功    能：读回 PAD1 触摸板的状态
* 入口参数：无
* 出口参数：返回 1 表示有手指按在 PAD1 上，返回 0 表示 PAD1 未被触摸
```

```
* 说    明: 可靠性较高
*********************************************************************/
char TouchPad_GetPad1(void)
   {
     return(PAD1_REG);
   }
```

在中断内，由于产生 CPU 时钟的倍频器稳定需要数毫秒时间，而读取充电时间需要较稳定的时钟（指令运行速度），因此略延迟后再调用扫描函数：

```
#pragma vector = BASICTIMER_VECTOR
__interrupt void BT_ISR(void)         // 1/32 秒一次中断（由 BasicTimer 所产生）
{
  int i;
  for(i=0;i<800;i++);                  // 略延迟数毫秒，等待时钟稳定
  TouchPad_ScanPad1();                 // 定时扫描触控键 PAD1 的状态
    __low_power_mode_off_on_exit();    // 唤醒 CPU（与触摸键无关）
}
```

仿照上例，读者可以尝试自行编写触控键 PAD2 的读取函数。完整的代码可参考光盘附带的《低功耗触控键》程序范例。

2．3 时钟系统与低功耗模式

读者若学习过其他种类的单片机，一定都了解“系统时钟”的概念，指的是供 CPU 以及内部设备的时钟节拍。而在 MSP430 单片机中，引入了“时钟系统”的概念。两词顺序颠倒含义却相差甚远：在 MSP430 单片机中，通过时钟系统不仅可以切换时钟源、通过软件随时更改 CPU 运行速度、为不同的外设产生不同频率的时钟、还可以在必要时关闭或降低某些设备的时钟以降低功耗。

时钟系统是 MSP430 单片机中最为关键的部件。通过时钟系统可以在功耗和性能之间寻求最佳的平衡点，为单芯片系统设与超低功耗系统提供了灵活的实现手段。

Ⅰ 时钟系统结构与原理

在 MSP430 单片机中，通过时钟系统的配置最终产生三种时钟：

MCLK:主时钟（Master Clock）。MCLK 是专为 CPU 运行提供的时钟。MCLK 配置得越高，CPU 执行速度就越快。因此 MCLK 一般都设在 1MHz 以上以发挥 CPU 性能；一旦关闭 MCLK，CPU 也随之停止工作。CPU 是系统中耗电较大的部件之一，但大部分应用中都只有少数时间需要 CPU 运算。因此在超低功耗系统中都通过间歇开启 MCLK（唤醒 CPU）的方式来降低功耗。

SMCLK:子系统时钟（Subsystem Master Clock），也称辅助时钟。单片机内部某些设备需要高速时钟（如定时器、ADC 等），SMCLK 为这些需要高速工作的设备提供时钟源。并且 SMCLK 是独立于 MCLK 的：当关闭 MCLK 让 CPU 停止工作时，SMCLK 可以被设为仍然打开，从而让外设继续工作。

ACLK: 活动时钟（Active Clock）。ACLK 一般是由 32.768K 晶体直接产生的低频时钟，在单片机运行过程中一般不关闭，用于产生节拍时基，或和定时器配合间歇唤醒 CPU。时钟系统中对三种时钟不同程度的关闭，实际上就是不同的休眠模式。关闭的时钟越多，休眠越深。当全部的时钟，包括 ACLK 也被关闭的时候，功耗降到最低（0.1uA）。

在 MSP430 单片机大部分的内部设备中，都能选择时钟源并对上述时钟再分频，因此应用极其灵活。

MSP43F42x 和 41x 系列单片机的时钟系统模块结构框图如下：

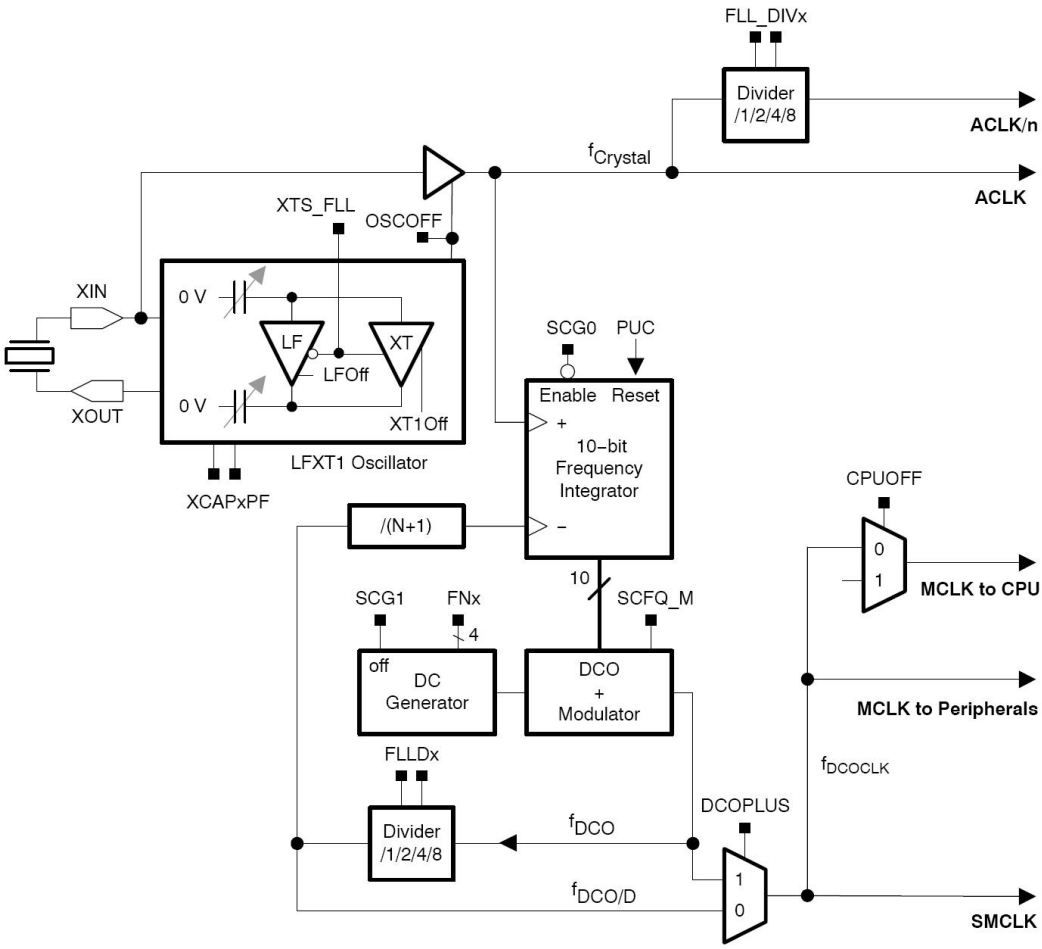


图 2.3.1 MSP430x42x/41x 单片机时钟系统

对于第一次见到结构框图的初学者，可能会感到无从下手。实际上，这是一种简化了的原理图，结构非常的清晰。为了便于读者理解模块结构框图，先简单介绍一下图中的各种标记的含义。

结构框图中的每个框表示了一个部件，每个方形黑点表示了一个控制位。若黑点引出线直接和某部件相连，说明该控制位“1”有效。若黑点直线末端带圆圈与某部件连接，说明该位“0”有效。

对于紧靠在一起的多个同名控制位，表示这些控制位的组合。用后面的字母 x 表示下标，高位在先。例如 FLL_DIVx 下面有 2 个黑点，说明有 2 个控制位 FLL_DIV1 和 FLL_DIV0，框内写着功能 Divider（分频）1/2/4/8 说明当 FLL_DIV1 和 FLL_DIV0 分别为 00、01、10、11 时，分频系数为 1、2、4、8。当出现多于三个控制位的组合时，用总线表示。例如 FNx 虽然只有 1 个黑点，但是下面的连接线上面写着“\4”，说明这是 4 位总线，FNx 代表 4 个控制位（FN3、FN2、FN1、FN0），共有 16 种组合。

梯形框表示多路选择器（MUX），它负责从多个输入通道中选择一个作为输出。通道选择关系由侧面的控制位来决定。

从结构框图中可以看出：只要通过软件配置各控制位，就可以改变硬件部件的连接关系、更改某些设置、开启或关闭某些部件、控制某些信号的路径和通断等等。这在其他功能模块中也会大量出现，甚至在某些模块中能通过软件直接设置模拟电路的参数。这些灵活的硬件配置功能，使得 MSP430 单片机具有极强的适应能力。大部分应中常用的硬件连接关系都可以通过软件配置出来，为“单芯片系统” 的实现了提供了方便。

所以，读结构框图和对结构图进行配置是进行 MSP430 单片机系统设计的基本功。读图过程首先可以将结构图大致划分为几个功能相对独立的部分，然后对照寄存器表和控制位功能表逐个理解，再整理各个部分之间的联系与制约关系。以图 2.3.1 为例，整个时钟系统可以分作两个部分：时钟振荡器和 FLL 倍频器。

在 MSP430F42x 单片机时钟系统中，ACLK 是由晶振及其振荡电路提供的。MCLK 和 SMCLK 时钟由 ACLK 倍频得来。从时钟系统框图中划分出晶体振荡器部分，如下：

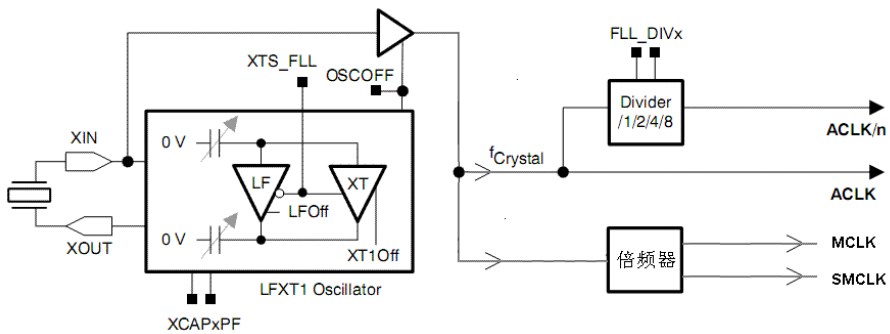


图 2.3.2 MSP430x42x/41x 单片机时钟振荡器

振荡器部分总共有 4 个控制位：XTS_FLL、OSCCAPx、FLL_DIVx、OSCOFF。以下用下划线表示复位后的默认设置。

- *XTS_FLL：* *选择晶振类型。* *0=低频晶体* *1=高频晶体* *(位于 FLLCTL0 寄存器)*

一般来说，MSP430 单片机推荐使用 32.768KHz 低频手表晶振，以获得低频 ACLK。但是某些特殊应用中可能会用到高频晶振产生高频的 ACLK。由于高频晶振和低频晶振特性不一致，振荡电路也有所不同，可以用该标志位选择振荡器种类。对于 450KHz 以下的晶振，该位置 0；对于 450KHz 以上晶振，该位置 1。

- *OSCCAPx：* *设置晶体匹配电容（每只引脚）。(位于 FLLCTL0 寄存器)*
  00≈ 1pF *01≈6pF* *10≈8pF* *11≈10pF* *（每只引脚）*

快捷宏定义： XCAP0PF XCAP10PF XCAP14PF XCAP18PF （总和）
或：OSCCAP_0 OSCCAP_1 OSCCAP_2 OSCCAP_3

一般的晶体振荡器电路在晶振两端都要对地外接 2 只匹配电容，与晶振的标称负载电容相等的时候，才能得到稳定、准确的震荡频率。MSP430 单片机内部集成了这两只电容，而且还能通过软件设置这两电容的大小，以匹配不同的晶振。对于最常见的手表低频晶振（32K），标称负载电容 12.5pF（每只脚）左右，应选则 10pF 选项（快捷宏定义中的容量是按两只引脚总和计算的），加上电路板上的分布电容刚好 12pF 左右。在高频晶振应用中，一般要求晶振外接两只 20~30pF 的匹配电容。超出了单片机内部所能提供的最大电容量，因此仍要外接电容。

- *FLL_DIVx： 设置 ACLK 输出分频系数。(位于 FLLCTL1 寄存器)、*
  00=无分频 01=2 分频 10=4 分频 11=8 分频
  快捷宏定义： FLL_DIV_1 FLL_DIV_2 FLL_DIV_4 FLL_DIV_8

ACLK 除了提供系统活动用之外，还可以从 P1.5 脚输出，供其他外部设备使用。输出频率的分频系数可以通过该控制位设置

- *OSCOFF： 关闭低频时钟振荡器 (位于 SR 寄存器)、*
  0=正常工作（开启） 1=关闭

关闭时钟振荡器后，系统中 ACLK 也随之停止。一般在进入最深的休眠模式（LPM4）前才关闭 ACLK，之后系统功耗降到最低。但因所有时钟都关闭，处理器内部没有任何模块可以唤醒 CPU，只能通过外部的 IO 中断或复位唤醒。

例 2.3.1 ：MSP430F42x 单片机外部接有 32.768KHz 晶振，为其配置时钟：

```
FLL_CTL0 &=~ XTS_FLL;                  // 设置振荡器类型为低频  （可省略）
FLL_CTL0 |=  XCAP18PF;                 // 设置晶振匹配电容 18pF 左右
```

例 2.3.2 ：MSP430F42x 单片机外部接有 1MHz 晶振，并要从 P1.5 输出 250KHz 时钟给某外部逻辑电路使用：

```
FLL_CTL0 |=  XTS_FLL;                  // 设置振荡器类型为高频
FLL_CTL0 |=  XCAP0PF;                  // 设置内部晶振匹配电容 0pF（电容需外接）
FLL_CTL1 |=  FLL_DIV_4;                // 设置对外输出 4 分频
P1DIR |= BIT5;                          // P1.5 设为输出
P1SEL |= BIT5;                          // P1.5 设为第二功能脚（ACLK）
```

在 4xx 系列单片机中，引入了锁频环倍频环路（FLL），对 ACLK 进行倍频产生高频时钟。由于 ACLK 来源于晶振，准确度很高，倍频后依然能得到准确的频率。可以供给定时器、波特率发生器等需要高频精确时钟的设备使用。

数字倍频环是一种非常巧妙的电路。它最核心的部件是数控振荡器和一个频率积分器（实际上是一个加减计数器）。对于频率积分器，每个 ACLK 脉冲将计数值加 1；数控振荡器的输出频率（f_{DCO}）经过（N+1）分频后的每个脉冲将计数减 1。计数器的累计结果又输出给数控振荡器，改变振荡器频率 f_{DCO}，构成反馈环。

ACLK 和（N+1）分频后的 f_{DCO} 对计数器进行“拉锯战”：若（N+1）分频后的 f_{DCO} 比 ACLK 频率略低，则对于计数器来说加法比减法频率高，计数结果不断增大。计数结果又

控制数控振荡器使 f_{DCO} 增大，最终让（N+1）分频后的 f_{DCO} “追上”ACLK 的频率。反之，若（N+1）分频后的 f_{DCO} 比 ACLK 频率略高，则对于计数器来说加法比减法频率低，计数结果不断减小。计数结果又控制数控振荡器使 f_{DCO} 减小，最终让（N+1）分频后的 f_{DCO} “落回”ACLK 的频率。

整个环路构成积分式负反馈。从控制理论角度分析，积分环节相当于无限直流增益，因此整个控制环路属于无静差系统。负反馈的最终结果是 $f_{DCO}/(N+1)=ACLK$，即

$$f_{DCO}=(N+1)\times ACLK$$

从而实现了倍频。这种方法用纯数字实现，和一般 CPU 内的 PLL（锁相环）倍频电路相比，功耗低得多。这种倍频方法存在一个缺点：虽然在宏观上 $f_{DCO}=(N+1)\times ACLK$，但在微观上看，频率是在微小的范围内不断变化。调整间隔时间约 31us（$1/f_{ACLK}$），而且稳定需要一定时间，因此对于时钟瞬时稳定度要求很高的场合，仍应该使用高频晶振。

这种频率的轻微“抖动”对于通过电磁兼容测试是有利的。因为时钟的能量谱由一个尖峰扩散到附近一个区域内。总面积（能量）不变的情况下，峰矮了许多。这对通过 EMI（电磁发射）测试是非常有利的。

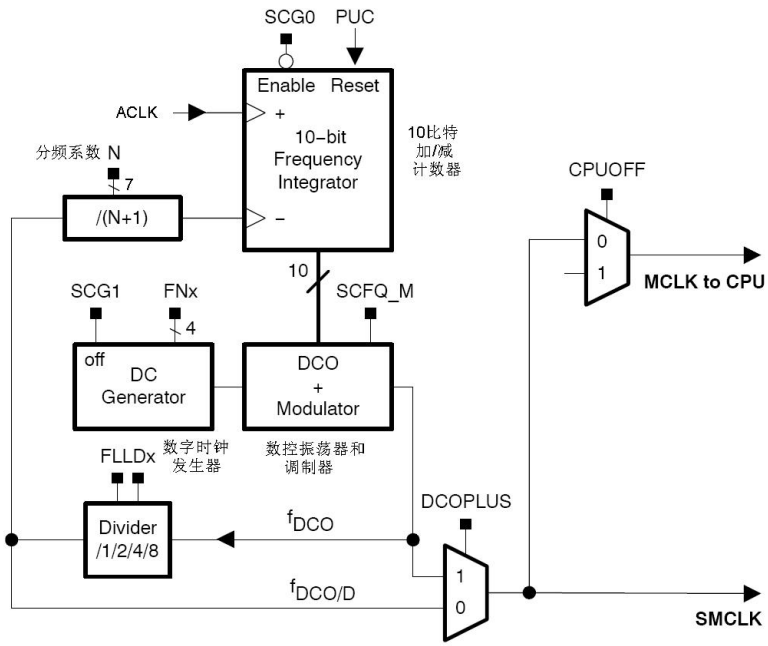


图 2.3.3 MSP430x42x/41x 单片机时钟倍频器结构

图 2.3.3 是从 2.3.1 中划分出来的倍频器部分电路。共有 8 个控制位：

- *N：环路分频系数。(位于 SCFQCTL 寄存器低 7 位)*
  倍频系数=N+1 （复位后 N=31，即 32 倍频）
  快捷宏定义：SCFQ_64K SCFQ_128K SCFQ_256K SCFQ_512K
  SCFQ_1M SCFQ_2M SCFQ_4M
- *DCOPLUS：DCO 额外分频允许 （位于 FLL_CTL0 寄存器）*

0=禁止额外的分频　1=允许额外分频

当 DCOPLUS 为 1 时，下面 FLLDx 的设置才有效。

- *FLLDx：　DCO 额外分频系数　（位于 SCFI0 寄存器）。*

  00=无分频　01=2 分频　10=4 分频　11=8 分频

  快捷宏定义：　FLL_DIV_1　FLL_DIV_2　FLL_DIV_4　FLL_DIV_8

对于整个倍频环路来说，上述 3 个控制位决定了反馈环路的总分频系数。因为输出频率除以分频系数等于 ACLK 频率，折算到输出就成了 ACLK 倍频系数。

通过系数 N 可以实现 2~128 倍频，再加上 FLLDx 的系数，最大能实现 1024 倍频。复位后的默认值 N=31、DCOPLUS=0，相当于对 ACLK 进行 32 倍频。对于 32.768KHz 晶振，MCLK 和 SMCLK 均为 32×32.768K=1.048MHz，在一般应用中可以无需更改。

- *FNx：　数字时钟发生器频率范围设置。　(位于 SCFI0 寄存器)*

  0000=0.65~6MHz　0001　1.3 - 12.1 MHz　001x　2 - 17.9 MHz

  01xx　2.8 - 26.6 MHz　1xxx　4.2 - 46 MHz

  快捷宏定义：　FN_2　FN_3　FN_4　FN_8

从上面对 FLL 电路的分析可知，倍频环路是一个积分累加式的调整过程。从开始工作到输出稳定频率需要一定的时间。如果数控频率发生器的中心频率（初始频率）恰好就在输出频率附近，那么调节过程就会很快。若偏离较远，则需要等待积分器将误差累计到足够程度才能将频率调整准确，过程耗时较长。数字时钟发生器通过 FNx 控制位提供 5 种振荡器频率范围供选择，根据输出频率来选择最合适的频率范围，使调节过程最快。一般来说复位默认值适合 1MHz 左右频率，快捷宏定义 FN_x 适合产生 xMHz 附近时钟频率。

- *SCQF_M：调制禁止。0=调制允许　1=调制禁止　(位于 SCFQCTL 寄存器)*

在数控振荡器中，产生的频率值是离散的，且分辨率较低。为了增加所产生频率的分辨率，采用了 32 周期调制的方法。让输出频率在相邻 2 个频率值中切换，通过调整二者时间比例在宏观上相当于够微调频率。这个过程由 FLL 倍频环路自动完成，用户无需干预。但可以通过将 SCQF_M 置 1 来禁止调制功能。禁止调制后的输出频率稳定，但有误差。

- *SCG0：　FLL 禁止。0=FLL 开启　1=FLL 禁止　(位于 SR 寄存器)*

该位置 1 后，将禁止倍频环中的频率积分器（计数器），此后 FLL 的输出频率将不再被自动调整。低功耗模式中使用。

- *SCG1：　时钟发生器禁止。0=时钟发生器禁止开启　1=禁止　(位于 SR 寄存器)*

该位置 1 后，将禁止时钟发生器。低功耗模式中使用。

- *CPUOFF：CPU 停止。　0=CPU 工作　1=CPU 停止　(位于 SR 寄存器)*

该位置 1 后，将关闭 MCLK，相当于停止 CPU 的工作，但不影响整个时钟系统的工作，也不关闭 SMCLK。低功耗模式中使用。

例 2.3.3 ：MSP430F42x 单片机外部接有 32.768KHz 手表晶振，CPU 需要 2MHz 左右时钟频率：

```
FLL_CTL0 &=~ XTS_FLL;                  // 设置振荡器类型为低频  （可省略）
FLL_CTL0 |=  XCAP18PF;                 // 设置晶振匹配电容 18pF 左右
SCFQCTL = SCFQ_2M;                     // 倍频至 2MHz(64 倍频，2.09MHz)
SCFI0  |=FN_2 ;                        // DCO 中心频率 2MHz 左右(1.3~12.1 MHz)
```

例 2.3.4 ： MSP430F42x 单片机外部接有 32.768KHz 手表晶振，CPU 需要 2.752MHz 时钟频率。2.752MHz 是 32.768KHz 的 84 倍。因此分频系数 N=84-1=83。

```
FLL_CTL0 &=~ XTS_FLL;                  // 设置振荡器类型为低频  （可省略）
FLL_CTL0 |=  XCAP18PF;                 // 设置晶振匹配电容 18pF 左右
SCFQCTL = 83;                          // 倍频至 2.752MHz(84 倍频)
SCFI0  |=FN_3 ;                        // DCO 中心频率 3MHz 左右(2~17.9MHz)
```

例 2.3.4 ： MSP430F42x 单片机外部接有 32.768KHz 手表晶振，CPU 需要 6.554MHz 时钟频率。6.554MHz 是 32.768KHz 的 200 倍。由于通过分频系数 N 最大只能实现 128 倍频，因此需要开启 DCOPLUS 利用 FLLDx 再额外倍频。200=2×100，可以将 N 设为 99，FLLDx 设为 2 倍频：

```
FLL_CTL0 &=~ XTS_FLL;                  // 设置振荡器类型为低频  （可省略）
FLL_CTL0 |=  XCAP18PF;                 // 设置晶振匹配电容 18pF 左右
SCFQCTL = 99;                          // 先 100 倍频
FLL_CTL0|=DCOPLUS;                     // 开启额外的倍频
SCFI0  |=FLLD_2+FN4;                    // 额外 2 倍频，DCO 中心频率 4MHz
```

上面的例子示范了通过倍频器灵活地配置 CPU 工作频率。倍频器使得 MSP430F4xx 系列单片机在不同应用中都能够使用同一款手表晶振，而且可以通过倍频器随时改变 CPU 主频和外设工作频率。使用倍频器的时候需注意，MSP430 单片机时钟频率存在上限，而倍频器可以产生超过上限频率的时钟，此时单片机功能将不稳定，甚至停止工作。

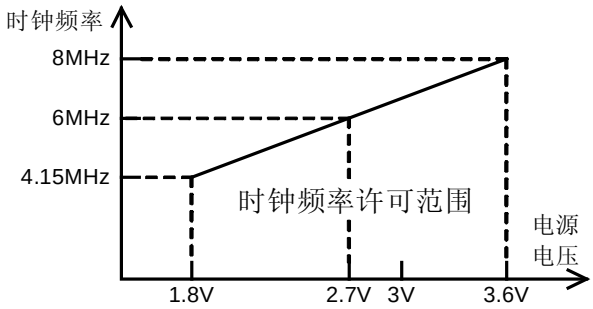


图 2.3.4 MSP430x4xx/1xx 单片机时最高钟频率与电源电压关系

MSP430 单片机的时钟频率上限与电源电压有关。对于 1 系列和 4 系列单片机来说，3.6V 电源时最大允许 8MHz 时钟频率，当电源电压下降，最高工作频率也随之下降。电源电压和最高工作频率的关系可参考上图。

例如某款产品使用 2 节 5 号电池，正常工作约 3V。考虑到电池寿命耗尽时电压降落到 2.5V，最高时钟不超过 5MHz 是安全稳定的。若考虑到用户可能将充电电池装入产品，正常工作电压仅 2.4V，电量耗尽时仅 2V。时钟取频率取 4MHz 以下才是安全的。

l 时钟错误处理

在电路板上各种元器件中，晶振是非常脆弱的。它的内部是很薄的石英晶体切片，受震后易碎，且震动对输出频率也有影响。此外，电路板受潮、进水都可能导致参数变化停振。MSP430 单片机内部自带了时钟检测电路，遇到时钟停振、不稳定等情况能产生中断。即使晶振停止时，对于 FLL 倍频器内的积分器来说没有加法时钟而只有减法时钟，导致不断降低 DCO 频率，最终 DCO 频率会被降到最低（由 FNx 所选频率范围的下限）但仍能为 CPU 输出时钟。这使得即使晶振停止时 CPU 仍能进行必要的应急处理。时钟错误中断错误相关控制位：

n *OFIE： 时钟错误中断允许 0=禁止 1=允许 (位于 IE1 寄存器)*

该控制位决定是否允许时钟错误中断。时钟错误中断是一个不可屏蔽中断（NMI 中断），不受总中断允许位的制约。只要 OFIE 位置 1，都会允许时钟错误中断。

n *OFIFG： 时钟错误中断标志位 0=时钟正常 1=时钟曾发生错误 (位于 IFG1 寄存器)*

该位是时钟错误中断标志位。只要时钟系统发生了错误都会将该标志位置 1。由于上电复位过程中晶振起振需要数毫秒时间，也会认为是时钟错误，从而将该标志位置 1，因此在开启时钟错误中断之前需要清除该标志。除了时钟错误之外，还有若干个中断公用了 NMI 中断入口，在 NMI 中断内需要判断 OFIFG 标志，并人工清除。

例 2.3.5 ：MSP430F42x 单片机外部接有 32.768KHz 手表晶振，CPU 需要 2MHz 左右时钟频率。若时钟发生错误则点亮 P2.0 口的 LED 作为指示：

```
#include "msp430x42x.h"            /*430 单片机寄存器头文件*/
void main( void )                  // 主程序
{ int i;
  WDTCTL=WDTPW+WDTHOLD;            // 停止看门狗
  FLL_CTL0 &=~ XTS_FLL;            // 设置振荡器类型为低频  （可省略）
  FLL_CTL0 |=  XCAP18PF;           // 设置晶振匹配电容 18pF 左右
  SCFQCTL = SCFQ_2M;               // 倍频至 2MHz(64 倍频，2.09MHz)
  SCFI0 |= FN_2 ;                  // DCO 中心频率 2MHz 左右(1.3~12.1 MHz)
  P2DIR |= BIT0;                    // P2.0 设为输出
  P2OUT &=~BIT0;                    // P2.0 设为低电平(LED 灭)
  for(i=0;i<100;i++);               // 略延迟，等待倍频器输出时钟稳定
  IFG1 &=~ OFIFG;                   // 清除上电过程引起时钟错误标志
  IE1 |= OFIE;                     // 打开时钟错误中断
    while(1)                       // 主循环
  {
     //...do some thing...          // 正常工作的程序
  }
}

#pragma vector = NMI_VECTOR        //NMI 不可屏蔽中断源
__interrupt void NMI_ISR(void)     //声明一个中断服务程序，函数名为 NMI_ISR()
{
  if(IFG1 & OFIFG)                 //如果是时钟错误引发的 NMI 中断
  {//-----------------时钟错误处理程序----------------------------
```

```
      P2OUT |= BIT0;                    //将 P2.0 置高(点亮 LED)
      IFG1 &=~ OFIFG;                   //清除时钟错误标志
    }
  }
```

运行上面的程序后，用手指触摸一下晶振的引脚（相当于改变了晶振的负载电容，会造成停振），LED 立即被点亮，说明错误处理程序被执行。实际应用中，若发生时钟错误，可以通过中断程序进行保存结果、清除定时器、复位单片机等应急操作。

Ⅰ 低功耗模式

超低功耗是 MSP430 单片机的一大特色。MSP430 系列单片机具有 5 种不同深度的低功耗休眠模式。在空闲时，通过不同程度的休眠，将内部各个模块尽可能地被关闭，从而降低功耗。关闭模块最简单的途径是关闭时钟，因此低功耗模式的管理是通过时钟系统来完成的。共有 4 个控制位参与：CPUOFF、SCG0、SCG1、OSCOFF。各控制位的功能在时钟系统中已经介绍过。通过这四个位的不同组合，构成了 5 种低功耗休眠模式。

表 2.3.1 MSP430F42x 单片机低功耗模式

部件/控制位 / 模式	CPU 处理器	FLL 倍频环	数字时钟发生器	晶体振荡器	基本功耗（3V 供电，32K 晶振）
	CPUOFF	SCG0	SCG1	OSCOFF	
Active（活动模式）	0（开启）	0（开启）	0（开启）	0（开启）	400uA /MHz
LPM0（低功耗模式 0）	1（关闭）	0（开启）	0（开启）	0（开启）	100uA
LPM1（低功耗模式 1）	1（关闭）	1（关闭）	0（开启）	0（开启）	50uA
LPM2（低功耗模式 2）	1（关闭）	0（开启）	1（关闭）	0（开启）	7uA
LPM3（低功耗模式 3）	1（关闭）	1（关闭）	1（关闭）	0（开启）	1uA
LPM4（低功耗模式 4）	1（关闭）	1（关闭）	1（关闭）	1（关闭）	0.1uA

从表中可以看出规律：随着休眠深度依次加深，时钟系统中被关闭的部件数目增加，而功耗也依次降低。各模式的特点与用法简单归纳如下：

活动模式:

正常工作状态，全部时钟均开启。功耗正比与 CPU 时钟速度（MCLK 频率）。对于 MSP430F4xx 系列，比例系数是 400uA/MHz。CPU 速度越慢功耗越低，但对于相同运算量的代码，速度降低一半，执行时间会加倍，总耗电量不变。因此通过降低 CPU 频率并不能有效地降低实际功耗。相反将 CPU 时钟设为较高频率，通过提高 CPU 速度节省运算时间，让 CPU 与其他模块更长时间处于休眠状态，反而有利于降低总功耗。

低功耗模式 0:

在低功耗模式 0 下，时钟系统仍然全部正常工作，只有 MCLK 的输出被禁止。结果是只关闭 CPU。此时 SMCLK 和 ACLK 仍然有效，且 SMCLK 与 ACLK 之间的倍频关系仍然成立。选择了这两时钟作为时钟源的其他模块会继续工作。CPU 被关闭后，程序将停止不再继续执行，直到被中断唤醒，或单片机被复位。因此在进入任何一个低功耗模式之前都必须设置好唤醒 CPU 的中断条件、打开中断允许位、等待被唤醒。否则程序将永远

停止。

低功耗模式 1:

在低功耗模式 1 下，不仅 MCLK 的输出被禁止，DC 发生器（数字时钟）也被关闭。结果是 CPU 被关闭、ACLK 仍然有效、SMCLK 仍然输出但频率和 ACLK 之间的倍频关系不再成立，SMCLK 只能作为一个粗略的高频时钟使用。对于某些应用中不需要准确时钟频率的模块，仍可继续使用 SMCLK 作为时钟。

低功耗模式 2:

在低功耗模式 2 下，MCLK 的输出被禁止、倍频环路被关闭、SMCLK 被关闭，仅打开数字时钟发生器和 ACLK。该模式一般不用，因为禁止 SMCLK 后打开数字时钟发生器已无意义。建议用更低功耗的 LPM3 来替代。与 LMP3 不同之处是 SCG0 依然打开，唤醒延迟比 LPM3 短，且唤醒后 SMCLK 时钟是准确的。

低功耗模式 3:

在低功耗模式 3 下，整个时钟系统中除了低频晶振的振荡器保持活动之外，其余全部被关闭。因此 MCLK、SMCLK 都被停止，仅留 ACLK 保持活动。只有选择 ACLK 作为时钟源的设备会继续工作。该模式下功耗仅 1uA，且活动的 ACLK 可以用于驱动液晶或驱动定时器产生中断周期性地唤醒 CPU，是最常用的低功耗模式。

低功耗模式 4:

在低功耗模式 4 下，整个时钟系统全部关闭，MCLK、SMCLK、ACLK 全部被禁止。这样单片机内部的所有模块都将停止活动。功耗也降到最低（0.1uA）。这种状态下单片机内部模块不可能再唤醒 CPU，只能依靠外部中断（如 IO 口中断）唤醒 CPU 继续执行程序，或者通过复位来唤醒 CPU 重新执行程序。对于电池来说，0.1uA 电流甚至小于电池自放电电流，所以该模式下功耗可以忽略不计。因此可以通过进入低功耗模式 4 实现关机功能。

当系统暂时空闲时，应尽可能进入低功耗模式。而且要根据系统中可以被关闭的模块和时钟决定低进入何种功耗模式，使休眠最深且仍能被唤醒。

表 2.3.2 MSP430F42x 单片机低功耗模式与时钟的关系

时钟 / 模式	MCLK（CPU 用）	SMCLK（高速设备用）	ACLK（低速设备用）
Active（活动模式）	开启	开启，频率准确	开启
LPM0（低功耗模式 0）	关闭	开启，频率准确	开启
LPM1（低功耗模式 1）	关闭	开启，频率不准确	开启
LPM2（低功耗模式 2）	关闭	关闭	开启
LPM3（低功耗模式 3）	关闭	关闭	开启
LPM4（低功耗模式 4）	关闭	关闭	关闭

低功耗模式的唤醒:

无论处于何种低功耗模式，只要有中断发生都会响应中断。进入中断前 CPU 会自动

将存有四个低功耗控制位的 SR 寄存器压入堆栈，并自动清除 SCG1、OSCOFF、CPUOFF 三个控制位，但不清除 SCG0 控制位。由于 LMP1、LPM3、LPM4 模式下 SCG0 是置位的，唤醒进入中断服务程序后，SCG0 仍然置位，DC 发生器仍被关闭。因此从上述 3 个模式唤醒后 MCLK 和 SMCLK 都是不准确的。若中断服务程序需要准确的 CPU 时钟，例如触控应用中计算电容充电周期需要很准确的时钟，则需在中断内人工清除 SCG0 标志位：

```
_BIC_SR(SCG0);              //清除 SR 寄存器的 SCG0 控制位
```

低功耗模式的退出:

当中断服务程序执行完毕，CPU 会自动地从堆栈中恢复 SR 寄存器。由于 SR 存放了低功耗模式控制位，退出中断服务程序后仍然保持原模式不变。这非常适合在中断内处理完全部任务，一旦执行完中断返回后立即休眠，等待下一个中断任务。

若希望中断结束后恢复到正常的活动模式，可以在中断结束前修改堆栈内 SR 值：

```
__low_power_mode_off_on_exit();     //退出中断时唤醒 CPU。注意开头两个'_'
```

这种方式非常适合替代流程中的等待过程。例如需要等待 A 事件发生后再执行 B 任务，可以设置好 A 事件的中断条件后休眠，等待被唤醒后继续执行 B 任务。等待的过程中系统是休眠的，节省了功耗。

内部函数

ICC430 编译器为低功耗模式的设置与控制提供了以下的内部函数：

```
__low_power_mode_0(); 或 LPM0;  //进入低功耗模式 0
__low_power_mode_1(); 或 LPM1;  //进入低功耗模式 1
__low_power_mode_2(); 或 LPM2;  //进入低功耗模式 2
__low_power_mode_3(); 或 LPM3;  //进入低功耗模式 3
LPM0_EXIT();                   //退出中断时清除 LPM0 相关控制位
LPM1_EXIT();                   //退出中断时清除 LPM1 相关控制位
LPM2_EXIT();                   //退出中断时清除 LPM2 相关控制位
LPM3_EXIT();                   //退出中断时清除 LPM3 相关控制位
LPM4_EXIT();                   //退出中断时清除 LPM4 相关控制位
__low_power_mode_off_on_exit(); //退出时唤醒 CPU。
__bic_SR_register(); 或 _BIC_SR();  //将 SR 寄存器的某些位清零
__bis_SR_register(); 或 _BIS_SR(); //将 SR 寄存器的某些位置位
```

I 低功耗模式的应用

1. 间歇工作:

实际的系统中，很多设备都不必一直连续工作，让大部分设备间歇工作，并尽可能延长工作时间间隔、减少工作时间、加深休眠深度。这是超低功耗系统设计的基本思想之一。

例 2.3.6： 在 MSP430 单片机组成的系统中，ACLK=32.768K，CPU 速度 1MHz。假设 P1.5 接有按键（按下为低电平）、P2.0 输出至 LED。要实现当按键被按下时 LED 亮，按键松开后 LED 灭。若 CPU 一直读取 IO 并处理，耗电很大（活动模式 400uA）：

```
#include "msp430x42x.h"            /*单片机寄存器头文件*/
void main( void )                  // 主程序
{
  WDTCTL=WDTPW+WDTHOLD;            // 停止看门狗
```

```
  FLL_CTL0 |=  XCAP18PF;            // 设置晶振匹配电容 18pF 左右
  P2DIR |= BIT0;                    // P2.0 设为输出，其余 IO 默认输入
  while(1)                          // 主循环
    {
       if((P1IN & BIT5)==0)     P2OUT |= BIT0;  //若键被按下，点亮 LED
       else                     P2OUT &=~BIT0;  //若键松开，关闭 LED
    }
}
```

考虑到键盘是个慢速的设备，对 IO 口每 10ms 读取一次的速度已经足够。在 10ms 间隔期间让 CPU 以及大部分设备休眠将节省大部分功耗。可以利用定时器产生中断，周期性地唤醒 CPU，并尽量选择 ACLK 作为定时器的时钟，在进入低功耗模式 3 后仍能保持活动。按照该方法将程序修改成间歇工作形式：

```
#include "msp430x42x.h"             /*单片机寄存器头文件*/
void main( void )                   // 主程序
{
  WDTCTL = WDTPW+WDTHOLD;           // 停止看门狗
  FLL_CTL0 |=  XCAP18PF;            // 设置晶振匹配电容 18pF 左右
  P1DIR |= BIT0+BIT1+BIT2+BIT3+BIT4;
  P2DIR |= BIT0+BIT1+BIT2+BIT3;     //悬空不用的 IO 口要置为输出
  P1OUT = 0;                        //否则不确定电平会造成 IO 耗电
  P2OUT = 0;
  BTCTL = BT_ADLY_8;//BasicTimer 时钟选为 ACLK，设为 1/128 秒(约 8ms)中断一次
  IE2 |= BTIE;                      // 允许 BasicTimer 中断
  _EINT();                          // 允许总中断
  LPM3;                             // 进入低功耗模式 3，等待被唤醒
  //------程序永远不会执行到这里----------------------
}

#pragma vector = BASICTIMER_VECTOR
__interrupt void BT_ISR(void)   // 1/128 秒一次中断（由 BasicTimer 所产生）
{
  if((P1IN & BIT5)==0)  P2OUT |= BIT0;      //若键被按下，点亮 LED
  else                  P2OUT &=~BIT0;      //若键松开，关闭 LED
}                                           //退出中断后仍保持原休眠状态
```

改为间歇工作后，功耗由 400uA 降至 5uA，且功能不变。上面的程序全部在中断内完成。也可通过休眠与唤醒机制来控制程序流程，写成下面的形式：

```
#include "msp430x42x.h"
void main( void )
{
  WDTCTL = WDTPW+WDTHOLD;
  FLL_CTL0 |=  XCAP18PF;            // 设置晶振匹配电容 18pF 左右
  P1DIR |= BIT0+BIT1+BIT2+BIT3+BIT4;
  P2DIR |= BIT0+BIT1+BIT2+BIT3;     //悬空不用的 IO 口要置为输出
  P1OUT = 0;                        //否则不确定电平会造成 IO 耗电
  P2OUT = 0;
  BTCTL = BT_ADLY_8;//BasicTimer 时钟选为 ACLK，设为 1/128 秒(约 8ms)中断一次
```

```
  IE2 |= BTIE;                          // 允许 BasicTimer 中断
  _EINT();                              // 总中断允许
  while(1)                              // 主循环
    {
        LPM3;    //休眠，仅留 ACLK，等待被唤醒。以下代码将每 1/128 秒执行一次。
       if((P1IN & BIT5)==0)     P2OUT |= BIT0;  //若键被按下，点亮 LED
       else                     P2OUT &=~BIT0;  //若键松开，关闭 LED
    }
}

#pragma vector = BASICTIMER_VECTOR
__interrupt void BT_ISR(void)   // 1/128 秒一次中断（由 BasicTimer 所产生）
{
   __low_power_mode_off_on_exit();     //退出中断时唤醒 CPU。
 }
```

2. 替代程序流程中的等待过程:

MSP430 单片机中，几乎所有的设备都能产生中断，目的在于让 CPU 无需查询即能等待设备。因此可以用休眠替代查询等待，设备在发生状态变化时将会主动唤醒 CPU 进行后续的处理。

例 2.3.7：从串口发送一字节数据。串口发送过程一般是先写入发送寄存器后等待发送完毕，然后才能发送下一字节：

```
void UART_PutChar(char Chr)
{
 TXBUF0=Chr;                                    // 写入发送寄存器
 while ((IFG1 & UTXIFG0)==0);                   // 等待数据发完
}
```

串口速度较慢，远低于 CPU 速度。例如以 2400bps 波特率发送 1 字节需要 4ms 时间；而以 1MHz 时钟执行 TXBUF0=Chr 赋值的过程却只需要 1us 时间。因此等待数据发完的过程要浪费 4000 个 CPU 周期用于查询。若将等待过程替换成休眠，则可节省大量的 CPU 耗电。假设使用 ACLK 作为串口模块的时钟，进入低功耗模式 3 后串口仍工作，而 CPU 及大部分时钟系统已经停止，由串口发送完毕中断唤醒继续执行：

```
void UART_PutChar(char Chr)
{
 TXBUF0=Chr;                                    // 写入发送寄存器
 LPM3;                                          // 休眠，等待数据发完中断唤醒
}
#pragma vector=UART0TX_VECTOR
__interrupt void UART_TX (void)
{
  __low_power_mode_off_on_exit();  //退出中断时唤醒 CPU。
}
```

当系统中只开启串口发送中断时，上面两个的程序功能完全等价，后者功耗更低。但在系统中开启了多个中断时，由于任何中断都可以唤醒休眠模式，有可能出现串口未发送完毕时发生了其他中断唤醒 CPU 继续执行下一次发送。这是不希望发生的。解决的办法

是设置一个全局变量标志位，用于识别中断源以决定唤醒后是否继续执行。

```
char UART_TxFlag;                    // 定义一个全局变量作为标志
void UART_PutChar(char Chr)          // 发送 1 字节数据的函数
{
 TXBUF0=Chr;                         // 将数据写入发送寄存器
 UART_TxFlag=0;                      // 清除全局变量标志位
 while(UART_TxFlag==0) LPM3;         // 只有串口发送中断唤醒 CPU 才能继续执行
}

#pragma vector=UART0TX_VECTOR        //串口发送完毕中断入口
 __interrupt void UART_TX (void)     //中断服务程序声明
 {
   UART_TxFlag=1;                    //全局变量标志位置 1，供唤醒后识别用
   __low_power_mode_off_on_exit();   //退出中断时唤醒 CPU。
 }
```

当程序休眠在 while(UART_TxFlag==0) LPM3;时，任何中断都可以将 CPU 唤醒。但只有串口发送完毕中断会将 UART_TxFlag 置 1。在 while 循环中检查到 UART_TxFlag 为 1 才会退出 while 循环继续执行后面的代码，否则会重新回到 LPM3 模式继续休眠。

3. 电源开关:

在所有的休眠模式中，LPM4 的功耗是最低的，仅 0.1uA。进入 LPM4 后单片机内部所有的部件都不再活动，仅保持 RAM 内数据和 IO 口状态不变。利用 LPM4 可以在不切断电源的情况下实现“软件关机”。

例 2.3.8：设计一个闪烁警告灯，工作时每秒亮 2 次，每次亮 125ms。不留机械式电源开关，用 2 个按键当开关：按 1 键开机，按 2 键关机。假设用 P1.6 键（低有效）开机、P1.5 键（低有效）关机、P2.0 口驱动 LED（高点亮）：

```
#include "msp430x42x.h"
char TimeCount=0;          //闪烁计时
char BT_IntervalFlag;      //定时中断全局变量标志
void main( void )
{
  WDTCTL = WDTPW+WDTHOLD;
  FLL_CTL0 |=  XCAP18PF;             // 设置晶振匹配电容 18pF 左右
  P1DIR |= BIT0+BIT1+BIT2+BIT3+BIT4;
  P2DIR |= BIT0+BIT1+BIT2+BIT3;      //悬空不用的 IO 口要置为输出
  P1OUT = 0;                         //否则不确定电平会造成 IO 耗电
  P2OUT = 0;
  BTCTL = BT_ADLY_125;  //BasicTimer 时钟选为 ACLK，设为 125ms 中断一次
  IE2 |= BTIE;                        // 允许 BasicTimer 中断
  P1IES |=  BIT6;                     // P1.6 下降沿触发中断
  P1IE |= BIT6;                       // 允许 P1.6 中断(开机键)
  _EINT();                           // 总中断允许
  while(1)                           // 主循环
    {  BT_IntervalFlag=0;
       while(BT_IntervalFlag==0) LPM3;
       //休眠，仅留 ACLK，仅能被 BasicTimer 唤醒。以下代码将每 1/8 秒执行一次。
```

```
        TimeCount++;                        //计数
        if(TimeCount>=4) TimeCount=0;       //产生 0~3 计数(0.5 秒)
        if(TimeCount==0) P2OUT |= BIT0;     //亮 125ms
        else             P2OUT &=~BIT0;     //灭 375ms
        if((P1IN & BIT5)==0)                //若 KEY1 被按下
         {
           P2OUT &=~ BIT0;                  //关闭 LED
           LPM4;                            //关机.ACLK 停止，仅有外部中断能唤醒
         }
      }
   }
//-------------------------------------------------------------------------
   #pragma vector = BASICTIMER_VECTOR // BasicTimer 定时器中断（1/8 秒）
   __interrupt void BT_ISR(void)      // 声明一个中断服务程序，名为 BT_ISR()
   {
     BT_IntervalFlag=1;               //全局变量标志位置 1，供唤醒后识别用
     __low_power_mode_off_on_exit();  //退出中断时唤醒 CPU。
   }
//-------------------------------------------------------------------------
   #pragma vector = PORT1_VECTOR        //P1 口中断源
   __interrupt void P1_ISR(void)        //声明一个中断服务程序，名为 P1_ISR()
   {                     //P1.6 口的 KEY2 键产生中断，将单片机从 LPM4 休眠中唤醒
     P1IFG=0;            //清除 P1 口中断标志位
     __low_power_mode_off_on_exit();  //退出中断时唤醒 CPU。
   }
```

当检测到 P1.5 按下时，进入低功耗模式 4。之后 ACLK 也被停止，定时器以及一切内部模块都停止工作。只有通过外部的中断（如 IO 口中断）能够唤醒 CPU。通过按键 2（P1.6）触发中断唤醒 CPU 后，程序才继续执行。

除了外部中断以外，复位操作也能唤醒休眠模式，但程序会重新开始运行。在实际应用中复位键也是一种很常用的电源开关形式。上面的程序中，可以删除 P1.6 中断相关的代码，用复位键替代开机键。

用复位键也能作为单键电源开关使用。由于复位操作不会改变 RAM 内的数据，可以在程序开始时对一个变量取反，再根据该变量值决定执行程序（开机）或是进入 LPM4（关机）。由于 C 语言的初始化程序在 main 函数执行之前会对所有 RAM 清零，为了保证电源标志变量能够不被清除，定义时需要加__no_init 关键字。

```
#include "msp430x42x.h"
char TimeCount=0;                       //闪烁计时变量
__no_init char PWR_Flag;                //电源标志，复位后不清零
void main( void )
{
  WDTCTL = WDTPW+WDTHOLD;
  P1DIR |= BIT0+BIT1+BIT2+BIT3+BIT4;
  P2DIR |= BIT0+BIT1+BIT2+BIT3;         //悬空不用的 IO 口要置为输出
  P1OUT = 0;                            //否则不确定电平会造成 IO 耗电
  P2OUT = 0;
  BTCTL = BT_ADLY_125; //BasicTimer 时钟选为 ACLK，设为 125ms 中断一次
```

```
  IE2 |= BTIE;                          // 允许 BasicTimer 中断
  _EINT();                              // 总中断允许
  if(PWR_Flag ==0)  PWR_Flag=1;         // 电源标志每次复位后取反
  else             PWR_Flag=0;
  if(PWR_Flag ==0)  LPM4;               //电源标志为 0 时关机，不再执行
 //-----------------电源标志为 1 时才执行主循环-------------------------
  while(1)                              // 主循环
    {
      LPM3;  //休眠，仅留 ACLK，等待被 BasicTimer 唤醒，以下代码将每 1/8 秒执行一次。
      TimeCount++;                      //计数
      if(TimeCount>=4) TimeCount=0;     //产生 0~3 计数(0.5 秒)
      if(TimeCount==0) P2OUT |= BIT0;  //亮 125ms
      else             P2OUT &=~BIT0;  //灭 375ms
    }
}

#pragma vector = BASICTIMER_VECTOR     // BasicTimer 定时器中断
__interrupt void BT_ISR(void)
{
   __low_power_mode_off_on_exit();     //退出中断时唤醒 CPU。
}
```

低功耗模式的引入，使得程序的思路和实现方法会发生一些变化。下一章中将详细讨论超低功耗软件的整体结构和常用编程方法。

2．4 Basic Timer 基础定时器

在低功耗模式的应用中，我们已经看到用间歇工作方式能够有效地降低功耗。实现间歇工作需要一个定时器用于定时唤醒。同时我们会发现，实际上只需要很粗略的定时就能实现该功能。类似的，在键盘扫描、查询等应用中也会出现只需要粗略定时功能的情况。

MSP430F4xx 单片机内部专门提供了一个产生周期节拍的定时器，叫做基础定时器（Basic Timer）。它能在无需 CPU 干预的情况下产生 2^N 周期的定时中断，供系统间歇唤醒用。当 Basic Timer 时钟选择为 32.768KHz 时，定时周期恰好 $1/2^N$ 秒，可以为 RTC 走时或秒表等计时程序提供精确时基。

与此同时，Basic Timer 定时器还为 LCD 的刷新提供时钟。通过 Basic Timer 的设置可以改变 LCD 的刷新频率。

Ⅰ Basic Timer 结构与原理

从结构图上看，Basic Timer 定时器更像是一个二进制分频器。此分频器结构分为了 2 级，每级 256 分频。对于第一级分频器（BTCNT1）来说，时钟只能是 ACLK，它的输出为 LCD 刷新提供时钟。而第二级分频器的时钟源可以时 ACLK、SMCLK 或是 ACLK 经

过第一级 256 分频后的输出（级联）。从第二级分频过程中可以选择输出抽头作为中断源，从而通过改变分频系数而改变中断间隔时间。两级分频器级联后最多构成 65536 分频（2^{16} 分频）。

Basic Timer 定时器总共有 5 个控制位：BTDIV、BTHOLD、BTSSEL、BTFREQx、BTIPx。都位于 BTCTL 寄存器。该寄存器在复位后保持不变，没有默认值，因此上电后一定要对该寄存器进行设置。

- ***BTDIV：** 预分频选择 0=无预分频 1= 256 分频 (位于 BTCTL 寄存器)*

当 BTDIV=0 时，第二级分频器 BTCNT2 的时钟源直接来自于 ACLK 或 SMCLK。当 BTDIV=1 时，第二级分频器的时钟源来自于 256 分频后的 ACLK。在一般情况，ACLK=32.768K 时，选择了预分频能产生长时间的定时中断，最长达 2 秒。

- ***BTSSEL：** 时钟源选择 0=ACLK 1=SMCLK (位于 BTCTL 寄存器)*

当 BTDIV=1 时，该位无效，时钟选为 ACLK/256。

快捷宏定义 BT_fCLK2_ACLK BT_fCLK2_MCLK BT_fCLK2_ACLK_DIV256

从结构图中可以看出，第二级分频器的输出决定了中断频率。而第二级分频器的时钟源是由 BTDIV 和 BTSSEL 两位共同决定的。考虑到选择 SMCLK 作为时钟源的目的是得到高频中断，而开启 BTDIV 分频的目的是得到低频中断，两者不会同时被选择。所以开启 BTDIV 后无论 BTSSEL 位高低，一律选 256 分频后的 ACLK 作为第二级时钟。

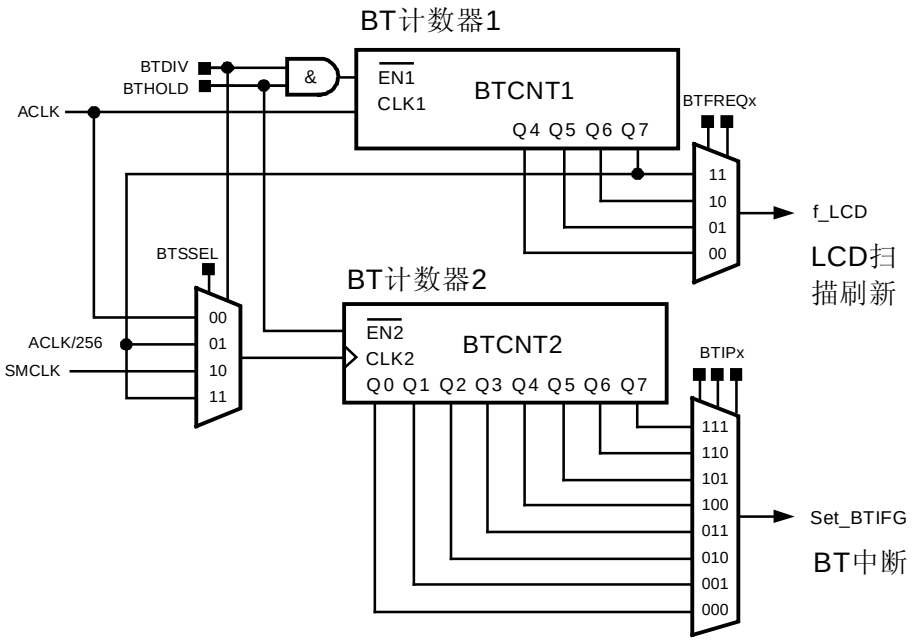


图 2.4.1 MSP430x4xx 单片机 Basic Timer 结构图

- ***BTHOLD：** Basic Timer 停止 0=正常运行 1=停止运行 (位于 BTCTL 寄存器)*

当 BTHOLD 控制位置 1 后，Basic Timer 将暂停运行，不再产生中断，计数器值将保持不变。可以将该标志位作为 Basic Timer 的启停开关使用。

注意第一级计数器的使能端由 BTDIV 与 BTHOLD 相与后控制。若 BTDIV=1（两级分频级联）时，BTHOLD 置 1 将两个计数器都停止，LCD 的刷新也随之停止。当 BTDIV=0 时，BTHOLD 置 1 只会暂停 BTCNT2，定时中断的发生被停止，但 LCD 的刷新仍然保持工作。因此在使用了 LCD 的应用中，若不希望暂停 Basic Timer 影响 LCD 显示，BTDIV 必须为 0。

在不使用 Basic Timer 的应用中，可以利用该控制位将 Basic Timer 关闭以节省电能。或者在只用 LCD 而不用定时中断的应用中，只关闭第二级，也能节省部分功耗。

- *BTIPx： Basic Timer 中断频率选择 (位于 BTCTL 寄存器)*

  000=f_{CLK2}/2 001=f_{CLK2}/4 010=f_{CLK2}/8 011=f_{CLK2}/16

  100=f_{CLK2}/32 101=f_{CLK2}/64 110=f_{CLK2}/128 111=f_{CLK2}/256

  快捷宏定义：BT_fCLK2_DIV2 BT_fCLK2_DIV4 BT_fCLK2_DIV8

  BT_fCLK2_DIV16 BT_fCLK2_DIV32 BT_fCLK2_DIV64

  BT_fCLK2_DIV128 BT_fCLK2_DIV256

  或 ： BT_ADLY_0_064（0.064ms） BT_ADLY_0_125（0.125ms）

  BT_ADLY_0_25 (0.25ms) BT_ADLY_0_5（1/2048s）

  BT_ADLY_1（1/1024s） BT_ADLY_2（1/512s）

  BT_ADLY_4 (1/256s) BT_ADLY_8（1/128s）

  BT_ADLY_16（1/64s） BT_ADLY_32（1/32s）

  BT_ADLY_64（1/16s） BT_ADLY_125（1/8s）

  BT_ADLY_250（1/4s） BT_ADLY_500（1/2s）

  BT_ADLY_1000（1s） BT_ADLY_2000（2s）

通过该控制位中三比特组合，选择分频器 2 的 8 个抽头之一作为中断源。配合 BTDIV 共可产生 16 种中断频率。

- *BTFREQx： 设置 Basic Timer 为 LCD 提供的刷新频率 (位于 BTCTL 寄存器)*

  00=f_{ACLK}/32 01= f_{ACLK}/64 10= f_{ACLK}/128 11= f_{ACLK}/256

快捷宏定义：

BT_fLCD_DIV32 BT_fLCD_DIV64 BT_fLCD_DIV128 BT_fLCD_DIV256

或： BT_fLCD_1K BT_fLCD_512 BT_fLCD_256 BT_fLCD_128

- *BTCNT1 寄存器： 保存着第一级计数器的计数值 （8 位）*
- *BTCNT2 寄存器： 保存着第二级计数器的计数值 （8 位）*

通过读上述两个寄存器，可以获得当前两级计数器的计数值；通过对上述两个寄存器的写操作，可以对 Bssic Timer 进行赋初值、清零等操作。为了防止读写操作和计数递增过程同时操作寄存器造成错误，一般通过 BTHOLD 控制位将计数停止后再读写上述两个寄存器。

通过对 BTCNT 寄存器的读写，可以将 BasicTimer 作为通用的定时器使用，产生高分辨率、精确的定时。但实际一般不这么使用。因为随意更改 BTCNT 的值会造成 LCD 刷新率的改变，而且用 TA 定时器产生精确定时比用 BasicTimer 定时器灵活的多。Basic Timer

定时器一般只用于简单的周期性定时，不需要操作 BTCNT 寄存器。当然，在定时器资源不够用的情况下，用 BasicTimer 产生精确定时也可是可行的。

例 2.4.1：在某 430 单片机系统中，ACLK 时钟频率为 32.768KHz。用 Basic Timer 定时器产生周期为 1/4 秒的定时中断，同时为 LCD 提供 512Hz 的刷新时钟。

由于 1/4 秒周期较长，从 32768Hz 分频到 4Hz 需要 8192 分频，大于 256 分频。因此一级分频器不够，需要 2 级级联使用。先对 ACLK 进行 256 分频后再进行 32 分频。LCD 时钟从 32768 分频到 512Hz 需要 64 分频。可以利用快捷宏定义对 BTCTL 寄存器进行设置：

```
BTCTL= BT_fCLK2_ACLK_DIV256 + BT_fCLK2_DIV32 + BT_fLCD_DIV64;
//      第二级时钟为 ACLK/256   第二级分频系数为 32   LCD 分频系数为 64
```

也可以更简单地写成：

```
BTCTL= BT_ADLY_250 + BT_fLCD_512;
//  时钟=ACLK，中断周期 250ms，LCD 刷新频率 512Hz
```

例 2.4.2：在某 430 单片机系统中，ACLK 时钟频率为 32.768KHz。用 Basic Timer 定时器产生周期为 1/1024 秒的定时中断，同时为 LCD 提供 256Hz 的刷新时钟。

由于 1/1024 秒周期较短，从 32768Hz 分频到 1024Hz 需要 32 分频，小于 256 分频。因此一级分频器足够。LCD 时钟从 32768 分频到 256Hz 需要 128 分频。

```
BTCTL= BT_fCLK2_ACLK + BT_fCLK2_DIV32 + BT_fLCD_DIV128;
//      第二级时钟为 ACLK   第二级分频系数为 32  LCD 分频系数为 128
```

也可以更简单地写成：

```
BTCTL= BT_ADLY_1 + BT_fLCD_256;
//  时钟=ACLK，中断周期约 1ms，LCD 刷新频率 256Hz
```

Basic Timer 中断

在 Basic Timer 第二级分频器的 8 个输出抽头中，被选中的抽头每次由 0 到 1 的跳变（计数进位）会产生中断标志。若 BasicTimer 中断被允许，则会引发中断。相关标志位有：

- *BTIE：Basic Timer 中断允许位 0=禁止 1=允许 (位于 IE2 寄存器)*

该控制位决定是否允许 Basic Timer 中断。当不需要使用 Basic Timer 中断时，关闭该控制位。但注意关闭 Basic Timer 中断并不停止 Basic Timer 的运行。若不使用 Basic Timer 应该将 BTHOLD 位置 1，以节省电能。

- *BTIFG：Basic Timer 中断标志位 0=无中断发生 1=中断发生 (位于 IFG2 寄存器)*

每次计数器溢出，都会将 BTIFG 自动置 1。此时若总中断和 BTIE 被允许，则会引发中断。Basic Timer 独占了一个中断源，在中断内无需再判断标志位。因此发生中断后 BTIFG 会被硬件自动清除。

例 2.4.3：在某 430 单片机系统中，ACLK 时钟频率为 32.768KHz。用 Basic Timer 定时器让 P2.0 口上的 LED 每秒闪烁一次，同时为 LCD 提供 256Hz 的刷新时钟。

```
void  main( void )
{
   WDTCTL = WDTPW + WDTHOLD;          // 停止看门狗
   FLL_CTL0 |=  XCAP18PF;             // 设置晶振匹配电容 18pF 左右
```

```
    P2DIR |= BIT0;                          // P2.0 口设为输出
    BTCTL = BT_ADLY_500 + BT_fLCD_256;      // 中断周期 500ms，LCD 刷新频率 256Hz
    IE2 |= BTIE;                            // 允许 BasicTimer 中断
    _EINT();                                // 允许总中断
    while(1)
    {
       ... ...                              //主程序
    }
}

#pragma vector = BASICTIMER_VECTOR // BasicTimer 定时器中断（1/2 秒）
__interrupt void BT_ISR(void)      // 声明一个中断服务程序，名为 BT_ISR()
{
 P2OUT ^= BIT0;                    // P2.0 取反
}
```

Ⅰ Basic Timer 的应用

1. 周期性唤醒 CPU:

当 MSP430 单片机系统欧进入低功耗模式后，周期性地将其唤醒，查询是否有需要处理的事件，这种方法叫做定时查询。在例 2.3.6 中已经看到定时查询法在应对慢速设备时，能够显著地降低功耗。定时查询是最基本也是最常用的低功耗程序结构之一。

Basic Timer 非常适合做周期唤醒 CPU 的定时器。一般都选择 ACLK 作为时钟源，在功耗仅 1uA 的 LPM3 模式下 Basic Timer 仍能保持活动。且 ACLK 频率较低，16 位计数器能产生长达 2 秒的定时周期。

2. RTC 计时:

Basic Timer 的结构就是一个二进制计数器，而作为时钟源的低频晶振一般都是 32.768KHz（2^{15}Hz），经过分频后恰好能产生（$1/2^N$）秒的定时节拍。因为不存在定时器重装初值等软件操作，即使中断被某些原因延迟，仍不影响计时精度。例如用 Basic Timer 产生 1 秒定时中断，在定时中断内调用第一章 1.7 节中的 RTC_Tick()函数，将得到一个准确的软件实时钟。

使用 Basic Timer 中断作为计时节拍时，走时误差只取决于晶振频率的误差。一般晶振的误差都在±20ppm 以内，且与环境温度有关。20ppm 误差折合到计时大约一个月误差 1 分钟。在某些计时精度要求更高的场合，可以选用更高精度和稳定度的晶振。也可以关闭内部负载电容，使用外部负载电容，并在外部负载电容上并联一个 5pF 左右的微调电容。通过调整该电容改变谐振工作点，从而实现对晶振频率的微调。

3. 获得更高分辨率:

Basic Timer 的结构使得它适合产生（$1/2^N$）秒的定时中断。如果对 BTCNT 进行读写，也能够产生周期为非 2 整数幂的定时中断。

例 2.4.4：在某 430 单片机系统中，ACLK 时钟频率为 32.768KHz。用 Basic Timer 定

时器产生 0.75 秒的定时中断。

```
    BTCTL  = BT_ADLY_1000 + BTHOLD; // 先将中断周期设为 1000ms，并暂停
    BTCNT1 = 0;
    BTCNT2 = 32;                    // 预置初值 32 (128 的 1/4),扣除 0.25 秒
    BTCTL  &=~ BTHOLD;              // 恢复 BasicTimer 的运行
    ... ...
#pragma vector = BASICTIMER_VECTOR // BasicTimer 定时器中断（1 秒）
__interrupt void BT_ISR(void)      // 声明一个中断服务程序，名为 BT_ISR()
{
    BTCTL  = BT_ADLY_1000 + BTHOLD; // 先将中断周期 1000ms，并暂停
    BTCNT1 = 0;
    BTCNT2 = 32;                    // 重置初值 32 (128 的 1/4),扣除 0.25 秒
    BTCTL  &=~ BTHOLD;              // 恢复 BasicTimer 的运行
    }
```

Basic Timer 能产生最接近的周期是 0.5 秒或 1 秒，无法直接产生 0.75 秒周期。但通过 BTCNT 寄存器可以修改定时值，从 1 秒定时中扣除 0.25 秒可以得到 0.75 秒定时。从结构图中看出，当 Basic Timer 取 1 秒周期时（Q6 抽头），BTCNT2 每隔 128 周期中断一次。如果在每次中断内扣除 1/4 周期，即赋初值 32，定时周期就变成了 0.75 秒。如果对 BTCNT1 也进行类似的修改，还能获得更高的分辨率。

一般不推荐将这样使用 Basic Timer，仅在定时器资源不够用的情况下才使用这种方式。因为修改 BasicTimer 计数值可能会导致 LCD 工作不正常，并且需要依靠软件值初值，对中断响应延迟非常敏感。从 LPM3 唤醒延迟较长（6us 左右）且具有一定不确定性，会导致计时不准确。

一般来说，Basic Timer 按照整个系统所需的最小节拍周期工作，需要更长的周期可以用软件模拟的方法实现，而不推荐直接修改 BTCNT。例如 BasicTimer 可以产生 0.25 秒中断，而 0.75 秒是 0.25 秒的 3 倍，通过一个全局变量累加 3 次即可得到 0.75 秒时基。

例 2.4.5：在某 430 单片机系统中，ACLK 时钟频率为 32.768KHz。用 Basic Timer 定时器为 0.25 秒、0.5 秒、0.75 秒、1.5 秒四个定时服务程序提供时钟节拍。

```
unsigned char Timer3=0; // 计时用全局变量
unsigned char Timer2=0; // 计时用全局变量
unsigned char Timer6=0; // 计时用全局变量
    ... ...
BTCTL  = BT_ADLY_250;   // 将中断周期设为 250ms
    ... ...
#pragma vector = BASICTIMER_VECTOR // BasicTimer 定时器中断（1/4 秒）
__interrupt void BT_ISR(void)      // 声明一个中断服务程序，名为 BT_ISR()
{
      ... ...                      // 这部分程序每 0.25 秒执行一次。
    Timer3++;                      // 0.25 秒累加一次
    if(Timer3>=3) Timer3=0;        // 每累加 3 次清零
    if(Timer3==0)
      {
        ... ...                    // 这部分程序每 0.75 秒执行一次。
      }
```

```
    Timer2++;                          // 0.25 秒累加一次
    if(Timer2>=2) Timer2=0;            // 每累加 2 次清零
    if(Timer2==0)
      {
        ... ...                    // 这部分程序每 0.5 秒执行一次。
      }
    Timer6++;                          // 0.25 秒累加一次
    if(Timer6>=2) Timer6=0;            // 每累加 6 次清零
    if(Timer6==0)
      {
        ... ...                    // 这部分程序每 1.5 秒执行一次。
      }
}
```

2.5 LCD 控制器

液晶（LCD）是最常用的低功耗显示设备。在 MSP430F4xx 系列单片机中，内部集成了 LCD 控制器，能够直接驱动段码液晶。LCD 控制器会产生 LCD 驱动所需的交流波形，并自动完成 LCD 的扫描与刷新。对用户呈现简单的 RAM 接口。程序中只需要写 LCD 显存对应的 RAM 区，即可直接改变 LCD 显示内容。

I LCD 基本知识

段码式 LCD 的工作原理：

LCD 通过改变透射率来实现显示，因其本身不发光，所以功耗极低。段码式液晶最基本的结构如下，主要由偏光片、玻璃板、透明电极、液晶和反光板构成。

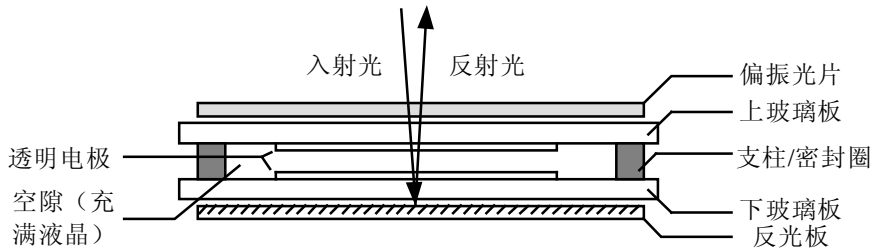


图 2.5.1 LCD 剖面示意图

在两个透明玻璃板上镀有一层极薄的导电膜构成一对透明电极，两电极之间的空隙内充满了液晶。上玻璃板顶部贴有一层偏振光片，下玻璃板底部贴有一层反光板。

当环境光入射 LCD 时，首先穿过偏振光片。偏振光片是一种特殊的光学元件，它只允许偏振方向相同的光完全透射。入射光的偏振方向与偏振光片固有的偏振方向夹角越大，透射率越低。环境光一般都是无偏振方向的，或者说各个方向的偏振光都有。当环境光投射过偏振光片之后，只留下一个方向的偏振光，其余的光线被滤除。

在透明电极不加电压的情况下，液晶所在空隙无电场，液晶分子呈自由状态，对光线没有偏转作用。入射光直接射到反光板后原路返回，又经过偏振光片。由于偏振方向一致，能完全透射出。所以从外界看这个区域是透明的。

当透明电极两端加电压时，液晶所在空隙形成电场，液晶分子受电场影响有规律地排列起来，液晶分子特殊的排列形状对光线有偏转作用。入射光被偏转后射到反光板再沿原路返回，又液晶被偏转的更厉害。最后经过偏振光片时由于偏振方向不一致，不能完全透射。从外界看来，这个区域因为射出光线弱于入射光线，所以看起来变黑了。

改变透明电极之间的电压，将改变电场强度，从而引起偏转程度的改变。最终的效果是透射率改变。驱动电压越高，偏转角越大，透射率越低，看起来就越黑。因此通过改变电极驱动电压可以调节对比度。这也是当电池快耗尽时 LCD 看起来发灰的原因。

当液晶长时间处于单方向电场时，很快会老化，导致寿命急剧下降。因此所有 LCD 的驱动电压都必须是交流波形。根据驱动电压的不同，分为 1/2BIAS、1/3BIAS 等方式。大部分液晶会采用扫描方式驱动以减少驱动所需管脚数目。根据扫描方式的不同，又分为静态、2-MUX、4-MUX 等方式。在目前的标准 LCD 片中，静态扫描方式和 4-MUX（1/3BIAS）方式最常见。

静态驱动方式:

LCD 中每个笔划有 2 个电极。静态方式下，LCD 所有段的其中一极全部连在一起，作为公共端（COM 端）。另一极对外引出，每一个电极对应一段笔划。因此每段笔划的亮灭都需要单独占用一根引脚来控制。

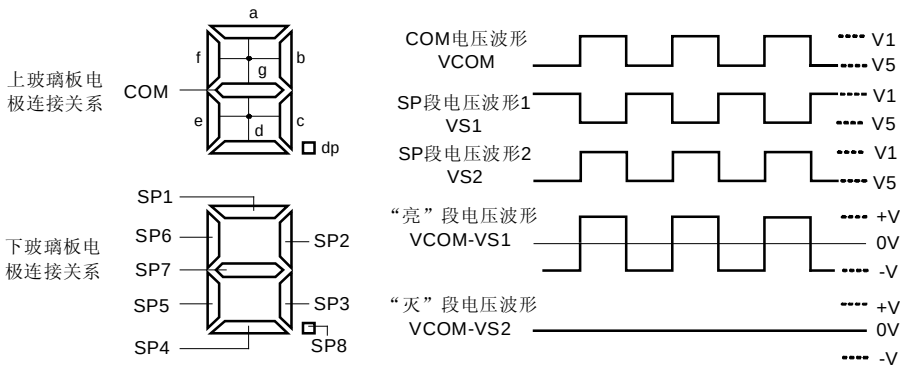


图 2.5.1 静态驱动方式驱动波形示意图

COM 端一直加有交流方波，图中的 V1 一般取 VCC，V5 一般取 GND。若某 SP 脚也加有同频同相的方波（图中 SP 波形 2），这段液晶极板间电压差为 0，不产生电场，因此该段透明（灭）。若某 SP 脚加有与 COM 端同频且反相的方波（图中 SP 波形 1），则该段液晶极板间存在 2 倍电压的方波，该段变黑（亮）。因此，通过切换某段的驱动电压是 SP 波形 1 或是 SP 波形 2 即可控制其亮灭。这种驱动方式硬件最简单，但需要占用大量引脚。每增加 1 位数字（8 段），就要增加 8 只引脚。在 MSP430F42x 系列单片机中，最多 32 只引脚用于段驱动，因此静态方式最多只能驱动 4 位数字显示。

2-MUX 驱动方式:

从上面的分析中可以看到，受到 LCD 控制器的引脚数量限制，静态方式无法在显示位数较多的场合应用。如果能像动态扫描数码管那样，将 LCD 的各段也分时扫描驱动，则能节省大量的引脚。或者同样数目的引脚能够驱动更多段 LCD。假设将一个数字内的 8 段分 2 次扫描完成显示，每个 COM 端只控制 4 段，那么每根段引脚可以控制 2 段。段控制引脚数目可以减少一半。这种方式就叫 2-MUX 驱动方式。

但是 LCD 的分时扫描比数码管要复杂的多，最主要的原因是加在每段上的电压必须是交流对称波形。因此 LCD 驱动波形都是经过精心设计的。以图 2.5.2 为例，在公共电极所在玻璃片上，共有 2 个 COM 端，其中 a、b、c、g 四段公用 COM0 端，f、e、d、dp 四段公用 COM1 端。另一片玻璃片上，每个电极控制 2 段，共有 4 个段电极（SP1~SP4）。

COM0 和 SP1 之间的电压差决定了 a 段的亮灭，COM0 和 SP2 之间的电压差决定了 b 段的亮灭，COM0 和 SP3 之间的电压差决定了 c 段的亮灭，依次类推，每个段都可以通过 COM 端和 SP 端的组合来控制。

液晶不同于数码管的另一特点是液晶分子排序需要一定的电压门限，即使液晶两极板之间加有电压，只要低于电压门限，液晶段将仍然保持透明（灭）状态。这一特点可以用于简化 LCD 驱动波形的设计。

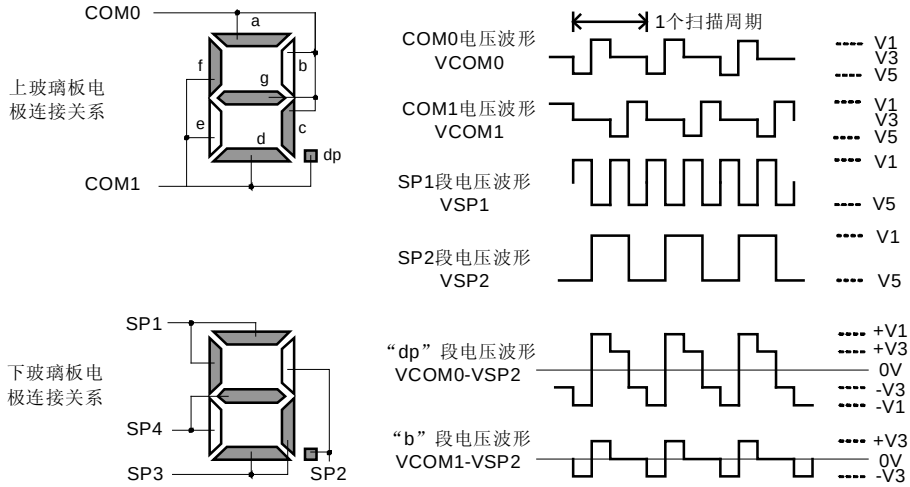


图 2.5.2 2-MUX 驱动方式与驱动波形示意图

上图给出了 MSP430 单片机 LCD 驱动器在 2-MUX 方式下产生的部分驱动波形。图中 V5 一般取 0V，V1 一般取 VCC，V3 一般取 VCC/2。因为用到电源电压的 1/2 作为中间电压，这种驱动波形被称为 1/2 偏压（1/2 BIAS）波形。

COM0 和 COM1 时间上错开了半个周期，可以看做分 2 次扫描。由于每个 SP 电极将要控制 2 个段，两个段有四种组合：全亮、亮灭、灭亮、全灭。因此共需要 4 种 SP 段驱动波形。图中只画出两种，用于示意在 2-MUX 方式下 LCD 的工作原理。小数点 dp 段受 COM0 和 SP2 共同控制：两者电压波形相减，能得到幅值为±V1 的对称交流电压波形，足够驱动该段点亮。对于 b 段，由 COM1 和 SP2 共同控制：两者电压波形相减，得到幅值为±V3 的对称交流电压波形，如果液晶分子排列的扭曲电压门限高于 V3，该段将仍处

于透明（灭）状态。

我们看到，将 SP2 波形加在某根引脚，就可以将它所连接的 2 个段置为一亮一灭状态。读者可以自行推算，将 SP1 波形加在某根引脚上，两个段都将亮。类似的原理，还可以设计出 SP3 和 SP4 波形，SP3 将两个段置为一灭一亮状态；SP4 波形将两个段全灭。选择 SP1~SP4 的其中一种加在一根引脚上，就能同时控制 2 段的状态。

由于每 2 段公用一只引脚，每增加 1 位数字（8 段），就要增加 4 只引脚。在 MSP430F42x 系列单片机中，32 只 LCD 段驱动引脚在 2-MUX 方式最多能驱动 8 位数字显示。

4-MUX 驱动方式:

在 2-MUX 驱动方式中我们已经看到扫描方式能够减少引脚数量。如果再增加扫描步骤，还能减少引脚数量。如果将 2-MUX 方式中的 COM 和 SP 互换，就成了 4-MUX 驱动方式，由于 COM 端变成 4 个，扫描也将分为 4 步。

图 2.5.3 示意了 4-MUX 方式的引脚连接关系。在上玻璃板，每个 COM 端与 2 个笔画段连接；下板每个 SP 端与 4 个笔划段连接。与 2-MUX 方式类似，通过公共端与 SP 端的之间的电压差可以分别控制 8 个笔画段。例如 COM3 和 SP2 之间的电压差决定了 a 段的亮灭、COM2 和 SP2 之间的电压差决定了 b 段的亮灭、COM1 和 SP2 之间的电压差决定了 c 段的亮灭，等等。

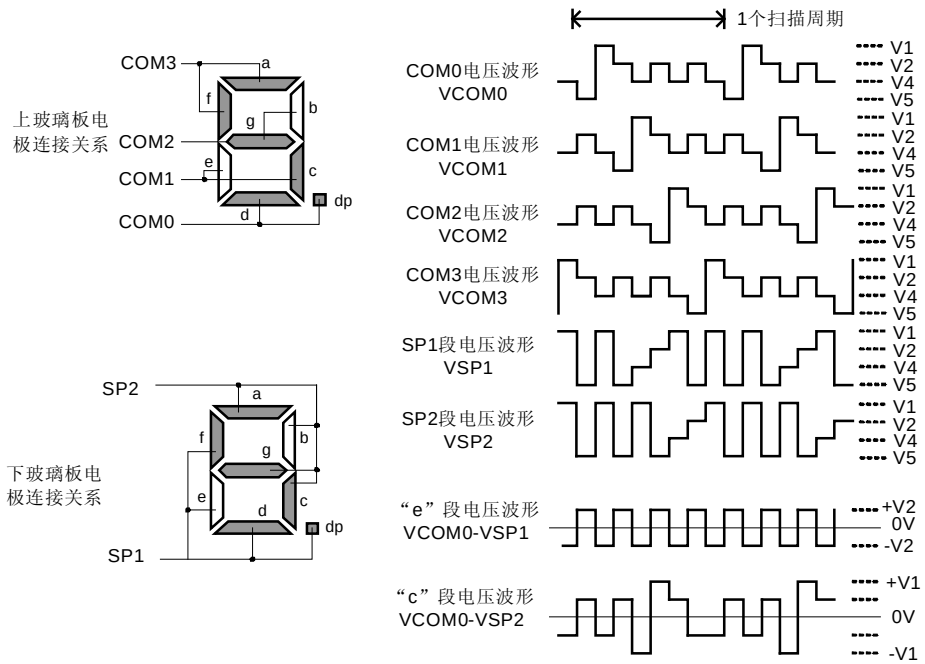


图 2.5.3 4-MUX 驱动方式与驱动波形示意图

由于 4-MUX 方式扫描步骤的增多，驱动波形也复杂得多。图中只画出了 MSP430 单片机 LCD 驱动器在 4-MUX 方式下产生的部分驱动波形。图中 V5 一般取 0V，V1 一般取 VCC，V4 一般取 VCC/3，V2 一般取 2VCC/3。因为用到电源电压的 1/3 和 2/3 作为中间电压，这种驱动波形被称为 1/3 偏压（1/3 BIAS）波形。

COM0~COM3 时间上错开了 1/4 个扫描周期，可以看做分 4 次扫描。由于每个 SP 电极将要控制 4 个段，4 个段亮灭有 16 种组合：0000、0001、0010 …… 1111。因此共需要 16 种 SP 段驱动波形。图中只画出两种，用于示意在 4-MUX 方式下 LCD 的工作原理。C 段受 COM0 和 SP2 共同控制：两者电压波形相减，能得到峰值为±VCC 的对称交流电压波形，足够驱动该段点亮。对于 e 段，由 COM0 和 SP1 共同控制：两者电压波形相减，得到幅值为±VCC/3 的对称交流电压波形，如果液晶分子排列的扭曲电压门限高于 VCC/3，该段将仍处于透明（灭）状态。

类似的原理，还可以设计出 SP1~SP16 波形，分别能将 4 个段置为 0000~1111 状态。选择 SP1~SP16 的其中一种加在一根引脚上，就能同时控制 4 段的状态。

由于每 4 段公用一只引脚，每增加 1 位数字（8 段），只需要增加 2 只引脚。在 MSP430F42x 系列单片机中，32 只 LCD 段驱动引脚在 4-MUX 方式最多能驱动 16 位数字显示，这在大部分应用中已经足够。

因为能够节省引脚，4-MUX 方式的液晶最常用。但 4-MUX 方式的液晶一般都是订制品，4-MUX 的通用液晶较少见。因为和老式设备兼容的原因，市售的通用液晶板一般都是静态驱动的。除此之外还有 3-MUX、8-MUX 等不太常见的驱动方式，原理类似。

Ⅰ LCD 与 MSP430 单片机的连接

在 MSP430F4xx 系列单片机的 LCD 控制器支持静态、2-MUX、3-MUX、4-MUX 四种驱动方式，不同方式下硬件连接关系略有不同。单片机与 LCD 相关的管脚分为下 3 组：

1. COM0~COM3：公共端，与 LCD 的 COM 端相连。
2. S0~S31：段驱动，与 LCD 的 SP 端相连。不同型号的单片机段驱动脚数量可能不同。
3. R03、R13、R23、R33：由外部分压电阻提供 LCD 驱动波形所需偏压 V1~V5。

静态驱动方式 LCD 的连接:

静态方式下，每一位数字需要 8 根段驱动引脚。例如下图中数字 1 由 S0~S7 驱动、数字 2 由 S8~S15 驱动。静态 LCD 只有 1 根公共 COM 端，与单片机的 COM0 相连。

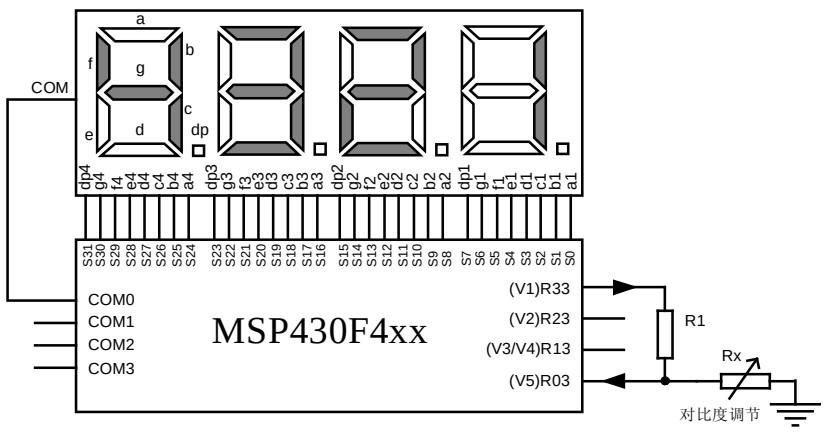


图 2.5.4 静态驱动方式 LCD 与单片机的连接

上一节已经分析过，V1 和 V5 之间的电压差决定了对比度。MSP430 单片机中 R33 管脚输出 V1 电压，通过 R1 和 Rx 分压得到 V5。因此调整电阻 Rx 改变 R1 两端的电压（V1-V5）即可调节对比度。一般 R1 取数兆欧，Rx 取数百千欧。若不需要调节对比度，将 R1 去掉，R03 引脚直接接地。

2-MUX 驱动方式 LCD 的连接:

2-MUX 驱动方式的液晶，由于每两段笔划并联在一起公用一根引脚，例如下图中 a 段和 f 段、b 段和 dp 段、c 段和 d 段、e 段和 g 段各公用一根引脚。因此驱动每一位数字需要 4 根段驱动引脚。图中数字 1 由 S0~S3 驱动、数字 2 由 S4~S7 驱动。2-MUX 方式 LCD 有 2 根公共端 COM0、COM1，与单片机的 COM0、COM1 相连。

除了 V1 和 V5 之外，2-MUX 方式的驱动波形需要中间电压 V3，一般由 2 只等值的电阻 R1、R2 串联分压得到。Rx 同样能改变 V1 与 V5 之间的电压差而调节对比度。R1、R2 一般取数百 K~1M 欧，Rx 取数百 K 左右。 若不需要对比度调节可以将 R03 脚接地。

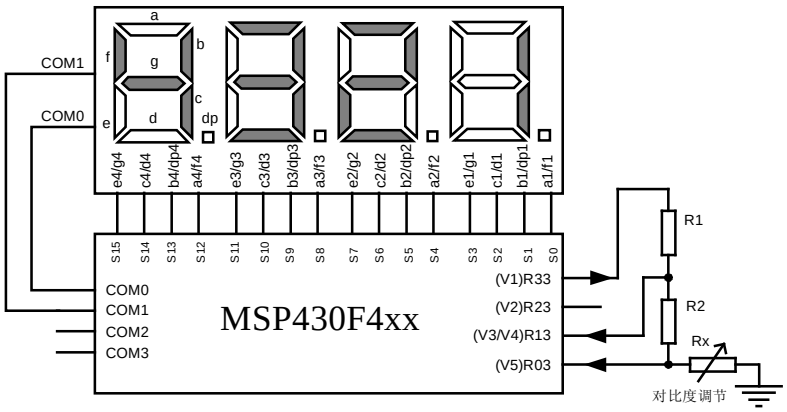


图 2.5.5 2-MUX 驱动方式 LCD 与单片机的连接

4-MUX 驱动方式 LCD 的连接:

4-MUX 驱动方式的液晶，每 4 段笔划并联在一起公用一根引脚。例如下图中 a、b、c、dp 四段公用了一根引脚，d、e、f、g 公用了一根引脚。

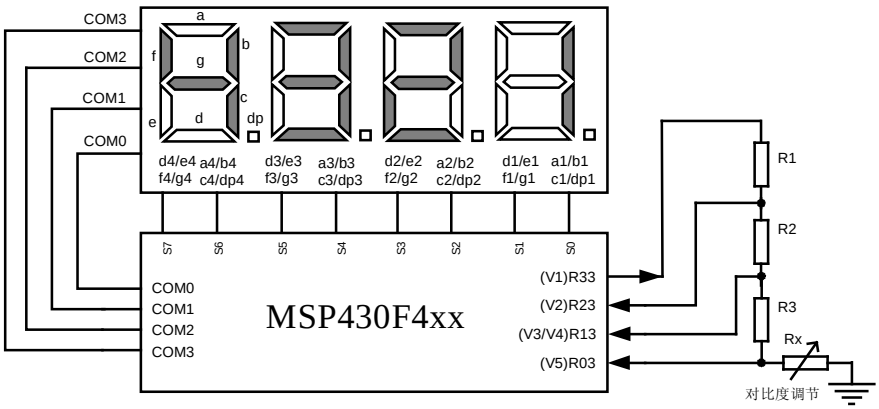


图 2.5.6 4-MUX 驱动方式 LCD 与单片机的连接

所以驱动每一位数字只需要 2 根段驱动引脚。例如图中数字 1 由 S0 与 S1 驱动、数字 2 由 S2 与 S3 驱动。4-MUX 方式 LCD 有 4 根公共端 COM0~COM3，分别与单片机的 COM0~COM3 相连。

除了 V1 和 V5 之外，5-MUX 方式的驱动波形需要两个中间电压 V2 和 V4，一般由 3 只等值的电阻 R1、R2、R3 串联分压得到。Rx 同样能改变 V1 与 V5 之间的电压差而调节对比度。R1~R3 一般取数百 K~1M 欧，Rx 取数百 K 左右。若不需要对比度调节可以将 R03 脚接地。

不同厂家的 LCD，各个 COM 端和 SP 端所公用的笔划可能有所不同；或者布线时调整 COM0~COM3 的连接关系，都不会影响 LCD 的正常显示，只导致的显存中各比特与实际的笔划段对应关系发生变化。利用第一章 1.7 节的宏定义法可以很容易的通过软件调整对应关系。

l LCD 控制器的结构与原理

虽然 LCD 控制时序比较复杂，但 MSP430 单片机内部的 LCD 控制器能通过硬件自动地产生 LCD 驱动所需的全部时序。只需要操作 LCD 控制寄存器即可选择液晶选择驱动模式。一旦模式（如 4-MUX 方式）被选定，产生时序的过程是全自动的，无需软件干预。对软件来说，只需要写显示缓存区即可直接控制 LCD 各段笔划的亮灭。

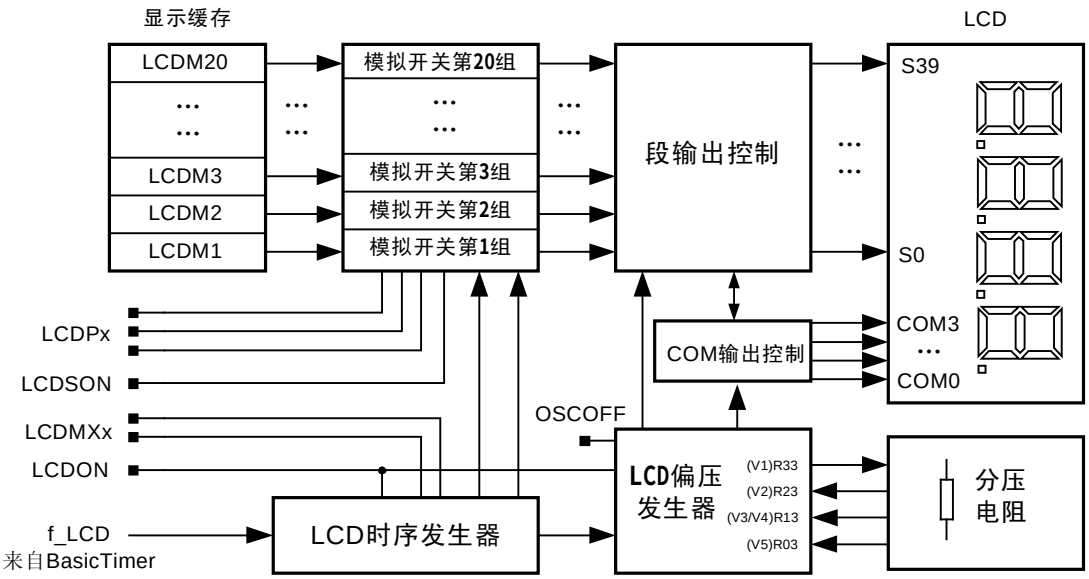


图 2.5.7 LCD 控制器内部结构

图 2.5.7 所示的 LCD 控制器内部结构比较复杂，但大部分都是无需干预的时序逻辑产生电路。用户需要操作的只有简单的 4 个控制位，都位于 LCDCTL 寄存器：

- *LCDPx：引脚功能选择　　(位于 LCDCTL 寄存器)*

  000：所有 Sx 引脚均作为 IO 口

001：S0~S15 管脚作为 LCD 的段驱动，其余作 IO
010：S0~S19 管脚作为 LCD 的段驱动，其余作 IO
011：S0~S23 管脚作为 LCD 的段驱动，其余作 IO
100：S0~S27 管脚作为 LCD 的段驱动，其余作 IO
101：S0~S31 管脚作为 LCD 的段驱动，其余作 IO
110：S0~S35 管脚作为 LCD 的段驱动，其余作 IO
111：S0~S39 全部管脚作为 LCD 的段驱动

快捷宏定义：

LCDSG0：所有 Sx 引脚均作为 IO 口
LCDSG0_1：S0~S15 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_2：S0~S19 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_3：S0~S23 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_4：S0~S27 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_5：S0~S31 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_6：S0~S35 管脚作为 LCD 的段驱动，其余作 IO
LCDSG0_7：S0~S39 全部管脚作为 LCD 的段驱动

由于 LCD 占用了单片机大量引脚，在某些单片机中（如 F41x 系列），LCD 段驱动脚 IO 口复用了同一批引脚。和在某些不需要使用 LCD 的应用场合，可以将这些段驱动引脚设为普通 IO 口使用。或者在 LCD 位数较少的场合，可以将剩余的段驱动引脚作为 IO 使用。若在某些段驱动引脚和 IO 口未复用的单片机上（如 F42X 系列），该控制位将不起作用。

- *LCDMXx： LCD 驱动模式选择 (位于 LCDCTL 寄存器)*

  00：静态驱动模式 01：2-MUX 模式 10：3-MUX 模式 11：4-MUX 模式

  快捷宏定义： LCDSTATIC LCD2MUX LCD3MUX LCD4MUX

  根据实际使用的 LCD 屏的模式，设置该组控制位。

- *LCDON： LCD 驱动器模块总开关 0=关闭 1=开启 (位于 LCDCTL 寄存器)*

  当 LCDON 置 1 后，整个扫描驱动电路、时序发生电路、以及偏压发生器才被开启，LCD 开始工作。将 LCDON 控制位置 0，可以关闭整个 LCD 模块，从而节省电能。

- *LCDSON： LCD 段驱动开关 0=关闭 1=开启 (位于 LCDCTL 寄存器)*

  当 LCDSON 置 1 后，控制段驱动电路的模拟开关组才开始工作。将 LCDSON 控制位置 0，整个屏幕将空白。利用该标志位周期性取反可以做出屏幕闪烁的效果。关闭该控制位虽然也能关闭屏幕，但整个 LCD 驱动电路仍在工作，并不能节省耗电。注意在 LCDMXx 的快捷宏定义中已经包括了将 LCDSON 置 1 操作，避免再加该位造成进位错误。

- *OSCOFF： 低功耗模式 4 的控制位 (位于 SR 寄存器)*

  当单片机进入 LPM4 休眠模式时，OSCOFF 会被置 1。它同时也控制着 LCD 偏压发生器，结果是一旦单片机进入 LPM4 休眠模式，LCD 模块的偏压发生器也被关闭。由于 OSCOFF 置 1 后整个单片机内部没有任何活动时钟，LCD 时序及扫描过程也随之停止，加

上 LCD 偏压被关闭，相当于自动关闭整个 LCD 模块。

例 2.5.1：某 MSP430 单片机系统，接有 4 位数字的静态 LCD，为其配置 LCD 控制器。

```
LCDCTL  = LCDSG0_5 + LCDSTATIC + LCDON; // S0-S31 作为段驱动，静态模式，打开 LCD
```

静态模式下，每位数字需要 8 个段驱动脚，4 位数总共需 32 根，因此 S0~S31 要作为 LCD 段输出使用。利用快捷宏定义 LCDS0_5 来设置。

例 2.5.2：某 MSP430 单片机系统，接有 6 位数字的 4-MUX 模式 LCD，为其配置 LCD 控制器。

```
LCDCTL  = LCDSG0_1 + LCD4MUX + LCDON; // S0-S15 作为段驱动，4-MUX 模式，打开 LCD
```

4-MUX 模式下，每位数字需要 2 个段驱动脚，6 位数总共需 12 根。但 LCD 段驱动控制位的最少的配置是 16 根，因此将 S0~S15 要作为 LCD 段输出使用。利用快捷宏定义 LCDS0_1 进行设置。

例 2.5.3：让 LCD 显示器的显示内容每秒闪烁 2 次，可以在 1/4 秒定时中断内调用：

```
LCDCTL  ^= LCDSON; // 定时将 LCD 段驱动开关取反，造成定时闪烁效果
```

由于液晶属于绝缘体，靠电场而非电流来改变显示状态，理论上没有耗电。但 LCD 在结构上相当于一个电容器（两个电极中间是绝缘的液晶），交流电加在电容两极仍会有充放电电流。所以扫描频率越高，电容充放电越频繁，耗电越大。但如果扫描频率太低，肉眼会看见 LCD 闪烁。LCD 的整个扫描时序的时钟来自于 BasicTimer 的第一级输出。通过 BasicTimer 的设置可以改变 LCD 的扫描刷新频率。实际应用中可以通过实验调整，取人眼看不到闪烁时的最低频率。一般 LCD 控制器加上 LCD 屏电容造成的耗电在 3uA~5uA 左右，屏幕增大、显示位数增多都会导致耗电的增加。

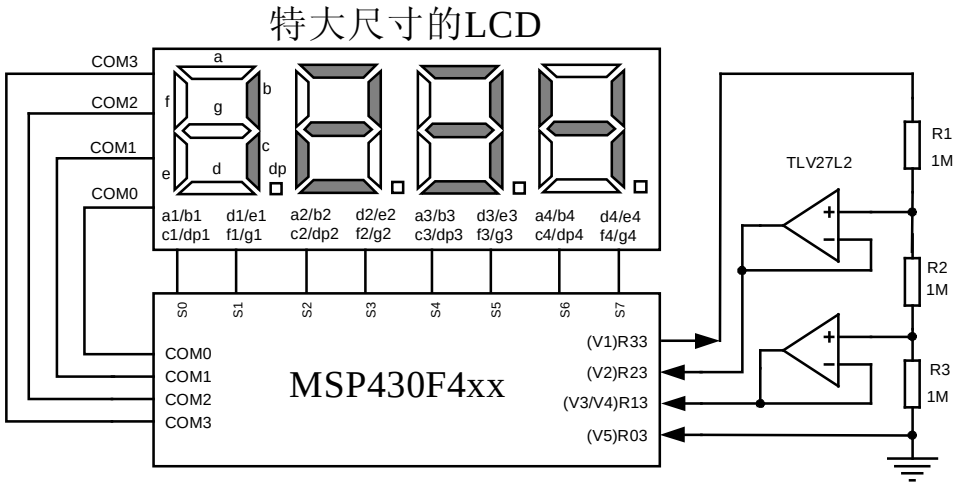


图 2.5.8　用运放为特大尺寸 LCD 提供偏压

液晶的另一个耗电之处在于偏压电阻。由于 LCD 驱动波形需要 VCC/3 或 VCC/2 等中间电压，用电阻分压的方法最容易得到。但是分压电阻本身会带来额外的耗电。为了降低该电阻的功耗，一般将阻值尽可能取大一些。由于极板之间构成电容，若电阻太大可能在扫描周期内不能将电容充满，造成电场强度不够，显示模糊。所以一般 LCD 面积越大，偏压电阻就要越小。一般来说，偏压电阻总阻值取 3M 欧左右，在一般小型屏幕上使用已

经足够。3V 供电时，额外增加约 1uA 功耗，尚可接受。

由于电容量正比于 LCD 面积（尺寸的平方），在驱动特大号的 LCD 屏幕时，可能会出现偏压电阻需要降到数十千欧的情况，这时偏压电阻上的电流将高达上百微安，不能被忽略了。遇到这种情况可以仍使用高阻值电阻做偏置分压，用超低功耗运放（如 TLC27L4）构成的跟随器，将偏置电压变成低阻源后再输入 LCD 控制器。这时运放自身会增加约 30uA 左右的耗电，但节省了百余微安的偏置电阻电流，总体上功耗仍然被降低了。

I LCD 显示缓存的操作

MSP430 单片机的 LCD 控制器提供了最多 20 字节的显存（不同型号数量不一样）用于控制 LCD 显示内容。在不同的驱动模式下，以及不同的硬件连接情况下，都会导致显存字节各比特与 LCD 笔划之间的对用关系发生变化，下面以 MSP430F42x 系列为例，说明显示缓存与显示笔划之间对应关系。MSP430F42x 系列单片机只有 16 字节显示缓存。

静态驱动模式下的显示缓存

从 LCD 结构图中可以看出，每字节显存映射到了相邻 2 根段输出引脚（S0 和 S1、S1 和 S2…）上。由于静态模式下，每个数字需要 8 根段驱动引脚，因此每位数字的显示需要占用 4 字节显存。

我们以图 2.5.4 的硬件连接为例，找出显存与笔画之间的对应关系。先画一个表格：

表 2.5.1 静态驱动模式下显示缓存与笔划对应关系

显存	地址	COM				COM				n	数字位
		3	2	1	0	3	2	1	0		
LCDM16	0A0H				dp4				g4	30	4
LCDM15	09FH				f4				e4	28	
LCDM14	09EH				d4				c4	26	
LCDM13	09DH				b4				a4	24	
LCDM12	09CH				dp3				g3	22	3
LCDM11	09BH				f3				e3	20	
LCDM10	09AH				d3				c3	18	
LCDM9	099H				b3				a3	16	
LCDM8	098H				dp2				g2	14	2
LCDM7	097H				f2				e2	12	
LCDM6	096H				d2				c2	10	
LCDM5	095H				b2				a2	8	
LCDM4	094H				dp1				g1	6	1
LCDM3	093H				f1				e1	4	
LCDM2	092H				d1				c1	2	
LCDM1	091H				b1				a1	0	
段驱动引脚		Sn+1				Sn					

表格中每行代表一字节显示缓存，需要找到各比特分别控制的笔划段填入每一行。每行用 8 个小格表示一字节中 8 比特，高位在最左边。我们知道液晶板靠 COM 和 SP 之间的电压差来驱动的。在图 2.5.4 中只连了 COM0，其余悬空。因此每字节缓存中 COM1~3 所对应的比特位是无用的，共有 6 比特是无内容的，表格中用灰色格子表示。

表格右侧的 n 表示段驱动引脚号，因为每字节映射到了相邻 2 根引脚上，所以每行都是偶数。最下方一行的 Sn 和 Sn+1 表示了具体的引脚。例如在 LCDM2 行，n=2，Sn 和 Sn+1 将是 S2 和 S3，说明 LCDM2 控制着 S3 和 S2 引脚。

下一步在电路图中找到 S3 和 S2 引脚分别控制的段是 d1 和 c1。填入表格中 LCDM2 所在行。类似的方法，可以依次找到所有的笔划对应关系。

例 2.5.4：让图 2.5.4 中的静态 LCD 显示器第三位上显示数字“3”。

数字“3”是由 a、b、c、d、g 段组成。查表 2.5.1，第三位数字由 LCDM9~LCDM12 控制。将这 4 字节中 a、b、c、d、g 段相应比特置 1，其余置 0，得到：

```
LCDM9 = 0x11;          //  a=1  b=1
LCDM10= 0x11;          //  c=1  d=1
LCDM11= 0x00;          //  e=0  f=0
LCDM12= 0x01;          //  g=1  dp=0
```

2-MUX 驱动模式下的显示缓存

在 2-MUX 模式下，每个数字需要 4 根段驱动引脚，因此每位数字的显示需要占用 2 字节显存。我们以图 2.5.5 的硬件连接为例，找出显存与笔画之间的对应关系。

表 2.5.2 2-MUX 驱动模式下显示缓存与笔划对应关系

显存	地址	COM（高位）				COM（低位）				n	数字位
		3	2	1	0	3	2	1	0		
LCDM8	098H			g4	e4			c4	d4	14	4
LCDM7	097H			b4	dp4			a4	f4	12	
LCDM6	096H			g3	e3			c3	d3	10	3
LCDM5	095H			b3	dp3			a3	f3	8	
LCDM4	094H			g2	e2			c2	d2	6	2
LCDM3	093H			b2	dp2			a2	f2	4	
LCDM2	092H			g1	e1			c1	d1	2	1
LCDM1	091H			b1	dp1			a1	f1	0	
段驱动引脚		Sn+1				Sn					

同样的，表格中每行代表一字节显示缓存， 用 8 小格表示 8 比特，高位在左。在图 2.5.5 中只连了 COM0 和 COM1，其余悬空。因此每字节缓存中 COM2 和 COM3 所对应的比特位是无用的，每字节中有 4 比特是无用的，表格中用灰色格子表示。

以 LCDM3 行为例，n=4，Sn 和 Sn+1 将是 S4 和 S5，说明 LCDM3 控制着 S4 和 S5 引脚。下一步在电路图中找到 S4 和 S5 引脚分别控制的段是 a2/f2 和 b2/dp2。再查 LCD 内部

连接关系图（图 2.5.2），找到 COM0 控制着 d、e、f、dp 段；COM1 控制 a、b、c、g 段。

S4 与 COM0 的交集是 f2 段， 填入表格中 LCDM3 所在行，低位 COM0 所在列。

S4 与 COM1 的交集是 a2 段， 填入表格中 LCDM3 所在行，低位 COM1 所在列。

S5 与 COM0 的交集是 dp2 段， 填入表格中 LCDM3 所在行，高位 COM0 所在列。

S5 与 COM1 的交集是 b2 段， 填入表格中 LCDM3 所在行，高位 COM1 所在列。

类似的方法，可以依次找到所有的笔划对应关系。

例 2.5.5：让图 2.5.5 中的 2-MUX 驱动模式的 LCD 显示器第三位上显示数字“3”。

数字“3”是由 a、b、c、d、g 段组成。查表 2.52，第三位数字由 LCDM5 和 LCDM6 控制。将这 2 字节中 a、b、c、d、g 段相应比特置 1，其余置 0，得到：

```
LCDM5 = 0x22;          //  a=1  b=1
LCDM6 = 0x23;          //  c=1  d=1  g=1
```

4-MUX 驱动模式下的显示缓存

在 4-MUX 模式下，每个数字需要 2 根段驱动引脚，因此每位数字的显示占用 1 字节显存。我们以图 2.5.6 的硬件连接为例，找出显存与笔画之间的对应关系。

表 2.5.3 4-MUX 驱动模式下显示缓存与笔划对应关系

显存	地址	COM（高位）				COM（低位）				n	数字位
		3	2	1	0	3	2	1	0		
LCDM4	094H	**f4**	**g4**	**e4**	**d4**	**a4**	**b4**	**c4**	**dp4**	**6**	**4**
LCDM3	093H	**f3**	**g3**	**e3**	**d3**	**a3**	**b3**	**c3**	**dp3**	**4**	**3**
LCDM2	092H	**f2**	**g2**	**e2**	**d2**	**a2**	**b2**	**c2**	**dp2**	**2**	**2**
LCDM1	091H	**f1**	**g1**	**e1**	**d1**	**a1**	**b1**	**c1**	**dp1**	**0**	**1**
段驱动引脚		**Sn+1**				**Sn**					

以 LCDM1 行为例，n=0，Sn 和 Sn+1 将是 S0 和 S1，说明 LCDM1 控制着 S0 和 S1 引脚。在电路图中找到 S0 脚控制 a1、b1、c1、dp1 四段；S1 脚引控制 d1、e1、f1、g1 段。再查 LCD 内部连接关系图（图 2.5.3），找到 COM0 控制着 d1 和 dp1 段、COM1 控制 c1 和 e1 段、COM2 控制 b1 和 g1 段、COM3 控制 a1 和 f1 段。

S0 与 COM0 的交集是 dp1 段， 填入表格中 LCDM1 所在行，COM0（低位）所在列。

S0 与 COM1 的交集是 c1 段， 填入表格中 LCDM1 所在行，COM1（低位）所在列。

S0 与 COM2 的交集是 b1 段， 填入表格中 LCDM1 所在行，COM2（低位）所在列。

S0 与 COM3 的交集是 a1 段， 填入表格中 LCDM1 所在行，COM3（低位）所在列。

S1 与 COM0 的交集是 d1 段， 填入表格中 LCDM1 所在行，COM0（高位）所在列。

S1 与 COM1 的交集是 e1 段， 填入表格中 LCDM1 所在行，COM1（高位）所在列。

S1 与 COM2 的交集是 g1 段， 填入表格中 LCDM1 所在行，COM2（高位）所在列。

S1 与 COM3 的交集是 f1 段， 填入表格中 LCDM1 所在行，COM3（高位）所在列。

例 2.5.6：让图 2.5.6 中的 4-MUX 驱动模式的 LCD 显示器第三位上显示数字“3”。

数字“3”是由 a、b、c、d、g 段组成。查表 2.5.3，第三位数字由 LCDM3 控制。将 LCDM3 字节中 a、b、c、d、g 段相应比特置 1，其余置 0，得到：

```
LCDM3 = 0x5e;          //  a=1  b=1  c=1  d=1  g=1
```

上面用列表格方法找段码关系的过程比较繁琐。在工程实践中，可以用实验的方法直接找到段码对应关系。

例 2.5.7：如在 4-MUX 方式的 LCD 中，有 8 位显示数字，分别受 LCDM1~LCDM8 的控制。用下面的程序可以直接确定显示缓存中各比特位与显示段码之间的关系：

```
LCDM1 = 0x01;          //  第 1 位数字中只点亮 BIT0 所对应的段
LCDM2 = 0x02;          //  第 2 位数字中只点亮 BIT1 所对应的段
LCDM3 = 0x04;          //  第 3 位数字中只点亮 BIT2 所对应的段
LCDM4 = 0x08;          //  第 4 位数字中只点亮 BIT3 所对应的段
LCDM5 = 0x10;          //  第 5 位数字中只点亮 BIT4 所对应的段
LCDM6 = 0x20;          //  第 6 位数字中只点亮 BIT5 所对应的段
LCDM7 = 0x40;          //  第 7 位数字中只点亮 BIT6 所对应的段
LCDM8 = 0x80;          //  第 8 位数字中只点亮 BIT7 所对应的段
```

上述程序在 MAGIC-430 学习板上的运行结果是：

第 1 位数字中亮了 d 段，　第 2 位数字中亮了 g 段，

第 3 位数字中亮了 b 段，　第 4 位数字中亮了 a 段，

第 5 位数字中亮了 dp 段，　第 6 位数字中亮了 e 段，

第 7 位数字中亮了 f 段，　第 8 位数字中亮了 c 段。

根据上述结果直接可以写出 MAGIC-430 板的显存各比特与显示段码的关系：

表 2.5.4 MAGIC-430 学习板上的显示缓存区各比特与笔划对应关系

LCDMx 中各比特位	D7	D6	D5	D4	D3	D2	D1	D0
显示段	**c**	**f**	**e**	**dp**	**a**	**b**	**g**	**d**

实际上，还有更简单的办法是在 EW430 的 Debug 状态下，暂停程序。再打开寄存器窗口，直接改写 LCDMx 的值，无需编写程序。

I　LCD 控制器的应用

对于编程者来说，对 LCD 的一切显示操作最终都通过读写 LCD 显示缓存区来完成。一般来说，段码 LCD 适合显示数值和部分字母。本节简单介绍常见的显示程序及其基本思路。本节中所有范例都针对 4-MUX 方式的 LCD，并假设显示缓存区各比特位与显示段码的关系同表 2.5.4。如果读者希望用于其他 LCD，可以自行移植。

数字的显示

为了显示数字，首先需要一张显示段码表。显示段码表实际上是 ROM 中的一个数组。数组中每个元素都是一个数字的字形。而且实际对应的数字恰好是该元素的下标。例如定义一个 LCD_Tab[]数组，数组中下标为 0 的元素存放着数字“0”的字形，数组中下标为 1 的元素存放着数字“1”的字形，依次类推。

参考第一章 1.7 节的方法，再根据表 2.5.4，可以写出段码表的定义。注意该数组是只读的，应该放在 ROM 里。所以数组声明的时候加了 const 关键字。

```
//-------------------------------------------------------------
```

```
/      *宏定义，数码管 a-g 各段对应的比特，更换硬件只用改动以下 8 行*/
//-------------------------------------------------------------------
#define d      0x01                    //  AAAAA
#define g      0x02                    // F    B
#define b      0x04                    // F    B
#define a      0x08                    //  GGGGG
#define DP     0x10  //小数点的定义     // E    C
#define e      0x20                    // E    C
#define f      0x40                    //  DDDDD   DP
#define c      0x80
#define NEG    0x02      //负号的宏定义，同 g 段
//-------------------------------------------------------------------
/                *用宏定义自动生成段码表，请勿修改*/
//-------------------------------------------------------------------
const char LCD_Tab[] =                 //LCD 段码表，放在 ROM 中
{
  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"
};
#undef a                          //清除宏定义，以免和变量名冲突
#undef b
#undef c
#undef d
#undef e
#undef f
#undef g
```

若需要显示数字“5”，查找显示表中下标为 5 的元素即可：

```
LCDM1 = LCD_Tab[5];                    //在第一位显示“5”
```

由于显示缓存 LCDMx 在 RAM 中是连续的，LCDM1 处于最低位。如果用一个指针指向 LCDM1 的地址，就可以通过指针或下标来操作显示缓存中任意单元的内容。下例中就是用 pLCD 指向 LCDM1 的地址，然后把 pLCD 当作数组操作，实际上就是操作 LCDM1 之后的显存单元。

例 2.5.7：编写在任意位置显示数字的函数：

```
/*********************************************************************
* 名    称：LCD_DisplayDigit()
* 功    能：在 LCD 上任意位置显示一个数字。
* 入口参数：Digit:      待显示数字    （0~9）
```

```
          Location:   显示位置      从左至右对应 76543210
* 出口参数：无
* 说    明：调用该函数不影响 LCD 其他位的显示。
* 范    例:     LCD_DisplayDigit(3,0); //在第一位（右侧最低位）显示 3
               LCD_DisplayDigit(2,1); //在第二位显示 2
               LCD_DisplayDigit(1,2); //在第三位显示 1  ---> 显示结果: 123
*********************************************************************/
  void LCD_DisplayDigit(char Digit,char Location)
  {  char  DigitSeg;              // 存放字形笔划的变量
     char *pLCD;                  // 存放 LCD 显存指针的变量
     DigitSeg = LCD_Tab[Digit];   // 得到待显示数字的字形笔划
     pLCD= (char *)&LCDM1;        // 获得 LCDM1 的地址
     pLCD[Location]= DigitSeg;    // 在 LCDM1 之后 Location 个单元显示出数字
  }
```

显示数值

为了显示数值，首先要将数值拆分成独立的数字，然后再将数字逐位显示出来。对于十进制数，拆分数字最简单的方式是不断除 10 并取余数。例如需要将 1234 拆分成 1、2、3、4，第一步可以对 10 取余，得到 4；然后将 1234 除 10 取整，得到 123，再将 123 对 10 取余，得到 3……。依次类推可以得到所有的数字。

例 2.5.8：编写在 LCD 上显示正整数的函数：

```
/*********************************************************************
* 名    称：LCD_DisplayNumber()
* 功    能：在 LCD 上显示一个正整数。
* 入口参数：Number:     待显示数字    （0~65535）
* 出口参数：无
*********************************************************************/
  void LCD_DisplayNumber(unsigned int Number)
  {  char  DigitSeg;                  // 存放字形笔划的变量
     char  DispBuff[5];               // 存放数字拆分结果的数组
     char i;                          // 循环变量
     for(i=0;i<5;i++)                 // 65535 最多 5 位数
       {
          DispBuff[i] = Number%10;    // 拆分数字，取余操作
          Number/=10;                 // 拆分数字，除 10 操作
       }
     for(i=0;i<5;i++)
       {
          LCD_DisplayDigit(DispBuff[i],i);   //依次显示拆分后的各位数字
       }
  }
```

该函数能够显示 0~65535 的数字。但对于有效数字前 0 没有作消隐。假设显示数字是 123，最终显示结果将是 00123。为了显示美观并符合数学习惯，一般都要将第一位有效数字之前的 0 消隐。消隐无效零的过程可以从拆分后数字的最高位开始向低位搜索，遇到 0 则用某个特殊标志替换掉，直到第一位有效数字为止。由于显示段码表最多只有 10 个元

素，取 255 作为消隐特殊标志不会影响正常的数据。在显示数字的过程中，若遇到消隐标志则显示空白，否则显示正常的数字。就实现了消隐功能。

例 2.5.9：编写带有无效零消隐的显示正整数函数：

```
/*********************************************************************
* 名    称：LCD_DisplayNumber()
* 功    能：在 LCD 上显示一个正整数，并消隐有效数字前的“0”。
* 入口参数：Number:     待显示数字     （0~65535）
* 出口参数：无
*********************************************************************/
  void LCD_DisplayNumber(unsigned int Number)
  {  char  DigitSeg;                  // 存放字形笔划的变量
     char *pLCD;                      // 存放 LCD 显存指针的变量
     char  DispBuff[5];               // 存放数字拆分结果的数组
     char i;                          // 循环变量
     pLCD= (char *)&LCDM1;            // 获得 LCDM1 的地址
     for(i=0;i<5;i++)                 // 65535 最多 5 位数
       {
          DispBuff[i] = Number%10;    // 拆分数字，取余操作
          Number/=10;                 // 拆分数字，除 10 操作
       }
     for(i=5;i>0;i--)                          // 从最高位开始消隐无效 0
       {
          if(DispBuff[i]==0)  DispBuff[i]=255;//从最高位开始，遇到 0 则替换成 255
          eles                break;          //直到第一个有效数字为止，停止替换
       }
     for(i=0;i<5;i++)                      //依次显示拆分并消隐 0 后的各位数字
       {
          if(DispBuff[i]==255)  pLCD[i]=0; //若被消隐，则清除该位（各段均为 0）
          else  LCD_DisplayDigit(DispBuff[i],i); //否则依次显示拆分后的各位数字
       }
  }
```

显示小数

在第一章 1.3 节介绍了最常用的小数表示方法就是将数字扩大 10^N 倍。然后显示的时候再人为添加小数点，使小数点后还有 N 位数。

例 2.5.10：编写显示小数的函数：

```
/*********************************************************************************
* 名    称：LCD_DisplayDecimal()
* 功    能：在 LCD 上显示一个带有小数点的整数。
* 入口参数：Number:显示数值  (0~65535)
           DOT   :小数点位数(0~4)
* 出口参数：无
* 说    明：不支持纯小数
* 范    例：LCD_DisplayDecimal( 12345,2); 显示结果：123.45 (2 位小数)
*********************************************************************************/
```

```
void LCD_DisplayDecimal( unsigned int Number, char DOT)
{
   char *pLCD = (char *)&LCDM1;    // 获取 LCDM1 的地址
   LCD_DisplayNumber(Number);      // 显示整数
   pLCD[DOT] |= DP;                // 添加小数点。(DP 是段码表中小数点位的宏定义)
}
```

以第一章 1.3 节的温度计算为例，为了保留 1 位小数的精度，计算时将温度结果 Deg_C 的数值人为地扩大了 10 倍。例如 234 表示 23.4 度。在显示的时候需要人为添加 1 位小数，因此调用 `LCD_DisplayDecimal(Deg_C,1)`，程序中将小数点添在了第二位上，因此显示结果变成了 23.4。

需要注意的是，对于纯小数，在消隐无效零的时候，只能消隐到到小数点前一位为止。上述程序在显示 0.012 时将显示出“ . 12”，不符合习惯。可以在消隐过程中添加对小数点位置的判断语句完成。参考光盘带的完整的显示程序库。

显示负数

对于负数，可以取绝对值变成正数，这样在显示时可以直接调用显示正数的函数。负数在显示形式上比正数只多一个负号。如果传入数据是负数，可以在最高位之前人为的添加负号，完成负数的显示。

例 2.5.11：编写能显示正数和负数以及小数的函数：

```
/*************************************************************************
* 名    称：LCD_DisplayDecimal()
* 功    能：在 LCD 上显示一个带有小数点的整数，支持负数。
* 入口参数：Number:显示数值  (-32768~32767)
           DOT   :小数点位数(0~4)
* 出口参数：无
* 范    例：  LCD_DisplayDecimal( 12345,2); 显示结果: 123.45 (2 位小数)
             LCD_DisplayDecimal(-12345,2); 显示结果:-123.45 (2 位小数)
*************************************************************************/
void LCD_DisplayDecimal(int Number, char DOT)
{
   char *pLCD = (char *)&LCDM1;    // 获取 LCDM1 的地址
   char Negative=0;
   if(Number<0)                    // 判断待显示数据是否为负数
     {
       Number=-Number              // 如果负数则取反变成正数
       Negative=1;                 // 并置标志
     }
   LCD_DisplayNumber(Number);      // 显示正整数
   pLCD[DOT] |= DP;                // 添加小数点。(DP 是段码表中小数点位的宏定义)
   if(Negative)  pLCD[5]= NEG;     // 如果是负数，在最高位前添加负号（NEG 是宏定义）
   else          pLCD[5]= 0;       // 如果是正数，清除负号位
}
```

显示浮点数

浮点数的运算不需要对阶，一般无需考虑溢出，所以使用非常方便。但是浮点数的显示相对困难一些，因为拆分数字和确定小数点位都比较复杂。一般来说有三种方法：一是根据浮点数的二进制表示方法，从二进制数据上解析并获得小数点、数值等信息。这种方法效率高，但是很可能不同的编译器浮点数表示方法不一样，造成不兼容。第二种方法是将浮点数转换成定点数后再拆分数字，效率较低但不存在兼容性问题。第三种方法是利用 C 语言提供的库函数来进行数据到字符的转换，简单快捷但效率最低。我们先以第二种方法为例。第三种方法在下一章中会看到：printf 函数对这类数值格式化输出问题的强大处理能力。

例 2.5.12：编写能显示浮点数的显示函数：

```
/*************************************************************************
* 名    称：LCD_DisplayFloat()
* 功    能：在 LCD 上显示一个浮点数，并能指定保留小数位数。
* 入口参数：Number:显示数值  (浮点数)
           DOT   :保留小数点位数(0~4)
* 出口参数：无
* 范    例：  LCD_DisplayFloat(12.345,2); 显示结果: 12.34 (2 位小数)
             LCD_DisplayFloat(12.34,3); 显示结果:12.340 (3 位小数)
*************************************************************************/
void LCD_DisplayFloat(float Number, char DOT)
{
   for(i=0;i<DOT;i++) Number*=10;             // 人为的扩大 10exp(DOT) 倍
   LCD_DisplayDecimal(int Number, char DOT);  // 作为定点小数显示
}
```

英文字母的显示

对于段码式液晶来说，受到笔划和字形的限制，只能显示出部分的字母。字母与数字一样，也是各段的组合。我们可以在段码表 LCD_Tab[]中 0~9 的后面继续添加英文字母字形的定义：

```
const char LCD_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"   //数字
  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"   //部分英文字母
}
```

之后可以用显示数字的方法显示字母。例如调用显示数字的函数：

```
    LCD_DisplayDigit(10,0); //在第一位（右侧最低位）显示表中第 10 个元素（“A”）
    LCD_DisplayDigit(11,1); //在第二位显示显示表中第 11 个元素（“B”）
    LCD_DisplayDigit(12,2); //在第三位显示显示表中第 12 个元素（“C”） à “CBA”
```

最终在 LCD 上的显示结果是“ CBA”。这种用大于 10 的数字表示字母的方法不直观。为了使用更加方便，可以用宏定义对字母进行定义，用便于记忆的符号来表示字母在段码表中的位置。最容易记忆的方法是用相应的英文字母，但是单个英文字母容易和变量重名，如 i,s,p,t 等。若将字母重复一遍，重名概率就小了：

```
  #define AA 10
  #define BB 11
  #define CC 12
  #define DD 13
  #define EE 14
  #define FF 15
```

当然，读者也可以根据自己的习惯，用其他方法来表示。借助宏定义，上例显示 ABC 的程序可以更加直观地写成：

```
   LCD_DisplayDigit(AA,0); //在第一位（右侧最低位）显示表中第 10 个元素（“A”）
   LCD_DisplayDigit(BB,1); //在第二位显示显示表中第 11 个元素（“B”）
   LCD_DisplayDigit(CC,2); //在第三位显示显示表中第 12 个元素（“C”） à “CBA”
```

在段码液晶上，由于只有 7 段笔划，无法显示所以的英文字母。图 2.5.9 给出了段码式显示设备上所能显示的所有英文字母。其中某些英文字母已经严重变形（如 K、M、W 等）、某些字母和数字重复（S、Z、O）。读者可以根据上图中的字母字形，写出完整的段码表。

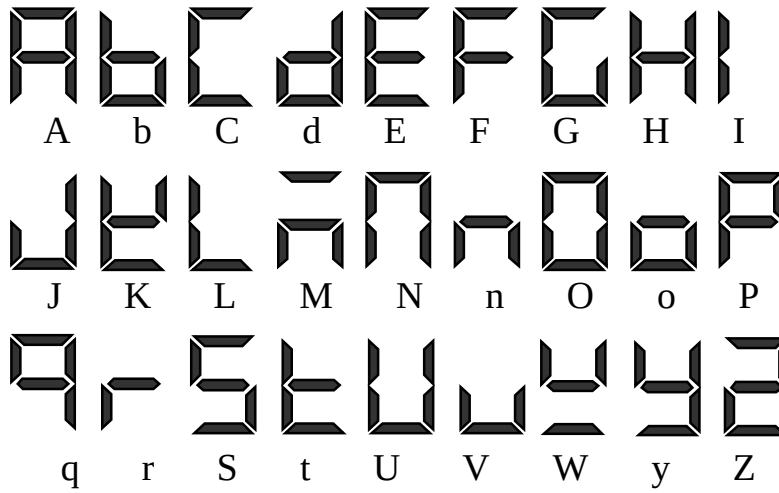


图 2.5.9 段码式液晶显示的英文字母

本节中给出的显示程序范例仅用于说明原理和设计思路，有很多细节问题没有考虑。比如数据溢出、小数点在最后一位时不应该被显示出来、纯小数的显示与零消隐程序的冲突、负号位置应该随显示数值位数而浮动、超过 65535 的数值显示、如何在数字后面显示单位……。

光盘里附带了完整的显示程序库，内部的各种显示函数考虑了上述所有的细节问题，在开发中可以直接使用。

2．6 存储器与 Flash 控制器

一般来说，在单片机中 Flash 存储器都用于存放代码，属于只读型存储器。但在全系列 MSP430 单片机上，都可以通过内置的 Flash 控制器，擦除或改写内部任何一段 Flash 的内容。此外，MSP430 单片机内部还专门留有一段 Flash 区域（InfoFlash）用于存放需要掉电后永久保存的数据。利用 Flash 控制器，可以实现较大容量的数据记录、用户设置参数在掉电后的保存、在线更新程序等功能。

Ⅰ MSP430 单片机的存储器组织结构

因为操作 Flash 存储器需要涉及存储器的物理地址，所以在使用之前，需要先了解 MSP430 单片机的存储器组织结构。MSP430 单片机的存储器组织结构采用冯诺依曼结构，因此 RAM 和 ROM 都统一编址在同一寻址空间内，没有代码空间和数据空间之分。实际上 MSP430 单片机具有 20 位地址总线，总寻址空间能够达到 1MB，但只在某些高端的型号上（如 FG461x 系列）才突破 64K。其余绝大部分的型号都具有如图 2.6.1 所示的 64KB 存储器空间组织结构。

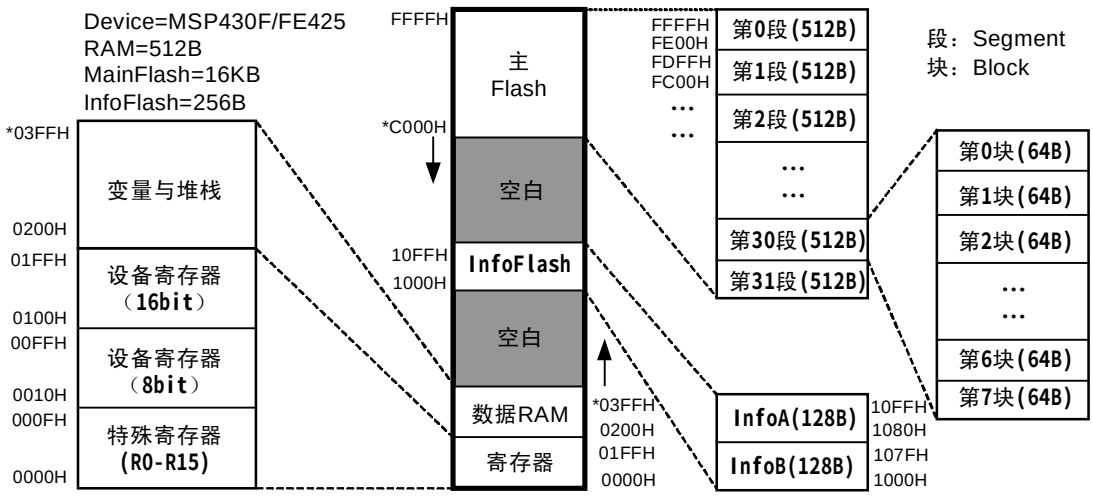


图 2.6.1 MSP430 单片机存储器组织结构

总的来看，MSP430 单片机内部的存储空间划分为三个区。位于存储器最低端的是数

据区，包括寄存器与数据变量、堆栈等，都属于 RAM。最高端是主 Flash 存储器，是存放程序代码的空间。中间还有一个 Flash 区，叫做信息 Flash(InfoFlash)，可以用作掉电后保存少量数据用。三个区域的中间有两段空白区域，读写该区域没有任何效果。MSP430 家族中不同型号的单片机，ROM 与 RAM 的容量有所不同，所以空白区域的大小也不同。

数据区

最低 16 个存储单元是寄存器区，对应着 R0~R15 共 16 个寄存器。其中 R0~R3 作为特殊寄存器：R0 是 PC（程序指针）、R1 是 SP（堆栈指针）、R2 是 SR 寄存器（前文中已出现）、R3 是常数发生器。R4~R15 是通用寄存器。MSP430 单片机中不再有累加器的概念，因为汇编指令中的源和目的数可以是任何一个寄存器。所以任何一个寄存器都可以做累加器，不再有累加器瓶颈问题。事实上，C 语言编译器已经帮我们屏蔽掉了 CPU 的细节和特征，关于汇编的指令、寄存器操作、寻址、跳转、调用、加减运算等操作对于 C 语言开发者来说是不可见的，编译器会自动选择最合适的指令来完成。当然，对于需要发挥 CPU 极限性能的设计来说，最后还是要通过汇编来优化和调整。

从第 17 单元（0010H）到第 255 单元（00FF）共 240 字节，以及从 0100H 到 01FFH 单元共 256 字节，是各种内部功能模块的寄存器。其中 8 位的寄存器位于 0100 以下的空间，16 位的位于 0100H 以上的空间。MSP430 单片机家族中不同的型号内部资源虽然不同，但即使在不同型号单片机上，同一模块的各种寄存器所占据的地址是相同的。这种模块化的结构使得 MSP430 家族在不断壮大的同时保持着很好的兼容性。对于 C 语言来说，内部设备的各种寄存器的地址在单片机头文件中已经被宏定义。对开发者来说只需要知道寄存器名称即可像变量一样使用。

从 0200H 开始是 RAM 区。由于不同型号的单片机 RAM 大小有所不同，致使 RAM 区的结束地址也不同。例如 MSP430F425 具有 512 字节的 RAM，则从 0200H~03FF 都是 RAM 区，以上的地址为空白区。我们看到空白区末尾是 0FFF，因此目前 MSP430 系列单片机的 RAM 最大不超过 4K（将来通过多个阵列会扩展到 128KB）。对于 C 语言编译器来说，会自动将各种变量、中间结果、堆栈存放在 RAM 区域。对程序员来说无需关心细节，编译结束后在信息窗提示的编译结果会给出 RAM 使用量的大小，只要不超过 RAM 区实际容量并稍留余量给堆栈用即可。

主 Flash 区

关于 Flash 存储器，先阐释几个基本概念。首先，Flash 的结构决定了写操作只能将存储单元中的各比特位从 1 改写成 0，不能将 0 改写成 1。所以 Flash 中每个单元可以一次性写入数据，数据一旦写入，在擦除前不能被再次改写。若两次写入同一单元，该单元的内容将是两次写入数据相或的结果，造成错误。Flash 可以被擦除，擦除后所有单元的比特位都恢复为 1，但擦除操作只能针对整个段进行。所以在改写某单元之前，必须先擦除整个段。而为了保留该段中其他的数据，擦除前需要备份整个段，所以单字节改写等随机存储操作效率很低。Flash 存储器较适合做大批量连续数据存储，而且一般控制器都会提供连续写功能以提高速度。

在 Flash 中，将每次能擦除的最小区块单位成为“段”（Segment），将每次能连续写入的最大区块单位称为“块”（Block）。

主 Flash 一般用于存放程序代码，通过 FET-Debugger 调试的时候，程序就是被下载到主 Flash 空间内。主 Flash 从最高地址向下排列，每个段的大小为 512 字节。例如 MSP430F425 单片机有 16KB 的程序空间，那么主 Flash 区将有 32 个段（图中段 0~段 31），它们占用着 C000H~FFFFH 存储单元，C000H 地址以下是空白区。不同单片机的 ROM 空间大小不同，因此主 Flash 的起始地址会有所不同。由于空白区起始地址为 1100H，因此 MSP430 系列单片机中最大的 ROM 空间是 59.75KB（将来通过多个存储阵列会扩展到 1MB）。

在主 Flash 中，第 0 段（FE00~FFFF）比较特殊，因为它存有中断向量表。在第 0 段的最高地址向下，每 2 字节存放着一个设备的中断向量（中断程序入口地址）。据体中断向量的数目因单片机而异。只有复位向量（位于 FFFE~FFFF）、不可屏蔽中断向量（位于 FFFC~FFFD）、看门狗中断向量（位于 FFF4~FFF5）是固定不变的，其余与单片机型号有关。实际上每个单片机的头文件中都对中断向量做了定义，编程者无需关心其实际地址。MSP430 将复位也作为一个中断来处理，复位向量指向了程序起始入口。MSP430 单片机程序都从低地址向高地址存放，因此复位向量一般指向主 Flash 的起始地址。例如 MSP430F425 单片机中，复位向量（FFFE~FFFF 单元）应被写为 C000H。中断向量表的操作是编译器自动完成的，所以在编译之前一定要指定芯片型号，若芯片型号不对，编译出来的代码可能会被安排在空白区，导致无法执行。

信息 Flash 区

上面已经分析过，对于单字节改写操作，需要备份该字节所处的整个 Flash 段。因此 Flash 段越大，改写操作的速度越慢。Flash 的另一特点是无论段大小，擦除操作所需时间是相同的。所以若将段分得很小，擦除速度会变慢。对于主 Flash 来说，一般只在下载、烧写过程需要连续写 Flash 操作，所以分段较大。但若需要改写其内容，将会很慢，且需要较大 RAM 空间用于保存整个段的其余数据。

为了解单字节改写决效率问题，MSP430 单片机中专门开辟了两个较小的段：InfoA 段（1000H~107FH）和 InfoB 段（1080~10FFH）。每个段只有 128 字节。特别适合保存菜单设置的参数等少量需要掉电保存的数据。改写数据时，即使在擦除前备份全部内容也只需要 128B 的 RAM 空间，大部分单片机都能提供。

除了每段的字节数比主 Flash 少之外，InfoFlash 与主 Flash 没有任何区别。换句话说，主 Flash 也能用于保存数据，且容量比 InfoFlash 大得多。但它与程序公用，要注意数据一定不能占用程序的空间。并且第 0 段是不能用与保存数据的，因为它存有中断向量表，一旦擦除程序就无法工作。

例如，对于 MSP430F425 单片机的程序，编译结果提示程序占用 4KB 空间，则主 Flash 还剩下 12KB 空间可以作数据存储用。扣除中断向量表所在的第 0 段，第 1 段~第 23 段都可以存放数据。

Ⅰ Flash 控制器结构与原理

Flash 存储器的读操作与 RAM 相同，但擦除操作与写操作都需要比较复杂的时序，还需要专门的编程电压发生器的配合，所以无法像 RAM 一样直接写。在 MSP430 单片机内，集成有 Flash 控制器，能够产生 Flash 写操作所需的时序以及编程电压，从而为用户提供一

定的读、写、擦除 Flash 的手段。

Flash 控制器的结构如图 2.6.2 所示，由 Flsah 时钟发生器产生 Flash 读写操作所需的时钟；编程电压发生器产生 Flash 写、擦除操作所需的较高电压；Flash 逻辑控制单元负责产生编程、擦除等操作时序，并通过总线将数据写入 Flash 或擦除 Flash。用户只需通过 3 个寄存器（FCTL1~FCTL3）即可对 Flsah 进行设置、写、擦除等操作。其中 FCTL1 是 Flash 操作指令寄存器，FCTL2 用于设置 Flash 模块的时钟，FCTL3 寄存器里包含了 Flash 状态、紧急退出处理等标志位。

在对 Flash 进行写或擦除操作之前，必须先设置 Flash 控制器的时钟源。对于 MSP430F4xx 系列单片机来说，在 Flash 的操作时，其时钟频率范围必须在 257KHz~476KH 之间，不同的单片机型号可能略有不同，但大致都在 250KHz~470KHz 左右的范围。

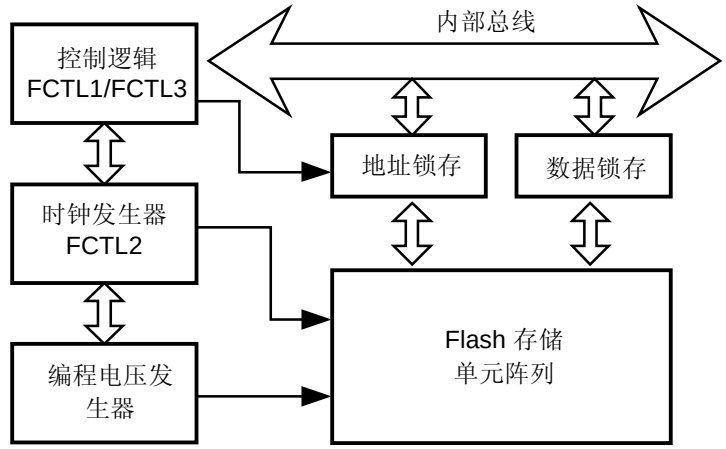


图 2.6.2 MSP430 单片机 Flash 控制器结构

Flash 存储单元的物理结构是一个栅极悬浮的 MOS 管，靠栅极与基底之间构成的电容上的电荷来实现记忆。在室温环境下，充入浮栅电容的电荷能保持约 100 年时间，但每次充放电（写-擦除）都会降低电容介质的绝缘性从而减少 Flash 的寿命。MSP430 单片机的 Flash 存储器理论上有 10 万次的擦写寿命。

若 Flash 时钟频率过高，将导致写入时序过短，注入 Flash 浮栅的电荷不足，可能导致写入失败，或者电荷维持时间达不到 100 年。若 Flash 时钟频率过低，对浮栅电容充电时间过长，会减少擦写寿命。

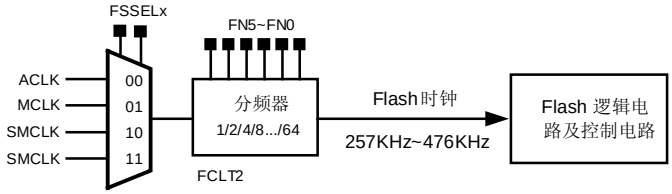


图 2.6.3 Flash 控制器内部时钟发生器结构

所以，在操作 Flash 存储器之前，都要设置正确的时钟频率。Flash 的时钟频率由 FCTL2 寄存器设置，结构如 2.6.3 所示。选择一个时钟源并进行 1~64 分频得到符合 Flash 控制器所需频率范围的时钟。

- *FSSELx：Flash 控制器时钟源选择 (位于 FCTL2 寄存器)*
  00=ACLK 01=MCLK 10=SMCLK 11=SMCLK
  快捷宏定义：FSSEL_0 FSSEL_1 FSSEL_2 FSSEL_3

- *FNx： 分频系数 (位于 FCTL2 寄存器)*
  分频系数 = FN5×32 + FN4×16 + FN3×8 + FN2×4 + FN1×2 + FN0×1 +1

- *FWKEYx：密码位 必须设为 0xA5，其余标志位才能写入 FCTL2 寄存器*
  快捷宏定义： FWKEY(0xA5) FRKEY(0x96)

由于 Flash 的擦除、写入等操作均属于不可恢复性操作，一旦误操作将永远丢失数据，甚至将程序本身毁坏。为了防止万一程序错乱（被干扰导致死机、误动作等非正常情况）时误操作 Flash 存储器，与 Flash 相关的三个寄存器都采用了密码核对机制。FCTL1~FCTL3 都是 16 位寄存器，所有的控制位都位于低 8 位，高 8 位用于核对密码。当高 8 位被写入 0xA5 时，低 8 位才能被更改。否则将认为程序非正常执行，会立即产生复位信号，将单片机复位。注意若读高 8 位密码，读回值是 0x96。

例 2.6.1：某 MSP430 单片机系统，ACLK=32.768KHz，MCLK=SMCLK=1.048MHz。为 Flash 控制器设置时钟。

Flash 控制器要求 250~470KHz 范围内的时钟源，显然 ACLK 过低，不能满足要求。而 MCLK 和 SMCLK 高于时钟范围，通过合适的分频系数可以得到符合要求范围的时钟。当对 MCLK 或 SMCLK 进行 3 分频后，能得到 349KHz 的 Flash 时钟，恰好处于最佳频率。

```
FCTL2 = FWKEY + FSSEL_2 + FN1;              //   SMCLK/3 =349KHz
```

例 2.6.2：某 MSP430 单片机系统，ACLK=32.768KHz，MCLK=SMCLK=7.62MHz。为 Flash 控制器设置时钟。

对 MCLK 或 SMCLK 进行 22 分频后，能得到 346KHz 的 Flash 时钟，处于最佳频率。由于 22=16+4+1+1，因此将 FN4、FN2、FN1 置 1。

```
FCTL2 = FWKEY + FSSEL_2 + FN4 + FN2 + FN1;  //   SMCLK/22 =346KHz
```

对 Flash 的工作模式设定通过 FCTL1 寄存器进行，相关的控制位有：

- *BLKWRT：“批量写”控制位 1=按块批量写入 0=正常写入 (位于FCTL1 寄存器)*

- *WRT：“写”控制位 1=Flash 进入“写”状态 0=退出写状态 (位于FCTL1 寄存器)*

- *MERAS：“批量擦除”控制位 1=全部擦除 0=单段擦除 (位于FCTL1 寄存器)*

- *ERASE：“擦除”控制位 1=进入擦除状态 0=退出擦除状态 (位于FCTL1 寄存器)*

可以看出 FCTL1 寄存器负责 Flash 的擦除以及写入操作。其中擦除操作有 3 种模式：单段擦除、全部擦除、仅擦除主 Flash。通过 MERAS 与 ERASE 两位的组合来选择：

MERAS	ERASE	擦除模式
0	0	退出擦除模式（复位后默认状态）
0	1	单段擦除
1	0	全部擦除主 Flash 区，保留 Info 区内容
1	1	全部擦除所有 Flash 内容

由于程序驻留在主 Flash 区，在全部擦除模式下执行擦除操作将会删除程序自身，因此除了在线升级程序等应用以外很少被使用。单段擦除模式是最常用的擦除方法。

MSP430 单片机的 Flash 控制器也提供了两种写入方式：正常写入和批量写入。在正常写入模式下，每次只能写入一字节（Byte，8bit）或一个字（Word，16bit）。由于每次写入过程都要等待 Flash 存储单元的电荷注入完毕才能进行下一次写操作，正常写模式下连续写入数据速度较慢。所以控制器还提供批量写入功能，以解决连续写入数据时的效率问题。在批量写入模式下，写入以块（64 字节）为单位，每次必须写入 64 字节的整倍数。为了在写入前暂存这些数据块，需要一定的 RAM 开销。

BLKWRT	WRT	写入模式
X	0	退出写模式（复位后默认状态）
0	1	正常写模式
1	1	批量写模式

为了保证 Flash 存储器的安全，除了 FCTLx 寄存器写入需要固定的密码之外，Flash 存储区本身还具有锁定控制位，当锁定位开启时，Flash 处于只读状态，不响应任何擦除或写入操作。锁定位以及其他一些状态标志位都处于 FCTL3 寄存器，其中常用的有：

- *LOCK：锁定位　1=Flash 被锁定　0=Flash 解除锁定　(位于FCTL3 寄存器)*
- *BUSY：忙标志位　1=Flash 正在执行当前操作，不允许再向 Flash 发出操作命令*
  0=Flash 空闲，向 Flash 发出操作命令　(位于FCTL3 寄存器)
- *WAIT：写入等待标志位　1=Flash 正在被写入，不允许再向 Flash 发出写操作指令*
  0= Flash 上一次写操作已完成，允许写操作 (位于FCTL3 寄存器)

每次对 Flash 执行擦写操作之前，都要先清除锁定标志位（LOCK）再执行操作，指令发出后，要等待忙标志消失后才能执行下一次操作。所有操作完成之后，要将锁定标志位恢复为 1，将 Flash 设为只读状态，以保证数据的安全。

当通过 FCTL1 将 Flash 的写模式开启并清除 FCTL3 内的锁定标志位之后，写 Flash 的操作类似于写 RAM。向 Flash 存储单元写入数据即可自动开启编程逻辑控制器、编程电压发生器等，它们会在 Flash 控制器的协调下自动完成向 Flash 内写入数据的过程。和 RAM 不同的是 Flash 写入速度较慢，需要判断 BUSY 标志位结束后才能进行下一次操作。

例 2.6.3：某 MSP430 单片机系统，向 Flash 存储器的 InfoA 段内的 1082H 单元写入数据 0x30。（假设 Flash 时钟已经设置好）。

```
unsigne char *Ptr=(unsigned char *)0x1082;//定义字节型指针，指向 0x1082 单元
FCTL1 = FWKEY + WRT;        //Flash 进入正常写状态
FCTL3 = FWKEY ;             //清除 Flash 的锁定位
_DINT();                    //Flash 操作期间不允许中断，否则将导致不可预计的错误
*Ptr=0x30;                  //向 0x1082 单元写入数据
while(FCTL3 & BUSY);        //等待操作完成
_EINT();
```

```
FCTL1 = FWKEY               //Flash 退出写状态
FCTL3 = FWKEY + LOCK;       //恢复 Flash 的锁定位，保护数据
```

上例比较适合写入少量数据，若需要向 Flash 内写入大量数据尽量使用批量写入模式，以提高效率。正常模式下每次写完一个数据，都会关闭编程电压发生器，在下次写操作前需要重新开启并等待电压稳定，而批量写模式下编程电压发生器将一直保持开启，省去了等待编程电压建立并稳定所需的时间，所以批量写模式效率较高。批量写模式下每字节写完后用 WAIT 标志判忙，每块结束后再用 BUSY 判忙。

例 2.6.4：在某 MSP430 单片机系统上，将数组 Array[128]内全部数据写入 InfoA 段。

InfoA 段的大小是 128 字节，包含了 2 个块（64 字节为一块），所以用批量写入模式，分两次写入：

```
unsigne char *Ptr=(unsigned char *)0x1080;//定义字节型指针，指向 InfoA 段起始
int i,j,k=0;
FCTL1 = FWKEY + WRT + BLKWRT;   //Flash 进入批量写状态
FCTL3 = FWKEY ;                 //清除 Flash 的锁定位
_DINT();                    //Flash 操作期间不允许中断，否则将导致不可预计的错误
for(j=0;j<2;j++)                //两个块
  {
     for(i=0;i<64;i++)          //每个块 64 字节
       {
          *(Ptr++)=Array[k++];  //依次写入数据
          while(FCTL3 & WAIT);   //等待字节写操作完成
       }
     while(FCTL3 & BUSY);       //等待块写操作完成
  }
_EINT();
FCTL1 = FWKEY                   //Flash 退出写状态
FCTL3 = FWKEY + LOCK;           //恢复 Flash 的锁定位，保护数据
```

当通过 FCTL1 将 Flash 设为段擦除模式并清除 FCTL3 内的锁定标志位之后，只需要向某个段内的任何存储单元写数据 0，即可执行擦除操作。擦除操作时同样需要判断 BUSY 标志位结束后才能进行下一次操作。

例 2.6.5：某 MSP430 单片机系统，需要擦除 Flash 存储器的 InfoA 段。（假设 Flash 时钟已经设置好）。

```
unsigne char *Ptr=(unsigned char *)0x1080;//定义字节型指针，指向 InfoA 段
FCTL1 = FWKEY + ERASE;      //Flash 进入单段擦除状态
FCTL3 = FWKEY ;             //清除 Flash 的锁定位
_DINT();                    //Flash 操作期间不允许中断，否则将导致不可预计的错误
*Ptr=0;                     //发出擦除指令的方法：向被擦除段内任意单元写 0，
while(FCTL3 & BUSY);        //等待操作完成
_EINT();
FCTL1 = FWKEY               //Flash 退出擦除状态
FCTL3 = FWKEY + LOCK;       //恢复 Flash 的锁定位，保护数据
```

除了擦除与写入过程以外，任何情况下都可以读 Flash 内容。读取 Flash 的速度与读取 RAM 速度相同（Flash 的读取速度上限约 10MHz，RAM 的读写速度上限能达到 100MHz，由于 MSP430F1xx/4xx 系列单片机最大 8MHz 主频，体现不出速度差异）。由于 MSP430

单片机采用冯诺依曼结构，读取 Flash 的方法与读取 RAM 内数据的方法完全相同。

例 2.6.6：某 MSP430 单片机系统，读取 Flash 存储器的 InfoA 段内的 1082H 单元内容，存于变量 Val 内。

```
unsigne char *Ptr=(unsigned char *)0x1082;//定义字节型指针，指向 0x1082 单元
unsigne char Val;
Val=*Ptr;                    //读取 0x1082 单元的内容
```

例 2.6.7：某 MSP430 单片机系统，读取 Flash 存储器的 InfoA 段内的全部内容，存于数组 Array[128]内。

```
unsigne char *Ptr=(unsigned char *)0x1080;//定义字节型指针指向 InfoA 起始单元
unsigne char Array[128];                  //128 字节数组
int i;
for(i=0;i<128;i++)                        //循环 128 次(InfoA 段共 128 字节)
  {
    Array[i]=Ptr[i];                      //依次读取 InfoA 的内容
  }
```

在 FCTL3 寄存器内还有一些不常用的控制位与标志位：

- ***EMEX：紧急退出控制位　1=紧急退出　0=正常状态　(位于 FCTL3 寄存器)***

在 Flash 写或擦除过程中，将该标志位置 1，会立即终止 Flash 的操作，FCTL1 寄存器的控制字将被清零（Flash 恢复为正常状态）。但当前所执行的操作是未完成的，操作的结果将是不可预知的。比如在系统检测到掉电时，只有数毫秒的电力可供继续工作，需要紧急停止 Flash 操作，去执行更加紧急的结果保存任务，可以使用该控制位。

- ***ACCIFG：非法访问标志　1=Flash 曾被非法访问过　0=访问合法(位于 FCTL3 寄存器)***

若通过 Flash 控制器操作了不存在的 Flash 空间（如空白区、RAM 区），将会引起该标识位置位。若 ACCVIE 中断允许控制位为 1，将会引发 NMI 中断。该标识位需要软件清除。

- ***ACCVIE：非法访问中断允许　1=允许 Flash 非法访问中断　(位于 IE1 寄存器)***

0=不允许 Flash 非法访问中断

利用 ACCIFG 引起的 NMI 中断可以捕获某些不可预期的错误，从而进行相应的紧急处理。例如某数据采集、记录系统的软件中不断向 Flash 区写数据，而软件设计上没有考虑存储区的边界问题，那么一旦数据存满后溢出，将访问到非 Flash 区域。这种状况下利用非法访问中断可以捕获到错误，进行某些紧急处理（如删除数据、复位单片机等）。

- ***KEYV：密码错误标志　1=曾经发生过 FCTLx 密码错误　(位于 FCTL3 寄存器)***

0=未发生过 FCTLx 密码错误

前文已述，为了 Flash 的安全，在写 FCTLx 寄存器时都要核对高 8 位的密码（固定值 0xA5）。一旦密码不对，将立即引起系统复位，同时会将该标志位置 1。在程序开始时若检测该标志位为 1，则可判定系统复位的原因是曾将错误的操作了 FCTLx 寄存器，可以进行某些必要的紧急处理（例如检验 Flash 内数据校验和是否正确、删除错误的数据、询问用户如何处理等操作）。

I Flash 控制器的应用

Flash 存储器的优点是掉电后数据不会丢失，且 MSP430 单片机可以通过程序来擦写 Flash，因此可以利用 Flash 来保存数据。通过擦除、写、读三种操作的配合，可以组合出丰富的数据存储功能。

连续数据记录

一般来说，在 MSP430 单片机中，Flash 的容量远大于 RAM 容量，且不受断电的影响，所以 Flash 特别适合做连续的数据记录使用。

例 2.6.8： 用 MSP430F425 单片机设计电压记录仪器，每秒采集一次输入电压，将电压的历史记录连续地保存在单片机内部存储器中。为数据连续记录编写函数。

MSP430F425 单片机共有 16KB 的主 Flash 存储空间，假设程序大小为 3.9KB（占用 8 个段），扣除中断向量表所在的第 0 段，还剩 11.5KB（23 个段）的 Flash 存储空间可供保存数据用。对照图 2.6.1 可以看出：从 C000H 到 C7FFH 的 8 段存储空间被程序占用；FE00H 到 FFFFH 的第 0 段被中断向量表占用；从 C800H~FCFFH 的 23 段是空余区间，可用于保存数据。给出一个程序范例（假设 Flash 时钟已设置正确）：

```
  #define START_ADDR      0xFE00      /*存储区起始地址*/
  #define END_ADDR        0xFCFF      /*存储区结束地址*/
  int *Flash_Ptr=(int *) START_ADDR;//整型指针(全局变量)，指向数据存储区起始单元

/*********************************************************************
* 名    称：Flash_RecordWord()
* 功    能：向 Flash 内连续地保存整形数据值。
* 入口参数：Word:       待保存的整型数据
* 出口参数：1 表示写入成功  0 表示写入失败（空间已满）
* 说    明：在重新记录之前需要擦除数据段，并将 Flash_Ptr 恢复为 START_ADDR
  *********************************************************************/
  char Flash_RecordWord (int Word)
  {
    if((unsigned int)Flash_Ptr > END_ADDR) return(0);//空间已满返回 0 表示失败
    FCTL1 = FWKEY + WRT;          //Flash 进入正常写状态
    FCTL3 = FWKEY ;               //清除 Flash 的锁定位
    _DINT();          //Flash 操作期间不允许中断，否则将导致不可预计的错误
   *Flash_Ptr = Word;             //向存储指针所指的单元写入数据
   Flash_Ptr++;                   //指针指向下一单元
   while(FCTL3 & BUSY);           //等待操作完成
   _EINT();
   FCTL1 = FWKEY                  //Flash 退出写状态
   FCTL3 = FWKEY + LOCK;          //恢复 Flash 的锁定位，保护数据
   return(1)                      //返回 1，表示写入成功
  }
```

在本例中，每秒记录一次数据（2 字节整型），11.5KB 空间可以连续记录 1 个半小时。若每隔 1 分钟记录一次数据，可以连续记录 4 天。数据存满后，将不再继续写入，需要擦除整个存储空间后才能重新开始数据记录。可以编写一个 Flash 数据清空函数来完成：

```
/*********************************************************************
* 名    称：Flash_RecordClear()
* 功    能：清除 Flash 内记录的所有数据。
* 入口参数：无
* 出口参数：无
* 说    明：在重新开始记录之前需要调用该函数，擦除数据段，并将 Flash_Ptr
           恢复为 START_ADDR
*********************************************************************/
void Flash_RecordClear ()
{ unsigned char *Ptr = (unsigned char *)START_ADDR;//指向数据起始地址的指针
  FCTL1 = FWKEY + ERASE;//Flash 进入单段擦除状态
  FCTL3 = FWKEY ;       //清除 Flash 的锁定位
   _DINT();             //Flash 操作期间不允许中断，否则将导致不可预计的错误
  while(1)
   {
     *Ptr = 0;                  //擦除一段
      while(FCTL3 & BUSY);      //等待擦除操作完成
      Ptr+=512;                 //指向下一段
      if((unsigned int)Ptr > END_ADDR) break;  //直到擦除到数据地址结束段
   }
  _EINT();
  FCTL1 = FWKEY;                //Flash 退出擦除状态
  FCTL3 = FWKEY + LOCK;         //恢复 Flash 的锁定位，保护数据
  Flash_Ptr=(unsigned int *) START_ADDR;   //写指针恢复成数据段起始地址
}
```

我们也可以编写一个循环队列，在数据存满后擦除最旧的一段数据并将最新的数据写入该段。这种方法可以一直记录数据，存储器内永远保存着最近一段时间的数据。该方法的程序留给读者自行完成。

MSP430 单片机的 Flash 控制器支持单字节（8bit）写入和字（16bit）写入，因此对于 char 型和 int 型的变量，均可以用指针赋值的方法写入 Flash。但对于 long、folat、double、long long 等超过 2 字节的变量，需要拆分成多个 8 位或 16 位的数据后才能写入。

例 2.6.9： 将 long 型的变量拆成 2 个 int 型变量：

```
  long int a=0x12345678;        //长整型变量
  long int b,c;
//------------------------------拆分------------------------------
  b=(unsigned long)a/65536;         //高 16 位(0x1234)
  c=(unsigned long)b%65536;         //低 16 位(0x5678)
//--------------------------- --恢复------------------------------
  a=(unsigned long)b*65536 + (unsigned long)c;  //将高低位拼合,恢复成长整型
```

注意负号会影响运算结果，所以强整为无符号长整型后再做除法运算。由于 65536 是 2 的整数幂，编译器会自动的用移位运算来完成，效率较高。

对于 float、double 等浮点变量无法用除法来拆分。可以利用 C 语言中的联合体将多字节变量和整型、字符型数据公用同一段存储空间，从而通过访问不同的数据成员来实现拆分。

例 2.6.10： 将 folat 型的变量拆成 4 个 char 型数据，存于 4 个 char 型变量内：

```
   union FloatChar          //声明一个浮点型与 4 个字节型的联合体
   { float Float;
     struct ByteF4         //为了不让四个 char 变量也公用同一地址，需要一个结构体
      { unsigned char  Byte_HH;        //最高位
        unsigned char  Byte_HL;        //次高位
        unsigned char  Byte_LH;        //次低位
        unsigned char  Byte_LL;        //最低位
      }Bytes;
   };
//--------------------------数据定义-------------------------------
   float f=3.1415926;                  //待拆分的浮点数
   char a,b,c,d;                       //四个 char 型变量
   union FloatChar F_Data;             //定义一个联合体，名为 F_Data
//---------------------------拆分-----------------------------------
   F_Data.Float= f;                    //对联合体中的浮点型成员赋值
   a = F_Data.Bytes.Byte_HH;           //得到最高字节
   b = F_Data.Bytes.Byte_HL;           //得到次高字节
   c = F_Data.Bytes.Byte_LH;           //得到次低字节
   d = F_Data.Bytes.Byte_LL;           //得到最低字节
//---------------------------恢复-----------------------------------
   F_Data.Bytes.Byte_HH = a;           //最高字节
   F_Data.Bytes.Byte_HL = b;           //次高字节
   F_Data.Bytes.Byte_LH = c;           //次低字节
   F_Data.Bytes.Byte_LL = d;           //最低字节
   f = F_Data.Float;                   //恢复出浮点数
```

这种方法可以最高效地得到各种变量的每个字节数据，能够达到汇编指令的效率极限（反汇编的结果只有四条 MOV 语句）。除了在数据保存中使用之外，串行通讯中也广泛使用该方法将数据拆成单个字节后传输，在接收方再恢复成原始数据类型。

随机数据存储

在连续存储的应用中，由于每次写入的地址均不重复（连续递增），因此不会出现两次写同一单元，或者改写某单元数据内容的情况。但在随机存储数据的应用中，需要随时读写任何一个存储单元，因而不可避免地会出现改写 Flash 内数据的问题。在保存菜单设置、保存系统状态、保存最新测量结果等应用中，都会用到随机数据存储。

Flash 内容的改写过程比较繁琐。首先，需要将被改写单元所在的整个数据段备份到 RAM 内，然后再擦除整个数据段。接下来写入被改写的数据，最后从备份的内容中恢复数据段内的其他数据内容。由于需要备份数据，效率很低，所以一般随机存储的应用中都尽可能的在 Info 段进行，因为 Info 段大小只有 128 字节，备份与恢复效率都比主 Flash 段要高。并且在备份的过程需要耗费与段大小同样容量的 RAM，大部分 MSP430 系列的单片机都具有 256 字节以上的 RAM 空间，足够备份一个 Info 段内容。

例 2.6.11：写一个数据段备份的函数，将一个数据段备份至 RAM。

```
 /*********************************************************************
  * 名    称：Flash_BackUp2RAM()
  * 功    能：备份 Flash 段内的所有数据至 RAM 中的数组。
  * 入口参数：Segment: 段起始地址
```

```
*          Array  : 备份数组名（首地址）
*         SegSize: 段大小（字节）
* 出口参数：无
*******************************************************************/
  void Flash_BackUp2RAM(unsigned int Segment,char *Array, int SegSize)
  { unsigned char *Ptr = (unsigned char *) Segment;//指向数据段起始地址的指针
    int i;
    for(i=0;i < SegSize; i++)     //依次备份段内每个字节数据
     {
       Array[i] = Ptr[i];         //备份一字节
     }
```

这里数组 Array 没有使用全局变量，因为全局变量所占的存储空间永远不会被释放，而备份的数据在写入完毕后就不再有用，没有必要让这些数据一直占用 RAM 空间。所以 Array 数组应该是随机写入函数内的局部变量，通过指针传递进备份函数。随机写函数执行完毕之后，Array 所在的空间即被释放，编译器会允许其他函数的变量覆盖使用这段 RAM 区。

在 RAM 开销比较紧张的应用中，或因数据量较大需要在主 Flash 内进行随机读写的应用中，以及因其他原因不能用 RAM 来进行备份时，可以使用另一段 Flash 存储器来进行备份。这种方法的效率更低，因为要擦除并写入另一个数据段，是速度较低的操作。并且要浪费一半的数据存储容量，例如用 InfoB 段备份 InfoA 段的数据，则 InfoB 段不能再用于数据存储，因为它在备份 InfoA 前需要被擦除。这种备份方法的优点是节省了大量的 RAM 开销。

例 2.6.12：写一个数据段备份的函数，将某个数据段备份至另一数据段。

```
/*******************************************************************
* 名    称：Flash_BackUp ()
* 功    能：将一个 Flash 段内的所有数据备份至另一段。
* 入口参数：Segment1: 数据段起始地址
*          Segment2: 备份段起始地址
*         SegSize:   段大小（字节）
* 出口参数：无
*******************************************************************/
  void Flash_ BackUp (unsigned int Segment1, unsigned int Segment2,int SegSize)
  { unsigned char *Ptr1 = (unsigned char *) Segment1;//指向数据段起始地址的指针
    unsigned char *Ptr2 = (unsigned char *) Segment2;//指向备份段起始地址的指针
    int i;
    FCTL1 = FWKEY + ERASE;    //Flash 进入单段擦除状态
    FCTL3 = FWKEY ;           //清除 Flash 的锁定位
    _DINT();                  //Flash 操作期间不允许中断，否则将导致不可预计的错误
    *Ptr2=0;                  //擦除备份段
    while(FCTL3 & BUSY);      //等待擦除操作完成
    FCTL1 = FWKEY + WRT;      //Flash 进入写状态
    for (i=0; i< SegSize; i++)
     {
       ptr2[i] = ptr1[i];     // 将数据段内容依次拷贝到备份段内
       while(FCTL3 & BUSY);   // 等待写操作完成
```

```
     }
     FCTL1 = FWKEY;                  //Flash 退出擦除状态
     FCTL3 = FWKEY + LOCK;           //恢复 Flash 的锁定位，保护数据
     _EINT();
   }
```

有了数据备份函数之后，才能进行 Flash 随机存储。

例 2.6.12：写一个可以随机改写 InfoA 段的函数，向 InfoA 段的某地址单元写入单字节数据。其中地址用 0~127 来表示（不要出现物理地址 0x1080~0x10FF），以便和其他单片机系统中的 EEPROM 程序保持兼容性。

```
#define   FLASH_SAVEADDR  (0x1080) /*Flash 数据存储区首地址(InfoA)*/
#define   FLASH_COPYADDR  (0x1000) /*Flash 备份存储区首地址(InfoB)*/

/*************************************************************************
  * 名    称：Flash_WriteChar()
  * 功    能：向 Flash 中随机写入一个字节(Char 型变量)
  * 入口参数：    Addr:存放数据的地址（0~127）
                 Data:待写入的数据
  * 出口参数：无
  * 范    例：Flash_WriteChar(0,123); 将常数 123 写入 0 单元（0x1080）
             Flash_WriteChar(1,a);    将 char 型变量 a 写入 1 单元 (0x1081)
*************************************************************************/
void Flash_WriteChar (unsigned int Addr,unsigned char Data)
  {
    unsigned char *Flash_ptrA;                     // Segment A pointer
    unsigned char *Flash_ptrB;                     // Segment B pointer
    int i;
    Flash_ptrA = (unsigned char *) FLASH_SAVEADDR;// 指向 InfoA 的指针
    Flash_ptrB = (unsigned char *) FLASH_COPYADDR;// 指向 InfoB 的指针
    Flash_BackUp(FLASH_SAVEADDR,FLASH_COPYADDR,128);//Flash 内的数据先保存起来
    FCTL1 = FWKEY + ERASE;      //Flash 进入单段擦除状态
    FCTL3 = FWKEY ;             //清除 Flash 的锁定位
    _DINT();                    //Flash 操作期间不允许中断，否则将导致不可预计的错误
    *Flash_ptrA =0;             //擦除数据段
    while(FCTL3 & BUSY);        //等待擦除操作完成
    FCTL1 = FWKEY + WRT;        //Flash 进入写状态
    for (i=0; i<128; i++)       //依次处理段内 128 个数据
      {
         if(i==Addr)            //对于被改写的数据所在的单元
          {
            *Flash_ptrA++ =Data;//对数据段相应单元写入新数据
            Flash_Busy();       //等待写操作完成
            Flash_ptrB++;       //跳过备份区内该单元的数据单元
          }
         else                   //对于其他不改变的数据单元
          {
            *Flash_ptrA++ = *Flash_ptrB++; // 从备份段内恢复原数据
            Flash_Busy();                  // 等待写操作完成
```

```
        }
      }
    _EINT();
  }
```

本例中使用了 InfoB 段来备份 InfoA 段的数据，节省了 128 字节的 RAM，但写入速度很慢。如果修改上例程序中的以下几句，使用 RAM 来备份数据，速度会快很多：

```
unsigned char BackUpArray[128];              //定义一个与段大小相同的数组
Flash_ptrA = (unsigned char *) FLASH_SAVEADDR;//指向数据段(InfoA)的指针
Flash_ptrB = BackUpArray;                     //指向备份数组的指针
Flash_BackUp2RAM (FLASH_SAVEADDR, BackUpArray,128);//Flash 内的数据先
                                                    //备份至 RAM
```

对于数据量不大的应用，没有必要备份数据段内全部 128 个字节的数据。例如某个应用中最多只需保存 16 字节数据，可以加一个宏定义：

```
 #define MAX_DATA_NUM   16                          /*最多 16 字节数据*/
```

把所有循环语句与数组定义语句中出现的 128 替换成 MAX_DATA_NUM 宏定义（16），可以节约 8 倍的时间，或节省 112 字节 RAM。

在光盘中附带了 Flash 存储器随机读写的程序库，不仅包含了字符型和 int 型变量的读写函数，还包含了长整型和浮点型数据的读写函数。该函数库采用了 Flash 段互相备份的方法，节省 RAM 但速度较慢，适合在菜单、人机界面、校准等对速度不敏感的场合使用。

Flash 存储器寿命问题

Flash 存储器的寿命有两层含义：一是数据保存时间，二是擦写次数。Flash 存储器在理想状况下数据保存时间能达到 100 年。Flash 时钟设置、编程电压和储存温度都会影响数据保存时间。MSP430 单片机要求 Flash 写入过程中时，电源电压必须在 2.7V~3.6V 之间。若果电压低于 2.7V 会导致数据保存时间的下降，这在电池供电的设备重要特别注意。特别是使用 2 节 1.5V 干电池供电的设计中，当电池寿命耗尽时，单节电压会下降到 1.1V 左右，此时 Flash 的写入已经不可靠了。另外若将充电电池（1.2V）替代干电池（1.5V）作为电源，Flash 写操作也是不可靠的。

MSP430 系列单片机标称的 Flash 擦写次数典型值是 10 万次，最低保证 1 万次。对于程序代码自身占用的存储单元来说，只在仿真和下载时才耗费 1 次擦写寿命，所以几乎不用考虑寿命问题。但对于程序中利用 Flash 控制器来擦写 Flash 而言，就不得不考虑寿命问题了。

例如某菜单程序，平均每小时被操作 20 次，每天平均工作 10 小时，则每天会对 Flash 中同一单元擦写 200 次。按 10 万次寿命来计算，工作寿命只有 500 天，约 1 年半。

再如数据记录程序，24 小时不停工作，每秒钟记录一次数据，每次占用 2 字节，计满后自动清除最旧的数据。那么写完一段 512 字节的主 Flash 需要 256 秒，假设有 20 段 Flash 可供使用（8KB），则每 5120 秒会重复写同一单元。按 10 万次计算，寿命可达 16 年。

对比上面 2 个结果，虽然后者操作频率远高于前者且 24 小时不停工作，但寿命远长于前者。原因在于后者均匀地消耗每一个 Flash 存储单元的寿命，而前者很快耗尽同一单元的寿命。这种寿命集中损耗的情况在随机数据存储中最常见，而且即使改写一字节也要擦写整个数据段，实际上随机存储很快会耗尽整个数据段的寿命。

在产品设计初期一定要规划好 Flash 的寿命，尽量不要让某些需要频繁改写的值保存在 Flash 中。特别是要根据菜单操作的频繁程度计算 Flash 存储单元寿命，至少保证 5 年的操作寿命。如果出现需要掉电后保存，且更新频繁的数据可以先利用 RAM 暂存，在断电前才存入 Flash 中。

例如在某电动自行车里程表的设计中，随着车轮旋转里程信息随时会更新，需要将总里程数保存在 Flash 内，以免更换电池时丢失里程数。假设轮圈周长 2 米，若每次更新里程后都立即将新里程保存进 Flash 内，则 20 万米（200 公里）后 Flash 寿命即被耗尽。在这个设计中可以用 RAM 保存里程，并在电池仓上安装一个开关锁，必须拨动该开关才能打开电池仓。当单片机检测到电池仓被打开后，才将 RAM 中保存的总里程数据写入 Flash，那么 Flash 寿命在更换 10 万次电池后才会被耗尽，相当于无限寿命。

在某些频繁操作的手持设备中，也可以将各种菜单参数和系统状态都保存在 RAM 中，使用 2.3 节所述的软件电源开关，当关闭电源开关按钮按下时，先将 RAM 内重要数据保存到 Flash 内，再进入 LPM4 实现关机，更换电池必须在关机后进行。由于开关机频率远低于操作频率，Flash 存储寿命得以延长。该方法的缺点是强行拆卸电池会造成数据丢失。

在某些无法控制电源开关的应用中，例如工业仪表，断电不由单片机主动控制。遇到需要频繁更新的数据且数据需要断电后仍被保存的情况时，可以在 VCC 和 GND 之间接一个较大的电容，再用比较器设计一个电压跌落检测电路，当电源电压跌落时引发中断，在中断内将数据保存进 Flash 内。由于电容上的储存的电荷能维持单片机继续工作数十毫秒，时间足够将数据存入 Flash。

Flash 调试选项的设置

MSP430 单片机的仿真和调试之前，都会自动通过 FET Debugger 调试器更新 Flash 内容，写入最新的程序。所以，每次更新程序的过程都会对将 Flash 内的数据擦除。例如在 2.6.8 中的数据采集程序中，假设发现了一个 Bug 并修改，需要进入调试状态检查问题是否已经被排除。一旦点击“Debug”按钮，FET Debugger 会自动擦除 MSP430 单片机内原有的程序并下载新的代码。这个过程会擦除主 Flash，因此原来存放的数据也全部丢失。类似的情况也会出现在菜单程序中，每次调试时擦除 Info 段会造成菜单全部设置丢失。

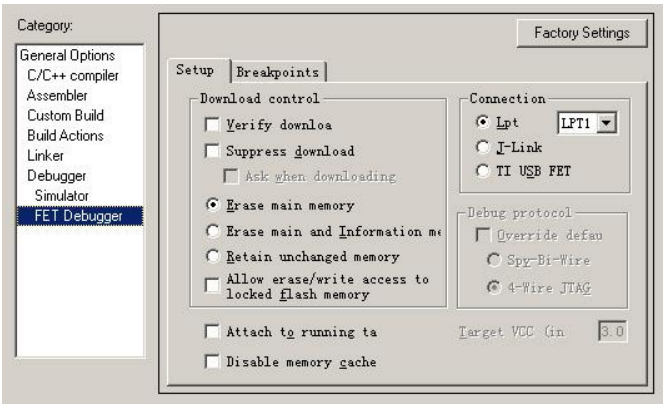


图 2.6.4 Flash 调试选项

为解决上述问题，EW430 开发环境中设有 Flash 调试选项页，用于设置调试过程中对

Flash 内原有数据的处理方法。在工程管理器的工程名上右键打开“Option”选项，在左边选择最后一项“FET Debugger”，打开 Setup 页。其中关于调试时 Flash 内的数据有 3 个选项：

Erase main memory：仅擦除主 Flash，保留 InfoFlash。选择该选项后，每次调试时只擦除程序空间并更新程序，不影响 Info 段的内容。所以若菜单等程序将设置参数保存在 Info 段内则不受调试的影响。

Erase main and Information memory：擦除主 Flash，也擦除 InfoFlash，是默认选项。每次点击调试按钮，都会擦除 Flash 内的全部数据。Info 段所保存的内容也随之丢失。

Retain unchanged memory：保留未改变的 Flash 内容。选择该选项后，每次调试时都会先将 Flash 内容全部读出，更新程序后将未改变的数据重新写回 Flash 内。这样无论主 Flash 还是 InfoFlash 内，非程序代码部分的内容均会被保留。对于例 2.6.8 的应用来说，若选择该选项，调试时主 Flash 内空余部分所记录的电压数据将被保留。但由于每次调试时都要进行读取、改写、重新写入的过程，速度较慢。

2．7　16 位 ADC (SD16 模块)

在 MSP430 系列的大部分单片机中，都集成了模数转换器（ADC）以及 ADC 所需的附件，如基准源、采样保持器、通道选择模拟开关等。部分单片机内部还集成了缓冲器、差分可编程放大器、温度传感器等部件，使得 MSP430 单片机非常容易地测量各种模拟量输入。整个系列涵盖了从最低成本的斜率 ADC，中端的 10 位、12 位 ADC，以及高端的 16 位 ADC。在各种测量应用中都可以找到合适的单片机，单芯片完成测量任务。

在 MSP430F42x 系列单片机中，集成了三个独立的 16 位 ADC，并且包含基准源、可编程增益放大器以及温度传感器，适合各种高精度测量应用。目前 16 位及以上的高分辨率 ADC 普遍采用了Σ－△调制技术，因此这类 ADC 也被称被Σ－△（Sigma-Delta）型 ADC。

Ⅰ Sigma-Delta 型 ADC 的原理

Sigma-Delta 型 ADC 是一类利用过采样原理来扩展分辨率的模数转换器件。从理论上说，用低分辨率的 ADC 对信号进行大量采样，只要调制方法得当，并且采样次数足够，仍能够恢复出高分辨率的测量值。例如用一根分辨率 1cm 的尺子，去测量长度为 1.5cm 的钉子，如果不允许目测估算的话，测出的结果只能是 1cm 或 2cm。假设我们将这枚钉子随机的扔在尺子上（保持与尺子平行，只是起始点随机），如果压住 1 根刻度线记为 1cm，压住 2 根刻度线记为 2 cm，由于 1.5cm 的钉子压住 1 根刻度线与压住 2 根刻度线的概率相等，那么重复 1000 次测量之后取平均，我们会得到 1.500cm 的高分辨率结果。对于 1cm~2cm 之间的不同钉子长度，测量结果 1、2 的概率不同，大量平均后能得到高分辨率的钉子实际长度。

上例中，“扔钉子”就是一种随机调制方法。随机调制的效率较低且难于用电路实现，

在电路中，更广泛使用的不是随机调制，而是Σ－△（Sigma-Delta）调制。取一个极端情况，当一个 ADC 的分辨率下降到只有 1 比特时，只要通过 Sigma-Delta 调制后大量采样，仍能恢复出高精度的原始电压值。在电路上，1 比特 ADC 实际就是一个比较器，1 比特 DAC 也可以用模拟开关来实现；加之滤波和量化工作也是全数字实现的，所以 Sigma-Delta 型 ADC 更像是数字器件而不是模拟器件。这最大可能的避免了模拟电路的漂移、批次性问题。因此 Sigma-Delta 型 ADC 可以集成在芯片上，并很容易达到高精度和高分辨率。

图 2.7.1 是简化后的一阶 Sigma-Delta 型 ADC 的原理框图。采用了一个比较器作为 1 比特 ADC，为了得到提高分辨率，采样时钟应该远高于读出时钟 fs。假设采样时钟是读出时钟的 K 倍，系数 K 被称为过采样率。图中 Sigma-Delta 调制器将输入信号转化为连续的“0、1”数据流。数据流的速率就是采样频率 Kfs。

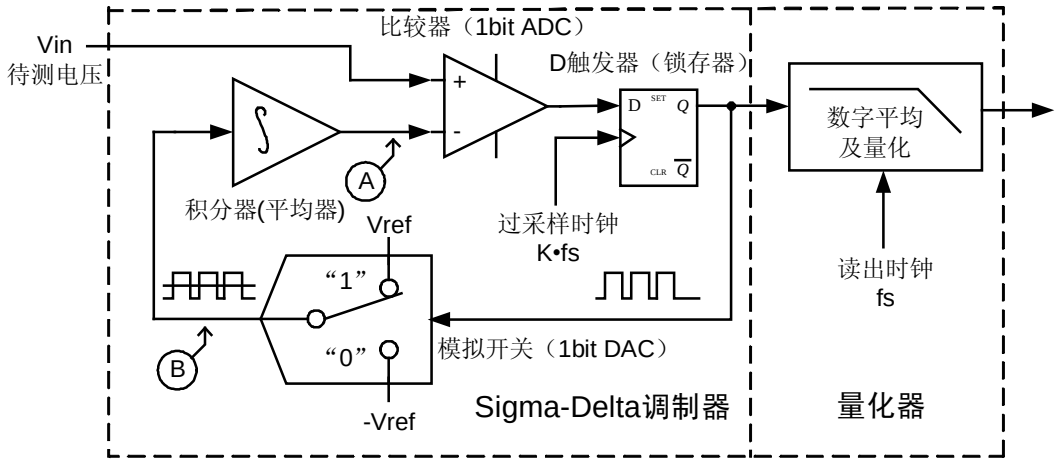


图 2.7.1 简化的 Sigma-Delta 型 ADC 原理框图

从 1 比特 ADC 输出的电平被 D 触发器周期性的采样形成数据流，数据流分为 2 路：一路给数字平均滤波和量化，另一路送到 1 比特 DAC（实际是个模拟开关），控制 1 比特 DAC 输出 0 或 Vref，再通过积分器取得平均值作为反馈。当输入电压 Vin 在-Vref ~+Vref 之间时，若 Vin 高于 A 点电压，比较器将输出高电平，D 触发器将数据“1”送给 DAC，DAC 输出等于基准电压 Vref，积分器将平均值升高，直到 A 点电压与 Vin 相等为止。反之，若 Vin 低于 A 点电压，比较器将输出低电平，D 触发器将数据“0”送给 DAC，DAC 输出为-Vref，将平均值降低，直到 A 点电压与 Vin 相等为止。

整个环路构成典型的负反馈，只要比较器的开环增益足够大，A 点电压会非常接近 Vin（虚短路）。A 点电压由 B 点信号求平均得来，而 B 实际上就是数据流。换句话说，数据流的平均值一定会非常接近 Vin / Vref 的比值。因此，从 D 触发器取数据流的另一分支，经过的数字平均及量化后，也能得到 Vin / Vref 的比值。只要提供一个稳定且精确的基准源 Vref，将数字平均量化器计算出的数据流平均值（ADC 转换结果）乘以 Vref，即可得到输入电压的值。

在过采样时钟一定的前提下，择较大的过采样率 K，能够提高分辨率，但读出速度会下降；反之，若降低过采样率 K 能够得到更快的采样速度但会牺牲分辨率。在扔钉子的例子中，若需要得到高分辨率的结果则要增加测量次数，但测量速度会变慢，反之减少次数

提高速度，测量分辨率下降。因此，能够在分辨率与速度之间自由选择也是 Sigma-Delta 型 ADC 的一大特色。

在实际的 Sigma-Delta 型 ADC 芯片中，都采用开关电容电路作为输入、减法器和积分器和基准切换电路。这样便于纯数字方法实现。很多 Sigma-Delta 型 ADC 内置有可编程增益放大器（PGA），非常方便与电桥、热电偶等微弱信号传感器连接。PGA 的实现其实是靠改变开关电容采样、积分与读出的速度比来实现的，仍然是纯数字电路实现，不存在模拟放大器的漂移、失调、上下轨等问题。

I SD16 模块的结构与原理

在 MSP430 单片机中，将内置的 16 位 Sigma-Delta 型 ADC 简称为 SD16 模块。MSP430F42x 系列的单片机中都含有 SD16 模块。

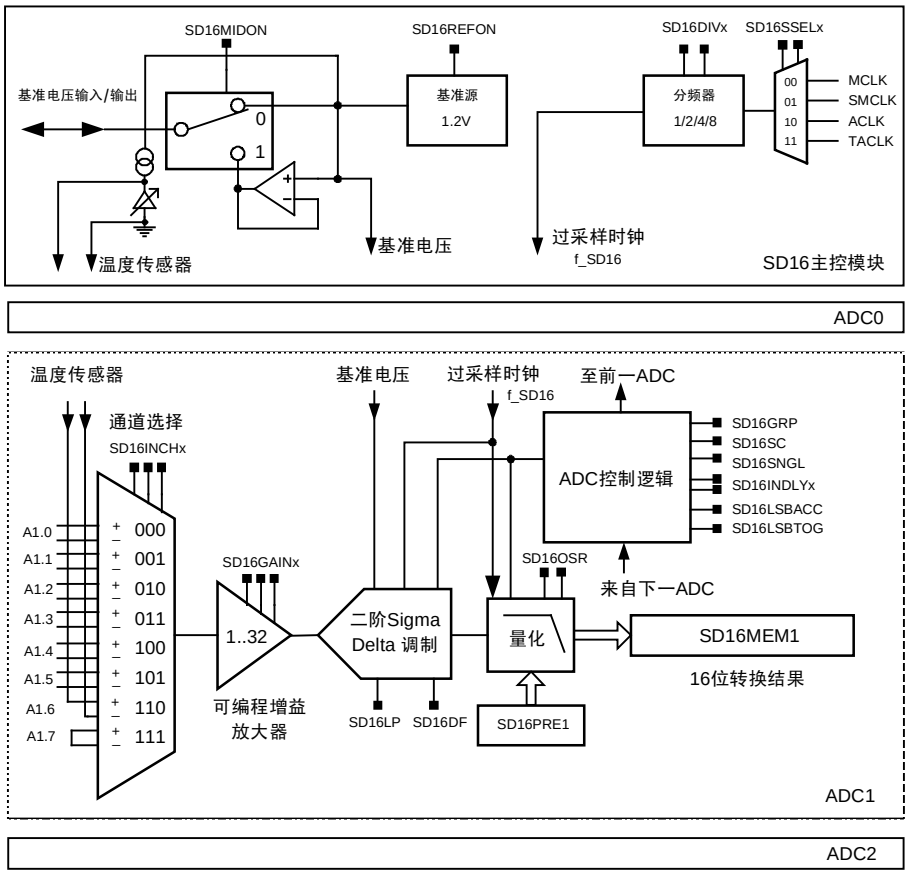


图 2.7.2　SD16 模块结构框图

图 2.7.2 是 SD16 模块的结构框图，可以看出 SD16 模块实际上包含了 3 个独立的 16 位 ADC，它们公用一个时钟源和基准电压源。每个 ADC 都有独立的控制寄存器组，并有 8 个差分输入通道，其中通道 6 接到了内部温度传感器，通道 7 短路（0V，校准用），通道 0~5 可以测量输入电压。在 MSP430F42x 单片机上，实际只有每个 ADC 的通道 0（A0.0、A1.0、A2.0）对外引出。在将来推出的引脚更多的芯片上才会引出其余通道。一般来说，

配置和使用 SD16 模块按照以下过程进行：

配置时钟及基准源：

使用 SD16 模块之前，首先要配置过采样时钟以及基准源，相关的控制位位于主控模块内：

- *SD16SSELx：SD16 模块时钟源选择 （位于 SD16CTL 寄存器）*
  00=MCLK 01=SMCLK 10=ACLK 11=外部输入(TACLK 管脚)
  快捷宏定义： SD16SSEL_0 SD16SSEL_1 SD16SSEL_2 SD16SSEL_3

- *SD16DIVx：SD16 模块时钟分频选择 （位于 SD16CTL 寄存器）*
  00=1 分频 01=2 分频 10=4 分频 11=8 分频
  快捷宏定义： SD16DIV_0 SD16DIV_1 SD16DIV_2 SD16DIV_3

在同样的过采样率下，采样时钟频率越高，得到同样分辨率所需的时间越短，这对减少工作时间降低功耗有利。由于 Sigma-Delta 型的 ADC 采用的是开关电容输入级，当采样频率过高时可能导致采样电容充电未满导致测量误差。所以一般根据实际情况选择最高的时钟频率。芯片的数据手册会给出推荐的最高工作频率。例如 MSP430F425 在 3V 工作电压条件下，SD16 模块推荐的工作频率 1MHz，当调制器处于低功耗采样模式时，推荐的时钟频率为 500KHz。

例 2.7.1：MSP430F425 单片机的 ACLK 为 32.768KHz，MCLK 和 SMCLK 被配置为 4.192MHz，为 SD16 模块配置 500KHz 左右的时钟。

ACLK 太低显然不适合做 SD16 的时钟。考虑到等待 ADC 转换完毕的过程中一直需要采样时钟，若选择 MCLK 作为时钟源则该过程中必须一直开启 MCLK。而 MCLK 也供 CPU 使用，因此等待转换完毕的过程中 CPU 无法进入任何低功耗模式。若选择 SMCLK 作为时钟源，等待转换完毕的过程可以让 CPU 进入 LPM0 休眠模式。对 SMCLK 进行 8 分频即可得到 502KHz 的时钟：

```
SD16CTL |= SD16SSEL_1 + SD16DIV_3;      //选择 SMCLK 作时钟，8 分频得到 502KHz
```

下一步需要为 SD16 设置基准源。SD16 的基准源可以由内部产生，可以对外输出，也可以从外部输入。和基准源相关的控制位如下：

- *SD16REFON：内部基准源开关 1=开启 0=关闭（位于 SD16CTL 寄存器）*

- *SD16MIDON：输出驱动器开关 1=开启 0=关闭（位于 SD16CTL 寄存器）*

- *SD16LP： SD16 低功耗模式 1=开启 0=关闭（位于 SD16CTL 寄存器）*

在 SD16 模块内部，集成了一个低功耗基准源和一个输出驱动器。当内部基准打开时，内部基准将向 3 个 ADC 提供 1.2V 基准电压，同时增加 200uA 左右的耗电，所以 ADC 采样结束后最好及时关闭基准源以省电。在某些应用中若内部基准源稳定度不能满足要求，可以关闭内部基准源，并使用外部基准源从 Vref 引脚向 MSP430 单片机提供基准，注意输入的基准电压必须在 1.0~1.5V 之间。

只要内部基准开启，基准电压也会对外输出，可以为外部其他的某些辅助测量电路提供基准源。但内部基准的负载能力有限，对外输出电流不能超过 200uA。若需要更大的输

出电流，则必须开启输出驱动器。输出驱动器开启后输出能力将达到 1mA。但输出驱动器自身会带来 400uA 左右的额外耗电。输出驱动器还要求至少 100nF 以上的外接电容，否则可能不稳定。

SD16LP 控制位用于开启 SD16 的低功耗工作模式。若该位置 1，SD16 的速度性能下降，换取更低的功耗。

例 2.7.2：SD16 模块采用内部基准源，并对外提供 1.2V 基准，外部负载约 3K 左右。

使用内部基准时 SD16REFON 必须打开，对外提供基准的负载电流大约为 1.2V/3K =400uA，超过了内部基准的输出能力，需要将 SD16MIDON 置 1 打开输出驱动器：

```
SD16CTL |= SD16REFON + SD16MIDON;        //选择内基准，并对外提供较强的输出能力
```

例 2.7.3：用 SD16 模块测量某压力传感器的输出，要求电压波动不影响精度并且尽量节省元器件成本。

压力传感器一般采用桥式电路，在金属梁上贴有 4 个对称的应变电阻片，当压力变化时金属梁发生形变，导致四个桥臂上的应变片电阻发生变化，输出电压差。但是压力桥输出幅度不仅正比于被测压力，也正比于与激励电压。因此为了得到准确的压力输出值，需要稳定且准确的激励电压。若增加稳压电路，将导致功耗和元件成本的上升。另一思路是从激励电压上分压出 1.2V 左右作为基准源，这样激励电压的变化将导致输出信号和基准源同时发生变化，两者变化比例相等。ADC 采样结果实际上是两者相除，电压的影响将被完全消除。

```
SD16CTL &=~ (SD16REFON + SD16MIDON);        //关闭内部基准，由外部提供基准源
```

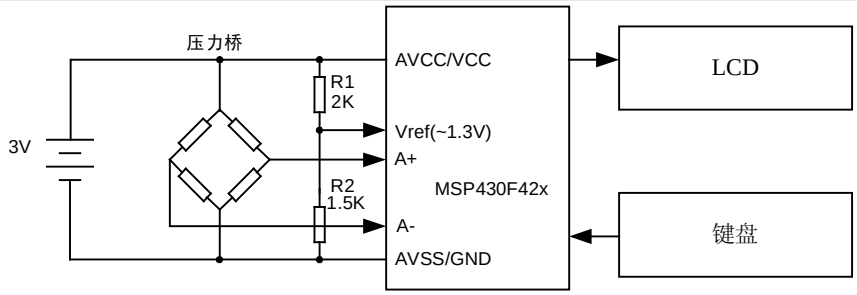


图 2.7.3　用比值测量法消除激励电压变化对压力测量的影响

这种方式被称为比值测量法（Ratiometric），对于输出正比于供电电压的各类传感器，都可以采用该方法消除电源电压变化带来的影响。

配置输入通道：

时钟与基准源配置后，可以开始使用 3 个 ADC，根据被测量的对象，选择输入通道与放大倍数。相关寄存器有：

- *SD16INCHx：输入通道选择　（位于 SD16INCTL0/1/2 寄存器）*
  000~101　：外部电压输入　110：温度传感器　111：0V（短路）
  快捷宏定义：SD16INCH_0　~　SD16INCH_7

- *SD16GAINx：PGA 增益选择　（位于 SD16INCTL0/1/2 寄存器）*
  000：1 倍（无放大）　001：2 倍　010：4 倍　011：8 倍

100：16 倍　　101：32 倍　　110、111：保留
快捷宏定义：SD16GAIN_1/2/4/8/16/32

- *INTDLYx：采样延迟选择　（位于 SD16INCTL0/1/2 寄存器）*
  00=4 个采样周期　01=3 个采样周期　10=2 个采样周期　11=1 个采样周期
  快捷宏定义：SD16INTDLY_0　SD16INTDLY_1　SD16INTDLY_2　SD16INTDLY_3

对于每个 ADC 来说，都可以通过 SD16INCHx 控制位来切换采样通道。其中通道 0~5 是外部输入（42x 系列中只有通道 0）、通道 6 是温度传感器、通道 7 是 0V（可用于校准零点）。若输入信号幅度太小，可以开启内部的可编程增益放大器（PGA），将信号预先放大后再进行采样。通过 SD16GAINx 控制位可以设置放大倍数 1~32 倍。例如图 2.7.3 中的压力传感器满量程只有几十毫伏的输出幅度，开启 16 倍增益即可放大到 ADC 的满输入幅度。内置 PGA 很大程度上简化了电路设计，无需在外部增加放大电路，同时也节省了功耗降低了成本。但内部 PGA 增益实际由开关电容的充放电率决定，不是真正的模拟电路，开启 PGA 会损失一定的有效分辨率，仅用于要求不高的场合。所以小信号高分辨率测量，或者计量级产品应用时仍建议使用外部的差分模拟放大器。

Sigma-Delta 型 ADC 的模拟量处理过程存在积分过程，数字量输出也需要数字滤波、平均、量化等过程，这些都是导致输出滞后于输入的因素。输出的滞后与平均对于抑制干扰有很大帮助，因为大部分干扰波形都是对称的，在积分平均的过程中能自相抵消。但在切换通道的过程中，会引入新的问题：一旦切换通道，也相当于输入突然跳动，需要一定的时间才能输出新的准确测量值。一般来说，切换通道将导致其后 1~4 次采样结果都是不准确的。可以通过软件去掉这些采样值，或者通过设置 INTDLYx 控制位，让 SD16 模块在发生变化后自动丢弃若干次采样值。若设置了 INTDLYx 控制位，只要可能引起输入突跳的因素，如通道号 SD16INCHx 改变、增益 SD16GAINx 改变、SD16SC 位发生变化，都会自动进行若干次空采样操作（不更新数据也不置结束标志）之后才开始正常采样。复位后默认值是 4 个周期，因此切换通道及增益后，第一次采样等待时间较长；单次采样每次都要操作 SD16SC 位，速度也较慢。

例 2.7.4：某仪器中使用 MSP430 单片机的 SD16 模块测量三种量：ADC0 采集某压力传感器的输出、ADC1 采集电池电压、ADC2 采集温度。为 SD16 配置通道寄存器：

考虑到压力传感器的输出幅度很小，按 30mV 的典型值计算，放大 16 倍后为 480mV，接近 ADC 满量程。电池电压可以经过分压后给 ADC1，无需放大。ADC2 应该选择通道 6，从内部的温度传感器获得输入。

```
SD16INCTL0 |= SD16INCH_0 + SD16GAIN_16;    // ADC0 从外部输入，放大 16 倍
SD16INCTL1 |= SD16INCH_0 + SD16GAIN_1;     // ADC1 从外部输入，放大 1 倍
SD16INCTL2 |= SD16INCH_6 + SD16GAIN_1;     // ADC2 采集温度，   放大 1 倍
```

配置 Sigma-Delta 调制器：

在 SD16 模块中，三个 ADC 均有独立的配置寄存器，能够各自工作在不同的模式下。图 2.7.2 中只画出了其中 ADC1 的内部框图，其余两个 ADC 有着与之完全相同的结构。常用的控制位有：

- *SD16OSRx：过采样率选择 （位于 SD16CCTL0/1/2 寄存器）*

  00：过采样率=256 01：过采样率=128 10：过采样率=64 11：过采样率=32

  快捷宏定义：SD16OSR_256 SD16OSR_128 SD16OSR_64 SD16OSR_32

当时钟一定时，过采样率越高，采样速度越慢，获得的有效分辨率越高；反之当降低过采样率时，能够提高采样速度但会损失有效分辨率。通过 SD16OSRx 控制位可以根据设计需要让 SD16 模块在速度和精度之间自由地选择。当过采样率被设定后，该 ADC 每次转换所需的时钟数等于过采样率。例如过采样率设为 256，那么每次采样与转换需要 256 个 ADC 时钟周期。

当过采样率为 256 时，SD16 模块才能达到标称的 16 位有效分辨率，若降低过采样率，16 位采样结果的末尾若干位将不停的跳动（量化噪声）而变得无意义。若通过某些软件算法人为继续提高过采样率，还可以获得超过 16 位的分辨率。

- *SD16DF： 数据格式 （位于 SD16CCTL0/1/2 寄存器）*

0=单极性(无符号二进制数) *1=双极性（有符号二进制数）*

一般来说，高精度的 ADC 都采用了对称差分输入结构，ADC 针对两个输入脚（A+和 A-）的电压差进行采样。由于干扰同时叠加在两个输入级上，相减后被抵消，因此差分输入的抗干扰能力很强。差分输入的另一优点在于能测量负压，即使两只引脚都处于正电压，只要 A-高于 A+，将会测出负数结果。这样整个电路中无需负电源，给设计带来了方便。对于正负结果，有两种表示方法，依靠 SD16DF 控制位来选择。

当 SD16DF=0 时，0V 输入时数字量输出 0x8000（32768），Vref/2 输入时数字量输出 0xffff（65535），-Vref/2 输入时数字量输出 0。

当 SD16DF=1 时，0V 输入时数字量输出 0，Vref/2 输入时数字量输出 0x7FFF（32767），-Vref/2 输入时数字量输出 0x8000（-32768）。

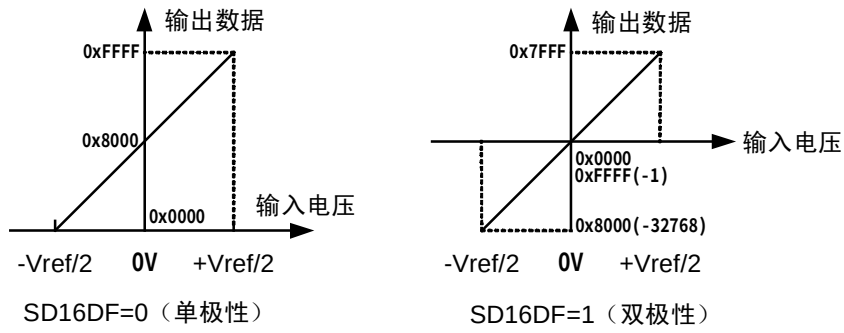


图 2.7.4 SD16 的两种数据输出格式

对于 C 语言来说，SD16DF=1 时，数据格式与 int 型变量完全相同，SD16DF=0 时，数据格式与 unsigned int 型变量相同，但是数据整体偏移了 0x8000。

对于使用内部 1.2V 基准来说，ADC 的量程将是-0.6~0.6V。在一般单端应用中，可以将 A-接地，这样只能测量正压，且损失一半的测量量程（只能测 0~+Vref/2 的电压），芯

片手册上给出每个引脚的最低电压极限能达到-0.3V，实测在单端应用中输入-0.2V~+0.6V时，ADC 都能达到精度指标。

ADC 采样:

与采样相关的控制器与寄存器有下列几组：

- *SD16SNGL： 采样方式 0=重复采样 1=单次采样 （位于 SD16CCTL0/1/2 寄存器）*
- *SD16SC： 开始采样 0=停止采样 1=开始采样 （位于 SD16CCTL0/1/2 寄存器）*
- *SD16IFG：采样结束标志 0=未结束 1=采样已完成（位于 SD16CCTL0/1/2 寄存器）*
- *SD16MEMx： ADCx 转换结果寄存器*

采样开始的命令通过 SD16SC 发出，之后 SD16 模块将自动开启相应的 Sigma-Delta 调制器进行采样。当采样结束时，相应的 SD16IFG 标志将被置 1。之后可以读取 SD16MEM 寄存器获得采样结果。每次读取 SD16MEM 寄存器后 SD16IFG 标志将会被自动清零，若不读取 SD16MEM 时也可以通过软件清除。

当 SD16SNGL=1 时，ADC 被配置为单次采样模式。每次采样结束后，SD16SC 会自动的被清除，采样过程也随之停止。当 SD16SNGL=1 时，ADC 被配置为连续采样模式，只要 SD16SC=1，采样过程结束后会自动启动下一次采样，直到软件清除 SD16SC 控制位为止。

例 2.7.5：用单次模式采集 ADC0 的电压输入值，存于变量 ADC_Result0 内。假设选择 2 分频后的 SMCLK 做时钟，内部基准源，PGA 增益=4，过采样率=256，数据格式为有符号二进制格式（双极性）。

```
int ADC_Result0;                                  //存放 ADC0 转换结果的变量
SD16CTL |= SD16REFON + SD16SSEL_1 + SD16DIV_1; //内基准，时钟=SMCLK/2
SD16INCTL0 |= SD16INCH_0 + SD16GAIN_4;     // ADC0 从外部输入，放大 4 倍
SD16CCTL0 |= SD16OSR_256 + SD16DF + SD16SNGL;
               //设置 ADC0 的过采样率=256，格式=有符号(双极性)，单次采样模式
SD16CCTL0 |= SD16SC;                              //发出开始采样指令
while((SD16CCTL0 & SD16IFG)==0);                  //等待 SD16IFG 标志变高(转换完毕)
ADC_Result0 = SD16MEM0; //读取 ADC0 的转换结果，将自动清除 SD16IFG 与 SD16SC 标志
```

例 2.7.6：用连续模式采集 ADC1 的电压输入值 128 次求平均，存于变量 ADC_Result1 内。假设选择 2 分频后的 SMCLK 做时钟，内部基准源，PGA 增益=1，过采样率=128，数据格式为有符号二进制格式（双极性）。

```
int  i,ADC_Result1;                               //存放 ADC0 转换结果的变量
long int ADC_Sum1;                                //暂存累计值的变量
SD16CTL |= SD16REFON + SD16SSEL_1 + SD16DIV_1; //内基准，时钟=SMCLK/2
SD16INCTL1 |= SD16INCH_0 + SD16GAIN_1;     // ADC1 从外部输入，放大 1 倍
SD16CCTL1 |= SD16OSR_128 + SD16DF;
               //设置 ADC1 的过采样率=128，格式=有符号(双极性)，默认连续采样模式
ADC_Sum1=0;                                       //清空累加变量
SD16CCTL1 |= SD16SC;                              //发出开始采样指令
```

```
  for(i=0;i<128;i++)                          //总共采集 128 次
    {
      while((SD16CCTL1 & SD16IFG)==0);        //等待 SD16IFG 标志变高(转换完毕)
      ADC_Sum1 += (int)SD16MEM1;//累加，读取 SD1MEM 将自动清除 SD16IFG 标志
     }
  SD16CCTL1 &=~ SD16SC;                       //停止采样
  ADC_Result1 = ADC_Sum1>>7; //计算平均值。右移 7 位相当于除以 128，但效率更高
```

由于 SD16 模块包含了三个独立 ADC，因此可以实现同时采样功能。但指令执行总有先后，只能依次向三个控制寄存器发出开始指令。为解决此类问题，SD16 模块为三个 ADC 提供了编组功能。该功能依靠 SD16GRP 控制位实现：

- *SD16GRP： ADC 编组控制位 （位于 SD16CCTL0/1/2 寄存器）*

  0=不参与编组　　1=与下一个 ADC 编为一组

当若干个 ADC 编组后，对下标最高的 ADC 的操作将等效于对所在组全部 ADC 操作。例如将 ADC0 的 SD16GRP 控制位置 1 后，ADC0 与 ADC1 编为一组，对 ADC1 的操作将全部同时作用在 ADC0 上。若将 ADC0 与 ADC1 的 SD16GRP 控制位都置 1，那么三个 ADC 被编为一组，对 ADC2 的操作将同时作用于三个 ADC。

例 2.7.7：用连续模式对三个 ADC 同时采集 128 次求平均，存于变量 ADC_Result[3]数组内。假设 ADC 时钟设为 SMCLK/2，使用内部基准，开启基准输出缓冲器。三个 ADC 均采集外部输入电压，1 倍放大。

```
  long int ADC_Sum[3];
  int ADC_Result[3];
  SD16CTL = SD16REFON + SD16VMIDON + SD16SSEL_1 + SD16DIV_1;
             // 开启内部 1.2V 基准源, 开启缓冲器，ADC 时钟选择为 SMCLK/2(524KHz)
  for (i = 0; i < 500; i++);                  // 略延迟，让基准电压稳定
  SD16INCTL0 |= SD16INCH_0+SD16GAIN_1;        // ADC0 输入选择为外部输入,1 倍放大
  SD16INCTL1 |= SD16INCH_0+SD16GAIN_1;        // ADC1 输入选择为外部输入,1 倍放大
  SD16INCTL2 |= SD16INCH_0+SD16GAIN_1;        // ADC2 输入选择为外部输入,1 倍放大
  SD16CCTL0 |= SD16DF + SD16GRP;              // ADC0 与 ADC1 编组,数据格式为有符号
  SD16CCTL1 |= SD16DF + SD16GRP;              // ADC1 与 ADC2 编组,数据格式为有符号
  SD16CCTL2 |= SD16DF + SD16IE;               // 打开 ADC2 中断,数据格式为有符号
              //ADC0/1/2 已经被编为同一组，对 ADC2 的操作将同时作用于 ADC0 与 ADC1
  SD16CCTL2 |= SD16SC;                        // 向 ADC2 发出"开始采样"命令
              //由于 ADC0/1/2 已经被编为一组，三个 ADC 将同时收到出"开始采样"命令
  ADC_Sum[0]=0; ADC_Sum[1]=0; ADC_Sum[2]=0;//清除累加值
  for(i=0;i<128;i++)                          //采样 128 次
   {
     while((SD16CCTL2 & SD16IFG)==0);//等待 ADC2 转换完毕(相当于等待全部采样完毕)
     ADC_Sum[0] += (int)SD16MEM0;             //ADC0 采样结果累加
     ADC_Sum[1] += (int)SD16MEM1;             //ADC1 采样结果累加
     ADC_Sum[2] += (int)SD16MEM2;             //ADC2 采样结果累加
   }
 SD16CCTL2 &=~ SD16SC;   // 向 ADC2 发出"停止采样"命令,相当于同时停止三个 ADC
```

```
ADC_Result[0]=ADC_Sum[0]>>7;
ADC_Result[1]=ADC_Sum[1]>>7;
ADC_Result[2]=ADC_Sum[2]>>7;          //求 128 次采样的平均值，右移效率比除法高
```

对于 ADC 采样的控制，还有下面几个不常用的控制位或寄存器：

- *SD16LSBACC： SD16MEMx 低 16 位访问控制位 （位于 SD16CCTL0/1/2 寄存器）*

  0：SD16MEMx 访问的是 ADCx 采样结果的高 16 位

  1：SD16MEMx 访问的是 ADCx 采样结果的低 16 位

在 SD16 模块中的三个 ADC 虽然是 16 位的，但是实际上数字平均及量化器具有 32 位的字长，过采样以及数字平均的结果将体现在高 16 位中，正常状态下，也就是 SD16MEM 所读取的结果，而低 16 位属于量化噪声对于测量来说无意义。

但在某些特殊的场合，可以利用该标志位分 2 次分别访问高 16 位和低 16 位来得到完整的 32 位量化结果。这样允许人为使用其他软件手段从量化噪声中获取额外的信息。例如可以将多次 32 位结果求平均，相当于获得更高的过采样率，从而使 ADC 的实际分辨率超过 16 位。

- *SD16LSBTOG： SD16ACC 取反控制位 （位于 SD16CCTL0/1/2 寄存器）*

  0：每次访问 SD16MEMx 后 SD16LSBACC 不变

  1：每次访问 SD16MEMx 后自动将 SD16LSBACC 取反

如果需要读出 32 位的量化结果，可以利用该标志位简化读取过程。将该标志位置 1 后，每次读取 SD16MEMx 后，都会自动将 SD16LSBACC 取反，这样只要对 SD16MEM 寄存器连续读 2 次，即可得到 32 位量化数据。

- *SD16PREx ： ADCx 的转换延迟（0~255 个 ADC 时钟周期）*

当多个 ADC 被编组后，开始采样控制位 SD16SC 置 1 将立即同时启动同组所有的 ADC 开始采样，同组的 ADC 也会同时转换完毕。若希望同组的 ADC 采样的开始时刻错开一定的时间差，可以利用该寄存器实现。当 SD16SC 置 1（发出开始采样指令）后，或 SD16PREx 寄存器被改写后的下一个转换周期将被延长 SD16PREx 个 ADC 时钟周期。之后的采样周期自动恢复为原值。

若将 3 个 ADC 并联，利用延迟功能可以提高 3 倍的采样速度。例如在过采样率设为 256 的情况下，每 256 个 ADC 时钟周期才能完成一次采样。如果将 3 个 ADC 的输入端并联采集同一个电压，并将 SD16PRE0 设为 0（ADC0 不延迟）、SD16PRE1 设为 85（ADC1 延迟 1/3 个转换周期）、SD16PRE2 设为 171（ADC2 延迟 2/3 个转换周期），那么第一次采样完毕之后，三个 ADC 的采样时间恰好错开 1/3 个转换周期，实际上将速度提高了 3 倍。这种用多个 ADC 来提高采样速率的方法被形象地称为“流水线采样法”。

l SD16 模块的中断

在上面的几个 ADC 采样的范例中，都采用了查询方式等待 ADC 采样与转换结束。由于 Sigma-Delta 型 ADC 的转换速度较慢，在等待的过程中关闭 CPU 节省部分功耗是有意

义的。参考 2.3 节给出的方法，可以在等待 ADC 转换结束的过程中将 CPU 休眠，再由 ADC 采样结束中断唤醒 CPU 读取转换结果，节省部分功耗。

与 SD16 模块中断相关的控制位与标志位有：

- *SD16IE： SD16 模块采样结束中断允许 （位于 SD16CCTL0/1/2 寄存器）*
  0=禁止采样结束中断 1=允许采样结束中断
- *SD16IFG： 采样结束标志 （位于 SD16CCTL0/1/2 寄存器）*
  0=采样未结束 1 =采样已完成
- *SD16OVIE： SD16 模块溢出（超量程）中断允许 （位于 SD16CCTL0/1/2 寄存器）*
  0=禁止溢出中断 1=允许溢出中断
- *SD16OVIFG： 溢出标志 （位于 SD16CCTL0/1/2 寄存器）*
  0=输入信号在测量范围内 1 =输入信号超量程，数据溢出
- *SD16IV： 中断向量寄存器。*

3 个独立的 ADC 中，每个 ADC 采样结束以及超量程都可以引发中断，所以总共有 6 个事件会引发 SD16 中断，这 6 个事件（中断源）共用了一个中断入口 SD16_VECTOR。需要在中断服务程序中通过软件判断 SD16IV 寄存器的值来确定具体的中断源。

表 2.7.1 SD16IV 寄存器值与中断标志位的关系

SD16IV 寄存器值	中断源	标志位	优先级
00H	无中断	-	-
02H	ADC 0~3 任一超量程	SD16CCTLx 寄存器中的 SD16OVIFG 标志位	最高 . . . . 最低
04H	ADC_0 采样完成	SD16CCTL0 寄存器中的 SD16IFG 标志位	
06H	ADC_1 采样完成	SD16CCTL1 寄存器中的 SD16IFG 标志位	
08H	ADC_2 采样完成	SD16CCTL2 寄存器中的 SD16IFG 标志位	

例如，当 ADC_2 采样完成时，SD16IFG 标志会被 SD16 模块自动置 1，SD16IV 寄存器的值也会变为 08H，若 ADC2 中断被允许（SD16CCTL2 中的 SD16IE=1）且总中断是开启状态，就会引发中断。在中断程序内判断 SD16IV 的值，知道是 ADC_2 采样完毕引发的中断，程序应该读取 SD16MEM2 的值。SD16MEM2 被读取后，SD16IFG 标志会被自动清除，SD16IV 恢复为 0。

当多个 SD16 中断同时产生时，SD16IV 会按照优先级顺序自动先处理优先级较高的中断。例如 ADC_2 和 ADC_1 都采样完毕时，SD16CCTL1/2 中的 SD16IFG 标志都被置 1。SD16IV 会优先处理 ADC_1 的中断，寄存器值变为 06H。当进入中断读取 SD16MEM1 后，

SD16CCTL1 中的 SD16IFG 标志被自动清除，SD16IV 变为 08H。中断结束之后，由于 SD16CCTL2 的 SD16IFG 标志仍为 1，还会再次引发 SD16 中断，读取 ADC_2 的转换结果。

对于 ADC 溢出的中断（SD16IV=02H），需要用 SD16OVIFG 去判断具体哪个 ADC 发生溢出，再做相应处理。

当只使用了一个 ADC 时，也可以不判断 SD16IV 的值，直接读取结果。

例 2.7.8：编写中断服务程，读取 3 个 ADC 的值，存于 ADC_Result[3]数组中；并在退出中断后唤醒 CPU。

```
/*********************************************************************
* 名    称：SD16ISR() ADC 采样结束产生的中断
* 功    能：保存 ADC 采样结果，并唤醒 CPU。
* 入口参数：无
* 出口参数：无
*********************************************************************/
 char ADC_Flag[3]={0,0,0};                  // 用于判断 SD16 中断已发生的标志位
 unsigned int ADC_Result[3];                // 存放采样结果的数组
  #pragma vector=SD16_VECTOR
  __interrupt void SD16ISR(void)            // 中断声明
  {
    switch (SD16IV)                         // 判断中断源
    {
      case 2:                               // SD16MEM 超量程
        break;                              // 不作处理
      case 4:                               // ADC0 采样结束
        ADC_Result[0]=SD16MEM0;             // 保存 ADC0 采样结果
        ADC_Flag[0]=1; break;               // 通知应用程序，中断已发生
      case 6:                                // ADC1 采样结束
        ADC_Result[1]=SD16MEM1;             // 保存 ADC1 采样结果
        ADC1_Flag[1]=1; break;              // 通知应用程序，中断已发生
      case 8:                                // ADC2 采样结束
        ADC_Result[2]=SD16MEM2;             // 保存 ADC2 采样结果
        ADC2_Flag[2]=1; break;              // 通知应用程序，中断已发生
    }
   __low_power_mode_off_on_exit();          // 退出中断后，唤醒 CPU
  }
```

有了中断服务程序，就可以将前面例子中所有的等待 SD16IFG 标志变 1 的程序替换为低功耗休眠。但要注意，SD16 模块一般都使用 SMCLK 作时钟源，CPU 只能进入 LPM0 休眠模式，其他休眠模式下 SMCLK 被关闭或不准确，SD16 模块将不能正常工作。还要注意休眠状态时，所有的中断都有可能唤醒 CPU，因此在软件中增加 3 个标志位 ADC_Flag[3]用于判断唤醒原因，若非 SD16 模块中断唤醒了 CPU，则继续回到低功耗休眠状态，只允许对应的 ADC 发生中断才能继续执行。

例 2.7.9：用单次模式采集 ADC0 的电压输入值，存于变量 Voltage 内。利用上例的中断服务程序，在采样过程中让 CPU 休眠以节省电力。（假设 SD16 模块参数已被设置妥）

```
  int Voltage;                      // 存放 ADC0 转换结果的变量
  SD16CCTL0 |=SD16IE;               // 允许 ADC0 的中断
```

```
  _EINT();                          // 开总中断
  ADC0_Flag=0;                      // 清除 ADC0 中断已发生软件标志
  SD16CCTL0 |= SD16SC;              // 发出开始采样指令
  while(ADC_Flag[0]==0)  LPM0;      // 休眠，且只有 ADC0 中断(转换完毕)能将 CPU 唤醒
  Voltage = (int)ADC_Result[0];     // 读取 ADC0 的转换结果(SD16MEM0 已被存于数组中)
```

例 2.7.10：利用连续转换模式编写单个 ADC 多次采样求平均函数 ADC16_Sample()，传入 2 个参数，第一个参数表示 ADC 编号。第二个参数表示平均次数。要求在采样过程中 CPU 休眠。（假设 SD16 模块参数已被设置妥，数据格式为有符号二进制格式）

```
/*****************************************************************
* 名    称：ADC16_Sample()
* 功    能：单个 ADC 采样函数。
* 入口参数：ADC_ID: 选择当前采样用的 ADC 编号(0~3):  0=ADC0  1=ADC1  2=ADC2
           AverageNum:采样平均次数(1~65535) 设为 1 即为单次采样。
* 出口参数：平均采样值。
* 范    例: val=ADC16_Sample(0,30); 返回 ADC0 连续采样 30 次的平均值,赋给 val
           val=ADC16_Sample(1,1) ; 返回 ADC1 单次采样值,赋给 val
*****************************************************************/
int ADC16_Sample(char ADC_ID,unsigned int AverageNum)
  {
    long int ADC_Sum=0;                 //累加值
    unsigned int *SD16CCTL;             //ADCx 控制寄存器选择指针
    int i;
    if(AverageNum==0) AverageNum=1;     //至少要采样 1 次
    switch(ADC_ID)                      //根据选择采样 ADC 编号决定指针指向的寄存器
     {
       case 0:SD16CCTL=(unsigned int *)&SD16CCTL0; //指针指向 ADC0 控制寄存器
              break;
       case 1:SD16CCTL=(unsigned int *)&SD16CCTL1; //指针指向 ADC1 控制寄存器
              break;
       case 2:SD16CCTL=(unsigned int *)&SD16CCTL2; //指针指向 ADC2 控制寄存器
              break;
     }
    *SD16CCTL |= SD16IE;                    //打开选中的 ADC 的中断
    _EINT();                                //开总中断
    *SD16CCTL |= SD16SC;                    //向选中的 ADC 发出"开始采样"命令
    for(i=0;i<AverageNum;i++)               //循环连续采样
    {
       while(ADC_Flag[ADC_ID]==0) LPM0;     //等待一次采样结束
       ADC_Flag[ADC_ID]=0;                  //清除软件标志
       ADC_Sum+=(int)ADC_Result[ADC];       //按有符号模式累加
    }                                       //采样次数达到
    *SD16CCTL &=~ SD16SC;                   //向选中的 ADC 发出"停止采样"命令
    *SD16CCTL &=~ SD16IE;                   //关闭相应 ADC 的中断
    return(ADC_Sum/AverageNum);             //求平均值
  }
```

程序中用了指针来操作寄存器。当需要选择多组寄存器中的一组进行设置时，使用指

针是一种常用的手段。

例 2.7.11：利用 SD16 模块提供的编组功能，对 3 个 ADC 同时采样。编写 3 个 ADC 同时多次采样求平均函数 ADC16_Sample3()，3 个采样结果通过指针返回，再用一个参数表示采样平均次数。要求在采样过程中 CPU 休眠。（假设 SD16 模块参数已被设置妥，数据格式为有符号二进制格式）

```
 /*************************************************************************
 * 名    称：ADC16_Sample3()
 * 功    能：三个 ADC 同时采样函数。
 * 入口参数：Result0: ADC0 采样结果存放地址
           Result1: ADC1 采样结果存放地址
           Result2: ADC2 采样结果存放地址
           AverageNum:采样平均次数(1~65535) 设为 1 即为单次采样。
  * 出口参数：无。（3 个采样结果通过 3 个指针返回）
  * 范    例：ADC16_Sample3(&a,&b,&c,30); 3 个 ADC 同时采样 30 次，采样结果的平均值
             存于 a,b,c 三个 int 型变量内。
*************************************************************************/
  void ADC16_Sample3( int *Result0, int *Result1,
                      int *Result2,unsigned int AverageNum)
  {
    long int ADC_Sum[3];              // 累加变量
    int i;
    if(AverageNum==0) AverageNum=1;   // 至少要采样 1 次
    for(i=0;i<3;i++){ADC_Sum[i]=0;}   // 累加值清零
    SD16CCTL0 |=SD16GRP;              // ADC0 编组
    SD16CCTL1 |=SD16GRP;              // ADC1 编组
        // ADC0/1/2 已经被编为同一组，对 ADC2 的操作将同时作用于 ADC0 与 ADC1
    SD16CCTL2 |=SD16IE;               // 开启 ADC2 中断
    _EINT();                          // 开总中断
    SD16CCTL2 |= SD16SC;              // 向 ADC0/1/2 同时发出"开始采样"命令
    for(i=0;i<AverageNum;i++)         // 循环连续采样 AverageNum 次
    {
      while(ADC_Flag[2]==0) LPM0;     // 等待一次采样结束(ADC2 结束时三个通道均
      ADC_Flag[2]=0;                  // 同时采样结束)，过程中 CPU 休眠。
      ADC_Sum[0] += (int)SD16MEM0;    // 按有符号模式累加
      ADC_Sum[1] += (int)SD16MEM1;    // 按有符号模式累加
      ADC_Sum[2] += (int)SD16MEM2;    // 按有符号模式累加
    }                                 // 采样次数达到
    SD16CCTL2 &=~ SD16SC;             // 向 ADC0/1/2 同时发出"停止采样"命令
    SD16CCTL2 &=~ SD16IE;             // 关闭 ADC2 中断
    SD16CCTL0 &=~ SD16GRP;            // 解除 ADC0 编组
    SD16CCTL1 &=~ SD16GRP;            // 解除 ADC1 编组
    *Result0=ADC_SumS[0]/AverageNum; // ADC0 采样结果的平均值
    *Result1=ADC_SumS[1]/AverageNum; // ADC1 采样结果的平均值
    *Result2=ADC_SumS[2]/AverageNum; // ADC2 采样结果的平均值
  }
```

由于三个 ADC 采样将有三个返回值。但对于 C 语言来说，函数只允许有一个返回值，

所以本例中通过三个指针传入函数，通过改变三个指针所指的变量值来实现三个数据的返回。这是一种常用的多值返回的方法。另一种方法是将三个变量定义为结构体，通过返回结构体来实现多个数据的返回，留给读者自行完成。

Ⅰ SD16 模块的电压测量应用

SD16 模块适用于各类高精度、高分辨率测量应用。加上差分输入结构以及内置的可编程增益放大器 PGA，具有很强的的共模抗干扰能力及小信号放大能力，特别适用于压力、称重等桥式传感器测量应用。

单端电压测量

直接测量电压是 ADC 最基本的应用之一。对于 SD16 模块来说，被测电压是 A+引脚与 A-引脚之间的电压差（差模电压）。ADC 的满量程输入电压为-Vref/2~ +Vref/2。同时必须满足 A+与 A-引脚都必须在-0.3V~VCC+0.3V 之间（共模电压），否则可能造成芯片损坏。如果在设计中将其中一只引脚（比如 A-）接到一个固定电压上，只改变另一只输入引脚的电压，称为单端测量模式。

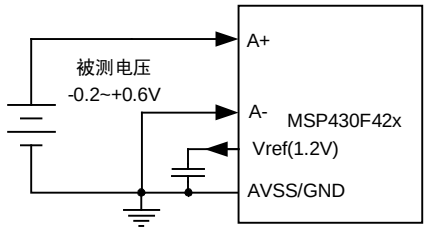


图 2.7.5a 单端测量电压

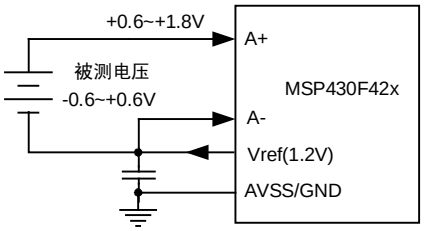


图 2.7.5b 正负量程单端测量电压

图 a 中，将 A-端接到 0V 电压（GND），从 A+引脚输入被测电压。由于芯片规定了任何引脚电压不能低于-0.3V，为安全起见，使用这种接法时 A+引脚的电压不能低于-0.2V。牺牲了一部分量程范围。

图 b 中，将 A-端接到 1.2V 电压（Vref），从 A+引脚输入被测电压。当输入电压在-0.6V~+0.6V 满测量范围内变化时，A+端的电压将在+0.6V~+1.8V 范围内，符合芯片的共模输入电压要求。该电路的特点是能测量满幅的正负电压，缺点是输入电压不能与单片机共地。单片机的电源必须与被测电压隔离。

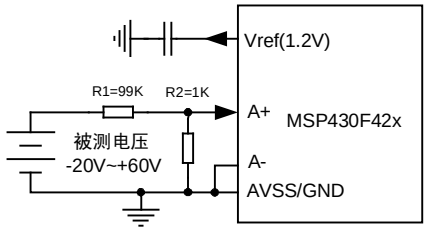


图 2.7.5c 单端测量高电压

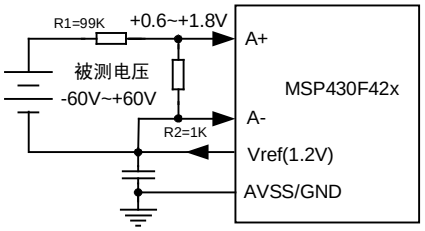


图 2.7.5d 正负量程单端测量高电压

图 c 在图 a 的基础上利用电阻分压扩大了 100 倍输入量程。图 d 在图 b 的基础上利用电阻分压扩大了 100 倍量程并保证了正负满量程输入。

上面 4 种测量电路中，量程都包含了负压。若只测量正电压时，相当于分辨率下降 1 倍（只有 15bit 有效读数）。很多应用中都会出现只测正电压的应用，可以使用图 e 的电路。A-被接到 Vref/2 上，输入电压缩小 2 倍后加到 A+上。相当于当被测电压为 0V~Vref 变化时，A+与 A-间的电压差为-Vref/2~+Vref/2，充分利用了 ADC 的 16bit 正负满量程。改变 R1 与 R2 的比值可以获得任何大小的对地正电压量程。

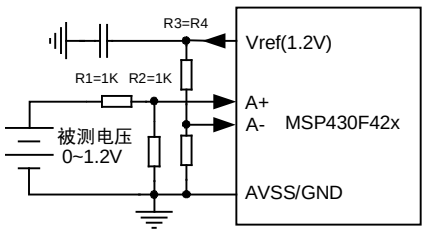


图 2.7.5e 对地正电压测量

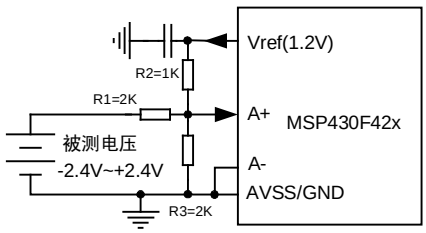


图 2.7.5f 对地正负电压测量

对于图 b 与图 d 中的正负电压测量，被测电压不能与单片机共地的缺点可以利用图 f 的电阻网络来解决。对于-2.4V~+2.4V 的输入，图中的三电阻网络将其变成 0~0.6V 的正电压。通过改变三个电阻的比例可以将任何正负电压量程都转成正电压。该电路同样存在损失一半量程的缺点。如果将图 e 中的 R2 接地端改为接 Vref，也可以实现对地正负电压测量，且不损失量程，如图 g 所示。改变 R1 与 R2 的比例同样能获得任意输入量程范围。

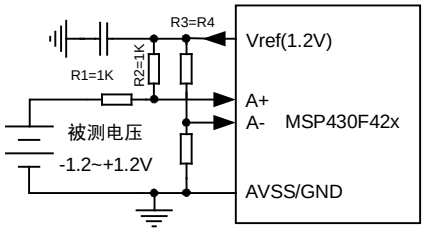


图 2.7.5g 不损失量程的对地正负电压测量

从上面的几个典型应用中可以看出，ADC 的差分输入结构为电路设计带来了很大的灵活性。不需要使用运算放大器，只需要电阻网络就能实现电平的搬移。读者在进行测量电路设计时遇到需要将电平搬移、抬升的情况，也应尽可能的利用差分端和电阻网络来实现，避免使用运放。以节省功耗、降低成本、且避免运放的失调、温度漂移等诸多问题。

对于 SD16 模块来说，只要对外输出 Vref，在 Vref 引脚外接一只 0.1uF 以上的滤波电容都是必不可少的。

差分电压测量

上面各例的测量电路中，两个输入端的输入阻抗不一致，因此无法发挥差分输入级抗干扰特性。以图 c 为例，假设整个电路置于一定强度的空间电磁干扰环境下，干扰对 A-端无影响（A-接地）；而干扰对 A+端的影响被 ADC 采集了。

如果在电路设计上使 A+与 A-的电压大小相反的变化，则称为差分测量方式。在差分测量方式下，若再将被测的电压源也设计成对 2 个引脚输入阻抗相等（电路对称），传输路径也设计成完全一致（例如用双绞线），那么空间中电磁干扰对两个引脚的影响相同，

相减后完全抵消，会具有极高的抗干扰能力。

图 2.7.6 是利用差分模式测量压力传感器输出的例子。当压力传感器受力时，R1 与 R4 变大，R3 与 R2 变小，电阻桥失衡产生很小的（数十 mV 级）电压差。该电压差正比于压力，因此可以通过测量电压差计算压力值。

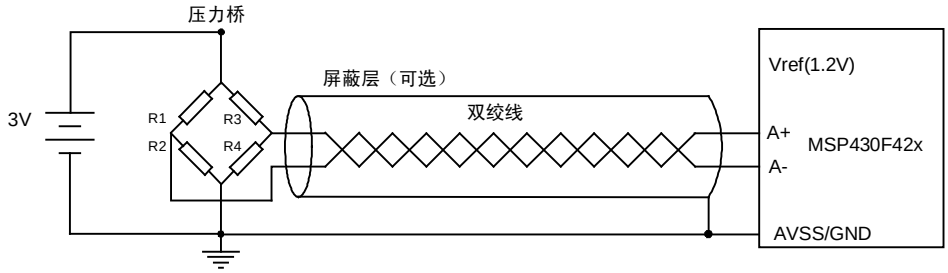


图 2.7.6 压力传感器的差分测量

传感器输出的电压差很小，且源阻抗较高，容易受到电磁干扰的影响。但由于该电路在结构上是完全对称的，信号的传输路径也是完全对称的，那么干扰对两根输入端的干扰是完全相同的。ADC 的差分输入级将两个信号相减后得到无干扰的原始信号。如果在双绞线上增加一个屏蔽层并接地，会有更好的效果。

需要注意的是，与单端方式一样，差分方式下，除了满足差分电压范围要求之外，也必须满足两个输入端的共模范围（-0.3V~VCC+0.3V）要求。图 2.7.6 中的压力传感器输出的两根信号线电压都在 VCC/2 附近，满足共模电压范围的要求。而下图 a 中的电路是无法工作的，因为 A+与 A-的共模电压无法确定。图 b 中的电路本意是希望通过一个小电阻上的压降来测量某 5V 系统的电源耗电，但 A+与 A-都远超出了 430 单片机的电源电压，且无限流保护电阻，会直接烧毁 430 单片机的输入引脚。

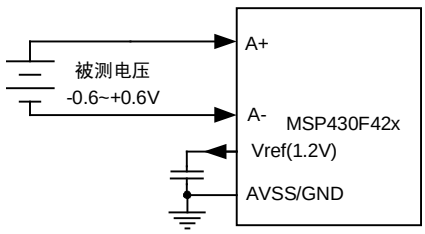


图 2.7.7a 错误的差分测量电路

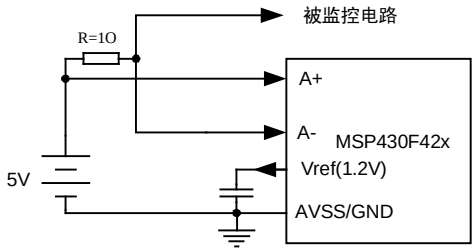


图 2.7.7b 错误的电流监测电路

一般来说，习惯将对地的信号称为单端信号，将正负对称的信号称为差分信号。但严格的说，差分信号与单端信号的最本质区别是源阻抗一致而非信号幅度相反。因为只有源阻抗相等的信号源，受同样强度的干扰时，产生的影响才会相同，在差分输入级才能被相减抵消。例如图 2.7.8a 中的电路，从三极管 E 和 C 级会输出大小相等但幅度相反的信号，看似一个差分信号，实际上三极管射级输出阻抗远低于集电极输出阻抗，同样的电磁干扰强度对 C 极信号的影响远大于对 E 极输出信号的影响，因此不是差分信号。图 2.7.9b 中的电路是一种常用的单端转差分电路，若运放的型号相同，则输出阻抗相同（且一般都很低），受干扰后的影响将相等，是差分信号。而且即使两个运放电路的放大倍数不同，导致输出

幅度不等，因为输出阻抗相等，仍然不改变其抗干扰能力。

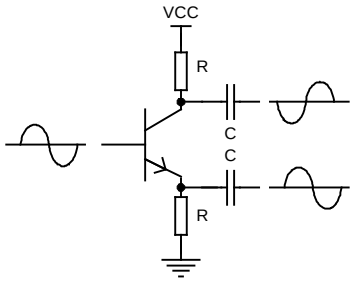


图 2.7.8a 非差分信号

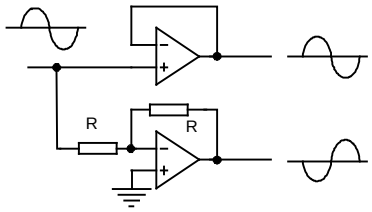


图 2.7.8b 差分信号

同样的，只有输入阻抗相等，且能实现相减的接收电路，才能叫做差分输入电路。SD16 模块内部的结构是完全对称的，共模抑制比达到 85dB（将共模电压衰减 17000 倍）以上（PGA 增益增加时会略有下降）。所以遇到高阻抗、微弱信号需要传输较远距离时，最好先转成差分信号后再传输。

比值（Ratiometric）测量

ADC 的测量本质是计算输入信号 Vin 与参考电压 Vref 的比值。因此利用 ADC 可以实现某些除法（比值、比例）计算功能。

图 2.7.9 是一种比值法测量电阻值的电路。图中 Rx 是被测电阻，R 是标准电阻，Rs 是保护电阻。根据电阻的定义，Rx=Ux/Ix。其中 Rx 是被测电阻，Ux 是被测电阻两端电压，Ix 是流过该电阻的电流。

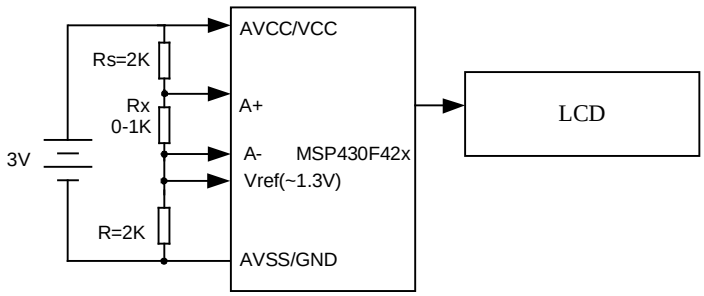


图 2.7.9 用比值法测量电阻值

因 ADC 的输入阻抗较高，假设 ADC 输入端吸收的电流可以忽略，那么 Rx 上的电流全部流过标准电阻 R，因此 ADC 的基准电压：

$$Vref = Ix \times R = (Ux / Rx) \times R \qquad (2.7.1)$$

ADC 的输入电压 Vin 等于 Rx 两端的电压 Ux：

$$Vin = Ux \qquad (2.7.2)$$

对于 SD16 模块，Vref/2 对应满量程，因此经过 ADC 采样后，得到数字量是二者的比值：

$$D = 2^{16} \times Vin / Vref \quad = 2^{16} \times UxRx / UxR$$

$$= 2^{16} \times Rx / R \qquad (2.7.3)$$

即： $$Rx = D \times R / 2^{16} \qquad (2.7.4)$$

取 R=2000 欧姆，即可精确测量 0~1000 欧姆的待测电阻，并获得 15bit 的分辨率（分辨至 0.03 欧）。当改变标准电阻 R 时，相当于改变电阻测量的量程。当 R 较大时，ADC 的阻抗不能忽略，可以将 Vin 与 Vref 通过运放跟随后输入 ADC。Rs 的作用是保证 Vref 的范围在 1~1.5V 之间，以符合 SD16 模块基准电压范围的要求。

在这个例子中，Rx 变化时或者供电电压变化时，被测电阻上的电压 Ux 是变化的，但通过 ADC 的比例运算将 Ux 相除约去。所以，通过比值法可以利用 ADC 的比例运算功能将某些不确定因素相除消去。因此利用比值法测量，可以无需提供稳定的激励量的电路，从而降低成本与功耗。一般电阻测量都用在被测电阻上加恒流源，测量端电压。用比值法避免了恒流激励电路，因为激励电流值在除法中被消去，不需要稳定电流一样能获得高精度。在图 2.7.3 中出现的比值法测压力的例子中，同样利用比值法消除了压力传感器激励电压波动的影响。

作为数值输入设备

一般的菜单中都会使用按钮来输入数据。例如留有 0~9 数字键盘，或者+/-键。对于某些需要凭经验调整的情况都不方便。例如对于搅拌机电机速度的调节，有经验的工人可以凭手感通过旋转调速器的旋钮调到最合适的速度。如果操作界面留的是按钮，让工人输入速度值反而会让操作员感到不适应。

利用电位器和 ADC 可以设计出通过旋钮输入数值的方法。下图中，Vref 被 R1 与电位器分压后得到 Vref/2 的电压，当电位器推至最下方位置时，ADC 采样值最小，电位器推至最上方时，ADC 采样值最大。将 ADC 值转换成电动机的速度指令，发送给电动机调速电路，从而实现调速功能。在调节搅拌速度、调节亮度等需要经验操作的场合，使用电位器比键盘具有更亲切的操作感。

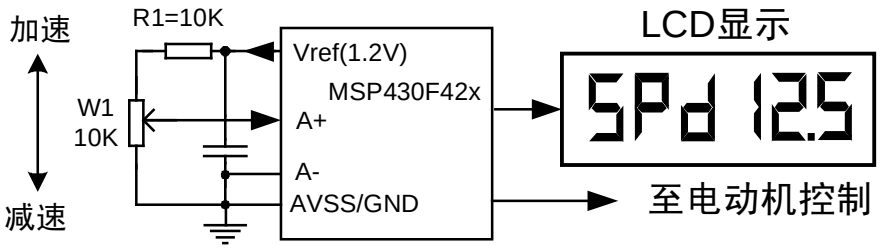


图 2.7.10　用 ADC 作为电机速度输入值

电位器输入比起键盘输入的缺点是不容易输入精确的数值。遇到需要精确输入的情况可以利用两个电位器，一只做粗调，另一只作为微调。

如果将两只电位器按照 X-Y 位置放置，可以做出类似“飞行摇杆”的控制输入装置，

例如对显微镜载玻片在水平面上位置控制。

电源检测

利用 ADC 可以测量出系统自身的电池电压。通过测出的电压值还可以估算出电池的剩余电量、剩余使用时间等参数。当检测到电源电压不足时，可以采取保存数据、关机等紧急处理手段。

Ⅰ SD16 模块的误差及校准

任何测量系统均存在误差。对于 SD16 模块来说，虽然属于高精度 ADC，但它和大部分 ADC 一样，在出厂时的初始误差是很大的。必须通过一定的校准手段后，才能实现精确测量。

基本概念

在了解误差之前，先明确几个易混淆的基本概念。首先，精度与分辨率是两个完全不同的概念。精度表示测量值与真是值之间的差异大小，分辨率表示最小能分辨的刻度大小。分辨率并不决定精度。也并非分辨率越高的测量系统其精度就越高。

例如一把 1 米的钢尺 A，刻度是 1cm，经过与标准尺比对，发现所有刻度的误差最大是 10um。我们说尺子 A 的精度是 10 微米，分辨率是 1 厘米。另一把皮尺 B 刻度是 1mm，经过与标准尺比对，发现所有刻度的误差最大是 100um。我们说尺子 B 的精度是 100 微米，分辨率是 1 毫米。A 尺的精度高于 B 尺，但 B 尺分辨率高于 A 尺。用 A 尺可以更精确地测量，用 B 尺可以分辨出更小的差异。例如用 A 尺测量并加工 10cm 的工件，比用 B 尺加工的更准确，但用 B 尺可以加工 10.5cm 的工件而 A 尺则不行。分辨率仅表示能分辨更小的差异而已，而并不表示测量精度。

对于 ADC 也是如此，一个 16 位 ADC，若测量误差达到 1%，精度不如一只准确的 8 位 ADC（最大测量误差 1/256）。但即使 16 位 ADC 的误差再大，它仍能分辨更小的电压变化，8 位 ADC 则不行。

第二个容易混淆的概念是元件稳定度与精度。一般来说，各种元器件参数都只保证稳定而并不保证精确。例如一般电阻的误差 5%，等级较高的也只有 1%精度。如果用 1%精度电阻构成分压网络，那么将导致至少 1%的分压误差。

同样的，对于 ADC 的基准源，一般来说 1.2V，但是手册上给出的误差范围是 1.14V~1.26V 之间。也就是说 ADC 的基准电压存在 5%误差。

ADC 内部的可编程增益放大器也同样存在误差，例如设置增益=1 时，手册上给出实际增益可能在 0.97~1.02 之间，折合约 3%的误差。

这些误差都是批次性误差，一旦元器件选定，这些误差值就被固定下来了。通过校准可以将其消除。只要元件的特性不改变，就不会带来新的误差。所以在设计中更加关心的并不是元件精度，而是元件的稳定度。

对于电阻，误差并不重要。因为校准后，只要电阻值不随环境温度变化，就不会引入新的误差。在决定精度的场合一般选用金属膜电阻，温度系数在 20ppm/℃以下。而且分压电阻的温度系数可以自相抵消。

对于内部基准源，手册上给出其温漂指标是 20ppm/℃，若不能满足要求，应该从外部提供更加稳定的基准源。

所以选择元件时首先要确定设备的工作温度范围与精度要求。例如用 MSP430 单片机设计的某仪器只在室温下使用，按 0~40 度考虑，内部基准源会变化 800ppm，带来 0.08%的误差。如果仪器本身精度等级为 0.5%，那么温度带来的误差基本可以不考虑。

再如用 MSP430 单片机设计的工业级计量设备，需要在-40~85℃工作，内部基准源会变化 2500ppm，带来 0.25%的误差。如果仪器本身精度等级为 0.1%，那么基准变化所带来的误差已大于仪器设计精度了，需要选用温飘更低的外部基准源，例如 MC1403（10ppm/℃）、LM399（1 ppm/℃）等。

SD16 模块的误差来源

ADC 的误差一般分为下面 5 种：

1. **零点误差（偏移误差）。**由于半导体工艺的对称性问题，差分输入级在实际输入 0V 的时候，输出可能不为 0。反映到 ADC 及测量结果上就是一个固定的偏移量。手册上给出 SD16 模块的零点偏差最大为 0.2%左右，折合约 130 个字的读数。在图 2.7.5.efg 以及 2.7.6 的电路中，如果桥壁电阻比例不等，也会造成零点偏移。记下零点偏移量，再从每次采样结果中扣除该值即可消除零点误差。SD16 模块中的通道 7（内部短路）就是用于测量 ADC 零点误差的。也可以通过在外部电路中增加人为调整措施（如调零电位器）来调整零点。
2. **增益误差。**基准电压误差、内部 PGA 增益误差、分压电阻误差都会导致增益误差。这也是最主要的误差来源。增益误差可以通过标定来消除：给定一个已知标准的输入，记下测量值，然后算出真实的增益记录下来。以后将真实的增益乘以采样数值即可得到真实的输入电压值。也可以通过在外部电路中增加人为调整措施（如调节增益的电位器）来调整增益。
3. **随机误差。**有时受到干扰、电路设计不良、信号源波动都可能造成 ADC 读数的跳动。通过改善电路、增加屏蔽措施、差分输入、增大信号的滤波时常数等措施都能较少随机误差。另外，随机误差在数值统计平均值为 0，因此也可以通过将多次采样结果求平均减少随机误差。
4. **漂移误差。**对于零点误差与增益误差来说，即使校准之后，也可能随温度与时间漂移。可以通过选用更低温度系数指标的元器件来降低温度漂移误差。通过出厂前的老化来降低元件随时间漂移的可能。
5. **非线性误差。**理论上 ADC 的读数与输入电压之间的关系应该是线性的。但实际中由于各种非理想因素，如半导体特性在不同电压下特性不一致、信号调理电路元件存在非线性、被测的传感器输出与测量物理量之间存在非线性等，都可能导致非线性误差。非线性误差可以通过分段校准、曲线拟合等方法消除。

SD16 模块的校准方法

如果 SD16 模块不经校准，其测量误差将可能达到 5%以上。这将使 16 位测量结果变得毫无意义。传统的校准方法一般通过电路进行，在电路设计中留出调节零点与增益的两只电位器，通过调节电位器消除零点偏移与增益误差。因为电位器存在接触点、旋转臂等

机械结构，电位器的阻值容易因为老化、震动、跌落等原因而变化。所以这种方法电路复杂、成本较高且稳定性不高。

SD16 模块与 CPU、存储器处于同一芯片系统，使得用数字校准方法更为容易。对于零点误差和增益误差来说，并不破坏输入电压与 ADC 读数之间的线性。由于只需要两点即可确定一条直线，因此只需要校准 2 个点，即可消除零点误差和增益误差。在输入量程范围内任意选择两个校准点 A 和 B，假设 A 点对应输入量 X1，ADC 采样值 D1；B 点对应输入量 X2，ADC 采样值 D2，那么对于采样值为 D 时，实际输入值为：

$$X = (X_2 - X_1)(D - D_1)/(D_2 - D_1) \qquad (2.7.5)$$

一般来说两个校准点应尽量远以保证校准效果。以图 2.7.11 的电压测量系统为例：将输入电压衰减 100 倍左右，从而将量程扩大 100 倍，图中的 SD16 模块能测量-20V~+60V 的输入电压。

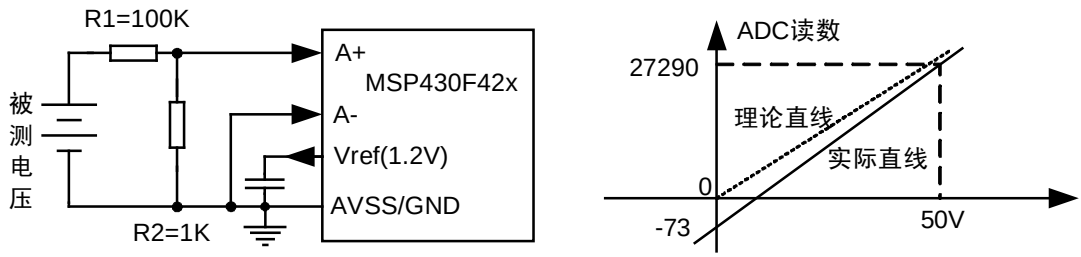


图 2.7.11 电压测量系统校准示意图

因为 ADC 存在零点误差，当输入 0V 时，实际 ADC 读数不为 0。同样，因为电阻误差、基准源误差、PGA 增益误差等，导致实际增益（直线斜率）也与理论值有偏差。取 0V 和 50.00V 两个点进行校准。假设输入 0V 时 ADC 读数为-73；输入 50V 电压时 ADC 读数为 27290。在直角坐标系中取（0,-73）与（50,27290）两点连一条直线，就是该测量系统的实际的输入电压与 ADC 读数之间的关系。对于任一 ADC 读数 D，有被测电压：

$$Ux = 50\times(D-(-73))/(27290-(-73)) = 50\times(D+73)/27363 \qquad (2.7.6)$$

校准的过程可以通过编写一段程序自动完成。例如进入校准界面之后，先提示用户将输入端短路（输入 0V）后按确定键。确定键被按下后将 ADC 的采样值 D1 记录下来存入 Flash 内。再提示用户输入 50.00V 已知的标准电压源后按确定键，确定键被按下后将 ADC 的采样值 D2 记录下来存入 Flash 内。之后按照 $Ux = 50\times(D-D1)/(D2-D1)$ 来计算被测电压值。

对于其他物理量的测量也可以用类似的方法，将所有环节的误差都消除。对于图 2.7.6 中压力测量的例子，假设压力传感器量程为 0~1Mpa，可以提示用户输入 0Mpa 和 1Mpa 的标准气压，记录下 ADC 读数，再通过两点直线公式将压力传感器本身的误差消除。

对于存在非线性的传感器或测量系统，可以校准多个点，再通过多段直线来减少非线性误差。

I SD16 模块的超低功耗应用

对于一个完整的测量系统来说，SD16 模块在测量过程中，下列部件都会引起功耗：

表 2.7.2 测量系统中各部件的功耗

部件	功耗（uA）	备注
基准源	175~260	SD16REFON=1
输出缓冲器	385~600	SD16MIDON=1
基准源外部负载	最大 1000	由外部负载决定
ADC	600~1550	PGA 增益越高越耗电 采样结束会自动关闭
CPU	400uA/MHz	保持活动
	100uA	进入 LPM0
信号调理电路	数百 uA~数 mA	由具体电路决定
传感器	数百 uA~数 mA	由具体电路决定

与前面各种内部设备数 uA 的功耗相比，SD16 模块中任何一个部件的功耗都是较大的。若将 ADC 相关的部件一直开启，总功耗可能达到毫安级。对于一节 200mAh 容量的纽扣电池来说，数天即可将电量耗尽。所以 ADC 采样部分必须间歇性工作，并对传感器、信号调理等外部电路进行电源管理。

例 2.7.11：以图 2.7.11 的电压测量系统为例，若对输入信号连续地采集，基准源、ADC、CPU 三个部分的功耗至少 900uA。假设电压采集的结果显示在 LCD 上供用户查看。那么连续的采集反而导致显示数字快速变化，看不清楚。采样速度降到每秒采样 2 次到 3 次已经足够，且能够明现地降低功耗。让 CPU 休眠在 LPM3 模式，用 BasicTimer 每 1/2 秒唤醒 CPU 进行采样及处理工作。在采样前才打开基准源，采样后及时关闭基准源：

```
#define ADC_0 (-73)
#define ADC_F  27290
#define VCAL   5000                              /*实测的校准参数*/
...
//-----------以下代码需每 1/2 秒运行一次--------------
   SD16CTL |=  SD16REFON;                        //开启内部基准源
   for(i=0;i<30;i++);                           //略等待 100us 以上，等基准稳定
   ADC_Result=ADC16_Sample(0,1);                //采样 ADC0,单次采样（例 2.7.10）
   SD16CTL &=~  SD16REFON;                      //关闭基准源
   Voltage=(ADC_Result-ADC_0)*(long int)VCAL/(ADC_F-ADC_0);//计算电压
   LCD_DisplayDecimal(Voltage,2);               //显示电压值，带 2 位小数 （例 2.5.10）
   LCD_InsertChar(VV);                          // 尾部添加单位：V
```

由于基准源从打开到稳定需要一段时间（手册上给出指标是 100us 以上），所以在开启基准源后需要略延迟才能进行采样。

取 SD16 时钟=500KHz、过采样率 OSR=256，那么每次 ADC 转换需要 500us 左右。由于第一次转换之前 SD16 模块会自动进行 4 次空采样操作，所以总共大约需要 2.5ms 时间。按照采样过程耗电 900uA、其余时间功耗 3uA（LCD 显示的耗电）计算，平均功耗为：

$I = (900uA \times 2.5ms + 3uA \times 497.5ms) / 500ms = 7.48uA$。这个值远小于 900uA。对于一节 200mAh 纽扣电池来说，能连续工作 3 年。

对于图 2.7.10 的例子，除了开启基准源以外，还必须开启基准源的输出缓冲器。同样的在采样结束之后要及时关闭基准源及缓冲器。使用缓冲器后，输出能力增强，在 Vref 引脚外接电容相同的情况下，基准稳定所需时间也会变短。

对于有外部调理电路以及外接有传感器的情况，外围电路的耗电也可能达到数毫安。有必要用一个开关对外围电路的电源进行控制，让外围电路也间歇性工作才能实现低功耗运行。在单片机 ADC 输入引脚的附近，一般都会安排一个 IO 口引脚。用这根引脚作为电源控制比较方便布线。例如 42x 系列单片机在 ADC 引脚的旁边留了 P2.2 口，可以用它控制外围电路的电源。

当外围电路耗电较小时（3mA 以内），可以直接用 IO 口输出高电平对外供电。当外围电路功耗较大时，可以将多个 IO 并联供电，或用一片 74HC 系列的驱动器（125、244、245、373、573 等）扩流后对外供电。由于 74HC 系列 CMOS 驱动器自身功耗不到 1uA，可以忽略不计。当模拟部分需要更大的电流或者工作电压高于单片机 IO 输出电压时，可以自行设计其他形式的电源开关。

例 2.7.12：以图 2.7.6 的压力测量系统为例，一般的压力桥传感器内阻约 1K 左右，3V 供电时耗电约 3mA 左右。若不对其电源进行控制，传感器的功耗将远大于单片机系统的功耗。三天即可耗尽一节 200mAh 的纽扣电池。降低功耗的方法是将压力桥的激励电源从 VCC 改为 P2.2 口，通过控制 P2.2 口的电平高低即可控制传感器的电源间歇性开启：

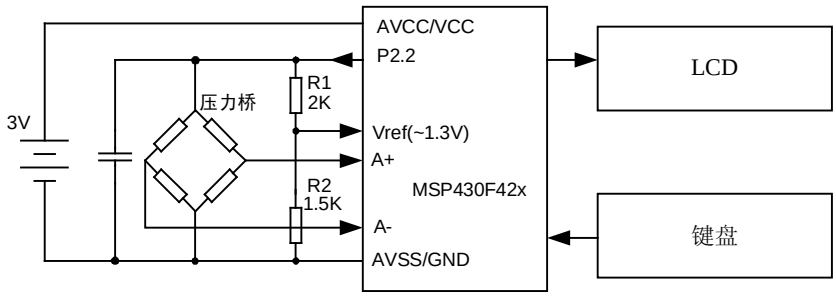


图 2.7.12　用 IO 口作为电源控制

在程序中，也应该增加相关的控制语句：

```
   ...
//-----------以下代码需每 1/2 秒运行一次--------------
   P2OUT |= BIT2;                          // 打开外部电路的电源
   for(i=0;i<100;i++);                     // 略等数百微秒以上，等电路稳定
   ADC_Result=ADC16_Sample(0,1);           // 采样 ADC0,单次采样（例 2.7.10）
   P2OUT &=~BIT2;                          // 关闭外部电路的电源
   ...                                     // 计算与显示
```

按照测量过程耗电 4mA、其余时间功耗 3uA（LCD 显示的耗电）计算，平均功耗为：$I = (4000uA \times 2.5ms + 3uA \times 497.5ms) / 500ms = 23uA$。对于一节 200mAh 纽扣电池来说，能连续工作 1 年时间。

SD16 模块的高精度应用

SD16 模块本身的性能是非常优秀的。在高精度应用中，外围电路的设计很大程度上决定了系统的总性能。特别是接地、布线等细节问题变得十分重要。

提高 SD16 模块分辨率的方法

SD16 模块本身的最高的有效分辨率是 16bit（过采样率=256，PGA 增益=1 时）。通过 Sigma-Delta 型 ADC 的原理分析可知，如果能够进一步增加过采样率，就能继续提高其分辨率。在测量钉子长度的例子中，连续随机测量 1000 次，求平均可以得到接近 1.500cm 附近的结果，如果重复多次，结果必然是在 1.500 附近随机波动的。假设 1.50 三位结果基本不变，把它称为“量化结果”，而波动的最后 1 位尾数称为“量化噪声”。如果不舍去量化噪声，将多次测量结果求平均，那么相当于更高的过采样率，能获得更高的分辨率。假设规定每次测量最多只能扔 1000 次钉子（相当于 SD16 的过采样率受到上限的限制），通过测量 10 次则相当于扔了 10000 次，能够获得平均 1.5000cm 的结果，分辨率提高了 10 倍。

在 SD16 模块中也用了类似的量化手段，32bit 量化结果中，高 16bit 是稳定的，将其存于 SD16MEM 内，低 16bit 每次是波动的，作为量化噪声被舍去了。但是通过 SD16LSBACC 控制字可以访问 32bit 全部的量化结果，如果人为再做平均滤波，可以从噪声中获得更高的分辨率。

例 2.7.13：编写采样函数，采样 ADC0 的输入，得到 18 位的分辨率的输出。

```
 long int ADC18_Sample()          /*ADC0 采样 128 次平均得到 18 位结果*/
 {
  long int var,sum=0;
  unsigned int i;
  SD16CCTL0 |= SD16SC;            //开始采集
  for(i=0;i<128;i++)              //总共采样 128 次
   {
     ADCFlag0=0;
     while(ADCFlag0==0)  LPM0;    //等待一次结束。CPU 在等待过程中休眠，省电
     SD16CCTL0 &= ~SD16LSBACC;    //清除 SD16LSBACC，SD16MEM 取高 16 位量化结果
     var = (long)SD16MEM0 << 16;  //读取高 16 位结果，放在 var 的高 16 位中
     SD16CCTL0 |= SD16LSBACC;     //置位 SD16LSBACC，SD16MEM 取低 16 位量化结果
     var |= SD16MEM0;             //读取低 16 位结果，放在 var 的低 16 位中
     var >>= 12;                  //将 var 保存的 32 位量化结果舍掉 12 位，变 20 位
     sum+=var;                    //累加(128 次)
    }
 SD16CCTL0 &=~SD16SC;             //停止采集
 sum/=128;                        //累加和除 128，求平均，去掉量化噪声
 return(sum>>2);                  //返回 18 位采样结果
}
```

由于直接对 32 比特数据累加容易溢出，所以先应该适当舍去末尾无意义的噪声。为了得到 18 比特的采样结果，运算前多保留了 2 比特数据（共 20 比特）。其中 16 比特是有效结果，4 比特噪声，通过 128 次平均，将噪声消除，获得噪声的均值。

受到 ADC 信噪比的限制，该方法提高分辨率的极限只能达到 18 比特左右。

Sigma-Delta 型 ADC 的输入级

在实际的 Sigma-Delta 型 ADC 芯片内部，输入级、减法器、积分器、基准切换电路，甚至内置可编程增益放大器（PGA）都采用开关电容电路电路实现，以便于纯数字方法实现。

开关电容型输入级也会带来很多问题。首先就是输入阻抗问题。图 2.7.13 中最右侧框内是 Sigma-Delta 型 ADC 的输入级等效电路。S1 和 S2 在采样时钟的控制下以 Kfs 的频率通断：半个周期 S1 接通 S2 断开，C 被充电，最终和输入电压一致；另外半个周期 S2 接通 S1 断开，电容的电荷被移走，实现一次采样。

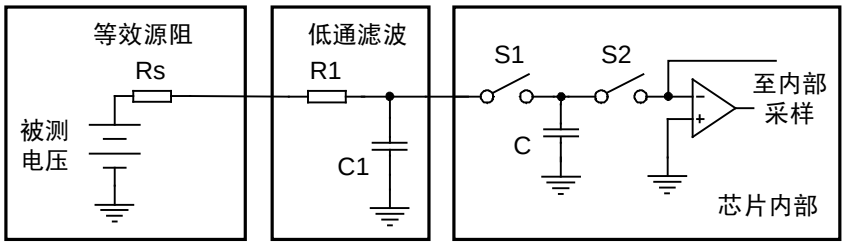


图 2.7.13 Sigma-Delta 型 ADC 的输入等效电路

对于被测电压信号来说，电容充放电相当于周期性地从被测电压上吸取电荷，宏观上等效于一股电流。所以对于被测电压来说，输入端等效为一个对地的输入阻抗 Z=1/(KfsC)。如果源阻抗较高，会因输入阻抗分压而引起误差。当然，如果源阻抗固定，这个误差能被校准过程消除，但是若被测源阻本身会变化，那么校准就毫无意义了。

另外，对于误差来说 ADC 的等效阻抗并不起着决定作用，而是电容 C 在充电周期结束时的电压决定了误差大小。很多带有 RC 一阶低通滤波器的电路（防混叠滤波）和 Sigma-Delta 型 ADC 连接时（图中的 R1、C1 部分），会产生额外的采样误差。原因是 S1 闭合时，相当于 C1 和 C 并联，由于 C 很小（7～20pF），而 C1 较大（数百 pF~数 uF），二者并联后电压会略有跌落。并联后 C 较大，并且电压接近 Vin，完全充电时间会很长，很可能在充电周期结束的时侯 C 上电压仍然和 Vin 之差大于 1LSB，引起采样误差。而且，R1、C1 越大，误差越明显。这一点与普通 ADC 完全不同，需要特别注意。TI 公司的手册上未公布具体数据，下表是作者根据实际经验总结的数据，当外部 RC 滤波电路取值小于表中数值时，造成的误差小于 1 个最低位（1 个字，或称 1LSB）。

表 2.7.3 小于 1LSB 误差对外部 RC 限制（OSR=256,f_SD16=500KHz）

最大 R1	外部 C1 容量					
增益	**0**	**50pF**	**100pF**	**500pF**	**1000pF**	**5000pF**
1	300kΩ	70kΩ	40kΩ	10 kΩ	5 kΩ	1 .5kΩ
2	150kΩ	35kΩ	20kΩ	5 kΩ	2.5 kΩ	700Ω
4	60kΩ	15kΩ	10kΩ	2.5 kΩ	1.2 kΩ	300Ω
8 以上	30kΩ	7kΩ	5kΩ	1.2 kΩ	600 Ω	150 Ω

开关电容输入电路带来的另一个困难是不允许 Sigma-Delta 型 ADC 与运放输出端直接连接。图 2.7.14 中运放输出直接和 Sigma-Delta 型 ADC 的输入连接，那么在 S1 闭合的瞬间，一个电容突然加在了运放的输出端，相当于突然对地呈现很小的阻抗，拉低运放的

输出。负反馈过程会努力消除这一电压降落，但由于输出受到运放压摆率的限制，不能立即上升，而达到稳定后又会过冲（反馈的重新稳定需要时间），从而会造成轻微的振铃现象，将这引起电容采压的误差。

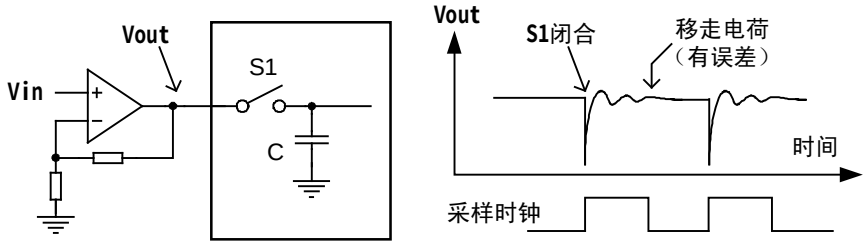


图 2.7.14　Sigma-Delta 型 ADC 与运放直接连接带来的误差

消除这一影响的办法是增加 RC 滤波器，C 提供大部分的电荷，R 将运放输出和电容隔开，使之呈现无跳跃的阻抗。RC 的取值要遵守表 2.7.13。

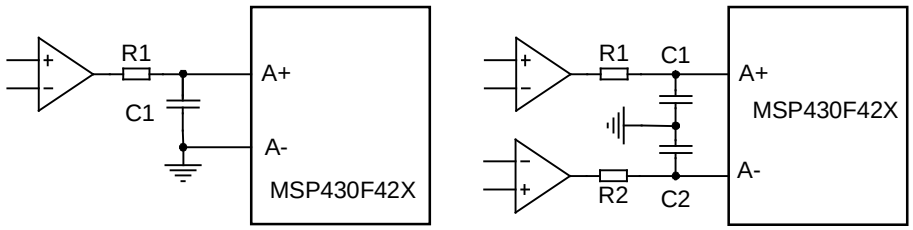


图 2.7.15　SD16 模块与运放的连接（*左：单端应用　右：差分应用*）

电源与布线

电源的质量（纹波、稳定度）对 SD16 模块高精度的应用有影响。所以 MSP430 单片机芯片提供了 2 套电源输入引脚：数字部分电源 DVCC 和 DGND、模拟部分电源 AVCC 和 AGND。在芯片内部，SD16 模块、基准源、PGA 等模拟电路全部使用 AVCC 引脚供电，AGND 引脚作为参考地。其余数字部分使用 DVCC 引脚供电，DGND 引脚作为参考地。

在管脚排列上看，这两对引脚靠得很近。在要求不严格的场合下，可以将 AVCC 与 DVCC 直接连接起来作为 VCC、将 AGND 与 DGND 直接连接起来作为 GND。但由于数字电源 DVCC 可能被数字电路高频率工作的干扰所污染。在要求较高的场合，需要为模拟部分单独提供高质量的 AVCC 电源。

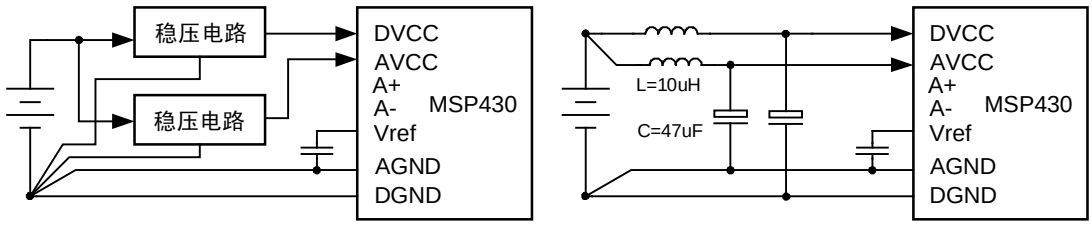


图 2.7.16　为 SD16 模块提供较高质量的电源

图 2.7.16 是常用的两种提供 AVCC 的方法。左图用 2 只稳压器分别为 AVC 与 DVCC 提供电源，两者互不干扰。在要求成本较低的情况下可以用右图的 LC 滤波电路，阻挡数字部分的干扰，使之不能进入模拟部分的电源。

作为参考电位，地线（GND）上各点的电压都应该相同，当作 0V 参考点。但实际中导线不可避免的存在电阻，只要有电流流过导线，各点之间就不可避免地存在电压差。对于模拟部分来说，地线不等电位将会导致误差。例如下图中的电路，数码管显示部分会消耗上百毫安电流，该电流通过地线回流至电源的途中，将在流过的导线上造成电压差。假设图中两段导线上的电压降分别为 V1 与 V2，那么调理电路接收到的电压与真实的传感器输出之间已经有了误差 V1，调理电路的输出与 ADC 实际接收到的电压之间有误差 V2。

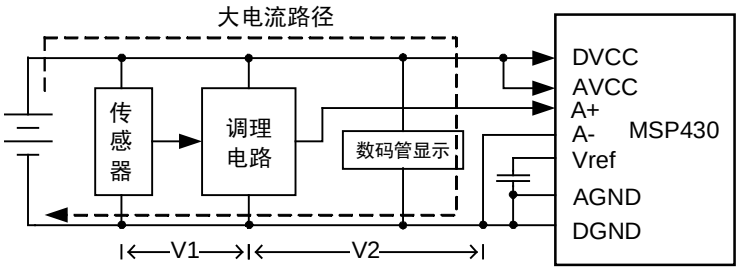


图 2.7.17 不合理的接地带来误差

在一般电路板中，铜皮厚度 50um 左右。按照 1mm（40mil）宽的导线计算，每厘米导线有 3.5 毫欧的电阻。按照电流 100mA 计算，即使 1 厘米的导线也将有 350uV 的电压差。对 SD16 模块来说，将导致 20 个字的测量误差。对于更加敏感的小信号传感器来说，会导致更加严重的误差。

减轻或消除地线电位差的方法有三种：减小地线电阻、减少电流以及改变拓扑结构。通过增大导线线宽、减少导线长度可以降低地线电阻，但最终受会到电路板面积与布局的限制，效果有限。减少电流需要改变电路设计，例如上例中将数码管显示改为 LCD 显示，耗电将下降到数微安，地线电阻就变得不敏感了。

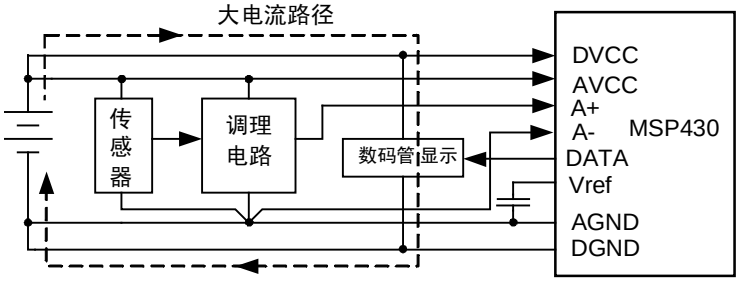


图 2.7.18 合理的接地

最佳的方法是改变电路拓扑结构。如图 2.7.18，将数码管显示部分的地线与数字地 DGND 连接到一起，模拟信号部分与 AGND 连接，走“专用通道”回到电源。大电流走 DGND 回到电源，整个 AGND 上没有大电流，不会产生电压差，因此不会带来误差。和精度关系密切的某些局部还可以采用一点接地的方法，例如将 A-端接到调理电路中最近的接地点，这样即使传感器或其他电路造成 AGND 上有较大的电流，也不会造成测量误差。

数字部分噪声容限很大，即使地电位差别数百毫伏，仍然不影响数字量的判别，因此让大电流、强干扰、高频等信号走数字地，一般不会对电路的工作造成影响。

严格地说，将 AGND 和 DGND 称为“模拟地”和“数字地”是不科学的。“模拟地”很容易被误解为“模拟电路的地线”，“数字地”很容易被误解为“数字部分的地线”。

事实上，“模拟地”的准确含义是“对地电位敏感的地线”，“数字地”的准确含义是“对地电位不敏感的地线”。无论任何电路，只要某几个部分之间要求地电位相等，否则会带来不良影响的，这段地线都要按照“模拟地”来对待。反之，如果某电路对地电位要求不严格，都可以当作“数字地”来对待。在实际应用中，大部分模拟电路对地电位敏感，而大部分数字电路都对地电位不敏感，因此习惯性称二者为“模拟地”和“数字地”。

此外，大电流、强干扰部分电路的地线还可以提取出来，作为“功率地”单独对待。例如图 2.7.18 中将数码管部分的 GND 单独拉一根地线回到电源，那么数码管的大电流对电路中其他任何部分都不会造成影响。

温差效应

两种不同的金属连接在一起，若连接点的温度与测量端的温度不同，就能在测量端测出温差电动势。温差电动势与金属材料有关，且大致正比与温差。

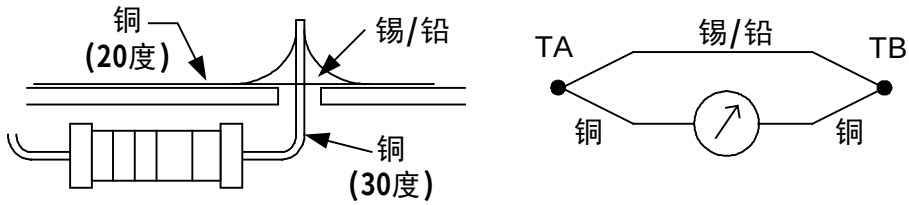


图 2.7.19　温差在电路板上产生的电压差

在电路板上，元件引脚与电路板导线大部分是铜制的，焊料大部分是铅锡合金。对于一个被焊接在电路板上的元件来说，假设自身发热造成管脚温度高于铜皮温度，那么铜-锡-铜三种金属构成热电偶，将会造成温差电动势。在电路板上，大部分温差电动势能够自相抵消，但是若整个电路板的温度存在梯度，有一部分温差电动势将无法自身抵消。10 度左右的温差在铜-锡之间将产生十几微伏的电压。如果恰巧多个温差电动势相互串联还可能加剧产生的电压差。对于数毫伏级别的某些被测量信号，这个误差是不可忽视的。所以在弱信号、高精度测量应用中，关键信号部分应尽量远离热源，避免热源在电路板上造成温差梯度。同时在设计上尽量降低系统耗电，避免元件自身发热。若电路中实在无法降低发热功率，就应该尽量增大发热元器件的散热面积，以降低温度。

I　SD16 模块的内部温度传感器

在 SD16 模块内，集成有一只温度传感器。任意一个 ADC 选择通道 6 都可以测量内部温度传感器的输出电压。通过温度传感器可以获得芯片内部的温度。如果单片机本身处于低功耗运行，几乎不发热，芯片的温度与环境温度是相等的。测量环境温度，不仅能作为温度计应用，还能够监控电路板或机箱内的温度，当检测到超温时，可以采取某些措施

（如断电、报警、停止功率部分电路等）避免事故发生。在高精度测量应用中，还可以通过测得的温度来做数字温度补偿。

温度传感器的使用

430 单片机内部温度传感器温度系数是：1.32mV/K (1.32mv/开尔文) 。其中开尔文温度是绝对温度，等于摄氏温度加 273K。在内部基准电压 Vref=1200mV 条件下，ADC 数据格式设置为“有符号”时，0V 对应采样值 0，Vref/2（600mV）对应 ADC 采样值 32767(忽略实际误差)。假设 ADC 采样值为 D，那么推导出传感器输出电压：

$$\begin{aligned} Vsensor &= (D/32768)\times Vref/2 \\ &= D\times Vref/65536 \\ &= D\times 1200/65536(mV) \end{aligned} \qquad (2.7.7)$$

Vsensor 除以温度系数（1.32mV/K）得到开氏温度：

$$\begin{aligned} K &= D\times 1200/65536/1.32 \\ &= D\times 909/65536 \end{aligned} \qquad (2.7.8)$$

再减去 273，得到摄氏温度:

$$DegC = D\times 909/65536 - 273 \qquad (2.7.9)$$

为了在定点运算时能保留 1 位小数（分辨至 0.1 摄氏度），计算过程中先扩大 10 倍，显示时加一位小数点。得到最终计算公式：

$$DegC = D\times 9090/65536 - 2730 \qquad (2.7.10)$$

例 2.7.14：编写一个温度计程序，显示环境温度（假设 ADC 没有误差）。

```
/*************************************************************************
* 名    称：主程序
* 功    能：对温度传感器采样、计算摄氏温度并显示在 LCD 上
* 入口参数：无
* 出口参数：无
*************************************************************************/
void main(void)
{
  int i;
  long int DegC;
  WDTCTL = WDTPW + WDTHOLD;                 // 关闭看门狗
  FLL_CTL0 |= XCAP18PF;                     // 设置晶振负载电容 18pF
  for (i = 0; i < 1000; i++);               // 略延迟，让时钟稳定
  SD16CTL = SD16REFON+SD16VMIDON+SD16SSEL0;
  // 开启内部 1.2V 基准源，开启缓冲器，ADC 时钟选择为 SMCLK(默认 1.048MHz)
  SD16CCTL0  |= SD16SNGL +SD16DF;
  // ADC0 工作在单次采样模式，输出数据格式为有符号
  SD16INCTL0 |= SD16INCH_6;       // ADC0 输入选择为通道 6(内部温度传感器)
  for (i = 0; i < 500; i++);          // 略延迟，让基准电压稳定
```

```
  BTCTL = 0;                              //BTCTL 在上电不会被清零，手动清零。
  LCD_Init();                             //LCD 初始化
  while (1)
  {
    SD16CCTL0 |= SD16SC;                  // 向 ADC0 发出"开始转换"命令
    while((SD16CCTL0 & SD16IFG)==0); //等待 ADC0 转换完毕
    DegC = ((long int)ADCresult * 9090)/65536 - 2730;    //计算摄氏温度
    LCD_DisplayDecimal(DegC,1);           //显示温度,带 1 位小数
    LCD_InsertChar(DT);                   //'''
    LCD_InsertChar(CC);                   //'C'  显示单位：  `C
    for(i=0;i<30000;i++);                 //延迟
  }
}
```

该程序工作时 CPU 及基准一直开启，功耗较大。如果将最后一行延迟替换成休眠（定时唤醒），并将基准也间歇开启，只在测量时启用，能实现超低功耗运行。完整的超低功耗温度计程序可参考光盘中附带的《温度计实用程序》范例。

温度传感器的校准

上述温度公式是在理想状况下求得的，实际上 ADC、温度传感器都存在误差，因此需要校准。在“SD16 模块的误差及校准”章节中提出的校准方法在这里同样适用，即输入两个已知温度(例如 0/100 度)，记录 ADC 读数，然后根据两点坐标得出新的直线公式，即可实现校准。

但是该方法存在两个难点：第一是标准温度产生比较困难（相对来说，电压表程序中产生已知电压容易得多）；第二是温度传感器位于片内，难以置入标准温度中（比如冰水混合物/沸水）。

这里提出一种相对简单的近似校准方法。一般温度计大部分情况测量的是室温附近的范围（0-50 度考虑），而传感器的输出比例系数是按绝对零度开始的，微小的比例误差乘以 273 都是不可忽略的。假设%5 的比例误差，在 27 度下（300K）会造成 19.8mV 误差，约 15 度，这个误差是难以接受的。又因为难以产生标准温度，所以比例系数难以校准。但偏移误差很容易通过显示值和普通温度计示数之差得到。我们可以将所有的误差都折算成偏移误差，这样虽然比例误差无法完全消除，但因为测温范围不大，影响也相对小得多。

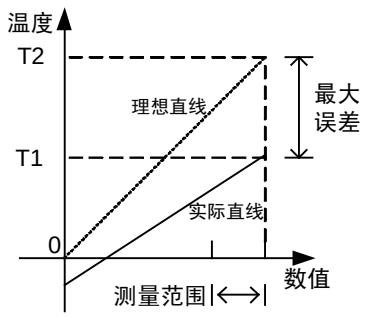


图 2.7.20a 未校准时的误差

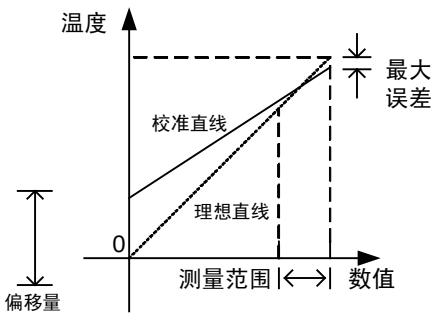


图 2.7.20b 校准后的误差

在 25 度左右的室温下，记下 430 显示的温度标与准温度计的温度差 T_OFFSET，在

显示的时候将这个误差扣除。按 0-50 度量程考虑，25 度必然是准确的，按量程 25 度程正负 25 度计算，5%比例误差造成的温度误差只有正负 1.25 度，精度已经基本足够。

例 2.7.14：在上例中，假设环境温度 24.0℃时，实测显示 26.8℃。那么折算成偏移误差为 **2.8**℃。将 **DegC** 的值在显示之前减去 **28**，就能得到较为准确的温度。

```
#define T_OFFSET (28)        /*25 度左右条件下，显示温度减去标准温度计温度*/
                             /*作为校准偏移值。注意要乘 10 倍，10=1.0 度*/
...
  while (1)
  {
    SD16CCTL0 |= SD16SC;              // 向 ADC0 发出"开始转换"命令
    while((SD16CCTL0 & SD16IFG)==0); //等待 ADC0 转换完毕
    DegC = ((long int)ADCresult * 9090)/65536 - 2730;    //计算摄氏温度
    DegC-=T_OFFSET;                        //扣除校准偏移量
    LCD_DisplayDecimal(DegC,1);            //显示温度,带 1 位小数
    LCD_InsertChar(DT);                    //'''
    LCD_InsertChar(CC);                    //'C'  显示单位：  `C
    for(i=0;i<30000;i++);                  //延迟
  }
```

对于样机的校准可以通过修改偏移量的宏定义进行。对于批量产品，也可以编写一个半自动的校准程序来完成。例如进入校准菜单后，提示按加减键修正温度偏移量，直到显示温度值与标准温度计示数相等为止，按确认键后将偏移量保存在 Flash 中。以后的温度计算都减去该偏移量。

2．8　16 位定时器 Timer_A

在全系列的 MSP430 单片机中，都带有一个 16 位定时器 Timer_A（以下简称 TA），用于精确定时、计时或计数。在普通的定时/计数器的基础上，还添加了 3 路（某些型号 5 路）捕获/比较模块，能够在无需 CPU 干预的情况下自动根据触发条件捕获定时器计数值或自动产生各种输出波形（如 PWM 调制、单稳态脉冲等）。

Ⅰ Timer_A 定时器主计数模块的结构与原理

Timer_A 定时器分为 2 个部分：主计数器和比较/捕获模块。主计数器负责定时、计时或计数。计数值（TAR 寄存器的值）被送到各个比较/捕获模块中，它们可以在无需 CPU 干预的情况下根据触发条件与计数器值来自动完成某些测量和输出功能。只需定时、计数功能时，可以只使用主计数器部分。在 PWM 调制、利用捕获测量脉宽、周期等应用中，还需要比较/捕获模块的配合。

与 Timer_A 定时器中的主计数器相关的控制位都位于 TACTL 寄存器中，主计数器的计数值位于 TAR 寄存器。每个比较/捕获模块还有一个独立的控制寄存器 TACCTLx，以及一个比较值/捕获值寄存器 TACCRx（x=0、1、2）。在一般定时应用中，TACCRx 可以提供额外的定时中断触发条件；在 PWM 输出模式下，TACCRx 用于设定周期与占空比；在捕

获模式下；TACCRx 存放捕获结果。

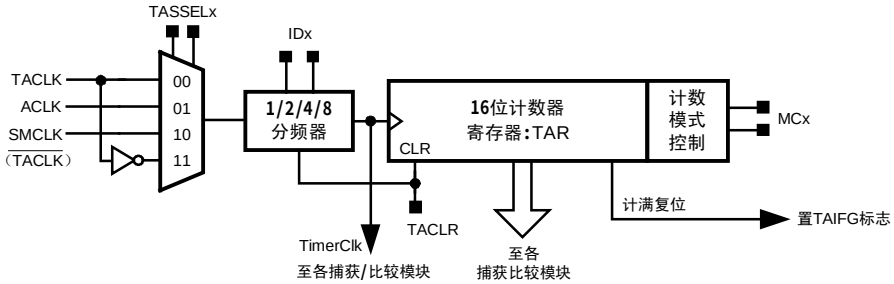


图 2.8.1 Timer_A 的主计数器结构

主计数器部分的结构见图 2.8.1，它包括时钟源选择、预分频、计数器与计数模式选择四个部分。相关的控制字有：

- *TASSELx：Timer_A 计数器的时钟源选择* *（位于 TACTL 寄存器）*
  00=外部引脚(TACLK) *01=ACLK* *10=SMCLK* *11=外部引脚(TACLK 取反)*
  快捷宏定义： *TASSEL_0* *TASSEL_1* *TASSEL_2* *TASSEL_3*

- *Idx：Timer_A 计数器的预分频系数* *（位于 TACTL 寄存器）*
  00=无分频 *01=2 分频* *10=4 分频* *11=8 分频*
  快捷宏定义： *ID_0* *ID_1* *ID_2* *ID_3*

通过上面两组控制位，可以设置定时/计数的时钟源。在低功耗应用，以及需要长时间定时、计时的情况下，可以选择 ACLK 作为时钟，加上预分频，最长的定时/计时周期可达 16 秒。在高分辨率、短时间的定时应用中，可以选择 SMCLK 作为时钟源。

若选择 TACLK 作为时钟源时，定时器实际上成了计数器，累计从 TACLK 管脚上输入的脉冲，上升沿计数。若选择 TACLK 取反作为时钟源，TACLK 的下降沿计数。

- *TACLR：Timer_A 计数器清零控制位* *（位于 TACTL 寄存器）*
  0=不清零 *1 =清零*

将该控制位置 1，可以立即将 Timer_A 计数器清零，无需通过软件赋值操作来实现。计数器复位后该标志位自动归零，因此读该标志位时将永远读回 0。

- *MCx：Timer_A 计数器的计数模式* *（位于 TACTL 寄存器）*
  00=停止 *01=增计数* *10=连续增计数* *11=增-减计数*
  快捷宏定义： *MC_0* *MC_1* *MC_2* *MC_3*

- *TAIFG：Timer_A 计数器溢出标志（中断标志）* *（位于 TACTL 寄存器）*
  0=未发生溢出 *1=曾发生溢出*

Timer_A 计数器提供了 3 种计数模式：增计数、连续增计数和增-减计数。在增模式下，每个时钟周期计数值 TAR 加 1。当 TAR 值超过 TACCR0 寄存器（捕获/比较模块 0 的设置值）时自动清零，同时会将 Timer_A 溢出标志位 TAIFG 置 1。如果 TA 中断被允许，还会引发中断。改变 TACCR0 寄存器可以改变定时周期，且不存在初值装载问题，非常适合产

生周期性定时中断，只要改变 TACCR0 的值即可随意调整定时周期。

在连续计数模式下，其操作与 8051 的定时器基本相同：每个时钟周期 TAR 值加 1，计数值超过 0xFFFF 后溢出，TAR 回到 0，同时将 TAIFG 置 1，或引发中断。如果中断内给 TAR 重新赋除值，也可以产生不同周期的定时中断。由于用增计数模式产生定时中断比连续模式更简单，一般不用连续模式来产生周期性定时中断；连续模式一般在捕获模式下使用较多，让计数器自由运行，利用捕获功能在事件发生时自动记录下计数值，通过对比几个计数值可以确定事件发生的准确时间或者准确的时间间隔。

在增-减模式下，计数值从 0 开始递增，计到 TACCR0 后，自动切换为递减模式，减到 0 后又恢复为递增模式，如此往复。在 TAR 从 1 递减到 0 的时刻，产生 TAIFG 中断标志。一般应用中，不用增减模式来定时或计数，而多用于 PWM 发生器。借助增-减模式，捕获/比较模块能够产生带死区的对称 PWM 驱动波形，可以直接驱动半桥电路，无需专门的死区产生电路。

在增计数模式或增-减计数模式下，向 TACCR0 写入 0 均可停止计数器。

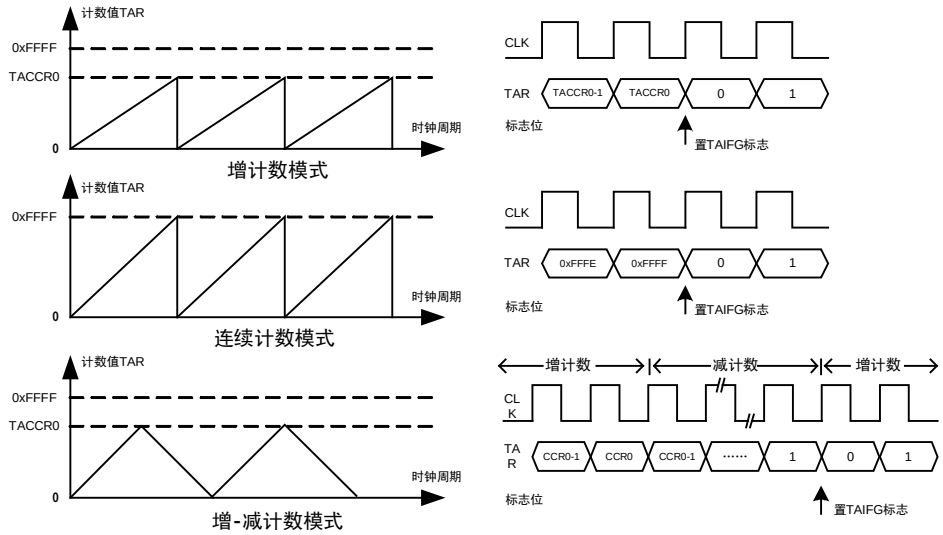


图 2.8.2 Timer_A 的三种计数模式示意图

例 2.8.1 在 MSP430 单片机中，为 Timer_A 配置时钟源及工作模式，使 Timer_A 在无需 CPU 干预的情况下，每隔 1.3125 秒溢出一次。（假设 SMCLK=MCLK=1.048576MHz，ACLK=32.768kHz）

首先 1.3125 秒时间较长，若使用高频的 SMCLK 作为时钟源 16 位计数器不够用。应该使用低频 ACLK 作为时钟源。再考虑周期性的定时，三种模式实际上都能实现，其中增计数模式最简单，无需重置初值等操作。最后计算 TACCR0 的值应该是 1.3125 秒乘以 ACLK 频率（32768Hz）得到设置值 43008。由于计数从 0 开始，实际应该设为 43007。

```
TACTL |= TASSEL_1 + ID_0 + MC_1;     //ACLK 做时钟,无分频,增计数模式。
TACCR0 = 43008-1;
```

例 2.8.2 在 MSP430 单片机中，为 Timer_A 配置时钟源及工作模式，使 Timer_A 在无

需 CPU 干预的情况下，每隔 123.45 毫秒溢出一次。（假设 SMCLK=MCLK=1.048576MHz，ACLK=32.768kHz）

对于 ACLK 来说，0.12345 秒乘 32758Hz 等于 4045.2，舍去小数 0.2 将导致万分之五的误差。这种情况可以考虑用更高频率的时钟来实现，虽然无法得到整数周期值，但整数部分的值会大很多，误差相对小得多。但 123.45 毫秒对 SMCLK 来说太长，16 位计数器不够用，16 位定时器最大定时长度是 65536 除以 1.048MHz 约等于 62.5 毫秒。可以考虑对 SMCLK 分频来得到合适的时钟。SMCLK 二分频后能得到最长 125 毫秒的定时，能满足要求：

```
TACTL |= TASSEL_2 + ID_1 + MC_1;     //SMCLK 做时钟,2 分频,增计数模式。
TACCR0 = 64723-1;
```

计算出 TACCR0 的值应该是 0.12345 秒乘以 SMCLK/2 频率（524.288kHz）得到设置值 64723.3。取整数 64723，尾数 0.3 对于 64723 来说，只造成十万分之五的误差，比用 ACLK 做时钟小了 10 倍。但这时 SMCLK 无法关闭，不能进入 LPM3 模式，功耗会增加。所以实际应用中应该根据定时周期、定时精度以及功耗等因素综合考虑来决定时钟源。

在任何模式下，若如果希望定时器暂停，可以将定时器模式设为停止（MC0=MC1=0）：

```
TACTL &=~ (MC0 + MC1);              //暂停。
```

在任何模式下，若如果需要定时器清 0，可以使用 TACLR 控制位，也可以对 TAR 进行赋 0 操作：

```
TACTL |= TACLR;                     // Timer_A 清零
TAR=0;                              // Timer_A 清零
```

对于分频系数=1 时，两者功能相同。对于分频系数不为 1 的情况，后者只清除了 Timer_A 计数值，而未对分频器的计数值清零，可能导致之后第一次计数的时间有误差。例如 8 分频时，应该每 8 个时钟周期 TAR 加 1。假设在运行到第 5 个周期时，对 TAR 赋值清零但未将分频器中的计数值“5”清除，之后只要 3 个时钟周期 TAR 加 1，导致第一次计数并非 8 个周期后，这种情况应使用 TACLR 控制位来对 Timer_A 清零。如果某些应用中希望清除计数值但保留分频尾数，可以对 TAR 赋值来清零。

Timer_A 定时器的捕获模块

除了主计数器之外，Timer_A 还带有 3 个（某些型号有 5 个）比较/捕获模块。每个比较/捕获模块都有单独的模式控制寄存器以及比较/捕获值寄存器。在比较模式（也叫输出模式）下，每个比较/捕获模块将不断地将自身的比较值寄存器与主计数器的计数值进行比较。一旦相等，就将自动地改变某个指定引脚（TAx 管脚）的输出电平。有 8 种改变电平的规律可以选择（8 种输出模式），从而能在无需 CPU 干预的情况下输出 PWM 调制、可变单稳态脉冲、移相方波、相位调制等常用波形。

在捕获模式下，用某个指定引脚（TAx 管脚）的输入电平跳变触发捕获电路，将此刻主计数器的计数值自动保存到相应的捕获值寄存器中。由于该过程纯硬件实现，无需 CPU 干预，因此不存在中断响应等时间延迟。可以用于测频率、侧周期、测脉宽、测占空比、门控计数等需要获得波形中精确时间量的应用。

比较/捕获模块的结构如图 2.8.2 所示。可分为 2 个部分，上半部分是捕获电路，下半部分是比较（波形输出）电路。图中所有的标志位都位于 TACCRx 寄存器内，三个模块之间互相独立，图中只画出了模块 2 的结构，其余两个模块的结构完全相同，只是寄存器下标不一样。每个模块的工作模式靠 CAP 标志位来选择：

- *CAP：Timer_A 捕获/比较模块工作模式选择　　（位于 TACCTL0/1/2 寄存器）*

  0=比较模式（波形输出）　　1 =捕获模式

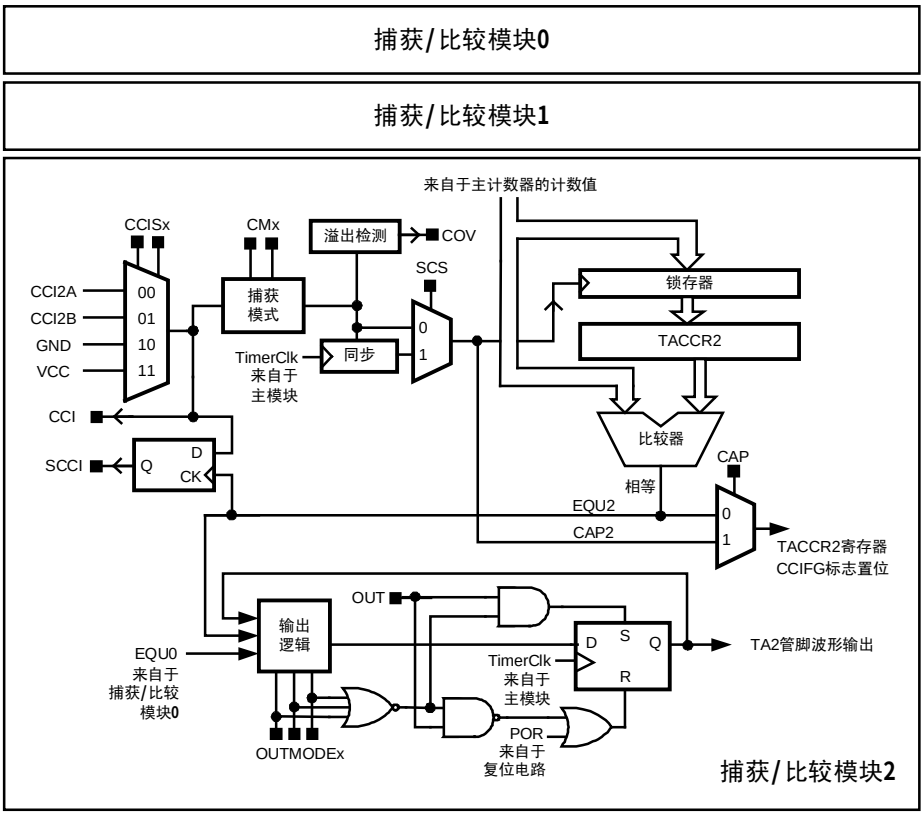


图 2.8.3　Timer_A 捕获/比较模块结构图

在捕获模式下，图中标有 CAP2 的信号线上升沿将主计数器的计数值通过锁存器锁存至 TACCR2 寄存器内，这个过程被形象的称为“捕获”（Capture）。该信号有 3 种来源，通过 CCIS 标志位来选择。对于每个信号源，还可以设置捕获触发沿：

- *CCISx：Timer_A 捕获模块的捕获源选择　　（位于 TACCTL0/1/2 寄存器）*

  00=CCIxA 引脚　　01 =CCIxB 引脚　　10=GND　　11=VCC

  快捷宏定义：CCIS_0　　CCIS_1　　CCIS_2　　CCIS_3

- *CMx：Timer_A 捕获模块的捕获触发沿选择　　（位于 TACCTL0/1/2 寄存器）*

  00=禁止捕获　　01 =上升沿　　10=下降沿　　11=上升或下降沿

快捷宏定义：CM_0　　CM_1　　CM_2　　CM_3

对于不同型号的芯片来说， CCIxA 和 CCIxB 信号所在的管脚会有区别，具体可以查阅相应的芯片手册。例如 MSP430F425 芯片中，CCI0A 和 CCI0B 都位于 P1.0；CCI1A 和 CCI1B 都位于 P1.2，CCI2A 和 CCI2B 都位于 P2.0。有的芯片将 CCIxA 与 CCIxB 分开至两个不同管脚上，以方便电路板布线，使用前利用 IO 口的 PxSEL 寄存器激活捕获输入功能。除了外部输入捕获触发信号源外，片内还有 2 个信号 VCC 和 GND 可供选择。当通过软件切换这两个信号时，可以产生上升沿或下降沿，相当于软件触发捕获。

触发沿可以根据实际需要来选择。例如测量方波周期时用上升或下降沿都可以，测量脉宽时用上下沿均可触发的模式比较方便。测量高电平周期时，先选择上升沿出发捕获一次，再改为下降沿捕获一次，两次数值差就是高电平周期。

由于捕获过程纯硬件实现，所以实时性很强（延迟仅十纳秒级）。CPU 可以通过查询或在中断内读取捕获值。根据两次捕获值之间的差即可计算出周期、脉宽等信息。即使读取略有延迟，也不影响捕获结果。这比单纯靠外部中断内保存计数器值的方法实时性高得多，且不要求 CPU 立即读取。但也可能遇到第一次的捕获值尚未来得及被 CPU 读取的情况下，第二次捕获条件又成立了，这种情况称为捕获溢出，会导致计算错误。为此，模块中留有一个标志位用于指示溢出：

- *COV：Timer_A 捕获模块的捕获溢出标志　　（位于 TACCTL0/1/2 寄存器）*

0=未发生溢出　　1 =发生了溢出

如果该标志位为 1，说明前一次的捕获值尚未被读取，新的捕获条件已经发生。应该舍去该结果或另作调整。该标志位必须通过软件清除。

- *SCS：Timer_A 捕获模块的同步捕获控制位　　（位于 TACCTL0/1/2 寄存器）*

0=异步捕获　　1 =同步捕获

当该标志位为 0 时，捕获过程的锁存直接由硬件电路控制，不受时钟的约束。当该标志位为 1 时，捕获触发信号经过 D 触发器与定时器时钟同步。假设捕获触发条件出现在计数值为 N 的时间段内，异步模式将捕获到数值 N，同步模式下将捕获到数值 N+1。一般建议工作在同步模式下，以避免数字逻辑部分出现竞争，产生毛刺。

可以通过标志位来获得捕获信号输入源的一些信息：

- *CCI：Timer_A 捕获模块的输入信号电平（异步）　（位于 TACCTL0/1/2 寄存器）*
- *SCCI：Timer_A 捕获模块的输入信号电平（同步）（位于 TACCTL0/1/2 寄存器）*

CCI 标志位的值与被选择的捕获触发源电平相同，实时地异步更新。SCCI 标志位是捕获同步的输入电平，每次计数值与捕获值相等时才更新。由于每次捕获后捕获值必然与计数值相等，会每次捕获成立后会自动更新该标志。

例 2.8.3　在 MSP430F42x 单片机中，捕获 3 路信号。要求其中从 P1.0 输入的方波上升沿出发捕获逻辑，P1.2 输入方波下降沿触发捕获，P2.0 输入方波上下沿都触发捕获。假设主计数器用 ACLK 作为时钟，连续计数模式，为 Timer_A 及三个捕获模块配置寄存器：

```
TACTL |= TASSEL_1 + ID_0 + MC_2;     //主计数器 ACLK 做时钟,无分频,连续计数模式。
```

```
TACCTL0 |= CAP + CCIS_0 + CM_1 + SCS;//模块0捕获模式，外部输入，上升沿同步捕获
TACCTL1 |= CAP + CCIS_0 + CM_2 + SCS;//模块1捕获模式，外部输入，下降沿同步捕获
TACCTL2 |= CAP + CCIS_0 + CM_3 + SCS;//模块2捕获模式，外部输入，双沿同步捕获
P1DIR &=~ (BIT0 + BIT2);             //P1.0与P1.2的方向设为输入
P2DIR &=~ BIT0;                      //P2.0的方向设为输入
P1SEL |= (BIT0 + BIT2);              //将P1.0与P1.2的第二功能激活（CCxIA/B）
P2SEL |=  BIT0;                      //将P2.0的第二功能激活（CC2IA/B）
```

l Timer_A 定时器的比较模块

当 CAP 控制位设为 0 时，比较/捕获模块工作在比较模式。此时 TACCRx 的值由软件写入，并通过比较器与主计数器的计数值进行比较。每次相等产生 EQU 信号，该信号触发输出逻辑，通过 OUTMODE 控制位可以配置输出逻辑，通过不同的变高、变低或翻转组合来产生各种输出波形。整个过程无需 CPU 的干预，软件中只需改变 TACCRx 的值即可改变波形的某些参数。对于不同型号的芯片，波形输出 TAx 所对应的管脚会有所不同，读者可参考相应的芯片手册。例如在 MSP430F42x 系列单片机中，TA0 对应 P1.0，TA1 对应 P1.2，TA2 对应 P2.0。某些芯片会将多个管脚对应一个输出以方便布线，使用时利用 IO 口的 PxSEL 寄存器激活输出功能。

n *OUTMODEx：Timer_A 比较模块的输出模式控制位* *（位于 TACCTL0/1/2 寄存器）*

n *OUT：Timer_A 比较模块的输出电平控制位* *（位于 TACCTL0/1/2 寄存器）*

表 2.8.1 Timer_A 比较模块的 8 种输出模式

OUTMODEx 控制位	输出控制模式	说明
000（模式 0）	电平输出	TAx 管脚输出电平由 OUT 控制位的值决定。
001（模式 1）	延迟置位	当主计数器计至 TACCRx 值时，TAx 管脚置 1。
010（模式 2）	取反/清零	当主计数器计至 TACCRx 值时，TAx 管脚取反。 当主计数器计至 TACCR0 值时，TAx 管脚置 0。
011（模式 3）	置位/清零	当主计数器计至 TACCRx 值时，TAx 管脚置 1。 当主计数器计至 TACCR0 值时，TAx 管脚置 0。
100（模式 4）	取反	当主计数器计至 TACCRx 值时，TAx 管脚取反。
101（模式 5）	延迟清零	当主计数器计至 TACCRx 值时，TAx 管脚置 0。
110（模式 6）	取反/置位	当主计数器计至 TACCRx 值时，TAx 管脚取反。 当主计数器计至 TACCR0 值时，TAx 管脚置 1。
111（模式 7）	清零/置位	当主计数器计至 TACCRx 值时，TAx 管脚置 0。 当主计数器计至 TACCR0 值时，TAx 管脚置 1。

模式 0（电平输出）：

在输出模式 0 下，TAx 管脚与普通的输出 IO 口一样，可以由软件操作 OUT 控制位来控制 TAx 管脚的电平高低。

例 2.8.4 在 MSP430F42x 单片机中，利用输出模式 0，通过软件将 TA2 引脚（P2.0）置高或置低：

```
//-----------------------设置-----------------------------------------
P2SEL |= BIT0;                    // P2.0 设为第二功能（TA2 输出）
P2DIR |= BIT0;                    // TA2 从 P2.0 输出 (不同型号单片机可能不一样)
TACCTL2 = OUTMOD_0;               // TA2 设为模式 0：软件控制
//-----------------------操作-----------------------------------------
TACCTL2 |= OUT;                   // TA2 输出设为高电平
TACCTL2 &=~OUT;                   // TA2 输出设为低电平
```

模式 1 与模式 5（单脉冲输出）：

利用比较模块的模式 1 和模式 5，可以替代单稳态电路，产生单脉冲波形。

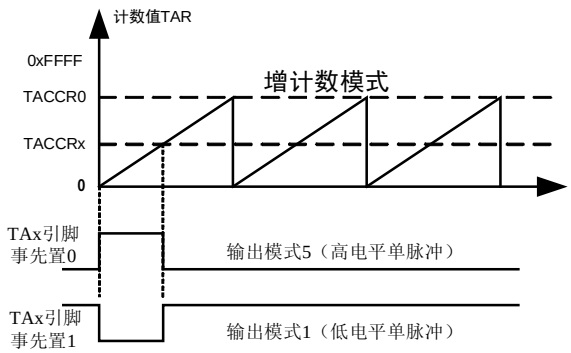


图 2.8.4 利用 OUTMODE 1、5 产生单脉冲的原理

在输出模式 1 下，当主计数器计至 TACCRx 值时，TAx 管脚置 1。如果通过 OUT 控制位事先将 TAx 的输出设为低，那么经过 TACCRx 个周期后，TAx 将自动变高。这样可以输出一个低电平脉冲。通过改变 TACCRx 的值，可以改变低电平脉冲的周期，且脉冲过程中无需 CPU 的干预。

在输出模式 5 下，当主计数器计至 TACCRx 值时，TAx 管脚置 0。如果通过 OUT 控制位事先将 TAx 的输出设为高，那么经过 TACCRx 个周期后，TAx 将自动变低。这样可以输出一个高电平脉冲。通过改变 TACCRx 的值，可以改变高电平脉冲的周期，且脉冲过程中无需 CPU 的干预。

例 2.8.5 在 MSP430F42x 单片机中，通过 P2.0 口控制三极管驱动一只长鸣型蜂鸣器，高电平鸣响。编写一个鸣响程序，要求鸣响的过程中不占用 CPU 的运行。

```
/*********************************************************************
 * 名    称：TA_Beep()
 * 功    能：利用 TA 单脉冲输出模式驱动蜂鸣器
 * 入口参数： Period: 鸣响周期(0~65535) ACLK 时钟个数
 * 说    明：鸣响过程不占用 CPU
 * 范    例：TA_Beep (500)蜂鸣器鸣响 500 个 ACLK 时钟周期
 *********************************************************************/
void TA_Beep(unsigned int Period)
{
   TACCTL2 =OUTMOD_5;  //设为模式 5，延迟清零
```

```
    TACTL|=MC_2 + TASSEL_1 + ID_0 + TACLR; //定时器 TA 用 ACLK，连续计数模式
    TACCTL2 |=OUT;      //TA2 输出设为高电平
    TACCR2= Period;     //TA2 高电平持续 Period 个 ACLK 周期后自动变低
}
```

对比下面的传统程序：

```
/*********************************************************************
 * 名    称： Beep()
 * 功    能： 通过软件延迟驱动蜂鸣器
 * 入口参数： Period: 鸣响周期(0~65535) 毫秒
 * 说    明: 鸣响过程会占用 CPU，无法释放。
 * 范    例: TA_Beep (500)蜂鸣器鸣响 500 个 ACLK 时钟周期
 *********************************************************************/
void Beep(unsigned int Period)
{
    int i;
    P2OUT |= BIT0;                  //开始鸣响
    for(i=0;i<Period;i++)           //延迟 Period 次
      {
          __delay_cycles(1000);     //每次约 1ms(在 1MHz 主频下)
      }
    P2OUT &=~ BIT0;                 //停止鸣响
  }
```

后者是较为通用的一种蜂鸣器驱动程序，不占用定时器资源。但后者在鸣响过程中，CPU 一直在做循环耗时间，直到设置时间到达才停止鸣响。鸣响的过程中程序将停在该函数内，导致后面的代码暂时无法执行。特别是当鸣响时间较长时，整个程序就会被“卡住”，对于该函数之后的任务暂时失去响应能力。

前者利用 TA 的硬件自动产生单脉冲波形，CPU 只需要数微秒时间来设置 TA 比较模块的参数，即可通过硬件自动产生蜂鸣器驱动波形。整个鸣响时间内，CPU 可以继续执行该函数后续的其他任务。

模式 3 与模式 7（PWM 输出）：

脉宽调制（PWM）是最常用的功率调整手段之一。所谓脉宽调制，顾名思义，是指在脉冲方波周期一定的情况下，通过调整脉冲（高电平）的宽度，从而改变负载通断时间的比例，达到功率调整目的。

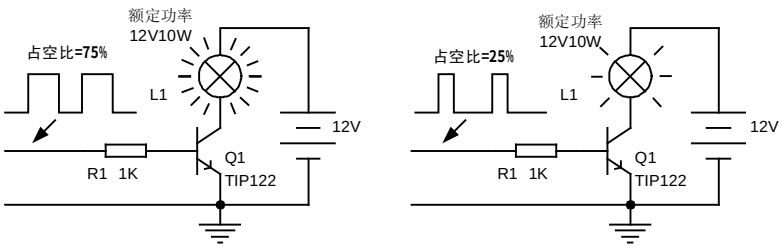


图 2.8.5 PWM 控制输出功率的原理

PWM 波形中，负载接通的时间与一个周期的总时间之比叫做占空比（Duty Cycle）。

占空比越大，负载的功率就越大。图 2.8.5 示意了 PWM 控制功率的原理：某灯泡亮度控制电路中，输入高电平将使三极管导通，点亮灯泡；输入低电平将使三极管截止，断开灯泡。在占空比 75%的条件下，灯泡有 75%的时间通电，25%时间断电，实际有效功率较大，亮度较高（左图）。在占空比 25%的条件下，灯泡有 25%的时间通电，75%时间断电，亮度较低（右图）。

如果 PWM 的频率足够高，以至于不足以表现出负载断续，那么从宏观上将看到负载实际功率是连续的。例如上例中如果 PWM 频率高到灯丝来不及冷却（约 100Hz 以上），那么将看到灯泡连续发光。

在 PWM 调整负载功率过程中，负载断开时三极管无电流通过，不发热。负载接通时三极管（开关器件）饱和，虽然通过有较大电流，但压降很小，发热功率也很小。所以使用 PWM 控制负载时，开关器件的总发热量很小。相比于串联耗散式的调整方法，效率高得多，适合大功率、高效率的负载调整应用。但 PWM 的缺点是负载功率的高频波动很大，不适于要求输出平稳的调节应用。

此外，PWM 控制本身属于开环控制，具有调节功能但不具备稳定负载的能力，也不保证输出结果线性正比于占空比。例如在电动机调速应用中，通过 PWM 控制可以改变电动机功率，但不能稳定电动机的转速，电动机的转速仍然会受到负载力矩的影响。在烤箱温度控制应用中，通过 PWM 控制加热器功率虽然能调温，但温度仍将受到环境因素的影响。在灯泡亮度控制中，灯丝在不同温度下电阻不同，虽然占空比越大灯泡越亮，但实际输出功率或实际亮度并不与占空比呈严格线性关系。

需要得到高精度、高稳定度、快速且无超调的控制结果时，需要反馈式控制系统。在 MSP430 单片机中，通过 ADC 的采集功能测量实际被控量作为反馈信号，结合 CPU 强大的计算功能实现各种反馈控制算法（如 PID 算法、模糊算法、最小拍控制等），最终通过 PWM 控制输出量，可以单芯片构成各种反馈式控制系统。

在输出模式 7 下，每次 TA 计数值超过 TACCRx 时，TAx 引脚会自动置低，当 TA 计至 TACCR0 时，TAx 引脚会自动置高。因此实际的输出波形就是 PWM 调制方波。如图 2.8.6，只需要改变 TACCR0 的值即可改变 PWM 方波周期，改变 TACCRx 即可改变从 TAx 引脚输出信号的占空比：TACCRx 越大，占空比越大。

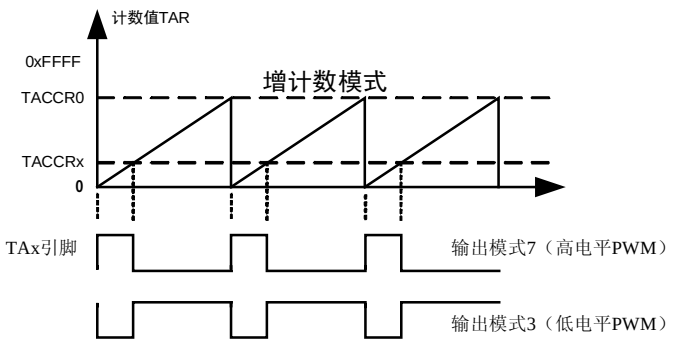


图 2.8.6　利用 OUTMODE 3、7 产生单脉冲的原理

在模式 3 下，与模式 7 刚好相反。TACCRx 越大，占空比越小。对于某些低电平接通

负载的电路，用模式 3 更符合习惯。模式 3 与模式 7 也常一起使用，用于产生两路对称的波形。

由于 TACCR0 被用于 PWM 周期的设定，因此通过 TimerA 产生的若干路 PWM 波形的周期都是一样的（由 TACCR0 的值决定）。且对于含有三个比较/捕获模块的 TimerA（Timer_A3），最多只能产生 2 路 PWM 波形。某些型号单片机中含有五个比较/捕获模块的 TimerA（Timer_A5），最多能产生 4 路独立的 PWM 波形。

例 2.8.6 在 MSP430F42x 单片机中，P1.2 口（TA1）与 P2.0 口（TA2）通过三极管控制两只灯泡的亮度。要求从 P1.2（TA1）引脚输出占空比 75%的 PWM 调制波形，从 P2.0（TA2）引脚输出占空比 50%的 PWM 调制波形。频率约 100Hz。

```
P1SEL |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
P1DIR |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
P2SEL |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)
P2DIR |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)

TACTL |= MC_1 + TASSEL_1 + ID_0; //定时器 TA 设为增量计数模式,ACLK
TACCTL1 =OUTMOD_7; //模式 7= 高电平 PWM 输出
TACCTL2 =OUTMOD_7; //模式 7= 高电平 PWM 输出
TACCR0=328-1;       //PWM 总周期 =328 个 ACLK 周期约等于 100Hz
TACCR1=246;         //TA1 占空比 = 246/328= 75%
TACCR2=164;         //TA2 占空比 = 164/328= 50%
```

此后只要通过软件改变 TACCR1 和 TACCR2 寄存器的值，即可改变两只灯泡的亮度。

例 2.8.7 在 MSP430F42x 单片机中，P1.2 口（TA1）与 P2.0 口（TA2）之间接有一只超声波发射器，要求从 P1.2（TA1）与 P2.0（TA2）引脚输出占空比约 50%、相位差 180 度的 40KHz 左右方波。假设 SMCLK 时钟频率=4.194304MHz

```
P1SEL |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
P1DIR |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
P2SEL |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)
P2DIR |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)

TACTL |= MC_1 + TASSEL_2 + ID_0; //定时器 TA 设为增量计数模式,SMCLK
TACCTL1 =OUTMOD_7; //TA1 模式 7= 高电平 PWM 输出
TACCTL2 =OUTMOD_3; //TA2 模式 3= 低电平 PWM 输出 （两路反向的方波）
TACCR0=105-1;       //PWM 总周期 =105 个 SMCLK 周期约等于 40kHz
TACCR1=52;          //TA1 占空比 = 50%
TACCR2=52;          //TA2 占空比 = 50%
```

两个输出引脚之间的电压刚好相反，因此在超声波发射器上实际获得了两倍的激励电压。在 0~50%范围内改变占空比即可调节超声波发射功率。

模式 2 与模式 6（带死区的 PWM 输出）：

PWM 调制不仅能用于功率调节，还被广泛地用于逆变器、开关电源、变频调速、斩波器等高效率功率变换应用。在这些应用中，因为涉及电压变换或者要求输入与输出之间隔离，常用变压器作为能量传递部件。变压器只能传递交流能量，所以在电路设计中常用

推挽电路或者半桥式电路来获得交流大功率信号。

例如在 2.8.7 中，用 MSP430 产生对称的 PWM 信号，驱动半桥开关产生交流方波，最后通过升压变压器将蓄电池的 24V 电压提升至 220V 左右输出。

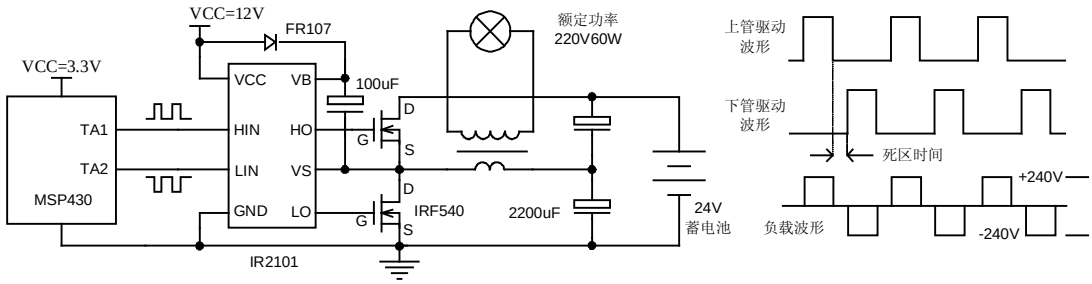


图 2.8.7 逆变器中的 PWM 死区示意图

为了得到对称的交流功率输出信号，上下两只 MOS 管必须轮流导通。由于 NMOS 的性能优于 PMOS，所以大部分桥式开关应用中上管也采用 N 沟道 MOS 管。但采用 NMOS 的情况下，上管的源极（S）电位是浮动的（0V 或 24V，根据下管波形而变）。由于 MOS 开关的通断由 G、S 间电压差决定，所以为了让上管导通，栅极 G 的驱动电压也需要随 S 浮动而变。有一类专为半桥驱动而设计的集成电路，叫做“浮栅驱动器”（Floating Gate Driver），专门解决上管浮动驱动问题。图中的 IR2101 就是一种常用型号，有兴趣的读者可以参阅它的 DataSheet 了解其工作原理。

当 TA1 输出高电平时，上管导通，负载上得到正半周波形，当 TA2 输出高电平时，下管导通，负载上得到负半周波形。如果 TA1 与 TA2 引脚输出 50%占空比，对称的波形（例 2.8.7），那么负载上将获得 220V 的交流电。

但这也随之会遇到一个问题：MOS 开关的导通与关断总会存在一定的延迟。如果同时命令上管关断、下管导通，会出现一个极短的瞬间上管来不及完全关闭时下管已经导通。此时相当于蓄电池直接短路，大电流会直接毁坏 MOS 管。为了安全地控制半桥电路，需要命令其中一只开关管断开后，略等待一段时间才接通另一只开关管。这段等待的时间被称为“死区时间”（Dead Time）。图 2.8.7 中示意了带死区的 PWM 波形，为了示意清楚，图中对死区时间作了夸大，实际应用中根据不同 MOS 管的导通延迟参数，取数百纳秒至数微秒时间以上即可保证 MOS 不会同时导通。实际应用中，半桥、推挽驱动、H 桥等电路中都存在对死区时间的要求。

TimerA 比较模块的模式 2 与模式 6 专门用于产生带有死区的 PWM 波形。模式 2/6 与模式 3/7 的区别是到达 TACCRx 门限后取反而非置电平。如果主计数器工作在增计数或连续计数方式下，模式 2 与模式 3 输出波形没有区别；模式 6 与模式 7 的输出波形也没有区别。均为普通的 PWM 波形。但若将主计数器设定为增减计数模式，情况会有所变化。由于增减计数模式下，每个周期内有 2 次计至 TACCRx 的时刻，因此利用模式 2/6 的取反功能可以产生 PWM 波形，且模式 2 与模式 6 所产生的波形不仅相位相反，而且不存在同时导通的时刻。

图 2.8.8 中，TA 计数器每次计至 TACCR1，TA1 引脚取反，增减计数模式下每周期 2 次计至 TACCR1 因此 TA1 引脚每周期取反两次，产生一个完整的方波。由于取反操作产

生的波形依赖于该管脚的初始电平，在模式 6 下计数器每次计至 TACCR0，TA 引脚置 1，给出了初始电平。类似的，在模式 2 下计数器每次计至 TACCR0，TA 引脚置 0，得到与模式 6 相反的波形。只要 TACCR1>TACCR2，两路 PWM 就不会有同时为高电平的时刻。

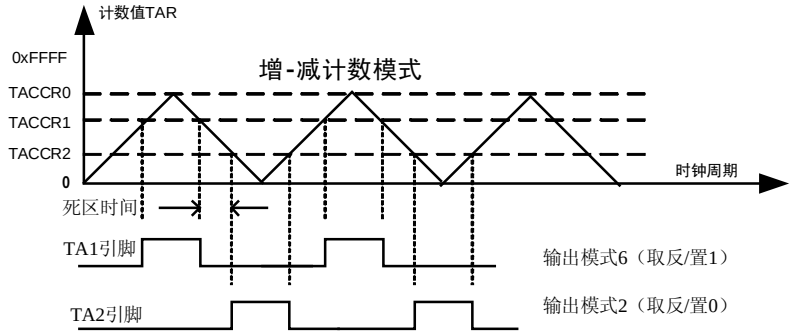


图 2.8.8 用模式 2、6 产生带死区的 PWM 波形

例 2.8.8 用 MSP430F42x 单片机驱动图 2.8.7 所示的逆变电路，要求输出交流电频率 400Hz 左右，正负半周各占 50%时间，死区时间 2us 左右。假设 SMCLK=1.048576MHz。

```
    P1SEL |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
    P1DIR |= BIT2;     //TA1 从 P1.2 输出 (不同型号单片机可能不一样)
    P2SEL |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)
    P2DIR |= BIT0;     //TA2 从 P2.0 输出 (不同型号单片机可能不一样)

    TACTL |= MC_3 + TASSEL_2 + ID_0; //定时器 TA 设为增-减计数模式,SMCLK
    TACCTL1 =OUTMOD_6; //TA1 模式 6= 高电平 PWM 输出
    TACCTL2 =OUTMOD_2; //TA2 模式 2= 低电平 PWM 输出 （两路反向的方波）
    TACCR0=1310;       //PWM 总周期 =2*1310=2620 个 SMCLK 周期约等于 400Hz
    TACCR1=656;        //TA1 占空比 = 50%左右
    TACCR2=654;        //TA2 占空比 = 50%左右      留 2 个周期的死区时间
```

推挽、桥式驱动模式下，每一路最大占空比 50%。每一路在 0~50%范围内调整占空比时，总输出功率在 0~100%之间调整。当每一路的占空比不到 50%时，剩余的时间都是死区时间，所以这种驱动方式调节功率时开关管是安全的。

例 2.8.9 在上例的基础上，编写一个输出功率设置函数。要求用 0~100 表示输出功率百分比作为传入参数。

```
/*********************************************************************
* 名    称：  Inverter_SetPower()
* 功    能：  设置逆变器的输出功率
* 入口参数：  Power:输出功率百分比 0~100 表示 0~100%
*********************************************************************/
void Inverter_SetPower(unsigned int Power)
{   int Duty;
    if(Power>99) Power=99;                      //最大只到 99%，留出死区。
    Duty=(unsigned long int)1310*Power/200;  //计算每一路的占空比
    TACCR1=1310-Duty;                           //设置 TA1 占空比
    TACCR2= Duty;                               //设置 TA2 占空比
}
```

模式 4（可变频率输出、移相输出）：

输出模式 4 下，TA 计数每次到达 TACCRx 值时，TAx 管脚电平自动取反。因此改变 TA 的计数周期可以改变 TAx 管脚的输出频率；同时若改变 TACCRx 值可以改变波形的相位。

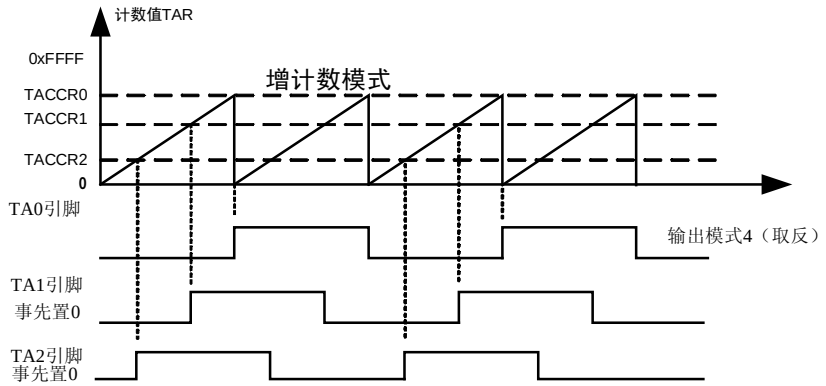


图 2.8.9　用输出模式 4 产生 3 路移相波形

如图 2.8.9，改变 TACCR0 的值即可同时改变三路输出波形的频率，改变 TACCR1 与 TACCR2 的值可以改变 TA1 与 TA2 输出波形与 TA0 波形之间的相位差。由于 TACCR1 与 TACCR2 最大只能等于 TACCR0，所以移相最大值只能滞后 0~180°。如果需要超过 180 度的移相，可以通过改变引脚初始值（反向 180 度）实现。

例 2.8.10 在 MSP430F42x 单片机中，通过 P1.0（TA0）输出引脚驱动一只喇叭，编写一个程序，用于设置喇叭的鸣响频率。SMCLK 时钟频率未知。

```
/*********************************************************************
* 名    称：Speaker_SetFreq()
* 功    能：设置音频输出方波的频率,单位 Hz。
* 入口参数：Freq: 输出频率
* 出口参数：无
* 范    例: Speaker_SetFreq(2500); 设置输出音频方波频率=2500Hz
*********************************************************************/
void Speaker_SetFreq(unsigned int Freq)
{
   unsigned long int F_TACLK;     //TA 定时器时钟频率
   int FreqMul,FLLDx;             //倍频系数、DCO 倍频
   unsigned long Period;
   TACTL |= MC_1 + TASSEL_2 + ID_0; //TA 定时器选择 SMCLK 做时钟，增计数方式。
   TACCTL0 = OUTMOD_4;            //每次到达 CCR0 取反（方波频率输出模式）
   P1SEL |= BIT0;                 //从 P1.0 输出 (不同型号单片机可能不一样)
   P1DIR |= BIT0;                 //从 P1.0 输出 (不同型号单片机可能不一样)
   FreqMul=(SCFQCTL&0x7F)+1;      //获得倍频系数
   FLLDx=((SCFI0&0xC0)>>6)+1;     //获得 DCO 倍频系数(DCOPLUS 所带来的额外倍频)
   F_TACLK=F_ACLK*FreqMul;        //计算波特率发生器时钟频率=ACLK*倍频系数
   if(FLL_CTL0&DCOPLUS) F_TACLK*=FLLDx; //若开启了 DCOPLUS,还要计算额外倍频
   Period=F_TACLK/Freq;           //计算方波时钟周期
   TACCR0=Cycle/2;   //每次到达 CCR0 电平取反，因此一个输出周期=两个 CCR0 周期
```

```
}
```

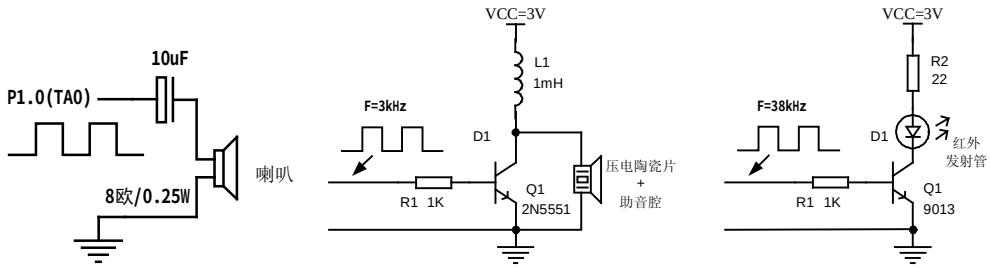
在这个程序中，为了适应各种情况，读取时钟系统的寄存器设置，计算出 SMCLK 的频率。这种方法使得调用该函数时不需要指定时钟频率，适应性和移植性都很强。

通过该函数可以实现各种频率的发音，替代蜂鸣器单调的“滴-滴”声。而且可以发出各种音符，如果结合延时程序控制时间长短和节拍，就可以实现乐曲的演奏。完整的乐谱解析与演奏请读者参考光盘中附带的《TA 定时器蜂鸣及音乐程序库》程序。该程序能解析手机铃声谱并播放，也能控制喇叭发出各种频率以及各种规律的鸣响声。

例 2.8.11 在 MSP430F42x 单片机中，通过 P1.0（TA0）输出引脚驱动一只红外发射管，发出 38kHz 调制的红外线。假设 SMCLK=4.194304MHz。

```
TACTL |= MC_1 + TASSEL_2 + ID_0; //TA 定时器选择 SMCLK 做时钟，增计数方式。
TACCTL0 = OUTMOD_4;             //每次到达 CCR0 取反（方波频率输出模式）
P1SEL |= BIT0;                  //从 P1.0 输出 (不同型号单片机可能不一样)
P1DIR |= BIT0;                  //从 P1.0 输出 (不同型号单片机可能不一样)
TACCR0=55;                      //38kHz 约等于 110 个 SMCLK 周期
```

目前红外遥控器大多采用 38kHz 的调制，配合一体化红外接收头，能获得 20 米左右的遥控感应距离。



例 2.8.10 的两种电路　　例 2.8.11 电路

图 2.8.10

在例 2.8.10 中，PWM 输出含有直流分量，需要通过一只电容隔去直流后驱动喇叭。喇叭的优点是能够发出宽范围频率的声音。如果使用低成本的压电陶瓷片作为发音器件，需要较高的驱动电压，可以采用第二种电路，利用电感通电后突然断开产生的高压驱动压电陶瓷片发出声音。如果恰好与助音腔共鸣，声音响度将超过 90 分贝。但是发音频率的范围很窄，一般都在 3kHz 附近。如果偏离共鸣频率，或者不加助音腔，声音强度将大幅度下降。用压电陶瓷片做高响度报警时，可利用 TA 可变频率发生器模式在 3kHz 附近不停扫频（类似警笛的声音），总能遇到共鸣点。

例 2.8.12 在 MSP430F42x 单片机中，从 P1.0（TA0）、P1.2（TA1）、P2.0（TA2）输出三路 50Hz 左右的方波，相位差 120° 。

```
TACTL |= MC_1 + TASSEL_1 + ID_0; //TA 定时器选择 ACLK 做时钟，增计数方式。
TACCTL0 = OUTMOD_4;
TACCTL1 = OUTMOD_4;               //三个模块都工作在方波频率输出模式
TACCTL2 = OUTMOD_4;
P1SEL |= BIT0;                  //从 P1.0 输出 (不同型号单片机可能不一样)
P1DIR |= BIT0;                  //从 P1.0 输出 (不同型号单片机可能不一样)
```

```
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)
P2SEL |= BIT0;                    //从 P2.0 输出 (不同型号单片机可能不一样)
P2DIR |= BIT0;                    //从 P2.0 输出 (不同型号单片机可能不一样)
TACCR0=328;                       //50Hz 约等于 656 个 ACLK 周期，TA0 输出
TACCR1=109;                       //TA1 输出滞后 TA0 信号 120°
TACCTL2 |= OUT;                   //TA2 输出初始相位反转（超前 TA0 信号 180 度）
TACCR1=219;                       //再滞后 60 度，超前 TA0 信号 120 度
```

例 2.8.13 在微弱信号检测学中，有一种从噪声淹没的信号中恢复微弱信号的电路叫做“锁定放大器”，它需要一个交流的激励信号以及同步信号，且根据被测系统的延迟不同，同步信号的延迟也要进行调整。

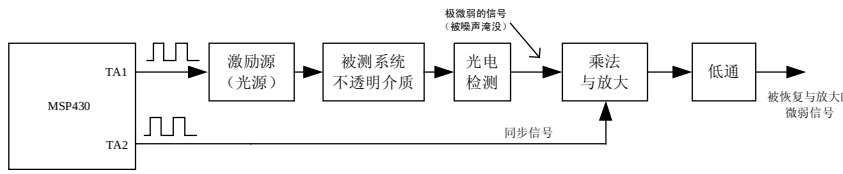


图 2.8.11 测量介质透光率的系统框图

图 2.8.11 框图所示的系统用于测量不透明介质（例如木板、铝箔）的微弱透射率。由于透射光极弱，光电变换能得到仅纳伏级电压，完全淹没在电信号噪声中。通过锁定放大器放大数万倍，并提取同频同相分量，能够将噪声相消且得到透射光强度的值。前提要求乘法环节中两路信号同频同相。由于被测环节与光电转换电路存在延迟，要求同步信号也要延迟相应的时间（一般根据实验调整）。该系统要求激励频率 1kHz 左右，两路相位差可调，MSP430 单片机 TA 比较模块的模式 4 能够用于产生该系统所需移相波形：（假设 SMCLK=1.048576MHz）

```
TACTL |= MC_1 + TASSEL_2 + ID_0; //TA 定时器选择 SMCLK 做时钟，增计数方式。
TACCTL1 = OUTMOD_4;               //三个模块都工作在方波频率输出模式
TACCTL2 = OUTMOD_4;
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)
P2SEL |= BIT0;                    //从 P2.0 输出 (不同型号单片机可能不一样)
P2DIR |= BIT0;                    //从 P2.0 输出 (不同型号单片机可能不一样)
TACCR0=1049-1;                    //1kHz 约等于 1049 个 SMCLK 周期
TACCR1=500;                       //
TACCR2=500;                       //两路输出相位差=0，以后通过软件更改，设置相位差
```

Timer_A 定时器的中断

Timer_A3 定时器的下列 4 种事件均能产生中断：

1. 主计数值计满复位，TAIFG 标志被置 1。
2. 计数值计至 TACCR0，或者捕获通道 0 发生捕获事件，CCIFG0 被置 1。
3. 计数值计至 TACCR1，或者捕获通道 1 发生捕获事件，CCIFG1 被置 1

4. 计数值计至 TACCR2，或者捕获通道 2 发生捕获事件，CCIFG2 被置 1

这四种事件占用了 2 个中断源，其中，事件 2（计至 TACCR0 或捕获通道 0 发生捕获事件）独占一个中断源 TIMERA0_VECTOR，其余 3 种事件公用另一个中断源 TIMERA1_VECTOR。对于需要紧急处理的捕获事件建议使用通道 0，因为它单独占有一个中断源，在中断内无需分支判断，响应速度最快。

和中断相关的标志位有：

- *TAIFG：Timer_A 主计数器计满复位标志* *（位于 TACTL 寄存器）*
- *TAIE：Timer_A 主计数器计满中断允许位* *（位于 TACTL 寄存器）*
- *CCIFG：Timer_A 比较/捕获模块中断标志* *（位于 TACCTL0/1/2 寄存器）*

比较模式下，当主计数器计至 TACCRx 时，该标志位置 1。在捕获模式下，当捕获条件发生，该标志位置 1。TACCTL0 内的 CCIFG 标志会在中断执行后自动清零，其余模块共用了中断入口，它们的 CCIFG 标志位会根据 TAIV 寄存器的值在执行相应的中断后自动清除。

- *CCIE：Timer_A 比较/捕获模块中断允许位* *（位于 TACCTL0/1/2 寄存器）*
- *TAIV：Timer_A 中断向量寄存器*

由于几个事件共用了 TIMERA1_VECTOR 中断向量，需要在中断服务程序中通过软件判断 TAIV 寄存器的值来确定具体的中断原因。

表 2.7.1 SD16IV 寄存器值与中断标志位的关系

TAIV 值	中断源	中断标志	优先级
00H	无中断发生	-	
02H	捕获/比较模块 1	TACCTL1 内的 CCIFG 标志	最高
04H	捕获/比较模块 2	TACCTL 2 内的 CCIFG 标志	.
06H	*捕获/比较模块 3	TACCTL 3 内的 CCIFG 标志	.
08H	*捕获/比较模块 4	TACCTL 4 内的 CCIFG 标志	.
0AH	主计数器计满溢出	TAIFG	.
0CH	保留		最低
0EH	保留		

注：带*号的中断源只对带有 Timer_A5 的单片机有效

例如，当捕获/比较模块 2 发生中断时（TAR 计到 TACCR2 的值，或捕获条件满足），TACCTL2 内的 CCIFG 标志会被自动置 1，TAIV 寄存器的值也会变为 04H，若 CCIE 中断被允许（TACCTL2 中的 CCIE=1）且总中断是开启状态，就会引发中断。在中断程序内判断 TAIV 的值，得值是捕获/比较模块 2 引发的中断，执行相应操作后返回，TACCTL2 中的 CCIFG 标志会被自动清除，TAIV 自动恢复为 0。

当多个 TimerA 中断同时产生时，TAIV 会按照优先级顺序自动先处理优先级较高的中

断。例如比较/捕获模块 1 与比较/捕获模块 2 都发生中断时，TACCTL1/2 中的 CCIFG 标志都被置 1。TAIV 会优先处理模块 1 的中断，寄存器值变为 02H。当进入中断处理完后退出，TACCTL1 中的 CCIFG 标志被自动清除，TAIV 变为 04H。中断结束之后，由于 TACCTL2 的 CCIFG 标志仍为 1，还会再次引发 TA 中断，处理模块 2 的中断事件。

第四章　MAGIC-430 学习板原理与使用说明

本章将指导

附录 1： MSP430 单片机选型表与封装(2007 年第 2 季度)

1.MSP430x1xx 无液晶驱动器系列

C：一次性编程 F：Flash 型	ROM (KB)	RAM (B)	I/O	16-Bit Timers		USART	I2C	DMA	SVS	BOR	MPY	比较器	温度传感器	ADC ch/res	附加	封装	价格 ($)
√				A	B												
MSP430F1101A	1	128	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DGV,DW,PW,24RGE	0.99
MSP430C1101	1	128	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DW,PW,24RGE	0.60
MSP430F1111A	2	128	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DGV,DW,PW,24RGE	1.35
MSP430C1111	2	128	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DW,PW,24RGE	1.10
MSP430F1121A	4	256	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DGV,DW,PW,24RGE	1.70
MSP430C1121	4	256	14	3	—	—	—	—	—	—	—	√	—	slope	—	20DW,PW,24RGE	1.35
MSP430F1122	4	256	14	3	—	—	—	—	—	√	—	—	√	5/10	—	20DW,PW,32RHB	2.00
MSP430F1132	8	256	14	3	—	—	—	—	—	√	—	—	√	5/10	—	20DW,PW,32RHB	2.25
MSP430F122	4	256	22	3	—	1	—	—	—	—	—	√	—	slope	—	28DW,PW,32RHB	2.15
MSP430F123	8	256	22	3	—	1	—	—	—	—	—	√	—	slope	—	28DW,PW,32RHB	2.30
MSP430F1222	4	256	22	3	—	1	—	—	—	√	—	—	√	8/10	—	28DW,PW,32RHB	2.40
MSP430F1232	8	256	22	3	—	1	—	—	—	√	—	—	√	8/10	—	28DW,PW,32RHB	2.50
MSP430F133	8	256	48	3	3	1	—	—	—	—	—	√	√	8/12	—	64PM,PAG,RTD	3.00
MSP430C1331	8	256	48	3	3	1	—	—	—	—	—	√	—	slope	—	64PM,RTD	2.00
MSP430F135	16	512	48	3	3	1	—	—	—	—	—	√	√	8/12	—	64PM,PAG,RTD	3.60
MSP430C1351	16	512	48	3	3	1	—	—	—	—	—	√	—	slope	—	64PM,RTD	2.30
MSP430F147	32	1024	48	3	7	2	—	—	—	—	√	√	√	8/12	—	64PM,PAG,RTD	5.05
MSP430F1471	32	1024	48	3	7	2	—	—	—	—	√	√	—	slope	—	64PM,RTD	4.60
MSP430F148	48	2048	48	3	7	2	—	—	—	—	√	√	√	8/12	—	64PM,PAG,RTD	5.75
MSP430F1481	48	2048	48	3	7	2	—	—	—	—	√	√	—	slope	—	64PM,RTD	5.30
MSP430F149	60	2048	48	3	7	2	—	—	—	—	√	√	√	8/12	—	64PM,RTD	6.05
MSP430F1491	60	2048	48	3	7	2	—	—	—	—	√	√	—	slope	—	64PM,RTD	5.60
MSP430F155	16	512	48	3	3	1	√	√	√	√	—	√	√	8/12	(2)DAC12	64PM,RTD	4.95
MSP430F156	24	1024	48	3	3	1	√	√	√	√	—	√	√	8/12	(2)DAC12	64PM,RTD	5.55
MSP430F157	32	1024	48	3	3	1	√	√	√	√	—	√	√	8/12	(2)DAC12	64PM,RTD	5.85
MSP430F167	32	1024	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	6.75
MSP430F168	48	2048	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	7.45
MSP430F169	60	2048	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	7.95
MSP430F1610	32	5120	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	8.25
MSP430F1611	48	10240	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	8.65
MSP430F1612	55	5120	48	3	7	2	√	√	√	√	√	√	√	8/12	(2)DAC12	64PM,RTD	8.95

2. MSP430x2xx 高速系列

C：一次性编程 F：Flash 型	ROM (KB)	RAM (B)	I/O	16-Bit Timers		USCI	USI2	DMA	SVS	BOR	MPY	Comp_A+	温度传感器	ADC Ch/Res	附加	封装	价格 ($)
√				A	B												
MSP430F2001	1	128	10	2	—	—	—	—	—	√	—	√	—	Slope	—	14PW,N,16RSA	0.55
MSP430F2011	2	128	10	2	—	—	—	—	—	√	—	√	—	Slope	—	14PW,N,16RSA	0.70
MSP430F2002	1	128	10	2	—	—	√	—	—	√	—	—	√	8/10	—	14PW,N,16RSA	0.99
MSP430F2012	2	128	10	2	—	—	√	—	—	√	—	—	√	8/10	—	14PW,N,16RSA	1.15
MSP430F2003	1	128	10	2	—	—	√	—	—	√	—	—	√	4/16	—	14PW,N,16RSA	1.50
MSP430F2013	2	128	10	2	—	—	√	—	—	√	—	—	√	4/16	—	14PW,N,16RSA	1.65
MSP430F2101	1	128	16	3	—	—	—	—	—	√	—	√	—	Slope	—	20DGV,DW,PW,24RGE	0.90
MSP430F2111	2	128	16	3	—	—	—	—	—	√	—	√	—	Slope	—	20DGV,DW,PW,24RGE	0.99
MSP430F2121	4	256	16	3	—	—	—	—	—	√	—	√	—	Slope	—	20DGV,DW,PW,24RGE	1.35
MSP430F2131	8	256	16	3	—	—	—	—	—	√	—	√	—	Slope	—	20DGV,DW,PW,24RGE	1.70
MSP430F2234	8	512	32	3	3	√	—	—	—	√	—	—	—	12/10	(2)OPAMP	38DA,40RHA	2.75
MSP430F2254	16	512	32	3	3	√	—	—	—	√	—	—	—	12/10	(2)OPAMP	38DA,40RHA	3.10
MSP430F2274	32	1024	32	3	3	√	—	—	—	√	—	—	—	12/10	(2)OPAMP	38DA,40RHA	3.55

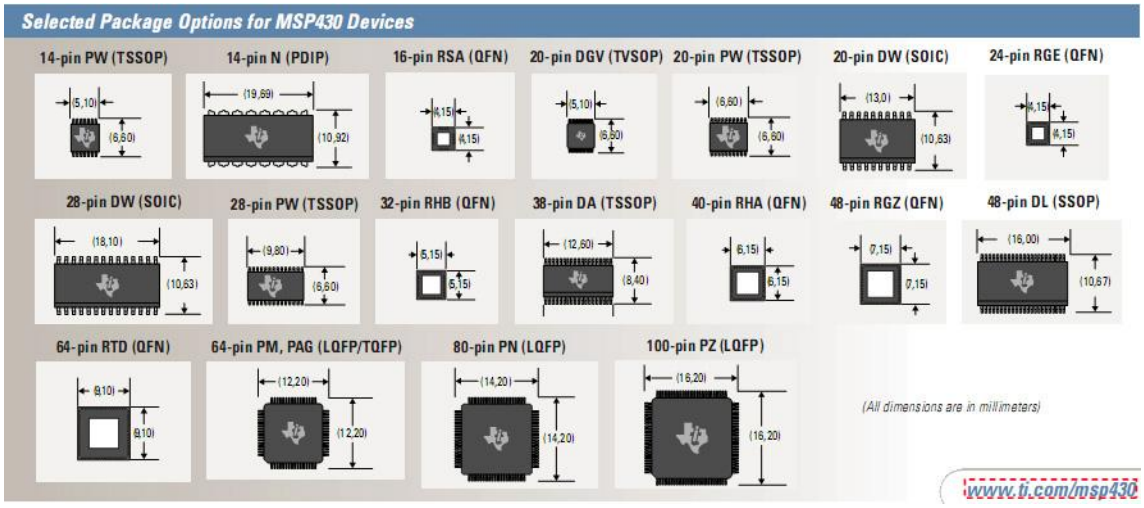
3.MSP430x4xx 系列带有 LCD 驱动系列

C：一次性编程 F：Flash 型	ROM (KB)	RAM (B)	I/O	16-Bit Timers		USART	USCI	LCD 段	DMA	SVS	BOR	MPY	Comp_A	温度传感器	ADC Ch/Res	附加	封装	价格 ($)
√				A	B													
MSP430F412	4	256	48	3	—	—	—	96	—	√	√	—	√	—	slope	—	64PM,RTD	2.60
MSP430C412	4	256	48	3	—	—	—	96	—	√	√	—	√	—	slope	—	64PM,RTD	1.90
MSP430F413	8	256	48	3	—	—	—	96	—	√	√	—	√	—	slope	—	64PM,RTD	2.95
MSP430C413	8	256	48	3	—	—	—	96	—	√	√	—	√	—	slope	—	64PM,RTD	2.10
MSP430F415	16	512	48	3,5	—	—	—	96	—	√	√	—	√	—	slope	—	64PM	3.40
MSP430F417	32	1024	48	3,5	—	—	—	96	—	√	√	—	√	—	slope	—	64PM	3.90
MSP430FW423	8	256	48	3,5	—	—	—	96	—	√	√	—	√	—	slope	Flow-meter	64PM	3.75
MSP430FW425	16	512	48	3,5	—	—	—	96	—	√	√	—	√	—	slope	Flow-meter	64PM	4.05
MSP430FW427	32	1024	48	3,5	—	—	—	96	—	√	√	—	√	—	slope	Flow-meter	64PM	4.45
MSP430F4250	16	256	32	3	—	—	—	56	—	—	√	—	—	√	5/16	DAC12	48DL,RGZ	3.10
MSP430F4260	24	256	32	3	—	—	—	56	—	—	√	—	—	√	5/16	DAC12	48DL,RGZ	3.45
MSP430F4270	32	256	32	3	—	—	—	56	—	—	√	—	—	√	5/16	DAC12	48DL,RGZ	3.80
MSP430F423	8	256	14	3	—	1	—	128	—	√		√	—	√	3/16	—	64PM	4.50
MSP430F425	16	512	14	3	—	1	—	128	—	√	√	√	—	√	3/16	—	64PM	4.95
MSP430F427	32	1024	14	3	—	1	—	128	—	√	√	√	—	√	3/16	—	64PM	5.40
MSP430FE423	8	256	14	3	—	1	—	128	—	√	√	√	—	√	3/16	E meter	64PM	4.85

MSP430FE425	16	512	14	3	—	1	—	128	—	√	√	√	—	√	3/16	E meter	64PM	5.45
MSP430FE427	32	1024	14	3	—	1	—	128	—	√	√	√	—	√	3/16	E meter	64PM	5.95
MSP430F435	16	512	48	3	3	1	—	128 /160	—	√	√	—	√	√	8/12	—	80PM,100PZ	4.45
MSP430F436	24	1024	48	3	3	1	—	128 /160	—	√	√	—	√	√	8/12	—	80PM,100PZ	4.70
MSP430F437	32	1024	48	3	3	1	—	128 /160	—	√	√	—	√	√	8/12	—	80PM,100PZ	4.90
MSP430FG437	32	1024	48	3	3	1	—	128	√	√	√	—	√	√	8/12	(2)DAC12, (3)OPAMP	80PN	6.50
MSP430FG438	48	2048	48	3	3	1	—	128	√	√	√	—	√	√	8/12	(2)DAC12, (3)OPAMP	80PN	7.35
MSP430FG439	60	2048	48	3	3	1	—	128	√	√	√	—	√	√	8/12	(2)DAC12, (3)OPAMP	80PN	7.95
MSP430F447	32	1024	48	3	7	2	—	160	—	√	√	√	√	√	8/12	—	100PZ	5.75
MSP430F448	48	2048	48	3	7	2	—	160	—	√	√	√	√	√	8/12	—	100PZ	6.50
MSP430F449	60	2048	48	3	7	2	—	160	—	√	√	√	√	√	8/12	—	100PZ	7.05
MSP430FG4616	92	4096	80	3	7	1	√	160	√	√	√	√	√	√	12/12	(2)DAC12 ,(3)OPAMP	100PZ	9.45
MSP430FG4617	92	8192	80	3	7	1	√	160	√	√	√	√	√	√	12/12	(2)DAC12 ,(3)OPAMP	100PZ	9.95
MSP430FG4618	116	8192	80	3	7	1	√	160	√	√	√	√	√	√	12/12	(2)DAC12 ,(3)OPAMP	100PZ	10.35
MSP430FG4619	120	4096	80	3	7	1	√	160	√	√	√	√	√	√	12/12	(2)DAC12 ,(3)OPAMP	100PZ	9.95

注：参考价格为美国离岸价，不含关税。1000 片报价。实际在不同国家或地区，随市场供求关系可能会略有调整。

4. MSP430 单片机封装列表：



附录 2： 常用纽扣电池参数

锂锰扣式电池规范表							
型号：	标称电压 (v)	标称容量 (mAH)	标准电流 (mA)	连续电流（最大） (mA)	脉冲电流（最大） (Max) (mA)	最大尺寸 (mm)	重量 (g.)
CR1216	3	25	0.1	0.2	5	Φ12.5×1.6	0.7
CR1220	3	38	0.1	0.2	5	Φ12.5×2.0	0.9
CR1225	3	50	0.1	0.2	5	Φ12.5×2.5	1
CR1530	3	85	0.1	0.3	7	Φ15.0×3.0	1.4
CR1616	3	50	0.1	0.3	8	Φ16.0×1.6	1.2
CR1620	3	70	0.1	0.3	8	Φ16.0×2.0	1.3
CR1632	3	120	0.1	0.3	8	Φ16.0×3.2	2.3
CR2016	3	75	0.1	0.3	10	Φ20.0×1.6	1.8
CR2025	3	150	0.2	0.5	15	Φ20.0×2.5	2.5
CR2032	3	210	0.2	0.5	15	Φ20.0×3.2	3.1
CR2320	3	130	0.2	0.5	15	Φ23.0×2.0	3
CR2330	3	260	0.2	0.5	15	Φ23.0×3.0	4
CR2430	3	270	0.2	1	15	Φ24.5×3.0	4.2
CR2450	3	550	0.2	1	15	Φ24.5×5.0	6.2
CR2477	3	950	0.2	1	15	Φ24.5×7.7	10.2