TI MSP430F5529PocketBoard

口袋板实验指导书

Rev 3.0

华清科仪（北京）科技有限公司
Huatsing Instruments

目　　录

第 1 章 MSP430F5529 口袋板实验开发系统

MSP430F5529 口袋板（MSP430F5529 Pocket Kit，以下简称 F5529PK）实验开发系统配合 TI MSP430F5529LaunchPad 开发板（以下简称 LP 板）使用，板子上集成了电子纸屏幕（电子墨水屏）、模拟滤波器、用户 LED 与按键、蜂鸣器、音频功放、温度传感器、DAC、SD 卡座、耳机插座、信号输入/输出接口等模块，大大扩展了 MSP430F5529LaunchPad 开发板的实验内容与应用领域。

通过口袋板上的众多模块，学生不但可以完成 MCU、数字电路、模拟电路等基础实验项目，而且可以完成音频播放、录音回放、温度测量等趣味性综合实验。口袋板非常适合学生在实验室之外（宿舍、家中）进行一些科技创新、电子设计大赛活动时使用。

1.1 MSP430F5529 微控制器介绍

1.1.1 MSP430F5529 特性

- 低电源电压范围：1.8V 至 3.6V
- 超低功耗：
  - 激活模式(AM)：
    所有系统时钟激活：
    在 8MHz，3.0V，闪存程序执行时为 290μA/MHz（典型值）
  - 待机模式 (LPM3)：
    带有晶振的安全装置、且电源监控器可用、完全 RAM 保持、快速唤醒；
    2.2V 时为 1.9μA，3.0V 时为 2.1μA（典型值）
  - 关闭（off）模式 (LPM 4)：
    3.0V 时为 1.1μA（典型值）
  - 关断（shutdown）模式 (LPM4.5)：
    3.0V 时为 0.18μA（典型值）
- 在 3.5μs 内从待机模式唤醒（典型值）
- 16 位精简指令集 (RISC) 架构、扩展内存、高达 25MHz 的系统时钟
- 灵活的电源管理系统
  - 具有可编程经稳压内核电源电压的完全集成低压降稳压器 (LDO)
  - 电源电压监控、监视、和临时限电
- 统一时钟系统
  - 针对频率稳定的锁频环路 (FLL) 控制环路
  - 低功耗低频内部时钟源 (VLO)
  - 低频修整内部参照源 (REFO)
  - 32kHz 钟表晶体 (XT1)
  - 高达 32MHz 的高频晶振 (XT2)
- 4 个配有 3，5，7 个捕捉/比较寄存器的 16 位定时器

- 2 个通用串行通信接口
- 全速通用串行总线 (USB)
- 具有内部共用基准、采样保持、和自动扫面功能的 12 位模数 (A/D) 转换器
- 电压比较器
- 支持 32 位运算的硬件乘法器
- 串行在板可编程，无需外部编程电压
- 3 通道内部 DMA
- 具有实时时钟模块的基本定时器

1.1.2 MSP430F5529 引脚图及结构框图

MSP430F5529 的引脚图如图 1.1.1 所示，结构框图如图 1.1.2 所示。

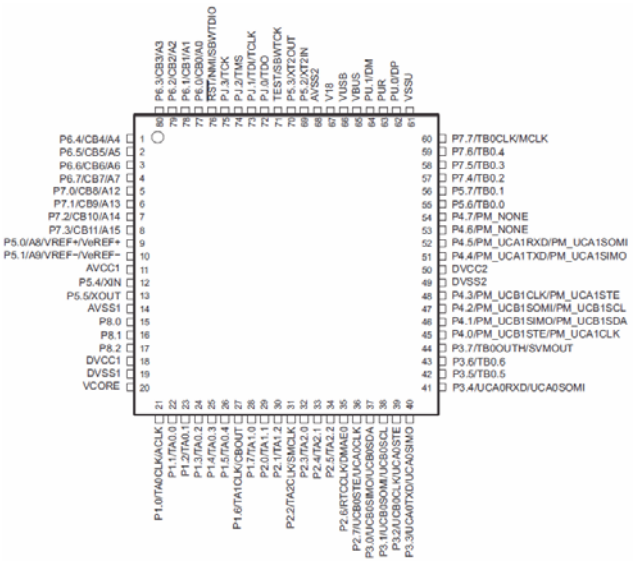


图 1.1.1　MSP430F5529 引脚图

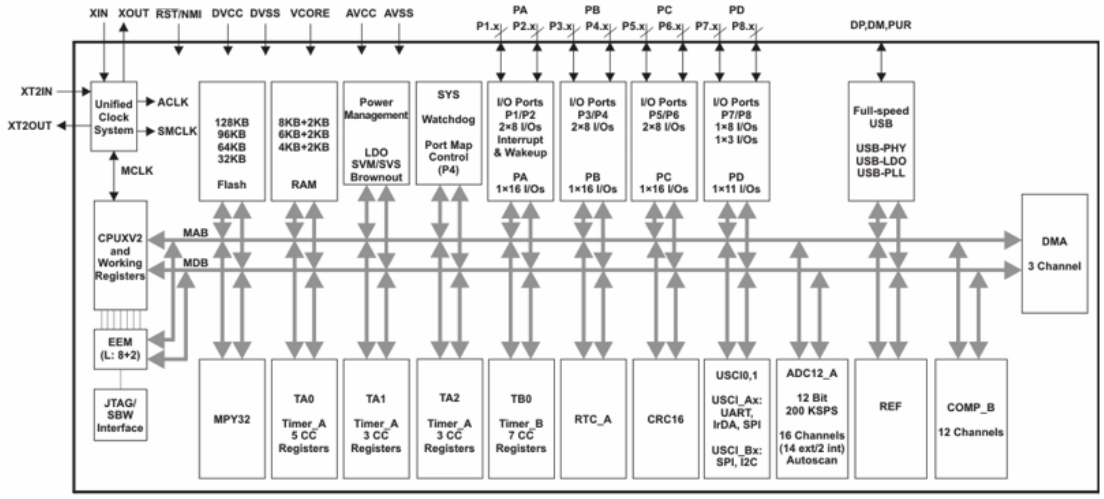


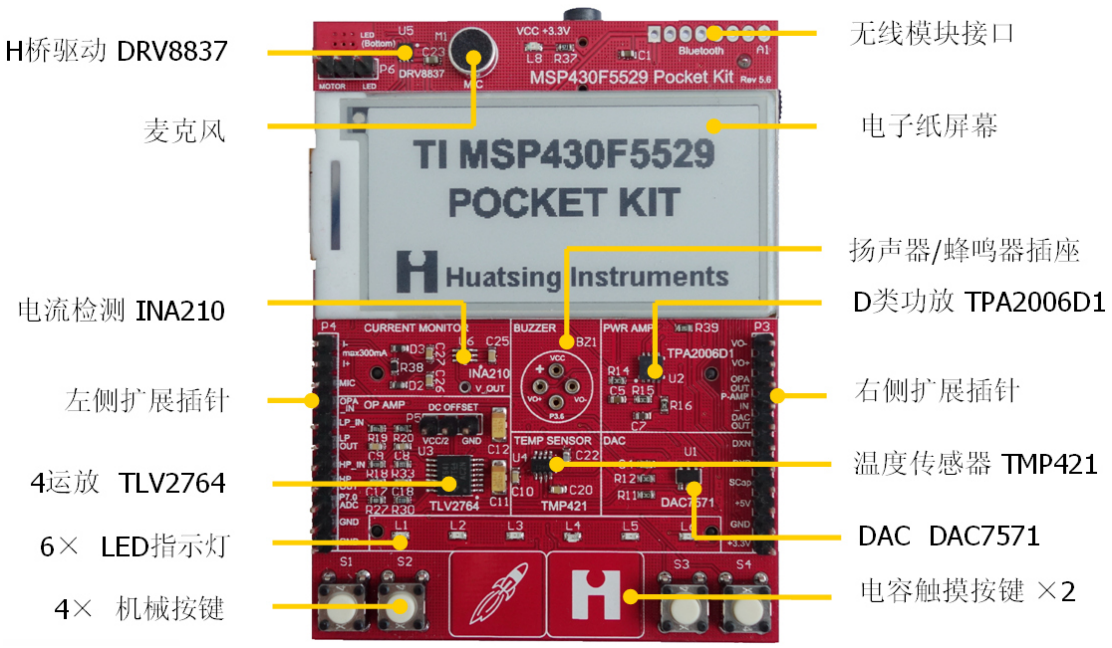
图 1.1.2　MSP430F5529 结构框图

1.2 MSP430F5529 口袋板硬件资源介绍

MSP430F5529 口袋板上主要集成了如下模块：

- 电子纸显示屏（电子墨水屏）：I2C 接口，分辨率 250×122
- 高功率 LED 模块
- 电流检测模块
- 有源滤波器模块：对用户开放一个二阶低通滤波器和一个二阶高通滤波器，其中低通滤波器的直流偏置电压可在）0V 与 1/2VCC 间进行切换
- 无源蜂鸣器/扬声器模块
- 音频功率放大器模块
- 温度传感器模块
- 串行 DAC 模块
- 用户 LED 模块：6 个 LED 灯可供用户编程使用
- 用户按键模块：4 个机械按键、2 个电容触摸按键可供用户编程使用
- 无线扩展模块接口
- 耳机插座：可以连接耳机，也可以做立体声 Line Out 接口使用
- MicroSD 卡插座
- BoosterPack 插针：用于连接 MSP430F5529LaunchPad 板子的上的 BoosterPack 插座（背对背连接）
- 信号接口：可以通过杜邦线将信号引入/引出

口袋板硬件资源图请看图 1.2.1（分别为口袋板正面与背面）。



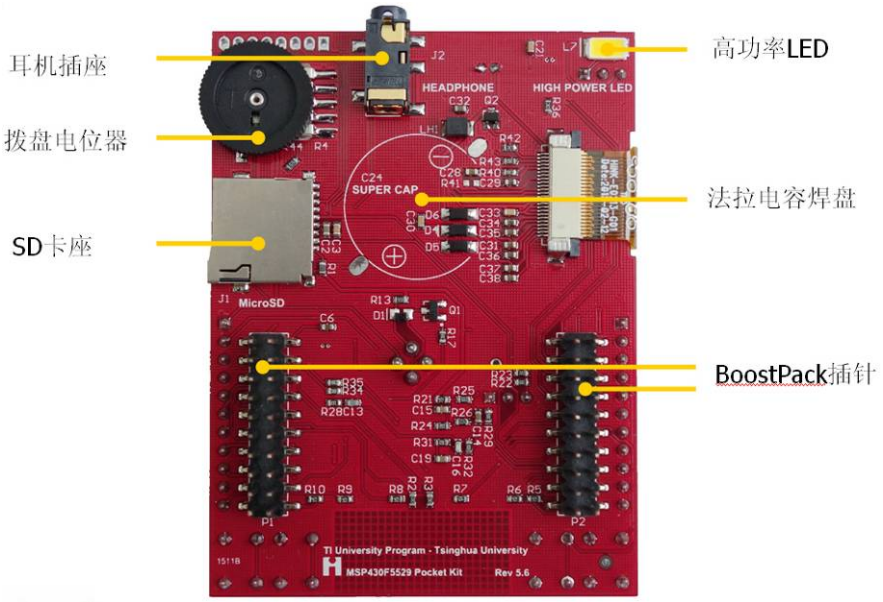


图 1.2.1　MSP430F5529 口袋板正面与背面资源图

1.2.1 电子纸屏幕（电子墨水屏）模块

口袋板正面上方是一块 2.1 英寸的电子纸屏幕（电子墨水屏），分辨率 250×122，SPI 接口。

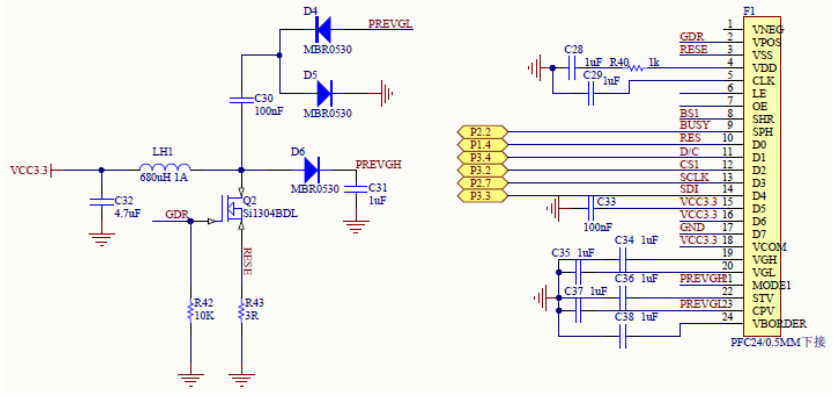


图 1.2.2　电子纸屏幕模块原理图

1.2.2 H 桥驱动模块与高功率 LED 模块

口袋板正面做上角是低压 H 桥驱动器 DRV8837，用它来驱动口袋板背面的高功率 LED（L7），也预留出了电机接口，可以用来驱动扩展的电机模块。

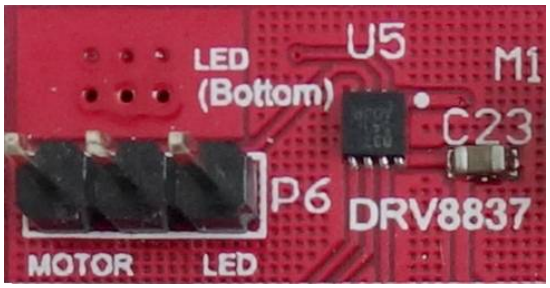


图 1.2.3　H 桥驱动模块

图 1.2.4　高功率 LED

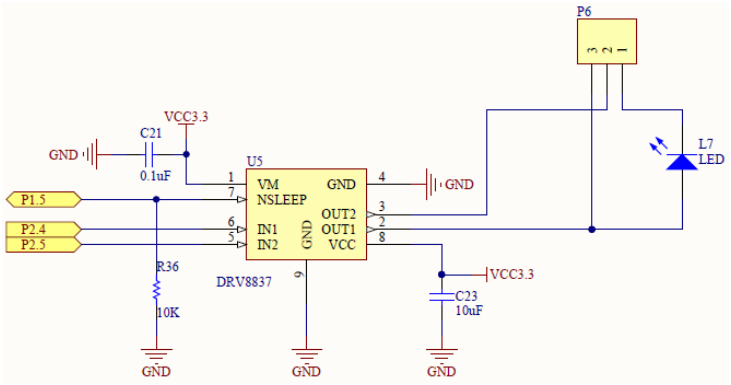


图 1.2.5　H 桥驱动模块原理图

1.2.3 电流测量模块

在口袋版屏幕下方就是电压输出型电流检测芯片 INA210，将模块串联接入电路中,就会测量出电路中的电流，输入电流与输出电流分别接入左侧扩展插针 P4.2（I+）与 P4.1（I-）端，最大输入电流不可超过 300mA。

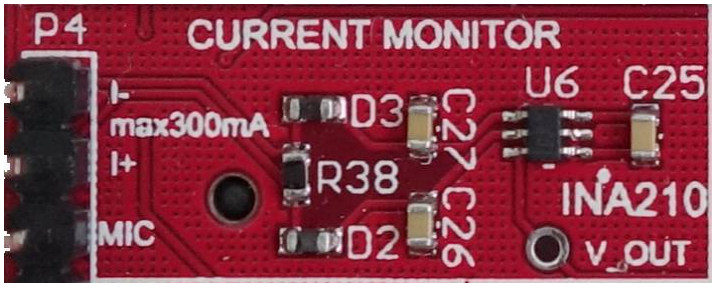


图 1.2.6　电流检测模块

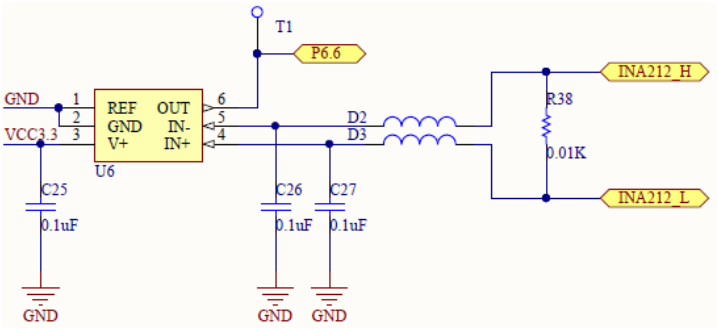


图 1.2.7　电流测量原理图

1.2.4 滤波器模块与前级放大模块

口袋板正面左下方是 TLV2764 四运放模块，该芯片内部集成了 4 个独立的运算放大器。利用其中 2 个运算放大器搭建了一个二阶有源低通滤波器和一个二阶有源高通滤波器，这两个滤波器都向用户开放，滤波器的输入输出管脚分别为左侧信号接口插针 P4.5（LP_IN）、P4.6（LP_OUT）、P4.7（HP_IN）、P4.8（HP_OUT），用户可以通过杜邦线引入引出信号。

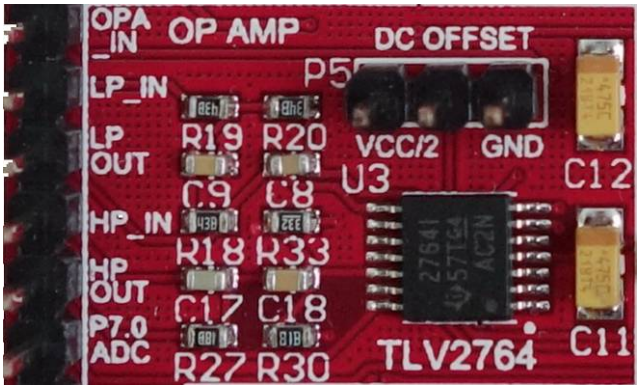


图 1.2.8　滤波器模块

低通滤波器采用的是多重反馈型（multi-feedback）型二阶低通滤波器，这种形式的滤波器比 Sallen-Key 型要多一个电阻，但是它在高频端的衰减特性要好的多，并且大信号输入时失真也小，低通滤波器的截止频率是 8KHz。

低通滤波器的偏置电压可以通过跳线在 0V 与 1/2VCC 间进行切换，因此可以无论本身有无直流偏置的信号都可以输入。

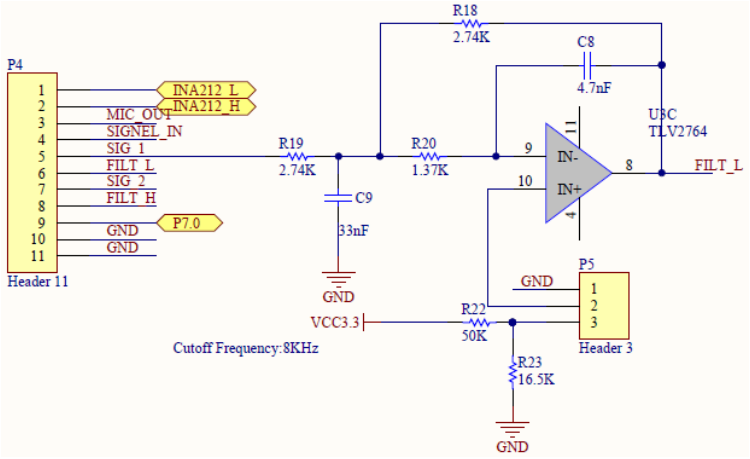


图 1.2.9　低通滤波器原理图

高通滤波器采用的是 Sallen-Key 结构，截止频率 500Hz。

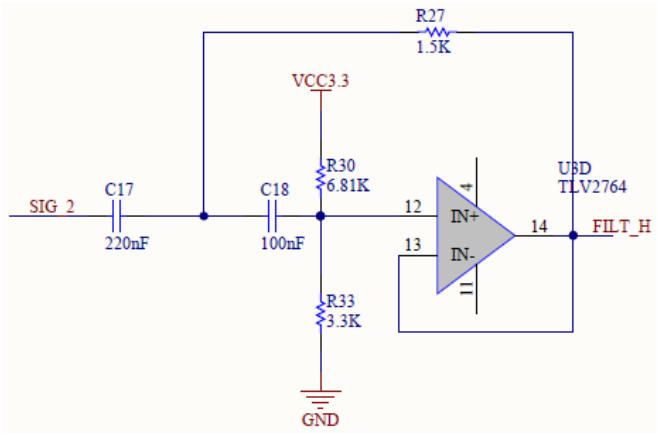


图 1.2.10　高通滤波器原理图

利用 TLV2764 的另外一个运算放大器搭建了一个对小信号进行电压放大的前级放大器，信号输入端为左侧扩展插针 P4.4（OPA_IN）。最简单直接的用法就是对口袋板上麦克风输出的微小信号进行前级放大：只需将麦克风输出信号 P4.3（MIC）与 P4.4（OPA_IN）用跳线短接即可。

1.2.5 蜂鸣器/扬声器模块

屏幕正下方是蜂鸣器/扬声器插座，这里有 4 个圆孔座，当把无源蜂鸣器的两个管脚插入上下两个圆孔座中，蜂鸣器作为无源蜂鸣器使用，我们可以通过 F5529 的 P3.6 口控制蜂鸣器的发声频率；当把蜂鸣器插入左右两个圆孔座中，是将无源蜂鸣器当做扬声器使用，因此我们一定要选用低阻抗（比如 16Ω）的无源蜂鸣器，否则会因为阻抗太大，声音很小。当然，我们也可以将正规扬声器的两个引脚接入这两个圆孔座。左右两个圆孔座分别与右侧扩展接口 P3.2（VO+）、P3.1（VO-）是连通的。

图 1.2.11　无源蜂鸣器模块

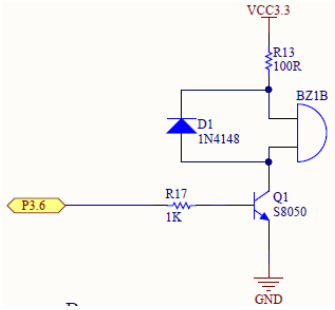


图 1.2.12　蜂鸣器模块原理图

1.2.6 温度传感器模块

蜂鸣器下方是 I2C 总线的双温区数字温度传感器芯片 TMP421，该芯片能够测量本地（Local）温度（芯片端的温度）与远程（Remote）温度，测量远程温度需要将一段 2-pin 等长的杜邦线一端连接在右侧扩展接口 P3.6（DXN）与 P3.7（DXP）上，另一端按一定规则连接一个指定型号的三极管，三极管端的文度即为远程温度。

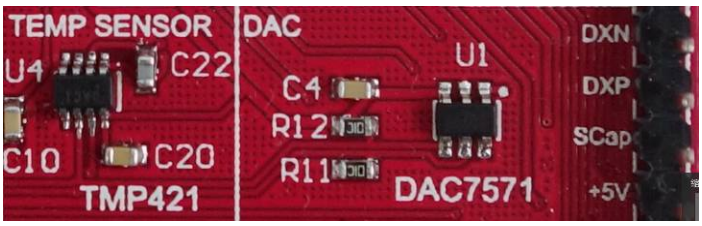


图 1.2.13　温度传感器模块

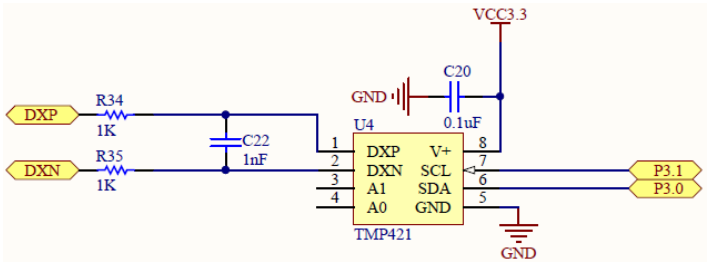


图 1.2.14　温度传感器模块原理图

1.2.7 DAC 模块

F5529 芯片内部没有 DAC 转换器，因此口袋板上配置了一片 I^2C 总线接口的 12 位精度芯片 DAC7571，DAC 的输出端在右侧扩展接口 P4.5（DAC OUT）。

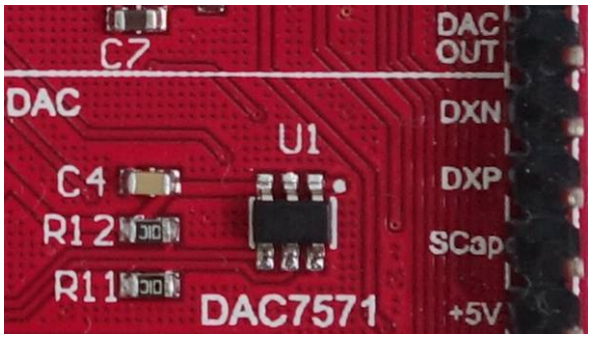


图 1.2.15　DAC 模块

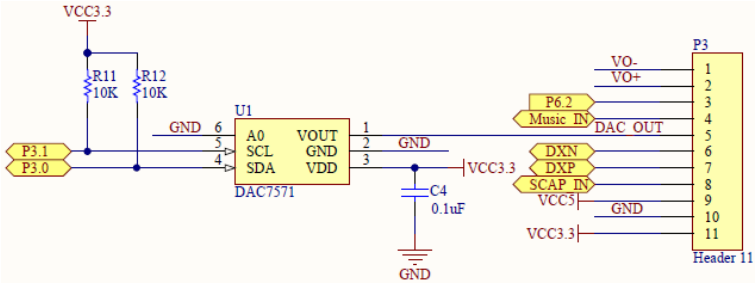


图 1.2.16　DAC 转换模块原理图

1.2.8 音频功放模块

蜂鸣器模块右侧是单声道 D 类功率放大器 TPA2006D1，该芯片采用 BTL 输出，没有“地”参考点，这点在使用与测量上要特别注意，不要造成短路。该芯片的两个输出端分别是右侧扩展插针 P3.1（VO-）、P3.2（VO+），这个两个插针又分别连通于蜂鸣器模块上 VO-与 VO+两个圆孔座。功放模块的输入端是 P3.4（P-AMP IN）

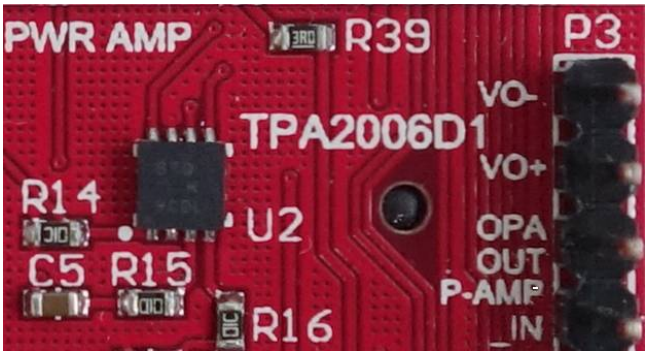


图 1.2.17　音频功放模块

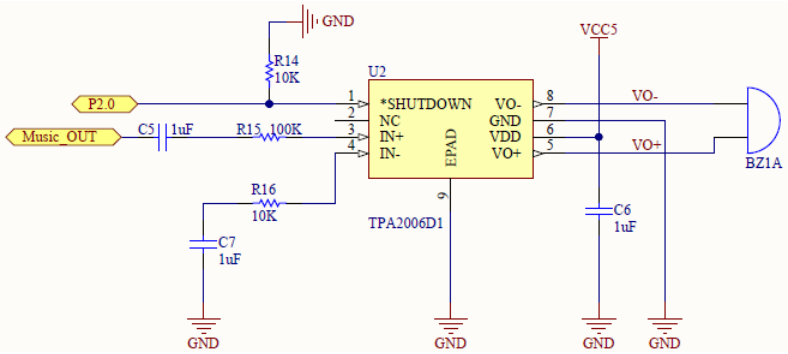


图 1.2.18　音频功放模块原理图

1.2.9 LED/机械按键/触摸按键模块

口袋板正面设置了 6 个 LED 指示灯（LED1—LED6），供用户编程使用。

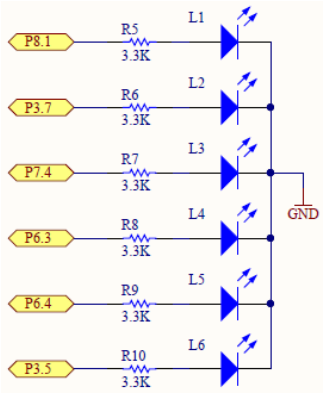


图 1.2.19　LED 模块原理图

口袋板正面下端设置了 4 个机械按键（S1—S4）与 2 个电容触摸按键（Pad—Pad2），供用户编程使用。

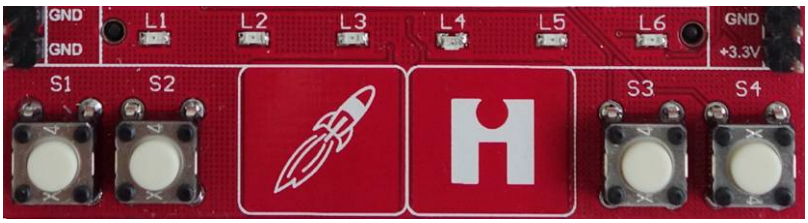


图 1.2.20　LED 模块按键模块

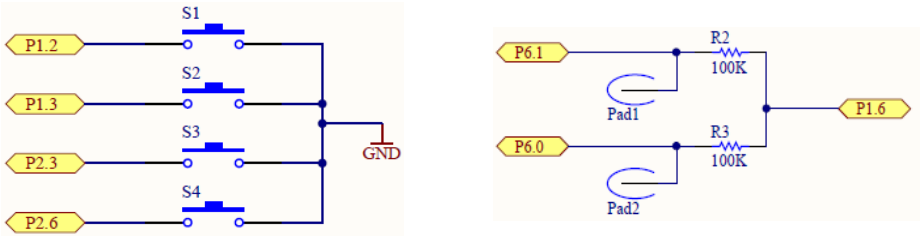


图 1.2.21　机械按键与触摸按键原理图

1.2.10 拨盘电位器

口袋板背面设置了一个双路 50KΩ 的拨盘电位器，一个抽头连接到到了 F5529 的 ADC 端（P6.5），另一个抽头连接到 TPA2006D1 芯片的输入端，同时也连接到耳机插座的输出端，起到对信号的衰减作用。

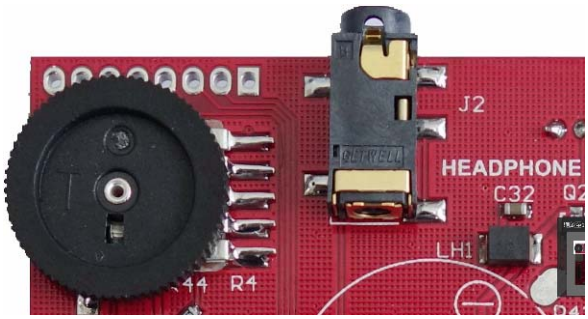


图 1.2.22　拨盘电位器模块

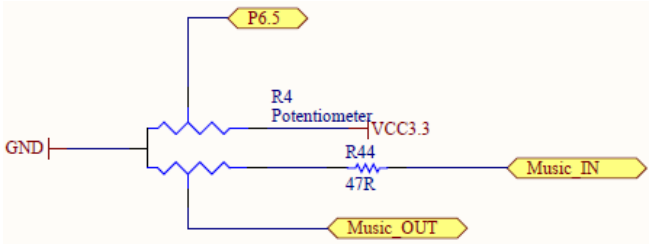


图 1.2.23　拨盘电位器模块原理图

1.2.11 耳机插座

耳机插座在口袋板背面上方，可以连接标准 3.5mm 耳机插头，由于整个信号链路都是单声道的，因此将左右声道并接到了一起。该插座也可以做 Line Out 接口使用，输出的信号电压受拨盘电位器控制。

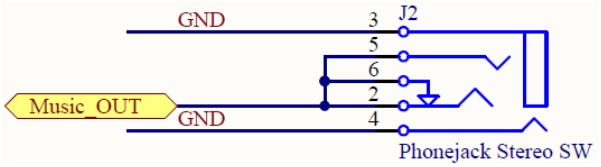


图 1.2.24　耳机插座原理图

1.2.12 SD 卡座

口袋板背面设置了 1 个 MicroSD 卡座，我们可以在 SD 卡中存放音频或图像文件，然后通过口袋板进行播放或显示。

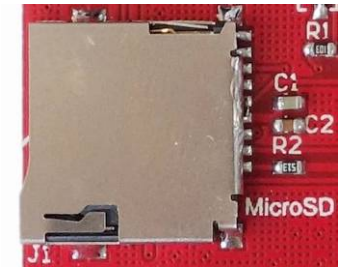


图 1.2.25　SD 卡座

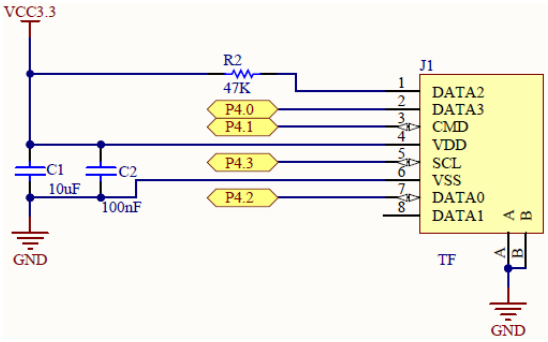


图 1.2.26　SD 卡座部分原理图

1.2.13 BoosterPack 接口

口袋板背面的 BoosterPack 接口用于和 MSP430F5529LaunchPad 进行连接，两块板子采用背对背的连接方式：**插入时请注意不要插错行，拔出时请注意不要把插针弄弯。**

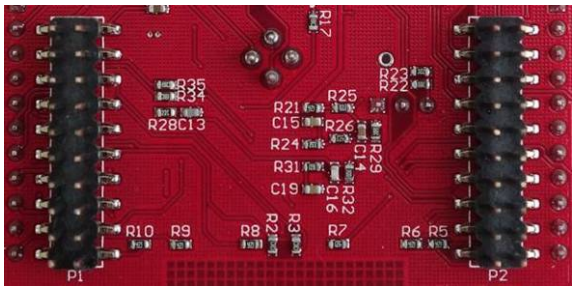


图 1.2.27　BoosterP 插针

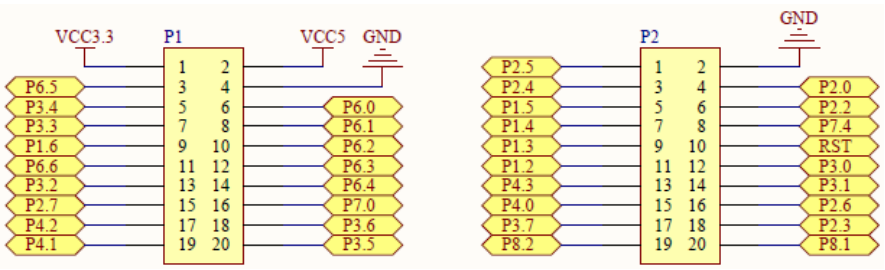


图 1.2.28　BoosterP 插针连接图

1.2.14 信号接口

口袋板正面中间两侧有两列单排插针作为一些信号端口使用，我们可以通过杜邦线将信号引入或引出。其中左侧插针主要是滤波器输入端与输出端；右侧信号是 DAC 模块输出端，功放模块输入端以及电源。

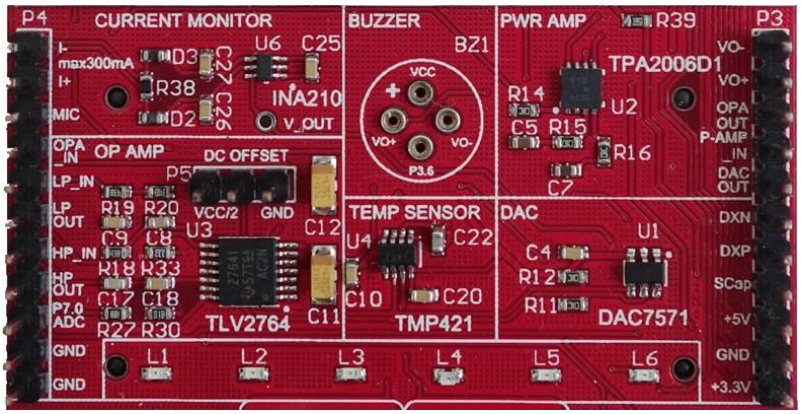


图 1.2.29　信号接口

信号接口（P4、P3）引脚定义见下：

位置	序号	定义	说明
P4（左侧）	1	I-	电流测试输入负端与正端，最大允许通过电流 300mA
	2	I+	
	3	MIC	麦克风输出信号
	4	OPA_IN	前级放大器输入信号
	5	LP_IN	低通滤波器输入端
	6	LP_OUT	低通滤波器输出端
	7	HP_IN	高通滤波器输入端
	8	HP_OUT	高通滤波器输入端
	9	P7.0/ADC	ADC 输入端，最大允许输入电压 3.3V
	10	GND	地信号，可以作为跳线帽收纳端子使用
	11	GND	
P3（右侧）	1	VO-	功放芯片 BTL 输出负端
	2	VO+	功放芯片 BTL 输出正端
	3	OPA OUT	前级放大器输出端
	4	P-AMP_IN	功率放大器输入端
	5	DAC OUT	DAC 输出端
	6	DXN	远程温度信号连接端
	7	DXP	远程温度信号连接端
	8	SCap	法拉电容正极
	9	+5V	+5V 信号
	10	GND	地信号
	11	+3.3V	+3.3V 信号

表 1.2.1 信号接口定义

1.2.15 无线通信接口

口袋板正面右上角有一个 8PIN 的无线通信接口,如图 1.2.30 所示。

图 1.2.30　无线通信接口位置

该通信接口包含了三线制 SPI 通信接口，采用 SPI 的无线模块都可以和本口袋板相连接。如图 1.2.31 所示。

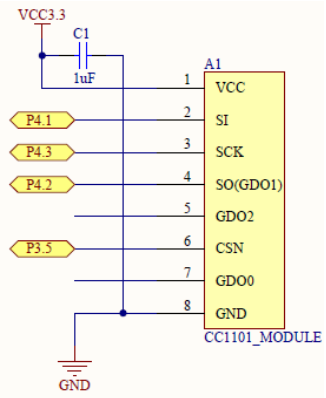


图 1.2.31　无线通信接口

第 2 章 集成开发环境 CCS 的安装与使用

CCS（Code Composer Studio）是 TI 公司研发的一款具有环境配置、源文件编辑、程序调试、跟踪和分析等功能的集成开发环境，能够帮助用户在一个软件环境下完成编辑、编译、链接、调试和数据分析等工作。CCS 是 MSP430 软件开发的理想工具，这里以 CCSv6 为例说明该软件的安装使用，其他版本大同小异。

2.1 CCSv6 的安装

（1）运行下载的安装程序 ccs_setup_xxx.exe（xxx 代表 CCS 版本号），当运行到如图 2.1.1 处时，选择 CCS 安装路径，默认路径是 c:\ti，但如果 C 盘装有还原卡或是空间很小，请选择安装到其他硬盘分区。

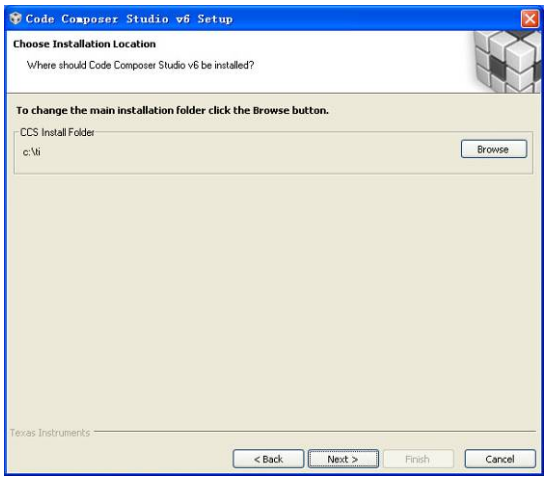


图 2.1.1 安装过程—选择安装目录

（2）单击 Next 得到如图 2.1.2 所示的窗口，为了安装快捷，在此只选择支持 MSP430 Ultra Low Power MCUs 的选项即可。单击 Next，继续安装。

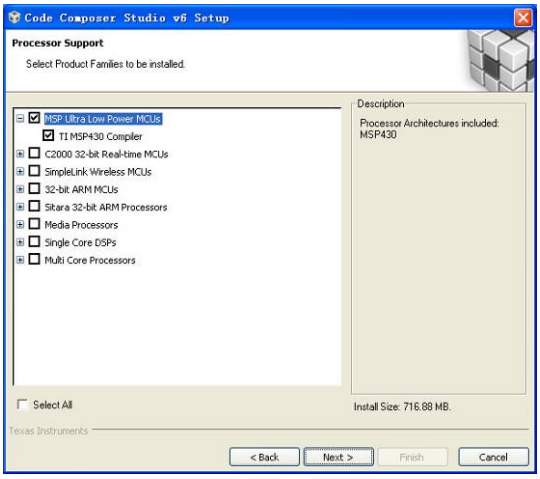


图 2.1.2 安装过程—选择安装项目

（3）在工具与软件安装选择页，请**不要**勾选任何工具与软件，因为这些组件都需要从网络上下载，这会让安装过程异常缓慢。如果需要这些软件，当软件安装完成后我们也能够在使用过程中下载安装。

图 2.1.3　APP Center

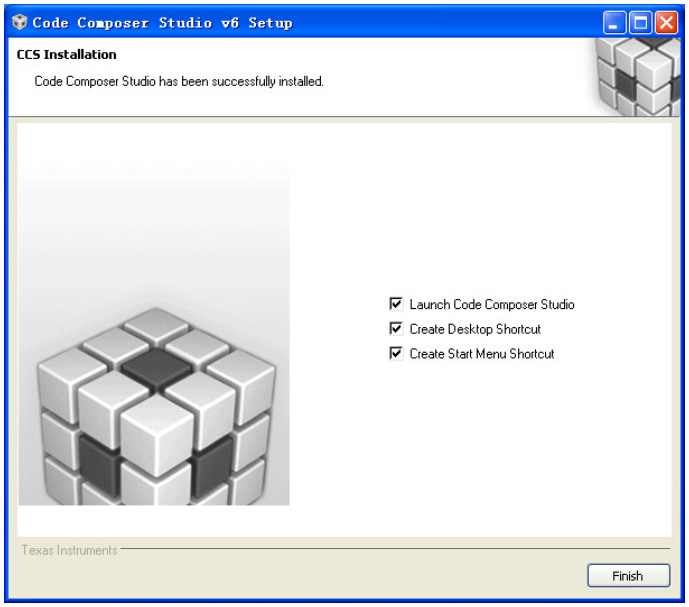


图 2.1.4　软件安装完成

（3）单击 Finish，将运行 CCS，弹出如图 2.1.5 所示窗口，我们可以在电脑中合适的区域建立工作空间（workspace），如果我们不再更换工作区，可以将"Use this as the default and do not ask again"选项勾选上。

图 2.1.5　Workspace 选择窗口

（4）单击 OK，至此软件安装配置工作全部完成，如图 2.1.6 所示。

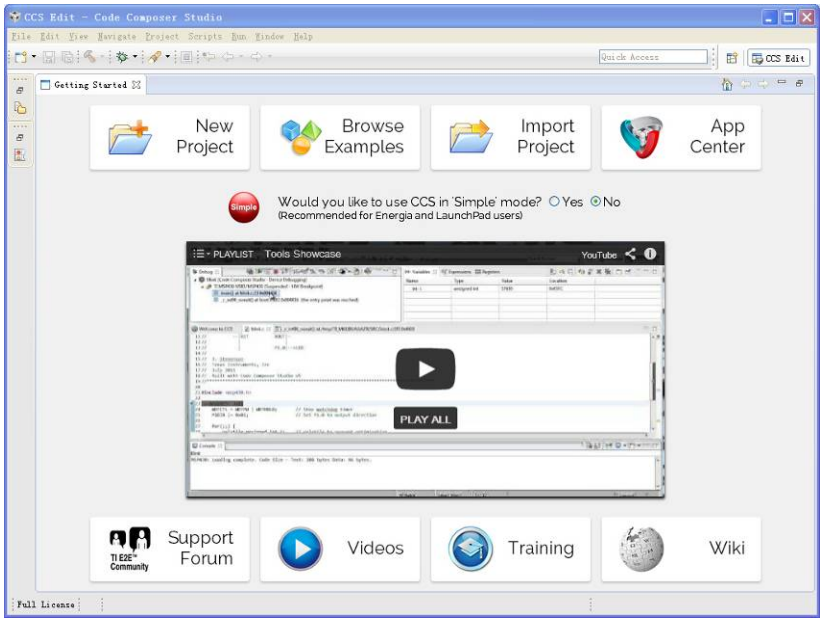


图 2.1.6　CCSv6 软件开发集成环境界面

2.2 利用 CCS 导入已有工程

（1）打开 CCS 后，点击 Import Project，或者在 Project 菜单栏中选择 Import CCS Project，打开如下界面。

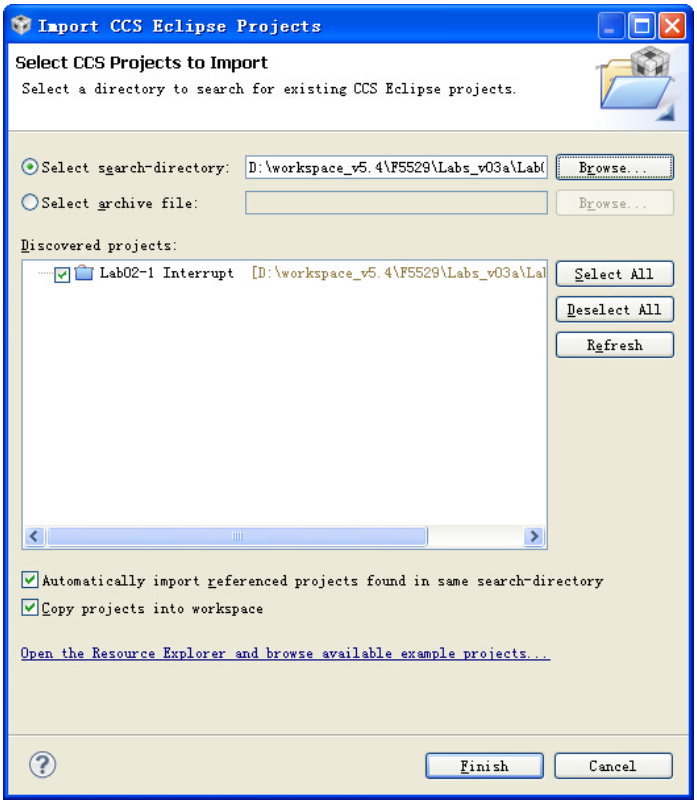


图 2.2.1　选择导入工程目录

其中第一项为需要导入的工程所在目录（源目录）；最下面的选项是问要不要把源目录中的工程复制到当前所建的 workspace（目标目录）中，我们选择“要”，然后单击 Finish，即可完成既有工程的导入。

2.3 利用 CCS 新建工程

（1）首先选择“New Project”工作区，然后将正确的器件系列，型号填入 Target 对话框。接着在在 Project name 中输入新建工程的名称，其余选择默认选项即可。

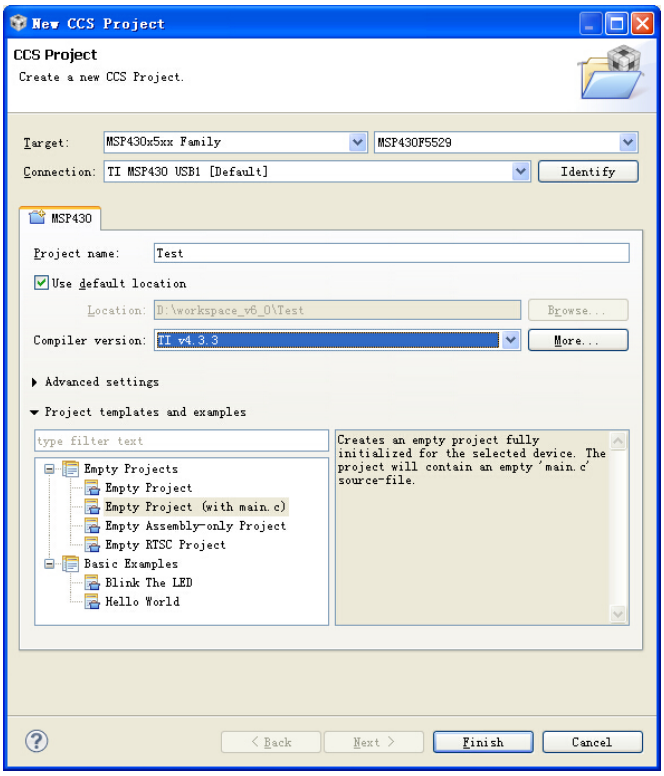


图 2.3.1　新建 CCS 工程对话框

2.4 利用 CCS 调试工程

2.4.1 启动调试器

（1）单击 CCS 工具栏上绿色的 Debug 按钮  ，就可以对工程文件进行编译、链接，生成可执行文件，然后下载到 F5529 芯片中。在编译过程中会出现下面的对话框，其中含义是问我们是否对对程序进行低功耗语法扫描，如果我们选择进行语法扫描（选择 Proceed），当语法正确但不符合低功耗规则时，编译器会给出警告信息；我们也可以选择不进行低功耗扫描（选择 Cancel）。

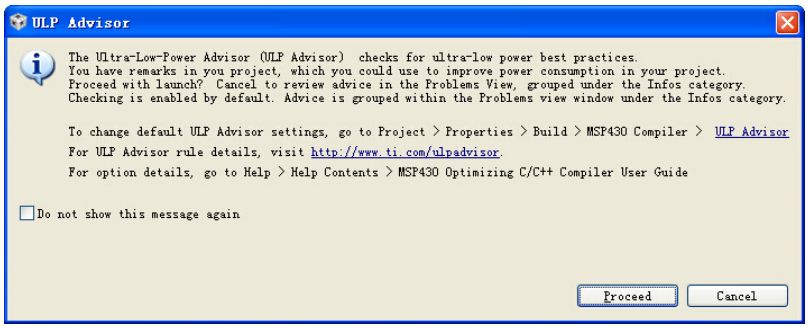


图 2.4.1　低功耗扫描选项

（2）编译下载完成后的 CCS 界面如下图所示：

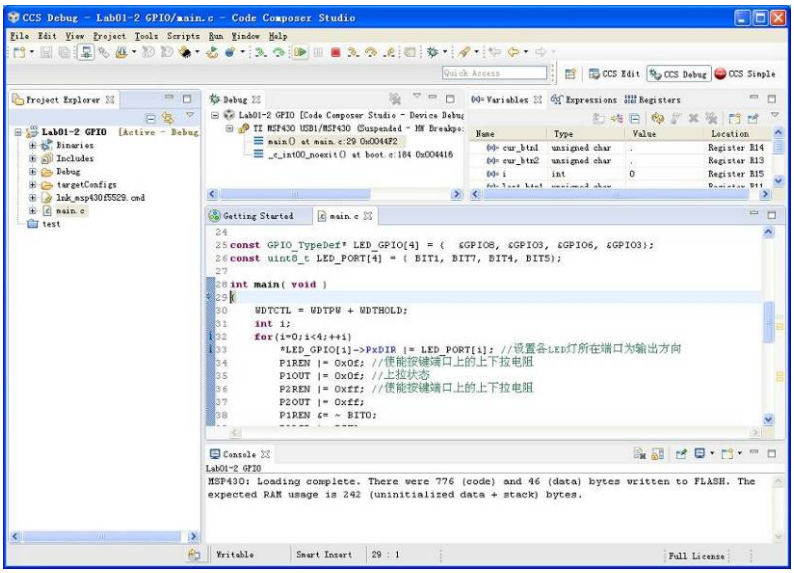


图 2.4.2　调试界面

单击运行图标  运行程序，观察显示的结果。在程序调试的过程中，可通过设置断点来调试程序：选择需要设置断点的位置，右击鼠标选择 Breakpoints→Breakpoint，断点设置成功后将显示图标  ，可以通过双击该图标来取消该断点。程序运行的过程中可以通过单步调试按钮  配合断点单步的调试程序，单击重新开始图标  定位到 main()函数，单击复位按钮  复位。可通过中止按钮  返回到编辑界面。

2.5 CCS APP Center

在 CCS6 中取消了资源管理器，代之以 CCS APP Center（CCS 工具栏 View 中第 1 项就是），界面如下：

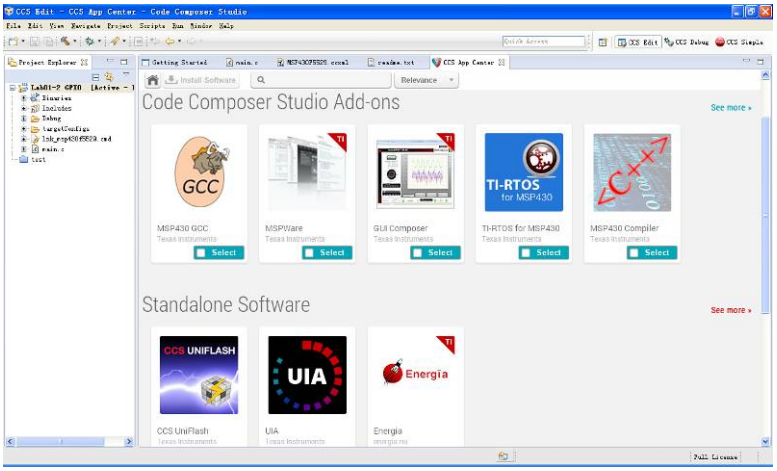


图 2.5.1　CCS APP Center

在 APP Center 中有众多应用，比如 MSP Ware（原 430Ware）、TI-RTOS（嵌入式操作系统）等等。这样资源都在服务器端（安装 CCS 时如果勾选了 APP Center 另当别论），需要连接网络，才可以下载到本地来使用。

第 3 章 MSP430F5529 片内资源及实验

MSP430F5529 口袋板套件能够完成基于 F5529 单片机的实验内容及外设模拟器件、传感器模块的实验，这一章首先介绍 F5529 片内资源及实验项目。

3.1 GPIO 模块

这一节作为学习 MSP430F5529 的入门实验，介绍了如何操作 MSP430 的 GPIO 接口，我们设计了两个简单的实验让大家了解 GPIO 的用法，这为后面更深层次的 IO 操作做了铺垫，因此必须熟练掌握。

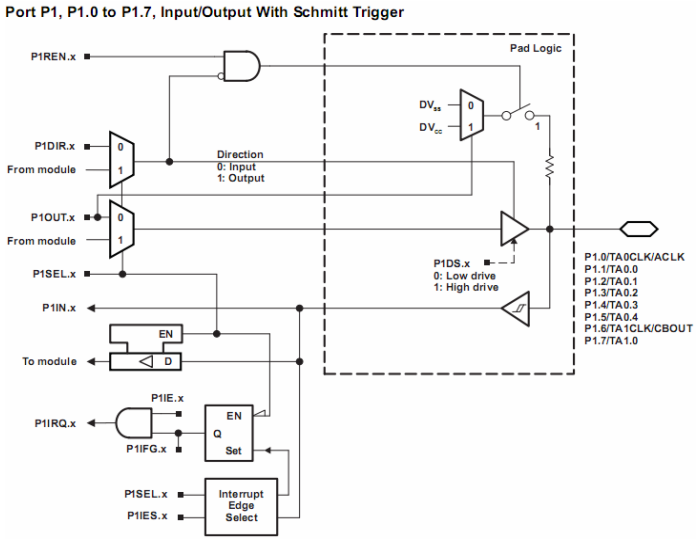
GPIO 是 MCU 中最基本、最常用的功能单元，信号、数据的输入输出都要依靠 GPIO 接口，GPIO 管脚也是 MCU 芯片上数量最多的管脚，现在 MCU 上的 GPIO 一般都具有多种复用功能，可以通过软件进行配置。

GPIO 是 MCU 与外界交互的重要途径，它具有如下的特性：

- 可以独立控制每个 GPIO 口的方向（输入/输出模式）
- 可以独立设置每个 GPIO 的输出状态（高/低电平）
- 所有 GPIO 口在复位后都有个默认方向（输入/输出）

MSP430F5529 上 GPIO 接口类型丰富，P1～P4 具有输入输出、中断和外部功能模块，这些功能可以通过他们各自 9 个控制寄存器的设置来实现。P1～P8 大多数端口包含 8 个 I/O 引脚，但一些端口可能含有较少引脚。每个 I/O 引脚单独配置输入或输出方向,每个引脚可以单独读或写。P1～P8 端口具有中断功能，每一个 P1～P8 I/O 引脚的中断可以单独启用和配置，在输入信号处于上升或下降沿时提供中断。在一些设备中还会附加具有中断能力的端口，并且包含了它们各自的中断向量。所有端口寄存器（除中断向量寄存器）是用命名的方式来操作的，例如：PAIV 不存在 P1IV 和 P2IV。通常 IO 的配置方式通过软件来实现。

下面看一下 IO 口功能框图，如图 3.1.1 所示。



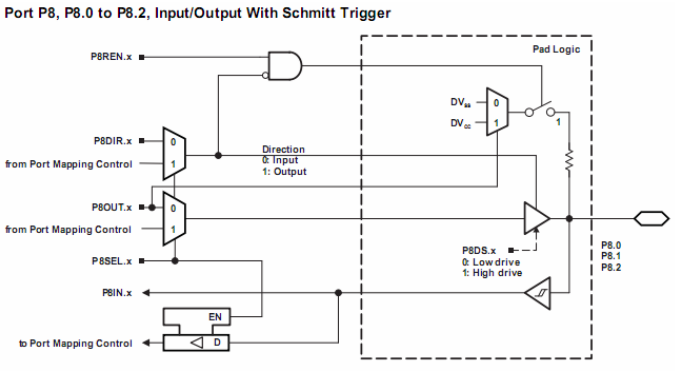


图 3.1.1 MSP430F5529 P1 与 P8 口功能框图

通过图 3.1.1 是不是感觉 IO 挺复杂的？

为什么 IO 口需要这么复杂的构造呢，我们来列举一下：

（1） 对集成电路来说，input 和 output 就是需要两套电路来实现的，并不是像一根导线那样信号能左右互通。这两套电路要有切换装置，否则输入输出会互相矛盾，于是有了 IO 方向寄存器 PxDIR。

（2） IO 口作为输入口时， 如何把输入信号 “告诉” CPU 呢？需要通过外部电路将寄存器 PxIN 置位或复位，这样 CPU 就能随时读取寄存器 PxIN 的值了。

（3） CPU 如何命令 IO 口输出某一电平呢？CPU 需要去写寄存器 PxOUT，然后会有相应的缓冲电路将 PxOUT 的信号传递到单片机的外部引脚处。

（4） 数字电路中实现高低电平有多种方式，上拉（强 0 弱 1） 、下拉（强 1 弱 0）还是图腾柱（强 1 强 0）的性质有很大差别，PxREN 寄存器用于控制内部上下拉电阻。

（5） 所有 MSP430 单片机的 P1 和 P2 口是带中断的。中断可不是“吱一声”就能有的，需要复杂的配套电路， 这就有了是否允许 IO 中断寄存器 PxIE， 中断标志位寄存器 PxIFG， 中断的边沿选择寄存器 PxIES。

（6） 对于高性能单片机，IO 口是高度复用的。何为复用呢，就是本来某功能最好是自己长出个引脚出来，现在被迫与 IO 共用引脚了。所以又需要选择开关了，PxSEL 和 PxSEL2 寄存器就是干这个的。

（7） 部分 MSP430 单片机的 IO 口带振荡器，可以在不附加任何外部器件的情况下，实现电容触摸按键的识别，最流行的电容触摸也可以有。

以上总总迹象表明， IO 口没有最复杂只有更复杂， 我们不关心用了多少硬件电路才实现了 IO 的这些功能，我们关心怎么配置寄存器去享用这些功能，根据下面实验来具体了解 IO 的使用。

3.1.1 Lab3-1-1 按键对 LED 灯的控制实验（查询方式）

3.1.1.1 实验介绍

通过按键对 IO 口 P1.2 的操作，查询实现高低电平检测，从而对与 LED 相连的 P8.1 的电平控制，实现 LED 灯的亮与灭。

3.1.1.2 实验目的

- 掌握对 IO 口的查询操作和 IO 基本操作的流程
- 对部分引脚功能有一个初步了解

3.1.1.3 实验原理

MSP430各种端口有大量的控制寄存器供用户操作，这最大限度提供了输入/输出的灵活性。下表是端口的各个寄存器的使用方式：

名称	缩写	BIT=1	BIT=0
方向寄存器	PxDIR	输出模式	输入模式
输入寄存器	PxIN	输入高电平	输入低电平
输出寄存器	PxOUT	输出高电平	输出低电平
上下拉电阻使能寄存器	PxREN	使能	禁用
功能选择寄存器	PxSEL	外设功能	IO端口
驱动强度寄存器	PxDS	高强度	低强度
中断使能寄存器	PxIE	允许中断	禁止中断
中断触发沿寄存器	PxIES	下降沿置位	上升沿置位
中断标志寄存器	PxIFG	有中断请求	无中断请求

表3.1.1 端口寄存器说明

通过表3.1.1可知读写 IO 口前，必须先设置 PxDIR，PxDIR 高电平代表 IO 是输出，低电平代表 IO 口是输入。CPU 读IO，实际上是读 PxIN 寄存器。CPU 写 IO，实际是写 PxOUT 寄存器。

提示：作为高阻输入 IO 时，务必关掉内部上下拉电阻开关 PxREN，否则输入就不是高阻态了。

数字电路的输出有高电平 1、 低电平 0 和高阻态 z。 决定高低电平的是 VCC 和 GND， 在初学者映像中，VCC 算电源，GND 不算电源，这是不准确的。GND 也是电源，确切的说 VCC 和 GND 都是电压源，所以GND 经常用 VSS 来代替。根据输出电平的“强弱”分类，可将 IO 输出分为推挽/推拉输出、上拉电阻输出、下拉电阻输出 3 种。

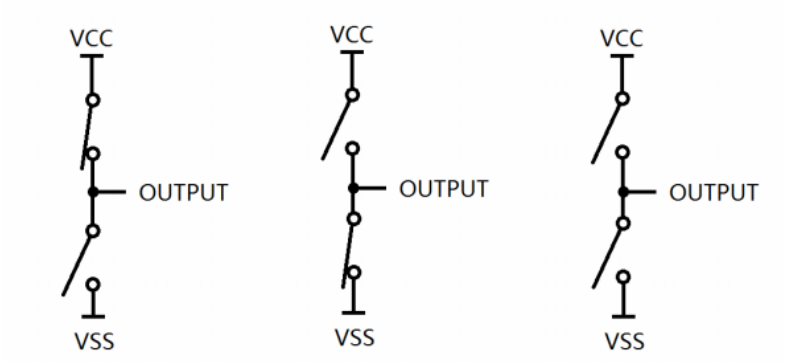
下面介绍一下三种输出方式：

推挽/推拉输出：图3.1.2是推挽/推拉电路简图。图中开关是受控的电子开关，由两个开关恩别连接VCC和VSS电路被称为推挽/推拉输出，该电路输出强1或者强0。

如图 3.1.2(a)所示，无论 OUTPUT 被接在什么电路上，OUTPUT 一定是高电平 1，这就是所谓的输出强 1。

如图 3.1.2 （b） 所示， 无论 OUPUT 被接在什么电路上， OUTPUT 一定是低电平 0， 这就是所谓输出强 0。

如图 3.1.2（c）所示，输出为高阻 Z，这代表不会对其他电路造成影响。当作为输入口时， OUTPUT 应置为高阻态。



（a）强 1 输出　　（b）强 0 输出　　（c）高阻输出

图 3.1.2　按键模块原理图

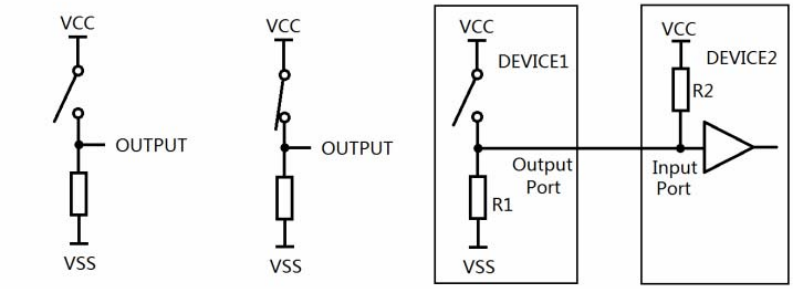
看到上图可能有些疑问， OUTPUT 为强 1 输出接在外部电路的 GND 上会怎样？会短路。 那么到底还是不是输出 1 呢。对于理想电压源 VCC，接在哪都得是 VCC，同样对于理想的地，谁接在上面都是 0V。当程序使得器件 1 输出强 0，器件 2 输出强 1，短路便发生了，是否会永久损坏硬件视短路持续时间的长短。

下拉电阻输出：图 3.1.3 所示为下拉电阻输出，该电路为强 1 弱 0 电路。

图 3.1.3 (a)输出为弱 0，为什么叫弱 0 呢？因为实际 OUTPUT 输出电平是不是 0 还取决于接什么样的负载。

如图 3.1.3 (c)所示， 器件 1 “打算”输出 0 给器件2，但是器件 2 不是高阻输入结构（比如 TTL 电平器件），那么器件 2 是否能正确识别输入为 0 电平，取决于 R1 和 R2 的比值，以及器件 2 的 1/0 识别门限值。下拉电阻 R1 越小，电平逻辑错误就越不易发生，但是这是以功耗为代价的。

图 3.1.3(b)的输出为强 1，无论 OUTPUT 接什么负载，均输出高电平 1。这就是强 1 弱 0 的由来。与推挽/推拉输出不同，两个下拉电阻输出的 IO 口相连，永远不会发生短路（前提是两 IO 的 VCC 必须等电位）。



（a）弱 0 输出　　（b）强 1 输出　　（c）下拉电阻对弱 0 输出的影响

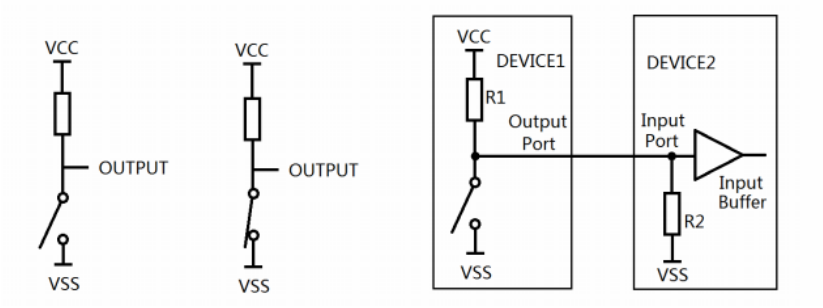
图 3.1.3　下拉电阻输出

上拉电阻输出：图 3.1.4 所示为上拉电阻输出，该电路为强 0 弱 1 电路。

图 3.1.4(a)输出为弱 1，为什么叫弱 1 呢？如果 OUTPUT 接的是高阻负载，输出肯定是 1，其他负载情况则需根据负载和上拉电阻的分压关系来计算。与下拉电阻输出类似。

图 3.1.4(c)所示，同样要根据负载情况合理设定上拉电阻的取值。

图 3.1.4(b)的输出为强 0，无论 OUTPUT 接什么负载，均输出低电平 0。这就是强 0 弱 1 的由来。同样两个上拉电阻输出的 IO 口相连，永远不会发生短路（前提是两 IO 的 VSS 必须等电位） 。



（a）弱1输出　　（b）强0输出　　（c）上拉电阻对弱1输出的影响

图3.1.4　上拉电阻输出

上拉电阻输出的应用范围非常广泛：

（1） 上拉电阻输出可用于两种供电电压器件之间的电平匹配。如图 3.1.5 所示，器件 1 供电电压为 VCC1， 器件 2 的供电电压为 VCC2。通过外置上拉电阻到 VCC2，器件 1 可实现与器件 2 的输出电平匹配。这种输出方式的芯片，说明书中会称为集电极开路输出（OC 输出）或漏极开路输出（OD 输出） ，两者区别是开关使用的是三极管还是场效应管。 凡是 OC 或 OD 输出的数字器件， 一定要外接上拉电阻到 VCC，否则器件将无法得到高电平。

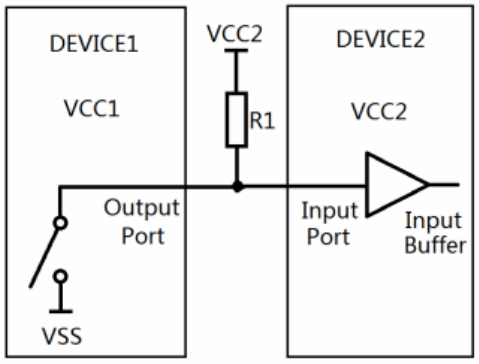


图3.1.5　上拉电阻输出用于电平匹配

（2） 上拉电阻输出还广泛用于实现“线与”逻辑。如图 3.1.6 所示，所有器件输出和输入同时进行（推挽/推拉无法做到这一点），任何一个器件都可以将公共总线电平拉低，但只有所有器件输出都为“高” ，公共总线的电平才能为高，这就是“线与”逻辑。 “线与”逻辑更独特之处在于，所有器件都可以通过输入缓冲一直“监视”总线电平的高低，这样就能知道总线电平是否与“自己期望电平”一致，从而判断是否有别的器件在与自己争夺总线。 “线与”逻辑广泛用于多机总线通信中，详见串行通信章节。

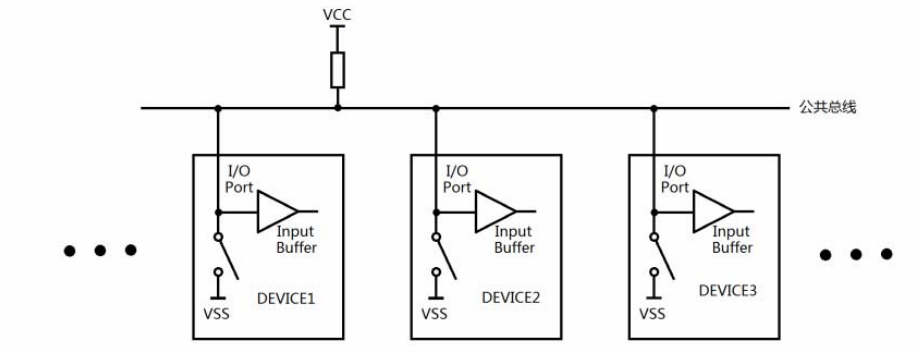


图3.1.6　由上拉电阻输出构成线与逻辑

下图为主板的S1按键与MSP430F5529 P1.2口连接：

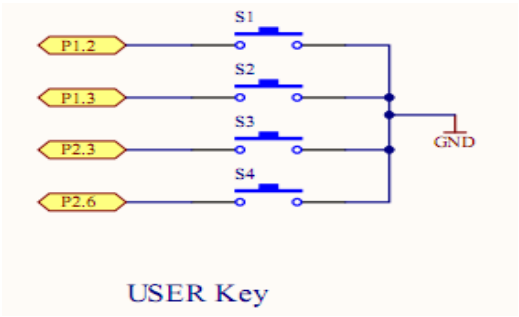


图 3.1.7　按键模块原理图

主板上方LED1指示灯与MSP430F5529 P8.1连接：

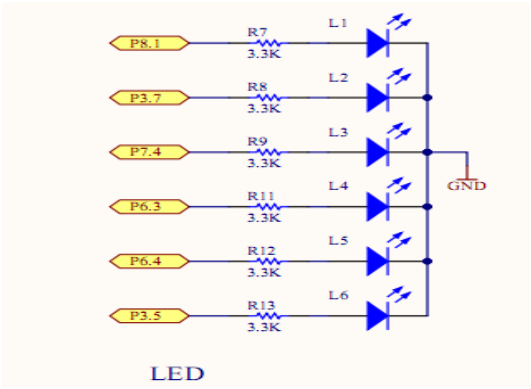


图3.1.8　LED指示灯模块原理图

PxIN 和 PxOUT 分别是输入数据和输出数据寄存器，PxDIR 为方向寄存器，PxREN 为使能寄存器：

```
#define PxIN          (PBIN_H)      /* Port x Input */
#define PxOUT         (PBOUT_H)     /* Port x Output */
#define PxDIR         (PBDIR_H)     /* Port x Direction */
#define PxREN         (PBREN_H)     /* Port x Resistor Enable */
```

3.1.1.4 程序分析

- 编程思路：

关闭看门狗定时器后，对 P8.1 的输出方式、输出模式和使能方式初始化，然后进行查询判断，最后对 P8.1 的电平高低分别作处理来控制 LED 灯。

- 程序流程图：

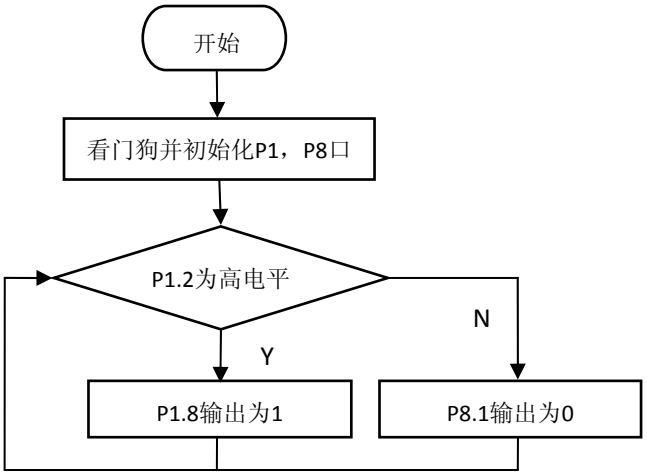


图3.1.9 程序流程图

- 关键代码分析

P1 P4 口初始化函数

```
#include <MSP430F5529.h>
int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;          //关闭看门狗定时器
   P8DIR |= BIT1;                    //设置8.1口为输出模式
   P1REN |= BIT2;                    //使能P1.2上下拉电阻功能
   P1OUT |= BIT2;                    //置P1.2为上拉电阻方式
```

循环判断部分

```
  while (1)
   {
      if (P1IN & BIT2)               //判断是否按下键，S1按键按下P1.2=0，抬起P1.2=1
       P8OUT |= BIT1;                //S2按键抬起，P8.1输出高（LED1点亮）
      else
        P8OUT &= ~BIT1;
   }
}
```

3.1.1.5 实验步骤与现象

- **实验步骤**

1. 将 PC 和板载仿真器通过 USB 线相连；
2. 打开CCS 集成开发工具，选择Project->Import Existing CCS Eclipse Project,导入所建文件夹中相应的工程；
3. 选择  对该工程进行编译链接，生成.out 文件。然后选择  ，将程序序下载到实验板中。程序下载完毕之后，可以选择  全速运行程序，也可以选择  单步调试程序，选择 F3 查看具体函数。也可以程序下载之后，按下  终止调试，软件界面恢复到原编辑程序的画面。再按下实验板的复位键，运行程序。（调试方式下的全速运行和直接上电运行程序在时序有少许差别，建议上电运行程序）。

- **实验现象**

  按下口袋板上机械按键 S1 后，LED1 熄灭，松开后恢复点亮。

3.1.1.6 实验思考

1. MSP430 的 IO 口都具备哪些功能？
2. 与 IO 口相关的寄存器有哪些？
3. 为何要关闭看门狗定时器？

3.1.2 Lab3-1-2 多个按键对 LED 灯的控制实验（查询方式）

3.1.2.1 实验介绍

该实验结合了各个按键的触发方式，将各个按键对应的 IO 口添加到一个结构体中操作 LED 灯，这使得代码简单易读，比上一个实验的简单查询方式更加规范，操作也更加方便。

3.1.2.2 实验目的

- 深入了解 IO 口的原理和操作方式
- 学习结构体定义各个指针变量的方法
- 掌握循环调用各个结构体内变量的方法

3.1.2.3 实验原理

按键与 LED 模块的电路连接在上一个实验中做过介绍，这里不再赘述，请阅读 3.1.1.3 章节。

由于 LED 灯不在同一个端口上，为了可以直接使用 int 进行索引操作，程序中将 IO 口的相关寄存器的地址声明为结构体并保存在数组中。即首先声明了保存寄存器地址的结构：

```
typedef     struct
{ const     volatile    uint8_t* PxIN;
volatile    uint8_t* PxOUT;
volatile    uint8_t* PxDIR;
volatile    uint8_t* PxREN;
volatile    uint8_t* PxSEL; } GPIO_TypeDef;
//之后声明所需操作的 IO 端口的结构体 GPIO3、GPIO6、GPIO7、GPIO8，如：
const  GPIO_TypeDef GPIO3 =    { &P3IN, &P3OUT, &P3DIR, &P3REN, &P3SEL };
const  GPIO_TypeDef GPIO6 =    { &P6IN, &P6OUT, &P6DIR, &P6REN, &P6SEL };
const  GPIO_TypeDef GPIO7 =    { &P7IN, &P7OUT, &P7DIR, &P7REN, &P7SEL };
const  GPIO_TypeDef GPIO7 =    { &P8IN, &P8OUT, &P8DIR, &P8REN, &P8SEL };
```

3.1.2.4 程序分析

- 编程思路

设置各个引脚变量并且初始化，然后循环检查按键是否按下，如果按下，就把 IO 电平取反。

- 程序流程图

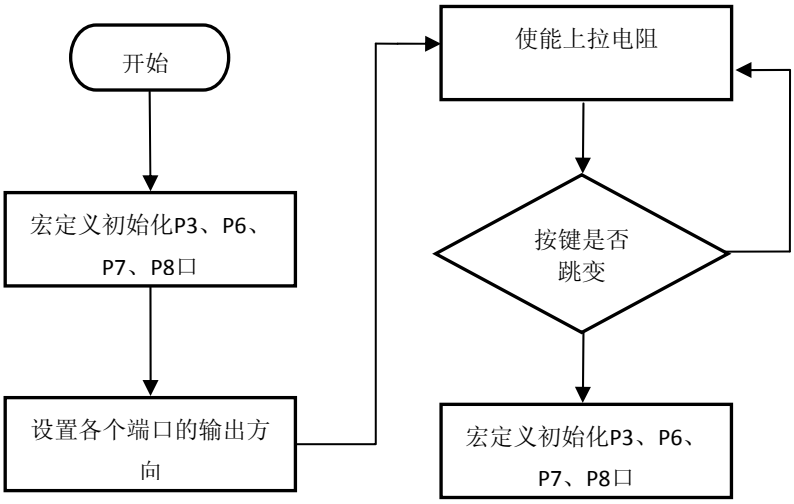


图3.1.5　程序流程图

- 关键代码分析

宏定义和参数定义

```
#include <msp430.h>

#include <stdint.h>
typedef struct                          //以指针形式定义P1口的各个寄存器
{
   const volatile uint8_t* PxIN;        //定义一个不会被编译的无符号字符型变量
   volatile unit8_t* PxOUT;
   volatile unit8_t * PxDIR;
   volatile unit8_t * PxREN;
   volatile unit8_t * PxSEL;
} GPIO_TypeDef;

const GPIO_TypeDef GPIO3 =
{&P3IN, &P3OUT, &P3DIR, &P3REN, &P3SEL};

const GPIO_TypeDef GPIO6 =
{&P6IN, &P6OUT, &P6DIR, &P6REN, &P6SEL};

const GPIO_TypeDef GPIO6 =
{&P7IN, &P7OUT, &P7DIR, &P7REN, &P7SEL};

const GPIO_TypeDef GPIO6 =
{&P8IN, &P8OUT, &P8DIR, &P8REN, &P8SEL};

const GPIO_TypeDef* LED_GPIO[4] = { &GPIO8, &GPIO3, &GPIO6, &GPIO3};
const unit8_t LED_PORT[4] = { BIT1, BIT7, BIT4, BIT5};
```

主函数

```
int main( void )
{
      WDTCTL = WDTPW + WDTHOLD;
      int i;
```

```
        for(i=0;i<3;++i)
            *LED_GPIO[i]->PxDIR |= LED_PORT[i]; //设置各LED灯所在端口为输出方向
            P1REN |= 0x1F; //使能按键端口上的上下拉电阻
            P1OUT |= 0x1F; //上拉状态
            P2REN |= 0xFF;
            P2OUT |= 0xFF;
            P1REN &= ~BIT0;
            unit8_t last_btn1 = 0x1F, last_btn2 = 0xFF, cur_btn1, cur_btn2, temp1, temp2;
    while(1)
        {
            cur_btn1 = P1IN & 0x1F;
            temp = (cur_btn ^ last_btn) & last_btn; //找出刚向下跳变的按键
            last_btn1 = cur_btn1;
            last_btn2 = cur_btn2;
            int i;
            for(i=2;i<4;i++)
                if(temp & (1 << i))
                    *LED_GPIO[i-2]->PxOUT ^= LED_PORT[i-2]; //翻转对应的LED
            cur_btn2 = P2IN & 0xff;
            temp1 = (cur_btn2 ^ last_btn2) & last_btn2;
            last_btn2 = cur_btn2;
                if(temp1 & (1 << 6))    //P2.6按下
                    *LED_GPIO[3]->PxOUT ^= LED_PORT[3]; //翻转对应的LED
                if(temp1 & (1 <<3))    //P2.3按下
                    *LED_GPIO[2]->PxOUT ^= LED_PORT[2]; //翻转对应的LED

    }
}
```

3.1.2.5 实验步骤与实验现象

- **实验步骤**

1. 将 PC 和板载仿真器通过 USB 线相连；
2. 打开CCS 集成开发工具，选择Project->Import Existing CCS Eclipse Project,导入所建文件夹中相应的工程；
3. 选择 对该工程进行编译链接，生成.out 文件。然后选择 ，将程序序下载到实验板中。程序下载完毕之后，可以选择 全速运行程序，也可以选择 单步调试程序，选择 F3 查看具体函数。也可以程序下载之后，按下 终止调试，软件界面恢复到原编辑程序的画面。再按下实验板的复位键，运行程序。（调试方式下的全速运行和直接上电运行程序在时序有少许差别，建议上电运行程序）。

4. 依次按下各个键 S1、S2、S3、S4（上层板）观察 LED 灯的变化。

- **实验现象**

依次按下口袋板机械按键 S1、S2、S3、S4 后，按键上方的 L1、L2、L5、L6 依次点亮，再一次按下时，对应的 LED 指示灯熄灭。

3.1.2.6 实验思考

如何处理按键产生的毛刺、抖动现象？

3.2 中断与低功耗工作模式

中断与低功耗工作模式是 MCU 两个重要功能，二者间存在在紧密联系，比如当我们从一种功耗模式进入到另外一种功耗模式往往都是通过中断实现的，因此我们将这两个功能放在一起介绍。

3.2.1 中断

中断装置和中断处理程序统称为中断系统，中断系统是单片机中重要的组成部分。中断是 CPU 对系统发生的某个事件作出的反应，暂停正在运行中的程序，转去执行处理相应的中断事件，完毕后返回被中断的程序继续运行。中断系统的使用大大提高了 CPU 的效率，中断的实现由软件和硬件综合完成。

MSP430F5529 的中断装置比较多，其主要的有 IO 口中断、定时器中断、串口中断、外部中断、ADC 转换中断、看门狗中断、捕获比较中断等，这些装置都能引发中断事件。中断分为不可屏蔽（nonmaskable）中断与可屏蔽（maskable）中断两大类，各种中断的优先级也是不同的，MSP430F5529 的一些中断源、中断标志及优先级等如下图所示。

Interrupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
Reset: power up, external reset watchdog, flash password	... WDTIFG KEYV	... Reset	... 0FFFEh	... Highest
System NMI: PMM		(Non)maskable	0FFFCh	...
User NMI: NMI, oscillator fault, flash memory access violation	... NMIIFG OFIFG ACCVIFG	... (Non)maskable (Non)maskable (Non)maskable	... 0FFFAh	... ...
Device specific			0FFF8h	...
...			...	...
Watchdog timer	WDTIFG	Maskable	...	...
...			...	...
Device specific			...	...
Reserved		Maskable	...	Lowest

图 3.2.1　中断资源表

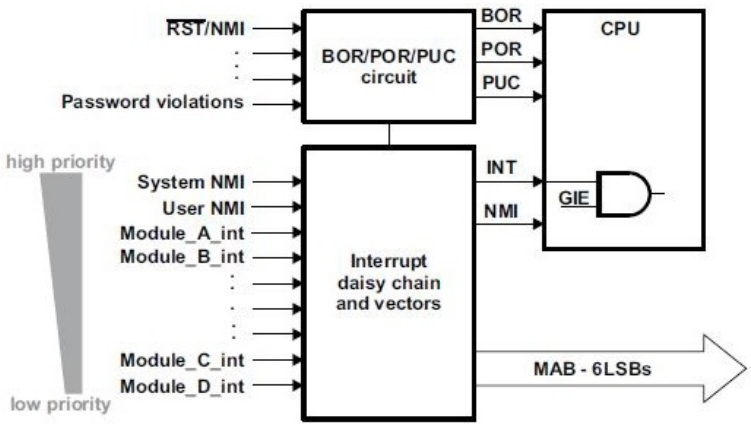


图3.2.2　中断资源示意图

IO 口中断：即外部中断，要使用外部中断要遵循以下原则：

（1） 通过 PxDIR 将 IO 方向设为输入。

（2） 通过写 PxIES，决定中断的边沿是上升沿、下降沿或两种情况均中断。

（3） 如果是机械按键输入，可以通过 PxREN 启用内部上下拉电阻，根据按键的接法，设定 PxOUT（决定最终是上拉电阻还是下拉电阻） 。

（4） 通过 PxIE 开启 IO 中断，通过_EINT()开启总中断。

（5） 在中断子函数中，通过 if 语句查询具体中断的 IO 口，如果是机械按键输入，还需有消抖代码。

（6） 根据具体 IO 的输入，编写事件处理函数。

（7） 清除中断标志位：PxIFG = 0。

使用按键时有个现象要特别注意，就是按键按下时会有抖动，那么就会对正常按键按下产生干扰，这时我们就需要消抖处理。

如图 3.2.3 所示，机械按键按下和弹起时，会有毛刺干扰。在一次按键过程中，会有若干次下降沿，只有 1 号是真正的按键事件。如何避免其他几次下降沿“中断”的影响呢？

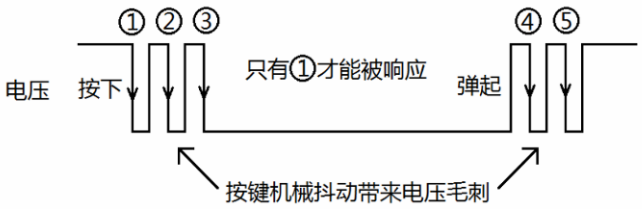


图3.2.3 机械按键抖动产生的电压毛刺

我们可以采用以下几种方式消除按键抖动：

方法 1： 理论上 Push_Key 的值应该等于 P1IN。 P1IN 和 P1IFG 的区别在于 P1IN 可能有双键被按下的可能性，而中断标志位 P1IFG 只记录最早按下的键。但有一个“干扰”因素，设置为输出口的 IO 其对应的 P1IFG 位也可能是 1，所以用了一个按位与操作代码 P1IFG&(~P1DIR)，将输出 IO 排除在外。

注：P1IN 为 IO 输入寄存器；P1OUT 为 IO 输出寄存器；P1IFG 为 IO 中断标志位寄存器；P1DIR 为 IO 方向寄存器，高电平代表 IO 设为输出，低电平代表 IO 口设为输入。

方法 2：足够长的延时，实际上就是阻塞 CPU，下降沿 1 检测到后，由于延时，下降沿 2 、 3 将不被检测。下降沿 4 检测到后，下降沿 5 也会因为延时不会被检测。

方法 3：延时是无法消灭下降沿 4 的。所以，另加了一条判据，当检测到下降沿 4 ，进中断，延时一段时间再看按键电平时，按键电平将是高，这就说明它是按键弹起阶段的第一个毛刺，可以被排除掉。另一个要说明的是，按键按下后，按键 IO 的电平是 0，而中断标志 IFG 是 1，正常按下时，两个值应该不等。所以判据代码是(P1IN&Push_Key)==0。

方法 4：在用 switch 判断具体中断 IO 时，没有用 P1IN 的值，因为 P1IN 有可能不止 1 个“1”（多个按键被按下）。而应该用 PIIFG 滤除输出 IO 影响后的 Push_Key，Push_Key 有且只有 1 个“1”。

3.2.2 低功耗工作模式

低功耗即是将能量消耗尽可能的降低，TI 的 MSP430 系列单片机最突出的特点之一就是超低功耗，它采用 RISC 内核结构，非常适用于应用电池的场合或手持设备，在超低功耗

方面 MSP430F5529 能够实现在 1.8V—3.6V 电压和 1MHz 的时钟条件下运行，耗电电流（0.1—400uA 之间），其中 MSP430F5529 提供功耗模式，且不同功耗模式间可以方便切换是实现系统超低功能的重要原因之一。

实现低功耗方面主要有三个方法：一是降低工作电压，MSP430 位控制器在 1.8V 低电压下仍可以工作；二是把暂时不用的模块功能给关闭掉实现节能，MSP430 把一个芯片分成了 N 个不同模块的部分，不用的部分功能模块关闭掉使电流近似为零；第三是根据实际情况适当降低工作频率，MSP430 可以有三个时钟源并产生更多的内部工作频率，内部各个模块根据实际需要工作在不同的频率，不用的时钟还可以关闭。

开发板上的主芯片 MSP430F5529 的工作模式有 6 种，包括活动模式（AM）和 5 种低功耗模式（LPM0—LPM5），图 3.2.5 表示了各个低功耗模式之间的关系。活动模式和低功耗模式的实现是由状态寄存器（SR）中的几个标志位（SCG1、SCG0，OSCOOFF、CPUOFF）共同来控制，其组合控制如下图 3.2.4 所示。在所有低功耗模式下 CPU 的 MCLK 都是禁止的，要在低功耗模式下转到活动模式必须由中断来唤醒，所以一般情况下进入低功耗模式前，应保持开启中断状态即确保 GIE 为置位状态。

SCG1 (1)	SCG0	OSCOFF (1)	CPUOFF (1)	Mode	CPU and Clocks Status (2)
0	0	0	0	Active	CPU, MCLK are active. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK, MCLK, or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0). FLL is enabled if DCO is enabled.
0	0	0	1	LPM0	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0). FLL is enabled if DCO is enabled.
0	1	0	1	LPM1	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0). FLL is disabled.
1	0	0	1	LPM2	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled. DCO is enabled if sources ACLK. FLL is disabled.
1	1	0	1	LPM3	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled. DCO is enabled if sources ACLK. FLL is disabled.
1	1	1	1	LPM4	CPU and all clocks are disabled.
1	1	1	1	LPM3.5 (3)	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, RTC operation is possible when configured properly. See the *RTC* module for further details.
1	1	1	1	LPM4.5 (3)	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, all clock sources are disabled; that is, no RTC operation is possible.

(1) This bit is automatically reset when exiting low power modes. Please refer to Section 1.4.1 for details.
(2) The low-power modes and, hence, the system clocks can be affected by the clock request system. See the UCS chapter for details.
(3) LPM3.5 and LPM4.5 modes are not available on all devices. See the device-specific data sheet for availability.

图3.2.4 低功耗模式控制寄存器（SR）组合控制操作模式

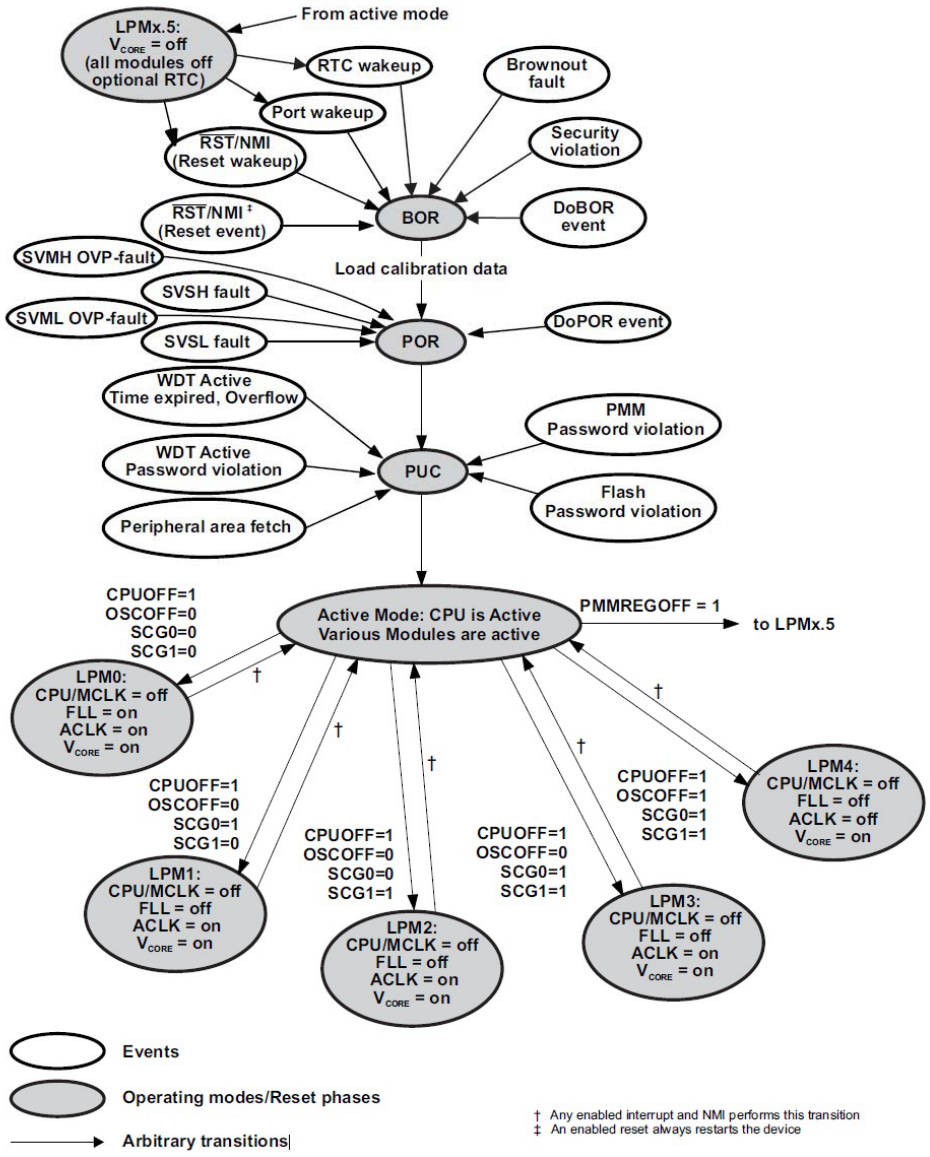


图3.2.5 不同功耗工作模式关系图

3.2.3 Lab3-2-1 按键对 LED 灯的控制实验（中断方式）

3.2.3.1 实验介绍

本次实验应用了 MSP430F5529 中断事件中的 IO 口中断，通过使用开发板上的 S1 按键来触发 P1.2 口的中断事件，在中断事件处理函数中改变 LED1 灯的状态。

3.2.3.2 实验目的

- 了解 MSP430F5529 中断系统
- 掌握 IO 口中断的使用和编程

3.2.3.3 实验原理

本实验应用的是 MSP430F5529 的 IO 口中断，主板 S1 上按键连接到 P1.2 口，当 S1 按键被按下时，P1.2 口电平由高变成低触发一个中断事件，然后在 P1 口的中断函数中填写代码改变 LED1 灯的状态。按键与 LED 灯部分的电路原理图请看 Lab1-1 中实验原理章节，这里不再赘述。

3.2.3.4 程序分析

- **编程思路**

整个编程过程比较简单，只需要配置好相应的 IO 口就可以了，需要设置 P8.1 口为输出状态以控制 LED1 灯状态，还需要使能 P1.2 口的上拉电阻，选择 P1.2 口中断沿，使能中断，清中断标志位，然后进入等待状态，最后再写一个 P1 口的中断服务函数，在函数中改变 LED1 灯的点亮状态。当有按键被按下既产生了中断事件，程序转向中断服务函数并改变 LED1 灯的状态。用到的寄存器有 P8DIR、P8REN、P8OUT、P1OUT、P1IES、P1FG、P1IE，具体功能可查看文档。写中断函数不同于写普通函数，中断函数不需要和普通函数那样声明，在 MSP430 中用拓展关键字“__interrupt”来表明，具体用法参考下面的代码。

- **程序流程图**

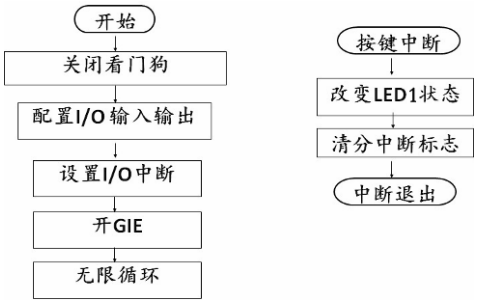


图 3.2.5　程序流程图

- 关键代码分析

main.c

```
int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                    // 关闭看门狗
  P8DIR |= BIT1;                               // 设置 P8.1 口方向为输出
  P8OUT &= ~BIT1;
  P1REN |= BIT2;                               // 使能 P1.2 上拉电阻
  P1OUT |= BIT2;                               // P1.2 口置高电平
  P1IES &= ~BIT2;                              // 中断沿设置（下降沿触发）
  P1IFG &= ~BIT2;                              // 清 P1.2 中断标志
  P1IE |= BIT2;                                // 使能 P1.2 口中断

  __bis_SR_register(LPM4_bits + GIE);          // 进入低功耗模式 4 开中断
  __no_operation();                            // 空操作
}

// P1 中断函数
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
   P8OUT ^= BIT1;                              //改变 LED1 灯状态
   P1IFG &= ~BIT2;                             //清 P1.2 中断标志位
}
```

3.2.3.5 实验步骤与现象

- 实验步骤

1. 打开开发软件 CCS5，创建一个 MSP430F5529 的空工程；
2. 完成以上代码的编写；
3. 将代码编译下载到开发板中并全速运行，按下开发饭上的 S1 按键，观察 LED1 灯亮灭状态；

- 实验现象

按下口袋板左下角的 S1 机械按键后，进入中断，按键上方的 L1 指示灯点亮；再一次按下 S1 时，L1 指示灯熄灭。

3.2.3.6 实验思考

1. 在实验现象中可能会出现按一次按键却会出现 L1 指示灯闪一次或者是多次的情况，这是为什么？如何解决？
2. 主函数中，没有调用中断子程序，中断子程序为什么能被执行？何时被执行?

3.2.4 Lab3-2-2 低功耗工作模式实验

3.2.4.1 实验介绍

TI 的 MSP430 是特别强调低功耗的单片机系列，具有多种低功耗模式，尤其是适用于采用电池供电的长时间场合，实验演示了 MSP430F5529 低功耗的编程方法，用过软件设置使 MSP430F5529 进入低功耗模式 1。

3.2.4.2 实验目的

1. 熟悉 MSP430F5529 实验板
2. 学习使用 TI 官方提供的函数库
3. 了解 MSP430F5529 的几种低功耗模式，掌握 MSP430F5529 低功耗设置的编程方法

3.2.4.3 实验原理

请参考 3.2.2 低功耗工作模式

3.2.4.4 程序分析

- **编程思路**

要使 MSP430F5529 进入低功耗模式主要就是对其控制寄存器（SR）的配置了，可直接调用库函数“__bis_SR_register(LPMx_bits)”使系统进入相应的低功耗模式（x 为模式编号），但是在进入低功耗之前要进行一些其他的配置，如关闭振荡器、配置驱动电平模式、配置未使用的 IO 口、设置禁止访问 USB 配置寄存器、禁用高低压侧 SVS 等，最后调用库函数进入低功耗模式 1。需要进行设置的寄存器包括 P5SEL、UCSCTL6、UCSCTL7、SFRIFG1、PxOUT、PxDIR、USBKEYPID、USBPWRCTL、PMMCTL0_H、SVSMHCTL、SVSMLCTL，具体寄存器功能可查阅文档。

- **程序流程图**

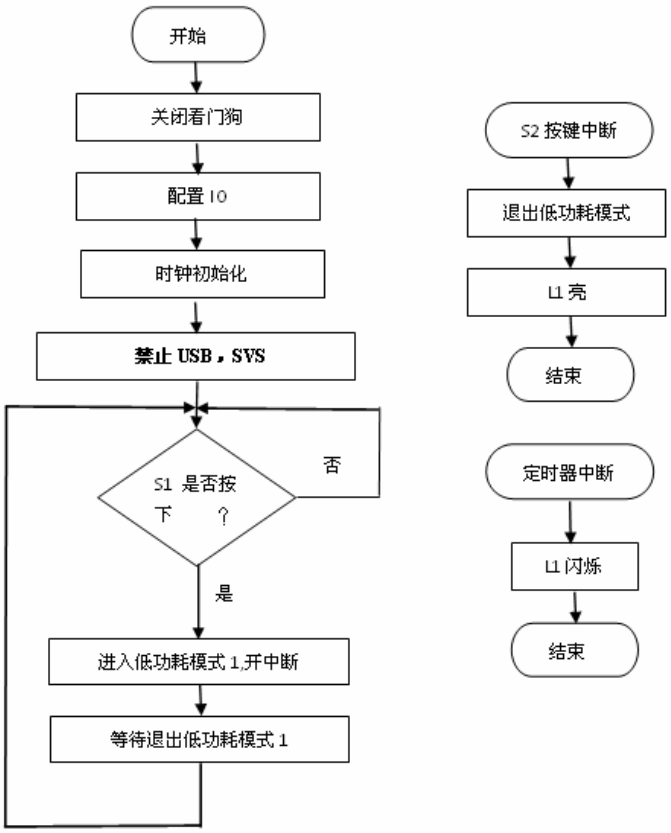


图3.2.6 程序流程图

- 关键代码分析

main.c

```
//*****************************************************************************
//     MSP430f5529   - 进入低功耗模式1实验例程
//     配置 ACLK = LFXT1 = 32KHZ，MCLK = SMCLK = 默认 DCO
//     禁用 VUSB LDO 和 SLDO ，    禁用SVS，禁用REF0;
//     main.c
//*****************************************************************************
#include <msp430.h>
unsigned int i;              //定义定时变量
unsigned char flag;       //设置按键标志位
int main(void)
{
   WDTCTL = WDTPW + WDTHOLD;                    // 关闭看门狗

   // XT1使能
   P5SEL |= BIT4+BIT5;                          //P5.4和P5.5外设功能，选择XT1
   UCSCTL6 &= ~(XT1OFF);                        // 配置寄存器打开XT1振荡器
   UCSCTL6 |= XCAP_3;                           //内部电容选择
   TA0CTL |= MC_1 + TASSEL_2 + TACLR;           //时钟为SMCLK,比较模式，开始时清零计数
器
```

```
    TA0CCTL0 = CCIE;                                  //比较器中断使能
    TA0CCR0   = 100;                                 //比较值设为50000，相当于50ms的时间间隔

    // 循环直到XT1和DOC振荡器稳定
    do
    {
      UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
                                                     // 清除XT1、XT2、DOC故障标志位
      SFRIFG1 &= ~OFIFG;                             // 清除OSC故障标志位
    }
    while (SFRIFG1&OFIFG);                           // 等待振荡器稳定
    UCSCTL6 &= ~(XT1DRIVE_3);                        // 振荡器已稳定，减少驱动器
    IOInit();                                       //IO口初始化
    USB_NOPOW();
    while(1)
        {
            if((P1IN & BIT2) == 0)                   //检测S1按下
                {
                flag = 1;                           //按键标志
                __bis_SR_register(LPM1_bits+ GIE);          // 进入低功耗模式1,开中断
                }
        }
    return 0;
}

                                    // P1中断函数
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
          flag = 0;                                  //按键标志
          LPM1_EXIT;                                 //退出低功耗模式
          P8OUT |= BIT1;                             //L1亮
          P1IFG &= ~BIT3;                            //清P1.3中断标志位
          __disable_interrupt();

}
                                    /*定时器中断函数*/
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
       i++;
       if(flag == 1)                                //判断按键
       {
            if(i==1000)
```

```
            {
                LPM1_EXIT;                                  //退出低功耗
                P8OUT ^= BIT1;                                  //形成闪灯效果
                __bis_SR_register(LPM1_bits+ GIE);          // 进入低功耗模式1,开中断
            }
            if(i==2000)
            {
                LPM1_EXIT;                                      //退出低功耗
                i = 0;                                      //i清零
                P8OUT ^= BIT1;                                  //形成闪灯效果
                __bis_SR_register(LPM1_bits+ GIE);          // 进入低功耗模式1,开中断
            }
        }
        else
            if(i > 2000)
                i=0;                                        //i清零
}
```

3.2.4.5 实验步骤与现象

- **实验步骤**

1. 打开开发软件 CCS5，创建一个 MSP430F5529 的空工程
2. 完成代码的编写
3. 将开发板与计算机连接，把程序编译并下载到开发板中，运行程序

- **实验现象**

上电后 L1 亮，S1 按下，L1 闪烁，表示系统进入低功耗模式 1；S2 按下，L1 常亮，表示系统退出低功耗模式 1；

3.2.4.6 实验思考

低功耗模式 LPM1 的 ACLK、SMCLK 分别处于什么状态？MSP430F5529 微控制器的 ACLK、SMCLK 可以通过相应的引脚 P1.0、P2.2 输出，利用示波器探测观察，方法是将相应的引脚的 SEL 功能选择寄存器位设置为 1，并把 DIR 方向寄存器位设置为输出。请编程实现完成，并用示波器观察。

3.3 定时器

介绍定时器前先介绍下系统时钟，定时器是基于系统时钟进行定时的。

MSP430 单片机工作的系统时钟被分为了 MCLK、SMCLK 和 ACLK 三个，使用时根据实际情况选择一种时钟。三种时钟的功能区别如下：

MCLK：主时钟（Main system Clock）,专为 CPU 运行提供的时钟。MCLK 频率配置的越高，CPU 执行的速度越快。虽然 CPU 速度越快功耗也越高，但高频率的 MCLK 可以让 CPU 工作时间更短。所以正确的低功耗设计并不是要尽量降低 MCLK，而是在不用 CPU 时立刻关闭 MCLK。在大部分应用中，需要 CPU 运算的时间都非常短，所以，间歇开启 MCLK（唤醒 CPU）的方法节能效果非常明显。

SMCLK：子系统时钟（Sub-main Clock），专为一些需要高速时钟的片内外设提供服务。比如定时器和 ADC 采样等，当 CPU 休眠时，只要 SMCLK 开启，定时器和 ADC 仍可工作（一般待片内外设完成工作后触发中断，唤醒 CPU 去做后续工作）。

ACLK：辅助时钟（Auxillary Clock），辅助时钟的频率很低，所以即使一直开启功耗也不大，当然也是可以关掉的。 辅助时钟可以供给那些只需低频时钟的片内外设， 比如 LCD 控制器， 还可用于产生节拍时基，与定时器配合间歇唤醒 CPU。

MCLK、SMCLK 和 ACLK 三者关系。需要用 MCLK 的时候不多，一般情况都处于休眠状态，以节约功耗。能只用 SMCLK 解决的问题，就别动用 MCLK，待 SMCLK 完成自己的任务后，再请 MCLK 出马。当没有实际任务的时候，MCLK 和 SMCLK 都可以休眠，但是要 ACLK 工作，检测到任务后唤醒 MCLK 和 SMCLK。

定时器是 MCU 中重要的功能单元之一，有着多种用法，能够实现多种功能。下面主要介绍 MSP430F5529 中的定时器 A（TIMER_A）、看门狗定时器（WDT）、实时时钟（RTC_B）。

定时器是根据时钟脉冲累积计时的，根据时钟源和分频系数来获取不同的时钟脉冲（定时器的工作过程实际上是对时钟脉冲计数）。每个定时器均有一个设定值寄存器和一个当前值寄存器。设定值寄存器存储编程时赋值的计时时间设定值，当前值寄存器记录计时当前值，这两个寄存器可以是 8 位、16 位或者 32 位二进制存储器。当前值寄存器的最大值乘以定时器的计时单位值即是定时器的最大计时范围值。定时器满足计时条件开始计时，当前值寄存器则开始计数，当当前值与设定值相等时定时器动作，产生相应的响应信号。

MSP430 单片机的定时器相当丰富。有定时器 A（TA）、定时器 B(TB)、实时时钟 RTC、看门狗定时器 WDT 等。其中看门狗主要用于出现软件故障后自动复位系统；TA 提供多路捕捉/比较模块，除了基本的定时功能，还可以用于 PWM 和脉冲测量；TB 与 TA 基本相同，具有提供计数器长度可编程、双缓冲式设定值寄存器、输出可设定为高阻态等高级特性，适用于一些有特殊要求的 PWM 操作；RTC 主要用来计时，可以提供日历模式的输出（可从分离的寄存器中直接读到二进制或 BCD 编码的年月日时分秒），也可用来计数。

3.3.1 Lab3-3-1 TIMER_A 实验

3.3.1.1 实验介绍

本实验使用了 MSP430F5529 中的 Timer_A 定时器模块，运用定时中断功能实现对 LED 指示灯的定时亮灭控制，并且控制蜂鸣器的发声频率。

3.3.1.2 实验目的

- 学习 Timer_A 定时器原理
- 熟练掌握 Timer_A 的定时中断
- 掌握蜂鸣器的使用方法
- 熟练应用 GPIO 控制 LED 亮灭

3.3.1.3 实验原理

◆ **TIMER_A 介绍**

定时器 A 是一个十六位的定时/计数器，MSP430F5529 中包含有 3 个定时器 A 的子模块和 7 个捕捉/比较模块。定时器 A 支持多重捕获/比较，PWM 输出和内部定时。定时器还有扩展中断功能，中断可以由定时器溢出产生或由捕获/比较寄存器产生。定时器 A 的特性包括：

- 四种运行模式的异步 16 位定时/计数器
- 可选择配置的时钟源
- 可配置的 PWM 输出
- 异步输入和输出锁存
- 对所有 TA 中断快速响应的中断向量寄存器

Timer_A 主要有 TAxCTL，TAxR，TAxCTLn，TAxCCRn，TAxIV、TAxEX0 几个寄存器（其中 x 指定时器的实例号，对 MSP430F5529 可以是 0~2，即 TA0~TA2；n 指捕捉/比较模块号）。其中最主要的是 TAxCTL 寄存器，用于设定 Timer_A 的输入时钟信号源、工作模式，复位、中断使能等。定时器 A 大致可分计数器和若干个捕捉比较模块。计数器在工作时由选定的时钟信号驱动计数，并且在计数归零时触发对应的定时器的 TAIFG；计数器配合几个比较/捕获寄存器便可实现定时、PWM、脉冲测量等功能。

TIMER_A 有 3 种定时/计数方式，方式选择由 TAxCTL 寄存器中的 MC 位控制：

MC	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TAxCCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TAxCCR0 and back down to zero.

图3.3.1 TIMER_A 模式选择

- 增计数模式（01）：设置 MCx=01，主定时器将工作在增计数模式下。与连续计数不同的是， CCR0 模块可以提前将 TAR 寄存器清零。如图 3.3.2 左侧所示，当 TAR 的值与 TACCR0 预设值相等时，TAR 被强迫清零。时钟表盘只能在灰色区域活动。

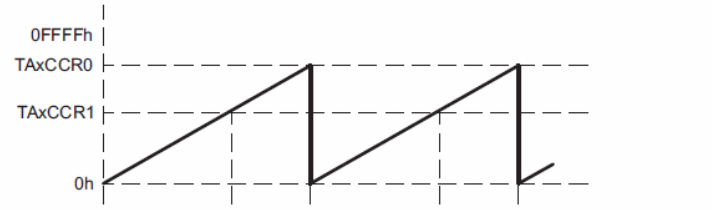


图3.3.2 增计数模式

- 连续计数模式（10）：设置 MCx=10，主定时器将工作在连续计数模式下。TAR 寄存器最大值 65535，计满则清零。如图 3.3.3 右侧所示，时钟的周期仅由时钟源的频率决定，频率越高则越快计数至 65535，周期越短。

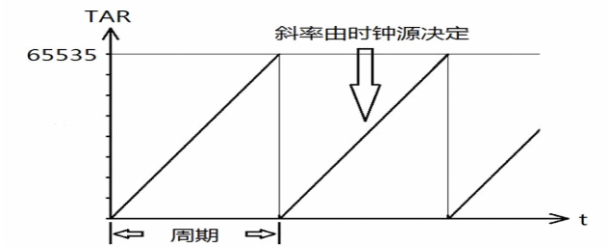


图3.3.3 连续计数模式

- 增减计数模式（11）：TAxR从0增加到TAxCCR0然后再从TAxCCR0减到0

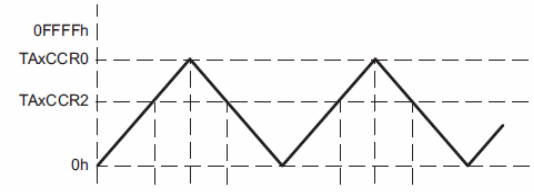


图3.3.4 增减计数模式

16 位定时/计数器寄存器 TAxR，随着分频后的时钟信号的上升到来沿增加/减少（由模式所决定）。TAxR 可以由软件读写。除此之外，定时器溢出时可以产生中断。TAxR 可以通过设置 TACLR 位来归零；在 UP/DOWN 模式下，设置 TACLR 也可以清除时钟分频器和计数方向。需要注意的是，操作定时器的相关寄存器前应当先停止定时器（中断使能、中断标志、TACLR 例外），以避免产生错误的运行结果。另外若定时器时钟与 CPU 时钟不同步，则在定时器运行中读取 TAxR 将产生不可预知的结果。一种可行的解决方案是在定时器运行时多读几次，通过软件表决的方式来确定正确的读数。对 TAxR 的写操作将会立即生效。

定时器的时钟源可以是内部时钟源 ACLK，SMCLK 或者或者外部源 TACLK。时钟源由 TAxCTL 寄存器的 TASSEL 域来选择，所选的时钟可以通过 ID 域选择进行 1，2，4，8 分频，之后时钟可以使用 TAxEX0 寄存器的 IDEX 域控制进一步进行 1，2，3，4，5，6，7，8 分频。

设置 TACLR 位将清除 TAxR 和分频器的状态。当 TACLR 位被清除时，TAxR 和分频器中的计数都将重新从 0 开始。

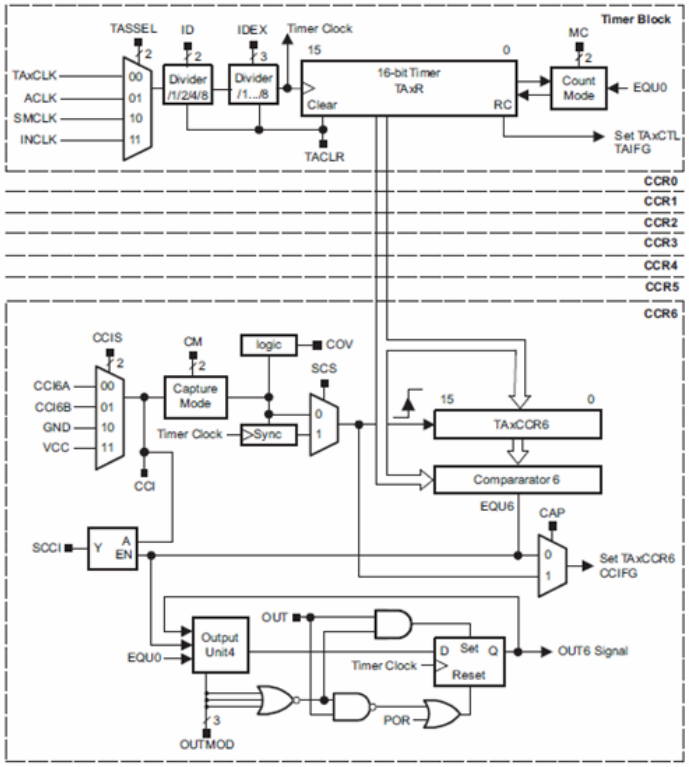


图3.3.5　TIMER_A 原理框图

◆ 蜂鸣器介绍

蜂鸣器按其是否带有信号源又分为有源和无源两种类型。有源蜂鸣器只需要在其供电端加上额定直流电压，其内部的震荡器就可以产生固定频率的信号，驱动蜂鸣器发出声音。无源蜂鸣器可以理解成与喇叭一样，需要在其供电端上加上高低不断变化的电信号才可以驱动发出声音。用跳线把图 3.3.6 中的 3 和 5 相连，4 和 6 相连，用定时器产生时序高低电平信号送入 P3.6 引脚实现对蜂鸣器的控制。

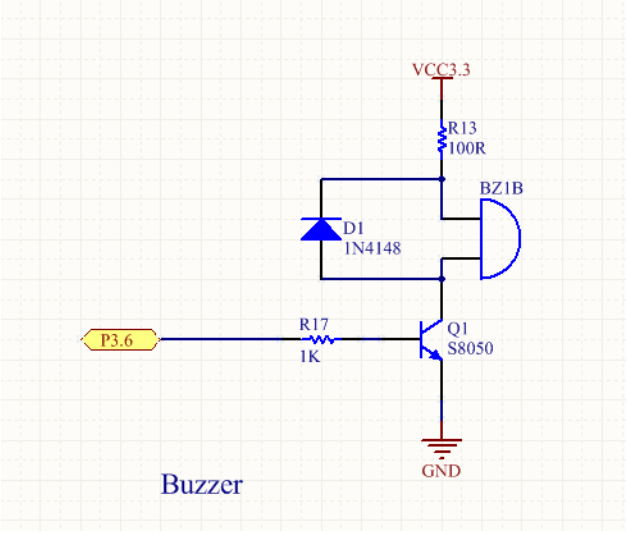


图3.3.6 蜂鸣器模块电路原理图

3.3.1.4 程序分析

- **编程思路**

实验开始必须先关闭看门狗定时器，然后配置好连接蜂鸣器和 LED 灯的 GPIO 寄存器，重点设置 TA0CTL 寄存器，使 Timer_A 工作在增计数模式，并选择 SMCLK 为其时钟源。使能比较器中断后，设定比较值为 50000，最后进入低功耗并打开总中断。

- **程序流程图**

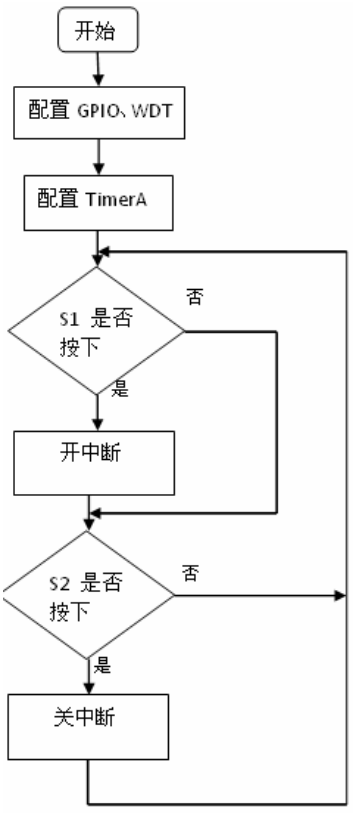


图3.3.7 程序流程图

- 关键代码分析

主函数

```
void main(void)
{
      WDTCTL = WDTPW | WDTHOLD;                    // Stop watchdog timer
      IO_Init();
     TA0CTL |= MC_1 + TASSEL_2 + TACLR;          //时钟为 SMCLK,比较模式，开始时清
零计数器
     TA0CCTL0 = CCIE;                             //比较器中断使能
     TA0CCR0   = 100;                             //比较值设为 50000，相当于 50ms 的时
间间隔
     while(1)
     {
      ScanKey();
      if(KeyFlag.S1==1)
      {
            KeyFlag.S1=0;
            __enable_interrupt();
            //__bis_SR_register(LPM0_bits + GIE);          //进入低功耗并开启总中断
      }
      if(KeyFlag.S2==1)
            {
                  KeyFlag.S2=0;
                   __disable_interrupt();
            }

     }
}
```

Timer_A 定时器中断服务函数

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)

{
      i++;
      if(i==500)
      {
      P8OUT ^= BIT1;                                 //形成闪灯效果
      i=0;
      }
      P3OUT ^= BIT6;                               //形成鸣叫效果
}
```

3.3.1.5 实验步骤与现象

- 实验步骤

1. 根据原理图，把蜂鸣器相关跳线接上。
2. 实现控制代码的设计，画出流程图；
3. 完成编码，并将代码烧录到开发板中；
4. 观察 LED 的状态变化，验证结果；
5. 听蜂鸣器的声音频率，验证效果。

- 实验现象

口袋板上 S1 按键按下后 L1 指示灯每隔50ms闪烁一次，同时蜂鸣器发出响声；S2 按键按下后蜂鸣器停止。

3.3.1.6 实验思考

1. Timer_A 定时器都有哪些工作模式，如何配置？
2. 如何通过定时器设定蜂鸣器发声的音调？

3.3.2 Lab3-3-2 看门狗定时器（WDT）实验

3.3.2.1 实验介绍

本实验使用了 MSP430F5529 中的看门狗定时器模块，运用看门狗定时中断功能实现了对 LED 的定时亮灭控制。

3.3.2.2 实验目的

- 学习看门狗定时器原理。
- 熟练掌握看门狗定时中断。
- 熟练应用 GPIO 控制 LED 亮灭。

3.3.2.3 实验原理

WDT（Watch Dog Timer）俗称看门狗，是单片机非常重要的一个片内外设。早期，普通 51 单片机内部没有它，则有使用专门的外扩看门狗芯片。在电子产品中，被称为“狗”的就是忠诚可靠的代名词。

什么是看门狗呢？看门狗实际就是一个定时器，只不过在定时到达时，可以复位单片机。这个功能对于实际工程应用中的产品非常有用。在很多应用中，单片机要经年累月的连续工作，如果期间单片机由于各种意外死机（俗称跑飞），则单片机就经年累月的不工作了。

有了看门狗，就可以避免这种意外的发生。单片机都是循环工作的，比如完成整个循环所需时间最长不超过 1 秒，则可以把看门狗定时器的定时值设为 1.2 秒，在主循环中加入看门狗定时值清零的代码（俗称喂狗）。这样一来，假如程序运行正常，则总会在看门狗定时器到点前“喂狗”，从而避免单片机复位。如果程序死机，则不会及时“喂狗”，单片机复位。复位后看门狗依然默认开启，继续守护着程序的正常运行。

综上所述，看门狗的原理就 8 个字“定时喂狗，狗饿复位”。

复位后单片机状态为：

（1） I/O 引脚切换成输入模式。

（2） I/O 标志位清除。

（3） 其它外围模块及寄存器实现初始化。

（4） 状态寄存器复位。

（5） CPU 从内存的 0FFFE 地址开始执行代码。

看门狗定时器（WDT）实际上是一个计数器，初始计数值为 0，处理器上电复位(PUC)后看门狗开始计数。如果程序运行正常，过一段时间 CPU 应发出指令让看门狗复位，重新开始计数；如果看门狗溢出就认为程序没有正常工作，产生看门狗中断。

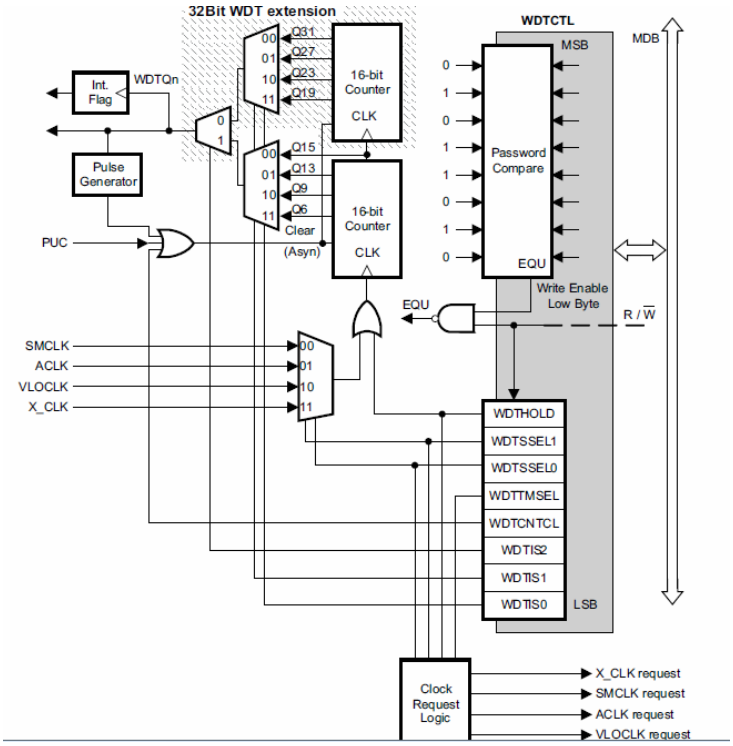


图3.3.8　WDT 定时器结构框图

WDT 由控制寄存器 WDTCTL 控制其计数器不可由程序访问。控制寄存器 WDTCTL 为 16 位寄存器，其高八位用作口令，低八位用作对 WDT 的实际控制，下面为控制寄存器各域的说明：

- WDTPW：　需要注意每次操作 WDTCTL 时必须写入本段，即对 WDTCTL 的操作均应当为 WDTCTL= WDTPW+....这种格式，否则将触发器件复位。
- WDTHOLD：　看门狗使能控制，向该位写入 1 时关闭看门狗。
- WDTTMSEL：模式选择，‘0’ 看门狗模式，‘1’ 定时器模式
- WDTCNCTL：计数器清除控制,当被置'1'后会自动复位为'0'，'0' 无动作，'1'计数器清零
- WDTSSEL：　选择 WDTCN 的时钟源，'0' 选择 SMCLK，'1'选择 ACLK。
- WDTISx：　看门狗的定时间隙选择。

Watchdog Timer Control Register (WDTCTL)

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Read as 069h WDTPW, Must be written as 05Ah							

7	6	5	4	3	2	1	0
WDTHOLD	WDTSSEL		WDTTMSEL	WDTCNTCL	WDTIS		
rw-0	rw-0	rw-0	rw-0	r0(w)	rw-1	rw-0	rw-0

WDTPW	Bits 15-8	Watchdog timer password. Always read as 069h. Must be written as 05Ah, or a PUC is generated.
WDTHOLD	Bit 7	Watchdog timer hold. This bit stops the watchdog timer. Setting WDTHOLD = 1 when the WDT is not in use conserves power.
		0 Watchdog timer is not stopped.
		1 Watchdog timer is stopped.
WDTSSEL	Bits 6-5	Watchdog timer clock source select
		00 SMCLK
		01 ACLK
		10 VLOCLK
		11 X_CLK; VLOCLK in devices that do not support X_CLK
WDTTMSEL	Bit 4	Watchdog timer mode select
		0 Watchdog mode
		1 Interval timer mode
WDTCNTCL	Bit 3	Watchdog timer counter clear. Setting WDTCNTCL = 1 clears the count value to 0000h. WDTCNTCL is automatically reset.
		0 No action
		1 WDTCNT = 0000h
WDTIS	Bits 2-0	Watchdog timer interval select. These bits select the watchdog timer interval to set the WDTIFG flag and/or generate a PUC.
		000 Watchdog clock source /2G (18:12:16 at 32 kHz)
		001 Watchdog clock source /128M (01:08:16 at 32 kHz
		010 Watchdog clock source /8192k (00:04:16 at 32 kHz)
		011 Watchdog clock source /512k (00:00:16 at 32 kHz)
		100 Watchdog clock source /32k (1 s at 32 kHz)
		101 Watchdog clock source /8192 (250 ms at 32 kHz)
		110 Watchdog clock source /512 (15,6 ms at 32 kHz)
		111 Watchdog clock source /64 (1.95 ms at 32 kHz)

图3.3.9 WDT 定时器寄存器

3.3.2.4 程序分析

- **编程思路**

本实验的关键在于 WDT 的定时功能应用，首先得配置好看门狗的工作模式和时钟源，然后通过中断的方式实现定时地对 LED 的亮灭进行控制，当按键按下不停喂狗时，不会产生中断。

- **程序流程图**

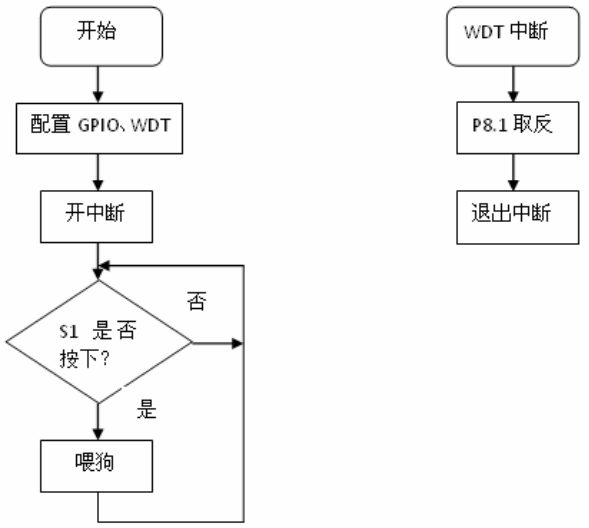


图3.3.10 程序流程图

- 关键代码分析

WDT 实验代码

主函数

```
#define                                                    WDT_ADLY_1000
(WDTPW+WDTTMSEL+WDTCNTCL+WDTIS2+WDTSSEL0)
                // 时钟为 ACLK，模式为内部时钟。

int main(void)
{
      WDTCTL = WDT_ADLY_1000;
      InitIO();
      SFRIE1 |= WDTIE;          // 开启看门狗中断

      __enable_interrupt();
      while(1)
      {
            ScanKey();
            if(KeyFlag.S1==1)
            {
                  KeyFlag.S1=0;
                  WDTCTL = WDT_ADLY_1000;
            }

      }
}
```

看门狗定时器中断服务函数

```
#pragma vector=WDT_VECTOR
__interrupt void WDT_ISR(void)
{
      P8OUT ^= BIT1;                                  // P4.5 口取反
}
```

3.3.2.5 实验步骤与现象

- **实验步骤**

1. 完成代码，并将代码烧录到开发板中；
2. 观察 LED1 的状态变化，验证结果。
3. 按下 S1 观察 LED1 变化

- **实验现象**

口袋板左下角 L1 指示灯闪烁表示系统工作异常，看门狗定时器（WDT）产生了中断，当我们不停的按下 S1 按键时，发出一个“喂狗”的信号（实际是让 WDT 定时器复位，不产生中断，表示系统工作正常），结果就是 L1 指示灯的状态不再变化（长亮或长灭）；当我们不按下 S1 键或按下的频率比 WDT 定时器产生中断的频率低的时候，L1 指示灯就以 1Hz 左右的频率不停闪烁。

3.3.2.6 实验思考

1. 如何灵活运用 WDT 实现负责任务的调度与切换？
2. 能否自定义 WDT 的中断时间间隔？
3. WDT 如何实现程序防跑飞?

3.3.3 Lab3-3-3 实时时钟（RTC）实验

在一个嵌入式系统中，通常采用 RTC 来提供可靠的系统时间，包括时分秒和年月日等。而且要求在系统处于关机状态下它也能够正常工作（通常采用后备电池供电），它的外围也不需要太多的辅助电路，典型的就是只需要一个高精度的 32.768KHz 晶体和电阻电容等。

3.3.3.1 实验介绍

该实验使用了 MSP430F5529 的 RTC_A 模块和外围的按键、TFT 液晶设备。RTC_A 用来计时，按键用来设置时间，液晶模块用来显示时间。

3.3.3.2 实验目的

- 熟练使用 RTC_A 的初始化和 RTC_A 中断
- 熟练应用按键和液晶显示

3.3.3.3 实验原理

◆ **RTC 特点**

实时时钟模块提供了具有日历模式、可编程闹钟和晶振频率校准功能的时钟。MSP430F5529 单片机有 RTC_A、RTC_B、RTC_C 三种实时时钟模块，RTC_A 模块可以配置为日历模式，也可以作为通用计数器使用；RTC_B 模块只能配置成日历模式，并且时钟源只能是 32768HZ 的外设时钟，能够在低功耗 LPMx.5 下工作；本实验用到的是 RTC_A 模块，它具有以下几个特点：

- 在日历模式中提供了秒钟、分钟、小时、星期、日期、月份和年份；
- 具有中断功能；
- 可以选择 BCD 码或者二进制格式；
- 具有可编程闹钟；

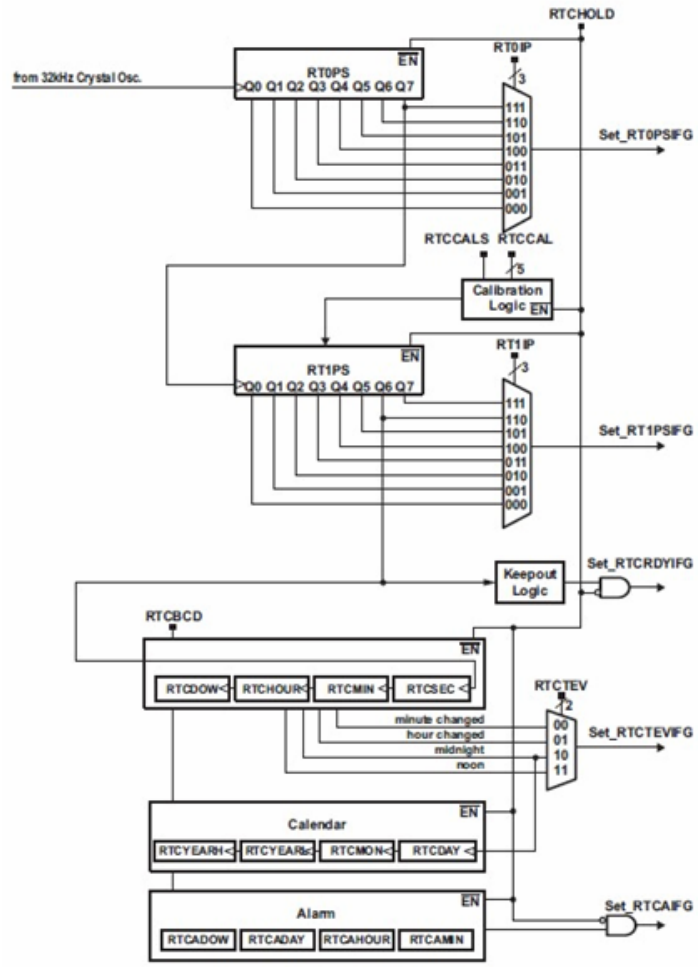


图3.3.11　RTC_A 结构框图

◆ **RTC_A 的初始化**

实时时钟模块的大多数寄存器没有初始化。在使用这个模块之前，用户必须通过软件进行配置。通过配置 RTCCTL01 寄存器的 HOLD 位来选择启动 RTC_A，配置 RTCBCD 位来选择数据格式。实时时钟将通过自动配置 RT0PS 和 RT1PS 为实时时钟提供一秒间隔的时钟。

◆ **RTC_A 的数据读取**

在日历模式下，实时时钟寄存器每秒更新一次。因为系统时钟实际上和实时时钟的时钟源不是同步的，为了防止在更新的时候读取实时数据而造成错误数据的读取，将会有一个禁止进入的窗口。RTCRDY 标明了可以进行读取的时段：仅有当 RTCRDY 被置位时可以安全的读取实时时钟寄存器。

◆ **RTC_A 的闹钟功能**

实时时钟模块提供了一个灵活的闹钟系统。通过设置闹钟的秒、分、小时、星期、月份、年份寄存器和对应的 AE 位，可以组合得到所需的闹钟功能。

实时时钟模块提供了 5 个中断源，每个中断源都有独立的使能位和标志位，分别是 RT0PSIFG、RT1PSIFG、RTCRDYIFG、RTCTEVIFG、RTCAIFG。这些中断标志共用同一个中断向量且具有优先次序，可由 RTCIV 判断当前请求中断的最高优先级的中断标志。

3.3.3.4 程序分析

- **编程思路**

首先初始化时钟、LCD，再初始化 RTC_A，将 RTC 的数据格式设为 BCD 格式、置高 HOLD 位，RTC 的小时、分钟、秒设为零；连接按键的 IO 设置为输入上拉模式，读取 RTC 时间并更新到显示屏上，扫描等待按键输入。在按键按下后，检测是哪个按键按下，然后响应该按键的功能，例如设置秒，分，时等等。

- **程序流程图**

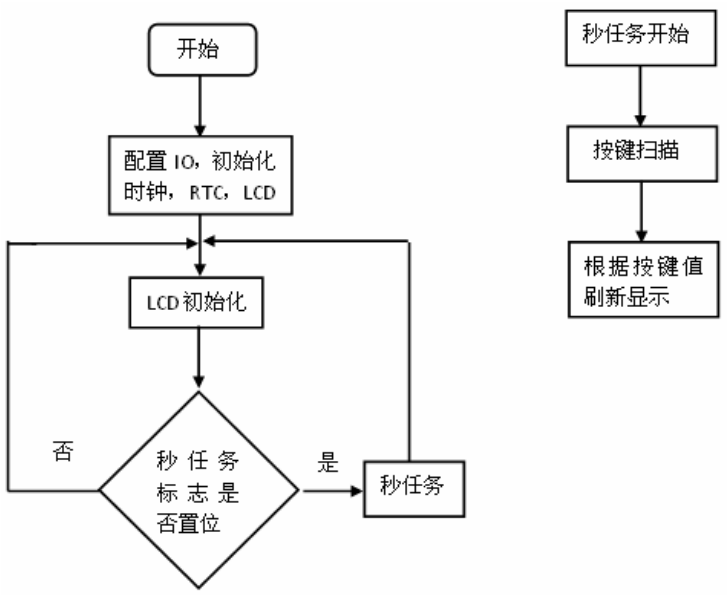


图3.3.12 程序流程图

- **关键代码分析**

实验 RTC 代码

实时时钟初始化

```
void Init_Rtc(void)
{                                                    // BCD 码日历格式输出

        RTCCTL01 = RTCBCD +    RTCHOLD + RTCMODE + RTCTEV_0;
        RTCSEC =    0x54;                            //初始化秒
        RTCMIN =    0x59 ;                           //初始化分钟
        RTCHOUR =0x21;                               //初始化小时
        RTCDOW = 0x02 ;
        RTCDAY =    0x24;                            //日期初始化
        RTCMON =    0x11 ;                           //初始化月份
        RTCYEAR = 0x2005;                            //初始化年份
        RTCCTL01 &= ~RTCHOLD;                        //打开 RTC 模块

}
```

3.3.3.5 实验步骤与现象

- **实验步骤**

1. 连接电源；
2. 完成编码，并将代码烧录到开发板中；
3. 观察液晶显示。
4. 按 S1 改变设置状态（秒，分，时.....），按 S2 和 S3 进行加减，按 S4 确认

- **实验现象**

上电后液晶显示时间信息，按第一下 S1 进入设置秒状态，屏幕下方会显示“SET SEC”，第二下按下 S1 会进入设置分状态，屏幕下方会显示“SET MIN”，以此类推设置时、日、月、年等。最后再按下 S1 会推出进入正常时间显示界面。

3.3.3.6 实验思考

1. RTC_A、RTC_B、RTC_C 三种实时时钟有何异同？
2. 如何使用 RTC_B 的闹钟功能？

3.4 模拟电压比较器 B 模块

比较器的原理是比较两个模拟信号的大小，将大或小的结果以数字 1、0 的方式输出。也就是输入是模拟信号，输出为数字信号。下面将介绍 3 种类型的模数转换器（ADC），逐次比较型、并行比较型和积分型，它们都是用比较器为核心元件。

比较器在逐次比较型 ADC 中的应用：

逐次逼近转换过程和用天平称物重非常相似。天平称重物过程是，从最重的砝码开始试放，与被称物体行进比较，若物体重于砝码，则该砝码保留，否则移去。再加上第二个次重砝码，由物体的重量是否大于砝码的重量决定第二个砝码是留下还是移去。照此一直加到最小一个砝码为止。将所有留下的砝码重量相加，就得此物体的重量。仿照这一思路，逐次比较型 A/D 转换器，就是将输入模拟信号与不同的参考电压作多次比较，使转换所得的数字量在数值上逐次逼近输入模拟量对应值。如图 3.4.1 所示，最终测的的结果就是天平中央的示数 101，也就是 5g。这种方法称为逐次逼近法，学名 SAR 型 ADC。对于 3 位分辨率的 SAR，需要测量 3 次。

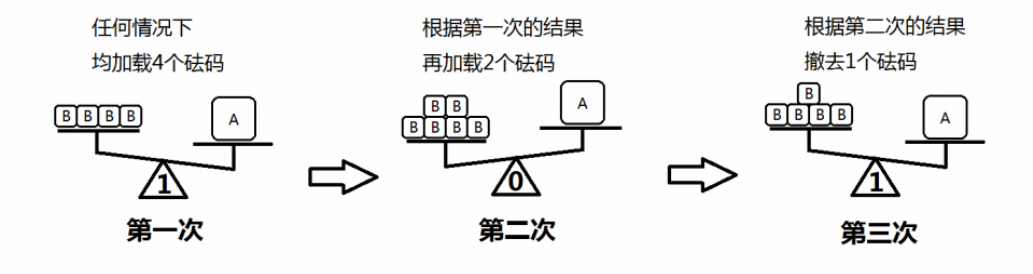
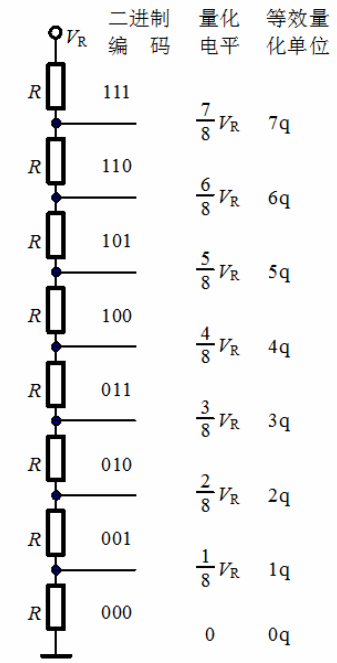


图 3.4.1　逐次比较型

比较器在并行比较型 ADC 中的应用：



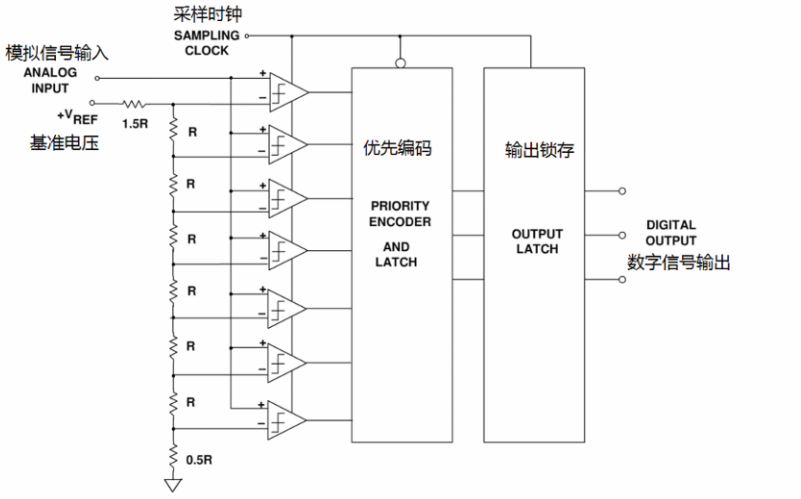


图 3.4.2　并行比较型

比较器在积分比较型 ADC 中的应用：

积分型（slope 型）ADC 的优点是结构简单，分辨率可以做得非常高（调整放电时间），并且抗干扰能力非常强（例如 50Hz 的工频电网干扰）。缺点就一个，只能测量缓变信号，所以非常适合在温度、压力等类型的传感器领域中使用。我们家用的各种厨房电子秤、体重电子秤都是应用积分型 ADC 的原理。

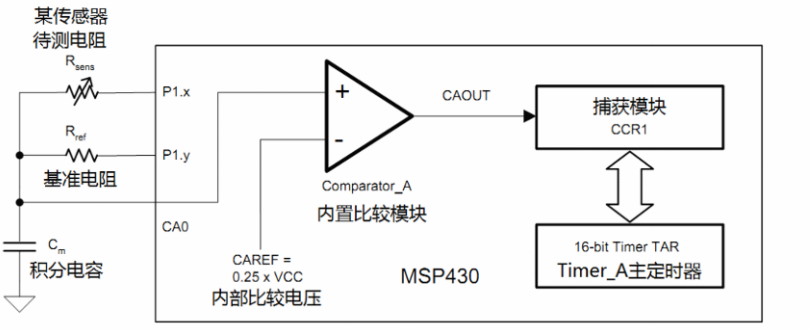


图 3.4.3　积分比较型

比较器和普通运放的区别：

前面我们讲过，比较器就是一类特殊的运放。

（1） 一般情况，不要混用普通运放和比较器，那样很可能是用家具当劈柴烧。

（2） 比较器的压摆率比普通运放高，简单说，就是“快”。

（3） 相比于比较器，普通运放内部有补偿电容，不易自激振荡。

（4） 输入端：普通运放正常工作时两输入端几乎没有压差，所以其输入端往往有电压嵌位保护。比较器的输入端则没有，混用时应特别注意。

（5） 输出端：运放输出端一般是图腾柱输出，多数不能满幅输出（轨至轨型除外）。比较器多数是集电极开路 OC 或漏极开路 OD 输出，需外接上拉电阻，以便匹配后级电平。图腾柱与 OC、OD 输出的区别可见 GPIO 章节。

本节主要是介绍 MSP430F5529 中模拟电压比较器 B 模块的使用，这是一个重要的模块，主板上的电容触摸按键（Pad1、Pad2）的功能实现就是依靠电压比较器 B。

- 比较器 B 特点

比较器 B 模块可用于精确的斜率式模数转换，电压监控和外部模拟信号的监控。比较器 B 有如下特点：同相端和反相端均有输入复用器；比较器输出端具有软件选择的 RC 滤波器；比较器的输出可以作为定时器 A 的捕获输入；端口输入缓冲由软件控制；具有中断功能，并支持在低功耗模式响应；可选择的参考电压发生器,电压滞回发生器；可使用共享参考源作为参考电压；超低功耗比较模式。

- 比较器的结构

由下图可以看出比较器 B 模块大概可以分为 5 个部分：模拟输入部分、核心部分、低通滤波部分、基准电压部分和比较器输出部分。

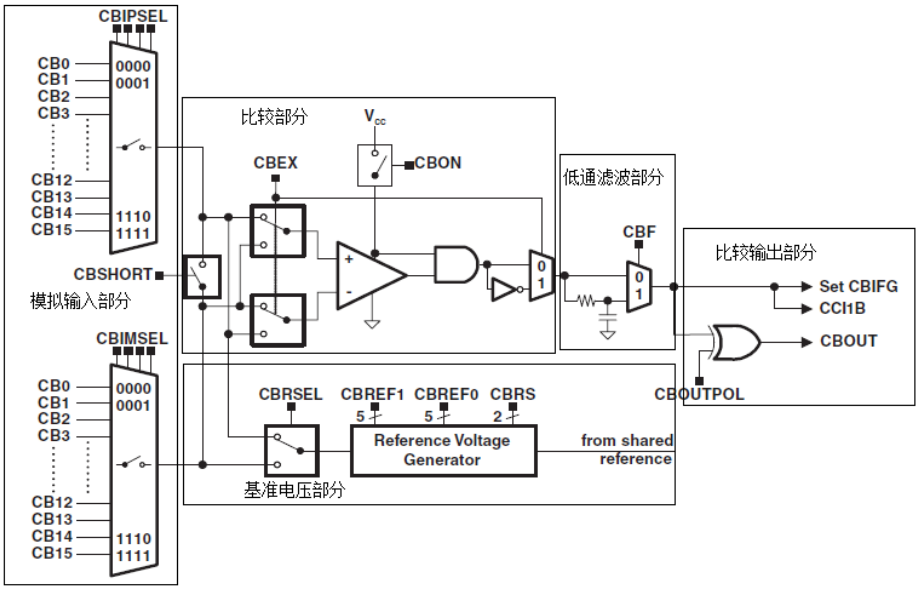


图 3.4.4　电压比较器 B 结构框图

- **模拟输入部分：**比较器 B 的正负输入端各自有 16 个通道选择，也都可以选择参考电压。CBCTL0 控制寄存器中的 CBIMEN 和 CBIPEN 位分别用于使能反相端和同相端的模拟信号输入，CBIMSEL、CBIPSEL 位分别用于选择反相端和同相端的输入通道。同时我们可以通过操作控制寄存器 CBCTL3 来禁止或者使能比较器相应的输入通道的输入缓存。

- **比较部分：**整个比较器 B 工作需将 CBCTL1 控制寄存器的 CBON 位置为 1，单片机上电复位时，此位为 0；CBCTL1 中的 CBEX 位控制输入方向。

- **基准电压部分：**主要由控制寄存器 CBCTL2 中的 CBRS、CBRSEL 、CBREF1、CBREF0 CBREFL 等位控制。CBRS 选择基准电压源，CBRSEL 位选择基准电压输出到同相端或反相端，CBREF1、CBREF0 控制基准电压生成器的电阻网络以对输入电压进行分压。

- **低通滤波部分：**使用 CBCTL 控制寄存器中的 CBF 位来控制此滤波器的功能开与关。此滤波器功能是用于消除比较器输出信号的毛刺，以保证信号的质量和中断请求的可靠性。

比较器 B 的输出既可以输出到内部模块也可以输出到外部引脚，也可以用于产生中断。比较器 B 模块可以在比较器输出的上升沿或者下降沿产生中断，所用的边沿由 CBCTL1 的 CBIES 选择。比较器 B 模块具有独立的中断向量，中断被响应后硬件会自动清除中断标志位 CBIFG。

Register	Short Form	Register Type	Address Offset	Initial State
Comp_B control register 0	CBCTL0	Read/write	0x0000	Reset with PUC
Comp_B control register 1	CBCTL1	Read/write	0x0002	Reset with PUC
Comp_B control register 2	CBCTL2	Read/write	0x0004	Reset with PUC
Comp_B control register 3	CBCTL3	Read/write	0x0006	Reset with POR
Comp_B interrupt register	CBINT	Read/write	0x000C	Reset with PUC
Comp_B interrupt vector word	CBIV	Read	0x000E	Reset with PUC

图 3.4.4 电压比较器 B 相关寄存器

3.4.1 Lab3-4-1 电压比较器实验

3.4.1.1 实验介绍

该实验使用了 MSP430F5529 的比较器 B，一路使用 P6.5 管脚，一路使用内部的 1.5V 参考电压，将两路信号进行比较；当 P6.5 管脚大于 1.5V 时触发中断，点亮 L1。

3.4.1.2 实验目的

- 熟悉比较器 B
- 学会使用低功耗工作模式

3.4.1.3 实验原理

比较器 B 的主要功能是指出两个输入电压 CPIP 和 CPIM 的大小关系，然后设置输出信号，比较器的两个输入电压各自有 16 个通道选择。在本次实验中 CPIP 选择的是 CB5 即 P6.5 管脚；CPIM 选择的是内部的 1.5V 参考电压；当 P6.5 管脚的电压大于 1.5V 时，触发中断，在中断里翻转触发的边缘，翻转 L1 的状态。

3.4.1.4 程序分析

- **编程思路**

本次实验只使用比较器 B 和 LED 电路，首先将连接 L1 的 P8.1 管脚设为输出模式，再初始化比较器 B，先选择 CNIP 的输入管脚为 CB5，关闭 P6.5 的输入缓冲。选择 CPIM 的管脚为内部参考电压 1（即 1.5V）,检测寄存器 CBCTL1 中 CBOUT 状态，根据 CBOUT 调整 P8.1 的亮灭。

- **程序流程图**

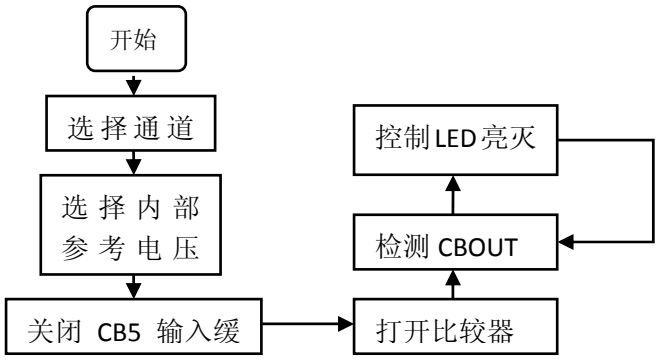


图 3.4.5 程序流程图

- 关键代码分析

实验 比较器 代码

主函数

```
int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // 关闭看门狗
  P8DIR |= BIT1;                            // P8.1/L1 设为输出
  P2DIR |= BIT2;
  P2OUT &= ~BIT2;                          //关闭背光
                                            // 配置比较器 B
  CBCTL0 |= CBIPEN + CBIPSEL_5;            // 使能正端，输入通道选择 CB5
  CBCTL1 |= CBPWRMD_1;                     // 正常的电源供应模式
  CBCTL2 |= CBRSEL;                        // 使能内部参考电压选择
  CBCTL2 |= CBRS_1+CBREFL_1;               // 参考电压选择为 1.5V
  CBCTL3 |= BIT5;                         // 关闭 CB5 输入缓存
  __delay_cycles(75);                       // 延时等待
  CBINT &= ~(CBIFG + CBIIFG);              // 清除中断标志
  CBINT    |= CBIE;                        // 使能比较中断，边缘选择为上升沿
  CBCTL1 |= CBON;                          // 打开比较器
  while(1)
  {
        if(CBCTL1&CBOUT)
        {
              P8OUT|=0x02;
        }
        else
              P8OUT&=~0x02;
  }

}
```

3.4.1.5 实验步骤与现象

- **实验步骤**

1. 完成编码，并将代码烧录到开发板中；
2. 通过调节电位器查看 L1 指示灯的状态变化；

- **实验现象**

旋转拨盘电位器，LED 灯指示出电位器输出电压的变化。

3.4.1.6 实验思考

1. 为什么要关闭 P6.5 的缓存？
2. 比较器 A 和比较器 B 有什么区别和共同点？

3.4.2 Lab3-4-2 触摸按键实验

3.4.2.1 实验介绍

本实验使用 MSP430F5529 触摸按键模块、LED 模块。利用 MSP430F5529 中的比较器 B 来判断触摸按键的充放电，通过 CPU 时钟算出电容的振荡频率，再判断是否有按键被触摸，如果有按键触摸则通过相应的 LED 显示当前的状态。

3.4.2.2 实验目的

- 学习电容触摸按键硬件电路原理；
- 学习触摸按键应用试验操作及编程思想；
- 学习定时器实现频率计的方法；
- 学习触摸电容程序资源。

3.4.2.3 实验原理

人体是具有一定电容的。当我们把 PCB 板上的覆铜画成如下形式的时候，就完成了一个最基本的触摸感应按键。

近年来，电容触摸这个词不时会在我们耳边回荡。比起机械按键，电容触摸按键的优点很多：成本低、外观花哨、寿命超长、容易实现防水防尘。

电容触摸的技术听起来很高端，但是原理却不复杂。 如图 3.4.6 所示，人手指接近金属会使金属对地的电容值增大，想办法测出电容值发生的变化，就能判断按键被按下。电容发生变化的原因是，电路板的材质和结构决定了板子上有寄生电容，当人体接触 PAD 时，人体就会和 PAD 产生寄生电容，这时电路中寄生电容值就发生了变化。

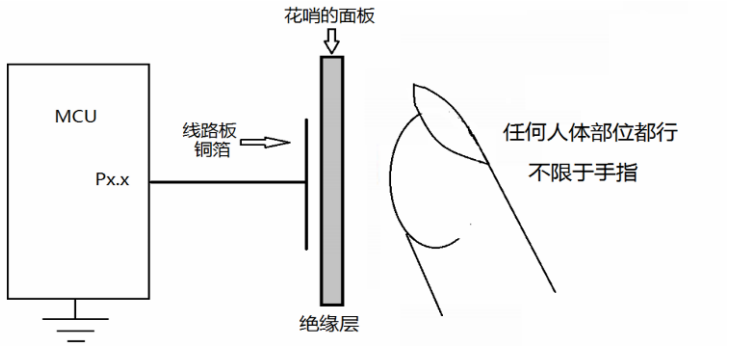


图 3.4.6 电容触摸原理

MSP430F5529 的比较器可以为比较结果 0 和 1 时设定两个阈值，这样便构成一个 RC 振荡器(电容电压低于较低阈值时充电、高于较高阈值时放电)，振荡频率与电容容值有关。当手指触摸到电容触摸按键以后，电容会由 C1 变化至 C2，振荡器的输出频率会发生变化，

因此只需在固定时间内，计算出振荡器的输出频率。如图 3.4.9 所示由于振荡频率较高(未触摸时在几百 KHz 量级)，可以直接用 CPU 时钟进行测量(即用变量自增来测量若干次振荡的耗时)。但为了防止测量中途出现中断影响测量，测量过程中应当关闭中断。如果在某一时刻输出频率有较大的变化的话，那就说明电容值已经被改变，即对应按键被按下了。

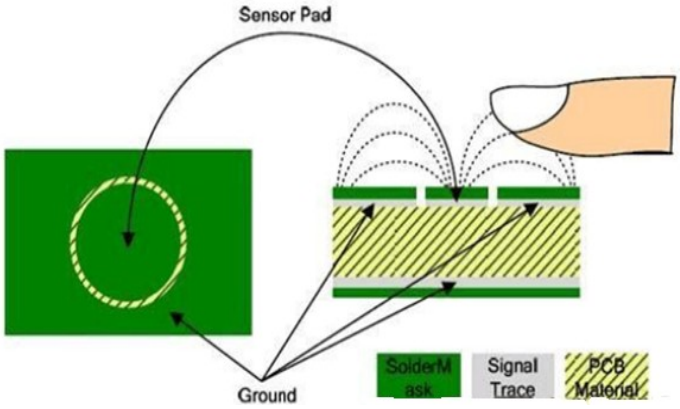


图 3.4.7　触摸按键工作原理图

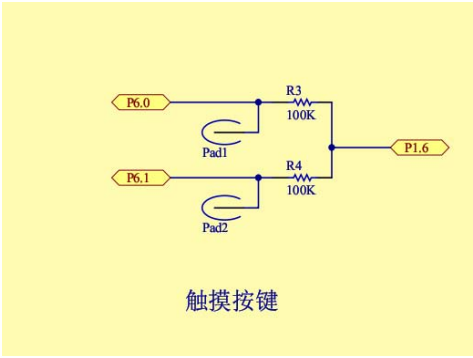


图 3.4.8　触摸按键模块电路原理图

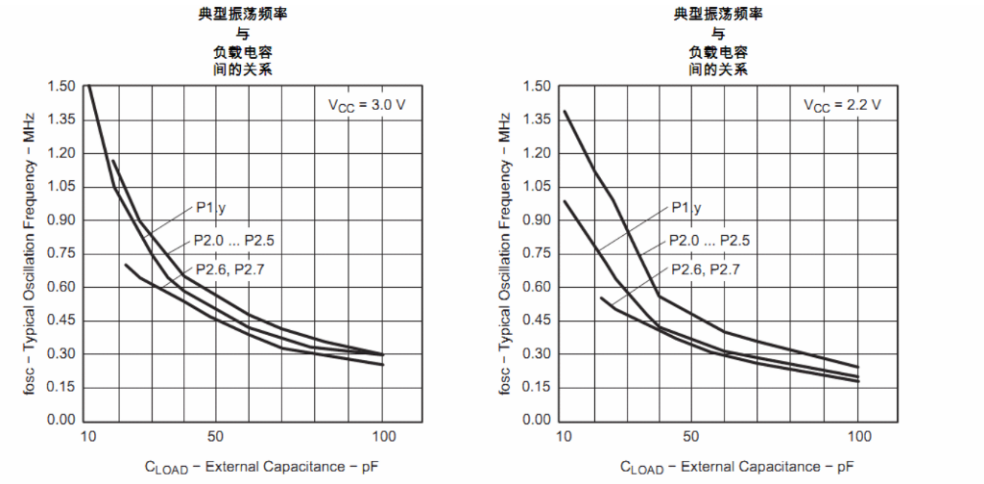


图 3.4.9　电容震荡特性

3.4.2.4 程序分析

- **编程思路**

先初始化触摸按键和控制 LED 灯的 IO 口，然后反复的测量各个触摸按键的频率值，当它们的频率的值大于某个门限值时，操作对应的 LED 灯。在每次测量中要加入适当的延时，防止反复测量导致 LED 误操作。

- **程序流程图**

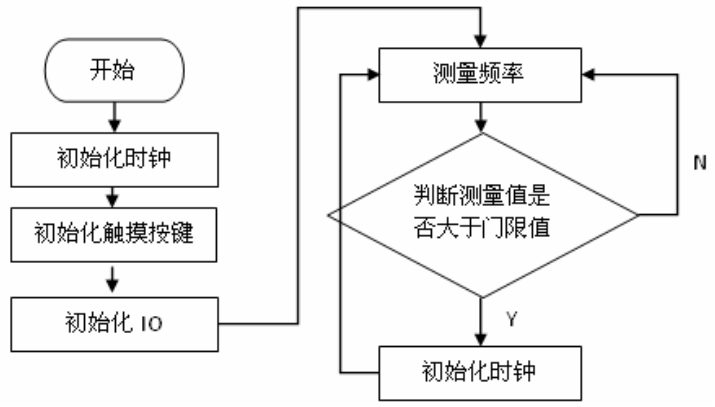


图 3.4.10　程序流程图

- **关键代码分析**

实验 触摸按键 代码

主函数

```
int main( void )
{
    initClock();                                          //初始化始终
    initCapTouch();                                       //初始化触摸按键
    for(i=0;i<2;++i)
    *LED_GPIO[i]->PxDIR |= LED_PORT[i];//设置各 LED 灯所在端口为输出方向
    _EINT();
    while(1)
    {
        uint32_t temp[2] ={0,0};
        for(i=1;i>=0;i--)
        {
          temp[i]= CapTouch_ReadChannel(i);           //测量触摸按键的频率值
          if((temp[i]>captouch_max)&&(lastflag[i]==0))
          {                              //当测量值大于门限值时翻转
              *LED_GPIO[i]->PxOUT ^= LED_PORT[i];   //对应的 LED 指示灯
              lastflag[i]=1;
          }
          else
              if(!(temp[i]>captouch_max))
              {
                  lastflag[i]=0;
              }
        }
        __delay_cycles(8000000);

    }
```

```
        return 0;
}
                                比触摸按键测量函数
uint16_t CapTouch_ReadChannel(int i)
{
   uint16_t cpu_cnt = 0;
   uint16_t osc_cnt = 0;
   CBCTL0 = CBIMEN + (i << 8);                    //外部信号加到负极
   P6OUT &= ~ALL_PORT;
   P6DIR |= ALL_PORT & ~(1 << i);
   CBCTL3 = 1 << i;
   uint16_t gie;
   _ECRIT(gie);
   while(1)
   {
      if(CBCTL1 & CBOUT)                          //控制电容的充放电
         P6OUT |= ALL_PORT;
      else
         P6OUT &= ~ALL_PORT;
      if(CBINT & CBIFG)
      {
         CBINT &= ~CBIFG;
         osc_cnt++;
      }
      if(osc_cnt >= OSC_CYCLES)
         break;
      cpu_cnt++;                                   //计算频率
   }
   _LCRIT(gie);
   CBCTL3 = ALL_PORT;
   P6DIR &= ~ALL_PORT;
   return cpu_cnt;
}
```

3.4.2.5 实验步骤与现象

触摸相应的按键 Pad1、Pad2，可以观察到对应的 LED 指示灯 L3、L4 的亮灭。

3.4.2.6 实验思考

请考虑用定时器来设计频率计，来计算触摸按键的电容。

3.5 通用串行通信接口----SPI 模式

通用串行通信接口（USCI）模块是 MSP430F5529 芯片上的重要通信接口，很多外设模块使用这一接口。5529 中 USCI 接口有 2 组：USCI_Ax 与 USCI_Bx，工作模式主要有：UART、SPI、I^2C 等等，不同的 USCI 模块所具有的功能也不尽相同，其中 USCI_Ax 模块支持 UART 模式、SPI 模式，该模块还可用于 IrDA 通信的脉冲整形以及 LIN 通信的自动波特率检测等；USCI_Bx 模块支持的特性包括 I^2C 模式和 SPI 模式。本节首先介绍 SPI 模式。

- **SPI 介绍**

在同步模式下，USCI 通过 3 个或者 4 个引脚把 MSP430 连接到一个外部系统中，这些引脚分别是：UCxSMIO，UCxSOMI，UCxCLK 和 UCxSTE。同步位 UCSYNC 被置位且模式选择位 UCMODE 位选择 SPI 时 USCI 模块工作于 SPI 模式。

- **SPI 的特点**
  - 具有 7 位或 8 位的数据位选择
  - 支持 3 线或 4 线 SPI
  - 支持主从模式
  - 独立的发送和接受寄存器
  - 分离的发送和接受缓冲寄存器
  - 支持低位在前或高位在前的收发
  - 独立的接收中断和发送中断
  - 主模式下时钟频率可编程
  - 从模式可工作在 LPM4 下

- **SPI 的结构**

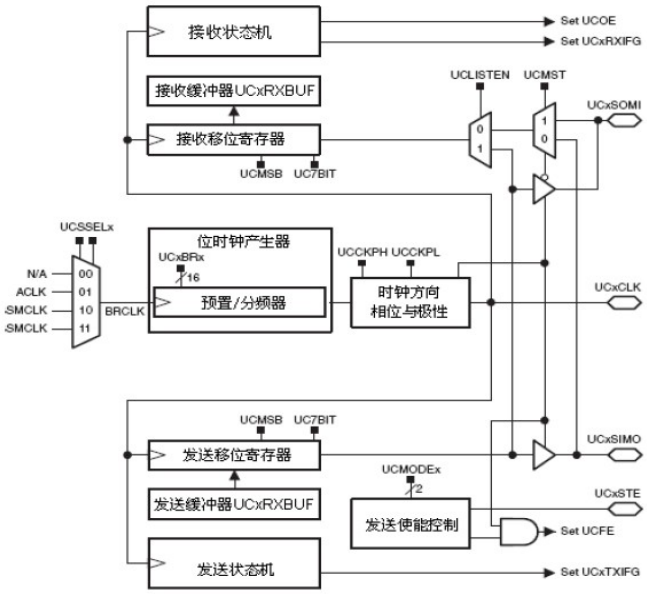


图3.5.1 SPI 接口结构框图

如上图所示，在 SPI 模式下，数据的发送和接收是由多个器件共享一个由时钟运行

的，该时钟是由一个主机提供的。UCxSIMO 是从机输入、主机输出，UCxSOMI 是从机输出、主机输入；UCxCLK 是 USCI 的 SPI 模式的时钟；UCxSTE 是从模式下的发送使能端，由主机控制，用于 4 线模式中选择一个从机接收和发送数据。

- **USCI 初始化和复位**

USCI 可以被 PUC 或者 UCSWRST 位复位。在 PUC 后，UCSWRST 位自动置位，保持 USCI 在复位状态。当置位时，UCSWRST 位复位 UCRXIE、UCTXIE、UCRXIFG、UCOE、UCFE 位并置 UCAXTXIFG 位。清除 UCSWRST 位可以是 USCI 进入工作状态。相应的 USCI 初始化/重配置的过程如下：

1. 设置 UCSWRST
2. UCSWRST=1 时初始化所有的 UCSI 寄存器（包括 UCxCTL1）
3. 配置端口
4. 软件复位 UCSWRST
5. 通过 UCRXIE 或 UCTXIE 使能中断（可选）

- **字符格式**

在 SPI 模式下的 USCI 模块支持由 UC7BIT 位来选择 7~8 位字符长度。在 7 位数据模式下，只能选择 LSB；UCMSB 位控制着数据发送的方向且选择低位在前还是高位在前，缺省情况下是低位在前。

- **主模式**

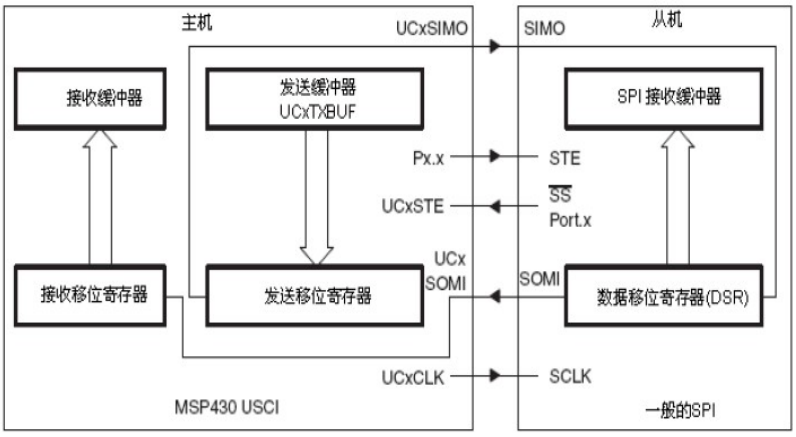


图3.5.2 主机与外部从机结构

如上图所示 MSP430 的 USCI 作为主机，当数据移动到数据发送缓冲区时，USCI 开始数据发送。当 TX 移位寄存器为空的时候，缓冲区的数据会移动到 TX 移位寄存器，开始在 UCxSIMO 口进行数据发送，并且由 UCMSB 位的设置决定高位在前还是低位在前。在相反的时钟边沿，在 UCxSOMI 端的数据被移位到数据接收寄存器。当一个字节的数据接收完成，被接收的数据就会从 RX 寄存器被移动到数据接收缓冲区 UCxRXBUF，而且接收中断标志位 UCRXIFG 被置 1。

发送中断标志位 UCTXIFG=1 意味着数据已经从发送缓冲寄存器被完整移动到发送移位寄存器，此时发送缓冲区已经准备好了发送一组新的数据，但并不意味着数据的发送和接受工作已经完成了。在主模式下，由于数据的接收和发送是并行工作的，如需要接收数据到 USCI，必须向发送缓冲区写入数据。

- 从模式

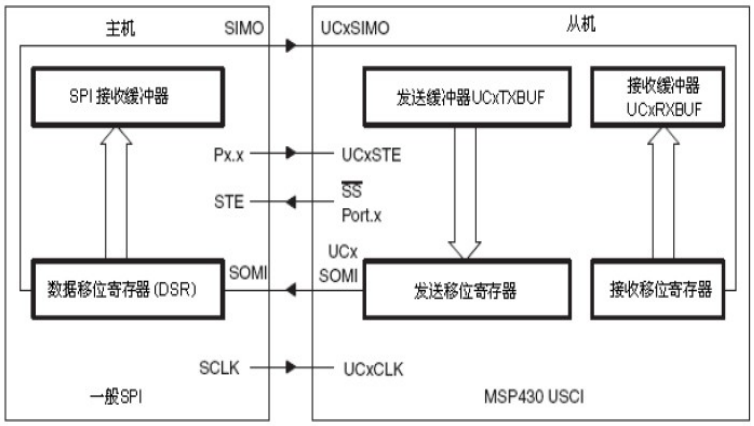


图3.5.3 从机与外部主机结构

如上图所示 MSP430 的 USCI 作为从机。UCxCLK 用于 SPI 的时钟输入而且必须有外部主机供电，数据传输的速率由这一时钟决定。待发数据应当在 UCxCLK 的起始边沿之前被写入并送至发送移位寄存器，并且会从 UCxSOMI 端口上发送出去。在时钟的另一边沿，UCxSIMO 端的数据会被移动到接收移位寄存器里，并且在预设位数的数据被接收完成后送入接收缓冲区 UCxRXBUF。当数据由接收移位寄存器移动到接收缓冲区时，接收的中断标志位被置 1，表明了数据已被接收。当之前传送的数据在新数据到来之前没有被接收缓冲区读到时会出现溢出错误，同时溢出错误标志位被置 1。

- **串行时钟控制**

UCxCLK 由 SPI 总线上的主机提供，当 UCMST=1 时，位时钟由 UCxCLK 引脚上的 USCI 位时钟产生器提供，并且由 UCSSELx 位来进行时钟选择；当 UXMST=0 时，USCI 时钟由主机的 UCxCLK 引脚提供，此时不使用位时钟产生器，并且 UCSSELx 位并不起作用。SPI 的数据接收器和发送器并行工作，使用同一个时钟源。位于位速率控制寄存器 UCxxBR1 和 UCxxBR0 中的 16 位值的 UCBRx 是 USCI 时钟源的分频数。主模式下可生成的最高频率的位时钟是 BRCLK。在 SPI 模式中不使用调制，同时 SPI 模式时的 USCI-A 模块下 UCAxMCTL 应该被清零。

- **SPI 中断**

每个 USCI 模块只有一个由发送和接收共享的中断向量，但 USCI-Ax 和 USCI-Bx 不共享同一个中断向量。发送中断标志位 UCTXIFG 是由发送器置 1 的，以便用来指示发送缓冲区 UCxTXBUF 做好接收下个字符的准备；此时如果 UCTXIE 和 GIE 也被置”1“，那么一个中断到来的时候就会产生一个中断请求。当一个字符被写入发送数据缓冲区时 UCTXIFG 会被自动复位。当 PUC 或者 UCSWRST=1 时会置位 UCTXIFG 并复位 UCTXIE。每次当接收一个字符并把字符装载到数据接收缓冲区时接收数据的中断标志位 UCRXIFG 就会被置 1，同时若 UCRXIE 和 GIE 被置 1 则也会有一个中断请求产生。UCRXIFG 和 UCRXIE 也会由 PUC 或者 UCSWRST=1 复位，当读取接收数据缓冲区时接收中断标志位也会自动复位。中断向量寄存器 UCxIV 可以被用来判断当前产生的是接收中断还是发送中断。

3.5.1 Lab3-5-1 SD 卡实验

3.5.1.1 实验介绍

本实验中需用到以下电路模块：USB 接口模块、SD 卡接口模块。单片机通过 USB 与主机通信，并通过 SPI 控制 SD 卡的读写。

3.5.1.2 实验目的

- 学习 SD 卡接口硬件电路原理。
- 学习 SD 卡读写程序资源。
- 学习单片机读取 SD 卡信息的操作及编程思想。
- 学习 USB 与 PC 机的通信操作及编程思想。

3.5.1.3 实验原理

SD 读卡器模块连接在 USCI_B1 接口上。Micro SD 卡（亦称 TF 卡）是一种极细小的快闪存储器卡，其格式源自 SanDisk 公司创造，主要应用于移动电话，但因它的体积微小和储存容量的不断提高，已经使用于 GPS 设备、便携式音乐播放器和一些快闪存储器盘中。它的体积为 15×11×1mm，差不多相等于手指尖的大小，是目前最细小的记忆卡。它也能通过 SD 转接卡来接驳于 SD 卡插槽中使用。目前 Micro SD 卡提供 128MB、256MB、512MB、1G、2G、4G、8G、16G、32G 和 64G 的容量。

如图 3.5.4 所示，TF 卡插上卡套后可作为 SD 卡使用，TF 卡有 8 个引脚，SD 卡是 9 个。TF 卡仅比 SD 卡少一个 VSS（也就是 GND），功能上没有区别。操作 SD 卡有两种模式，SD 卡模式和 SPI 模式，单片机读写 SD 卡一般都是用 SPI 模式，本文也仅介绍 SPI 控制方式。

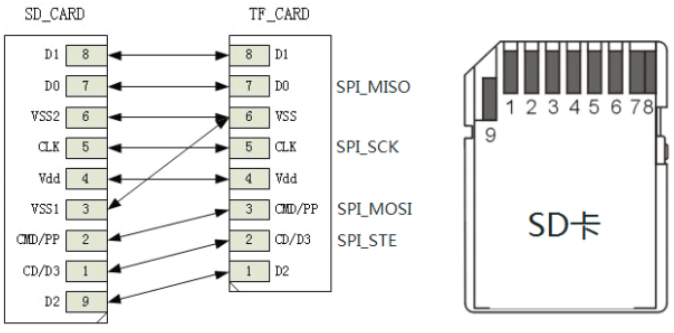


图3.5.4 SD 卡与 TF 卡管教对应

我们生活中用的各种存储器，如 U 盘和各种数据卡（SD 卡只是数据卡的一种）都是基于闪存（Flash）原理，而且是闪存中的 NAND 类型（详见存储器章节）。NAND 型 Flash 的好处就是成本低。NAND 的缺点是读写均需要以 512 字节（1 扇区）为单位来进行。所以，操作 SD 卡具有以下特点：

（1） 每次读写前，有非常复杂的初始化操作。

（2） 每次读写都必须以 512 字节的一扇区为单位来进行。

（3） 扇区操作时，很多单片机的全部 RAM 都不到 512 字节，这时，可以存前半部分数据先做处理，舍弃后半部分（空读不存）。下次再存后半部分数据，空读前半部分。当然，类似原理可以把 1 扇区数据分成更多部分多次处理。

（4） SD 卡中任何文件都是从一个扇区的头开始存，占 N 个扇区，第 N 的扇区未用的字节将无法被利用。所以读写 SD，实际是以扇区为“地址”操作的。

基于以上特点，我们仅讲解以下知识点：

（1） SD 卡（TF 卡）使用 SPI 控制的硬件连接。

（2） 如何初始化 SD 卡。

（3） 如何读取一个扇区数据。

（4） 如何写入一个扇区数据。

如下图 3.5.5 所示 TF 卡和单片机的 SPI 接口相连，根据程序中的 USB MSC(USB mass storage device class)模块为 PC 机提供了一个 U 盘的操作接口，使得 PC 机可以像操作 U 盘一样对 SD 卡进行操作。这一接口对 SD 卡的操作是通过程序中的 SD 卡模块的盘级接口部分实现的。

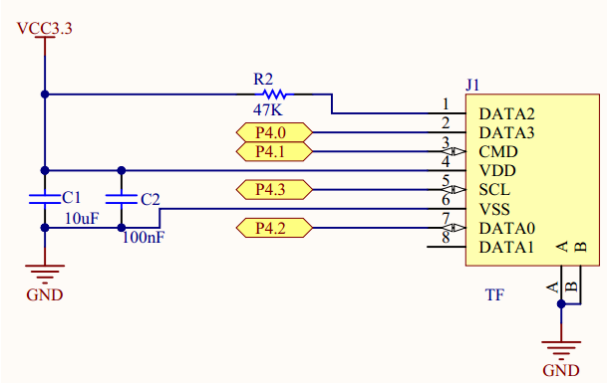


图3.5.5 SD 卡模块电路原理图

SD 卡的命令字：如图 3.5.6 所示，SD 卡命令是有 6 个字节。

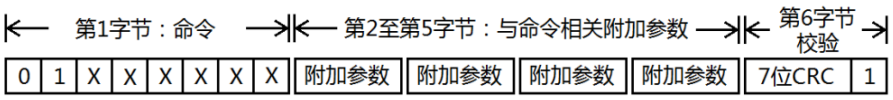


图3.5.6 SD 卡命令字结构

（1）第 1 字节是命令（CMD0、CMD1 等）其中头两位一定是 01，所以，第 1 字节数据为“命令的代码+0x40”。例如：CMD17，命令字的第一字节即为 0x11+0x40=0x51。

（2）第 2 至第 5 的 4 个字节，是与命令有关的附加参数，可能是 32 位地址，也可能为空。

（3）第 6 字节的高 7 位是 CRC 校验值，最后一位一定是 1。

如图 3.5.7 所示为 SD 卡发送命令的时序图，图 3.5.7 的时序中，凡是发送 0xFF 的情形均是空发 CLK 的意思。我们根据时序，编写“万能”代码。当新遇到一个编程任务的时候，我们可以先只编写顶层应用层，需要什么底层函数只管凭空捏造，假设已经存在就行

了。最后再写底层驱动，这样出来的底层就是万能的底层了。

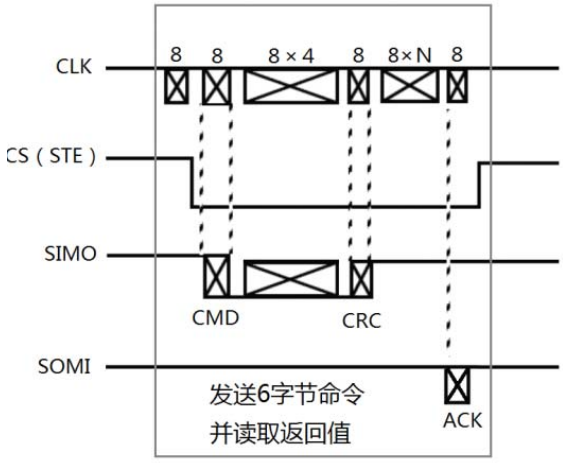


图3.5.7 SD 卡发送命令时序图

在发送完命令时，需要等 N 个字节的空时钟才会有应答。

SD 卡的复位：

由于 SD 上电后默认就是 SD 模式，所以需要给复位命令然后再设置为 SPI 模式。复位需要给 SD 卡发送 CMD0（0x40）命令。

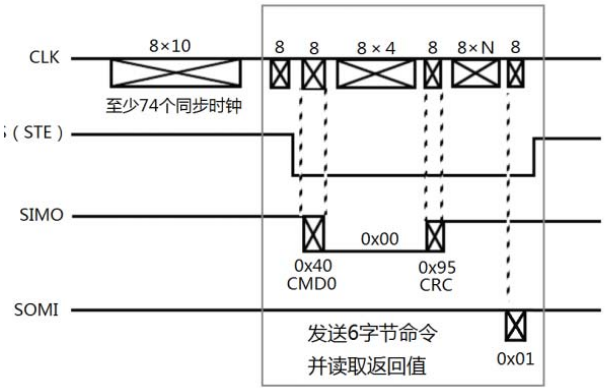


图3.5.8 SD 卡的复位时序

发送 0xFF 十次（实际上需要至少 74 个同步时钟，也就是 74 位），然后 CS 拉高，然后发送 0xFF ，然后 CS 使能，然后发送 0x40，然后发送 0x00 四次，然后再发送 CRC 校验码，然后再发送若干字节的 0xFF 作为同步时钟直到收到 0x01 为止，最后 把 CS 拉高。

SD 卡复位代码有以下注意事项：按 SD 卡规范，命令往往要发多次才能有应答，所以大量地方需要超时判断。

SD 卡初始化为 SPI 模式：

在完成 SD 卡复位以后，就可以写 CMD1 命令初始化为 SPI 方式了。如图 3.5.9 所示，和复位模式差不多格式，只不过没有了 74 个同步时钟，发送数据是 0x41，CRC 校验位随便写，SD 相应字节为 0x00。由于只有 SD 模式才会 CRC 校验，所以图 3.5.9 中的 CRC 校验位置随便写什么都行。

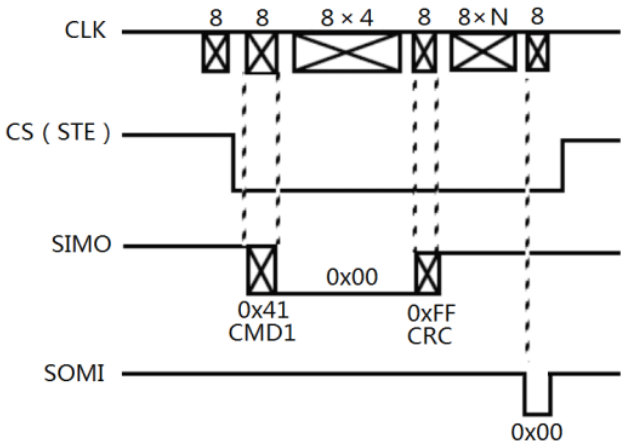


图3.5.9 SD 卡设置为 SPI 模式

CS 拉高，然后发送 0xFF，然后 CS 使能，然后发送 0x41，然后发送 0x00 四次，然后再随便发送点什么作为 CRC 校验码（比如 0xFF），然后再发送若干字节的 0xFF 作为同步时钟直到收到 0x00 为止，最后 CS 拉高。

SD 卡初始化时要注意，CLK 不能高于 500KHz。

SD 卡的读操作：

SD 卡只能按 512 字节一个扇区为单位来读数据，还可以用 CMD16 命令更改为每次读写 N 个扇区。我们这里采用默认的 1 个扇区操作。如图 3.5.10 所示：

CS 拉高，然后发送 0xFF，然后 CS 使能，然后发送 CMD17（0x51），然后将 32 位扇区地址分 4 字节发送（高位字节在前），然后再随便发送点什么作为 CRC 校验码（比如 0xFF），然后再发送若干字节的 0xFF 作为同步时钟直到收到 0x00 为止，然后再发送若干字节的 0xFF 作为同步时钟直到收到 0xFE，开始接收 512 字节数据接收 2 字节 CRC 校验数据，最后 CS 拉高。

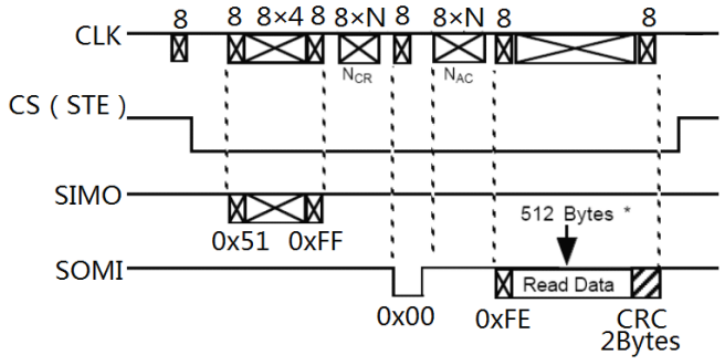


图3.5.10 SD 卡读一扇区

SD 卡写操作：

如图 11.10 所示，SD 卡最小写入单位也为 1 个扇区。

CS 拉高，然后发送 0xFF ，然后 CS 使能，然后发送 CMD24（0x58），然后将 32 位扇区地址分 4 字节发送（高位字节在前），然后再随便发送点什么作为 CRC 校验码（比如 0xFF），然后再发送若干字节的 0xFF 作为同步时钟直到收到 0x00，然后再发送若干字节的 0xFF 作为同步时钟，然后发送 0xFE 作为数据头，然后开始发送 512 字节数据，然后随便发送 2 字节 CRC 校验数据，然后接收到 SD 应答 xxx0_0101b，然后发送若干字节 0xFF 作为同步时钟，然后等待 SOMI 变高，最后 CS 拉高。

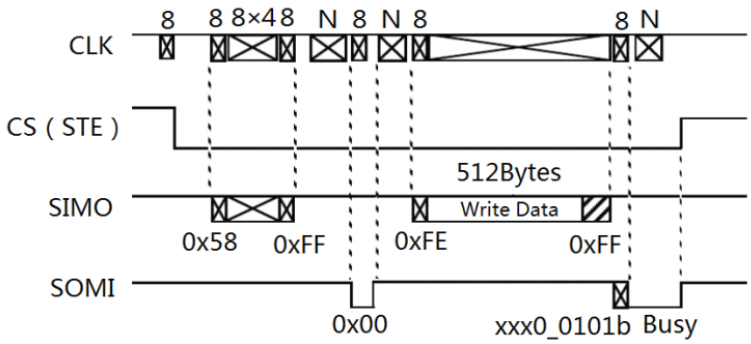


图3.5.11 SD 卡写一扇区

3.5.1.4 程序分析

- **编程思路**

  SD 卡工作有两种模式，既 SD 模式和 SPI 模式，通过拉低 CS 线并发送 REST 命令进入 SPI 模式；然后再初始化 SD 卡和 USB 接口。通过 FAT 文件系统对 SD 进行访问。

- **程序流程图**

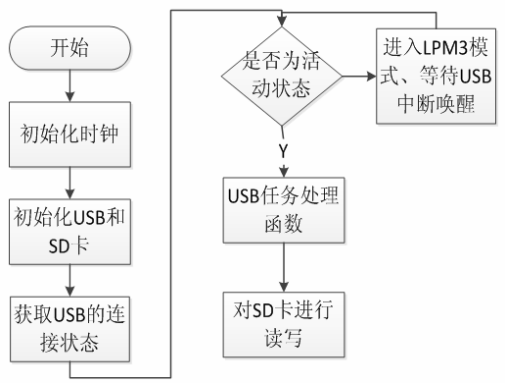


图3.5.12 程序流程图

- **关键代码分析**

实验 SD 卡读写 代码

```
void main (void)
{
    WDTCTL = WDTPW + WDTHOLD;                          //关看门狗
    Init_Ports();                                      //IO 初始化
    SetVCore(3);
    Init_Clock();                                      //时钟初始化
    disk_initialize(0);                                //SD 卡初始化.
    USB_init();                                        //USB 初始化
    USB_setEnabledEvents(kUSB_allUsbEvents);           //使能 USB 事件
    RWbuf_info = USBMSC_fetchInfoStruct();
    if (detectCard()){
     //SD 卡插入检测
        mediaInfo.mediaPresent = kUSBMSC_MEDIA_PRESENT;
    }
```

```
        else {
            mediaInfo.mediaPresent = kUSBMSC_MEDIA_NOT_PRESENT;
        }
        mediaInfo.mediaChanged = 0x00;
        mediaInfo.writeProtected = 0x00;
        disk_ioctl(0,GET_SECTOR_COUNT,&mediaInfo.lastBlockLba);
        mediaInfo.bytesPerBlock = BYTES_PER_BLOCK;                    //块尺寸
        USBMSC_updateMediaInfo(0, &mediaInfo);

        USBMSC_registerBufInfo(0, &RWbuf[0], NULL, sizeof(RWbuf));   //数据交换和内存管理
        TA0CCTL0 = CCIE;                                              //使能 TA0 中断
        TA0CCR0 = 32768;                                              //TA0 时钟选择
        TA0CTL = TASSEL_1 + MC_1 + TACLR;
//ACLK = 32kHz, 升模式,清除 TAR
        if (USB_connectionInfo() & kUSB_vbusPresent){        //如果 USB 已经连接但是 USB 总线
不能握手
            if (USB_enable() == kUSB_succeed){               /重新使能 USB
                USB_reset();                                 //USB 复位
                USB_connect();                                        //重新连接 USB
            }
        }
        __enable_interrupt();                                         //开中断
        while (1)
        {
            switch (USB_connectionState())                            //USB 连接状态
            {
                case ST_USB_DISCONNECTED:                             //没有连接
                    __bis_SR_register(LPM3_bits + GIE);               //低功耗 3
                    _NOP();
                    break;

                case ST_USB_CONNECTED_NO_ENUM:
                    break;
                case ST_ENUM_ACTIVE:
                    __disable_interrupt();
                    if ((USBMSC_poll() == kUSBMSC_okToSleep) && (!bDetectCard)){
                        __bis_SR_register(LPM0_bits + GIE);
                    }
                    __enable_interrupt();
                    while (RWbuf_info->operation == kUSBMSC_READ)     //读数据
                    {

                        DRESULT dresult = disk_read(0,                    //物理驱动号 0
                            RWbuf_info->bufferAddr,                   //指针指向使用的地址
                            RWbuf_info->lba,                          //第一个 LBA 操作
                            RWbuf_info->lbCount);                     //操作快号
                        switch (dresult)
                        {
                            case RES_OK:
                                RWbuf_info->returnCode = kUSBMSC_RWSuccess;
                                break;
                            case RES_ERROR:                           //文件被转移
                                if (!checkInsertionRemoval()){        //核对 SD 卡状态
                                    RWbuf_info->returnCode =
                                        kUSBMSC_RWMedNotPresent;
                                }
                                break;
                            case RES_NOTRDY:
```

```
                    RWbuf_info->returnCode = kUSBMSC_RWNotReady;
                    break;
                case RES_PARERR:
                    RWbuf_info->returnCode = kUSBMSC_RWLbaOutOfRange;
                    break;
            }
            USBMSC_bufferProcessed();
        }
        while (RWbuf_info->operation == kUSBMSC_WRITE)
        {
            DRESULT dresult = disk_write(0,                    //同上
                RWbuf_info->bufferAddr,                        //同上
                RWbuf_info->lba,                               //同上
                RWbuf_info->lbCount);                          //同上
            switch (dresult)
            {
                case RES_OK:
                    RWbuf_info->returnCode = kUSBMSC_RWSuccess;
                    break;
                case RES_ERROR:
                    if (!checkInsertionRemoval()){
                        RWbuf_info->returnCode =
                            kUSBMSC_RWMedNotPresent;
                    }
                    break;
                case RES_NOTRDY:
                    RWbuf_info->returnCode = kUSBMSC_RWNotReady;
                    break;
                case RES_PARERR:
                    RWbuf_info->returnCode = kUSBMSC_RWLbaOutOfRange;
                    break;
                default:
                    RWbuf_info->returnCode = kUSBMSC_RWNotReady;
                    break;
            }
            USBMSC_bufferProcessed();
        }
        if (bDetectCard){                                      //检查中断标志
            checkInsertionRemoval();                           //核对卡状态
            bDetectCard = 0x00;                               /清除中断标志
        }
        break;
    case ST_ENUM_SUSPENDED:
        __bis_SR_register(LPM3_bits + GIE);
        break;

    case ST_ENUM_IN_PROGRESS:
        break;

    case ST_NOENUM_SUSPENDED:
        __bis_SR_register(LPM3_bits + GIE);
        break;

    case ST_ERROR:
        _NOP();
        break;

    default:;
```

```
            }
        }
    }
```

3.5.1.5 实验步骤与现象

- **实验步骤**
  1. 插入 SD 卡（4G 以下，FAT16 格式；4G 以上、FAT32 及 NTFS 格式的卡片无法识别）。
  2. 将主板右侧的 USB 接口（J1）与 PC 机相连。
  3. 在 PC 机上将发现 PC 机已连接到一个 U 盘，这样我们就可以像 U 盘一样操作 TF 卡。
- **实验现象**

  PC 机端会出现一个可移动磁盘盘符，可以按照 U 盘方式读写。

3.5.1.6 实验思考

FAT32 或 NTFS 分区格式的 SD 卡为何无法识别？

3.5.2 Lab3-5-2 电子纸（电子墨水屏）显示实验

F5529 口袋板上没有采用常见的点阵液晶、段式液晶、TFT 等屏幕，而是采用了电子纸（电子墨水屏）这种革新的信息显示设备，它与传统的显示屏幕有很大不同，下面略作介绍。

1. 电子纸的分类：

如同其他显示技术一样，电子纸主要也分为以下以下几类：

- **电泳显示技术（EPD）**：电泳显示技术系将黑、白两色的带电颗粒封装于微胞化液滴结构中，由外加电场控制不同电荷黑白颗粒的升降移动，以呈现出黑白单色的显示效果。由于 EPD 技术可呈现出高反射率、高对比的黑白显示效果，因此十分适合做电子纸，F5529 口袋板上使用的电子纸屏幕采用的就是这种技术。

- **电子粉流体显示技术(QR-LPD)**
- **胆固醇液晶显示技术(Ch-LCD)**
- **双稳态向列液晶显示技术（Bi TNLCD)**

这几种电子纸技术我们没有使用到，这里也就不展开介绍了，大家如果有兴趣可以问问度娘，呵呵。

2. 电泳型电子纸技术详解：

- **原理**

电泳型电子纸技术（以下简称电子纸）是一种“微胶囊电泳显示”技术。其基本原理是悬浮在液体中的带电纳米粒子受到电场作用而产生迁移。电子纸是一种薄膜状的材料，它是由成千上万个微小的胶囊状颗粒（称为微胶囊）涂在一种塑料基材上而制成的。微胶囊是电子纸的基本单元，它里面包含了两种不同颜色的纳米粒子。

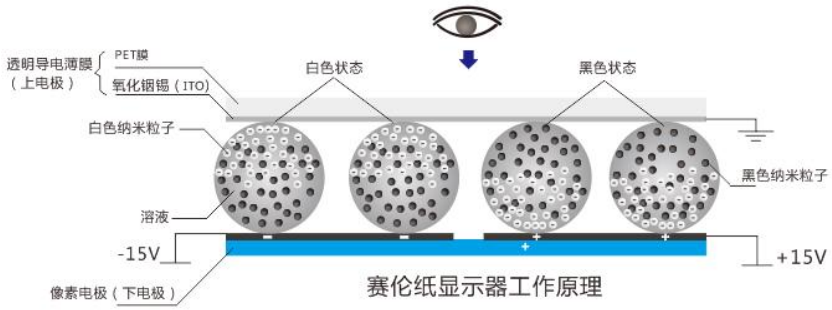


图 3.5.13　电泳型电子纸工作原理

- **结构**
  - **电子纸膜片：**这是电子纸显示模组的核心材料，负责显示人眼实际看到的图案。
  - **底板：**作为电子纸显示屏的像素电极（下电极），用于控制电子纸每个像素的黑白变化。底板有多种类型可选，包括 PCB、FPC、TFT 玻璃、PET 等，实际应用时可根据具体需求选择不同的底板。电子纸膜片可通过层压的方式贴合在底板上。
  - **驱动芯片：**可根据控制指令和信号产生相应的逻辑电平和时序，用于控制底板每个像素（或段码）的工作时序和状态，并使电子纸能够显示所需图案。

- **透明保护膜：**一种高分子塑料薄膜，具有很强的防水汽透过性。用层压机将其紧密贴合在电子纸膜片与底板上面，可有效防止水汽侵入赛伦纸膜片，避免电子纸因受潮而损坏。
- 封边胶：一种特殊的化学胶水，将其均匀涂在透明保护膜的四周边缘处，起到隔离水汽的作用。可避免水汽从透明保护膜四周渗入进去而对电子纸膜片造成损坏。

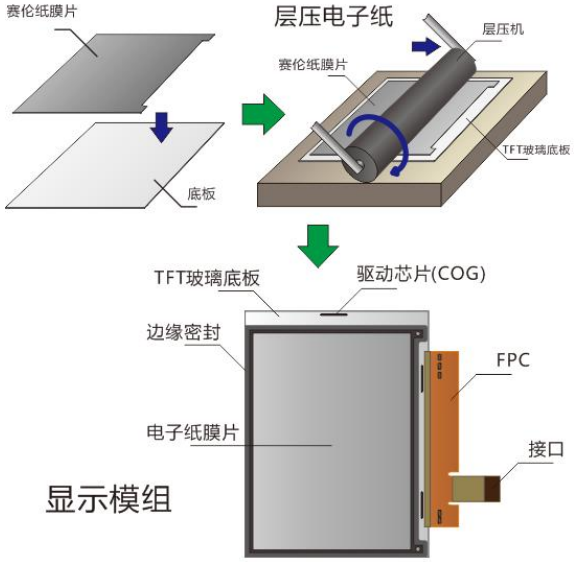


图 3.5.14　电子纸结构

3. 优势与不足：

电子纸作为一种革新的信息显示的新方法和设备，区别于其他显示技术的具有以下几个突出优势：

- **易阅读性：**

  电子纸是靠反射环境光来显示图案的，它具有纸张印刷般的效果。与传统透射式液晶显示屏（TFT 等）相比，即使是在强烈的阳光底下，依然清晰可视；可视角度几乎达到了 180°。另外电子纸显示柔和、无闪烁，因此“电纸书”都是采用电子纸做屏幕。

- **超级省电：**

  电子纸只有在刷新屏幕的时候才会消耗电能，而且断电后能保持断电前最后一帧图片的显示，这是其他屏幕都做不到的，再加上不需要背光，因此电子纸适合作为电子标签使用。

- **轻薄灵活：**

  与其他显示屏幕相比，电子纸不管是在厚度还是重量上都有明显的优势，最薄可以做到 0.1mm，和纸张的厚度差不多。如果采用塑料薄膜作为基材，还具有可弯曲的特点。

  以上说的都是电子纸的优势，下面说说电子纸目前的不足，这些技术不足，以及成本上的居高不下今后随着技术发展会逐渐得到解决的。

- **刷新速度慢：**尤其是电泳型电子纸刷新速度是比较慢的，目前无法做动画或动态视频显示。
- **色彩还原不好：**目前电子纸色彩还原还不如 TFT 等屏幕真实、鲜艳，无法很好的显示彩色照片，再加上成本原因，目前电子纸以点色灰阶型为主。

3.5.2.1 实验电路介绍

F5529 口袋板使用了一片分辨率为 250×122 的 2.1 英寸的电子纸，SPI 接口。

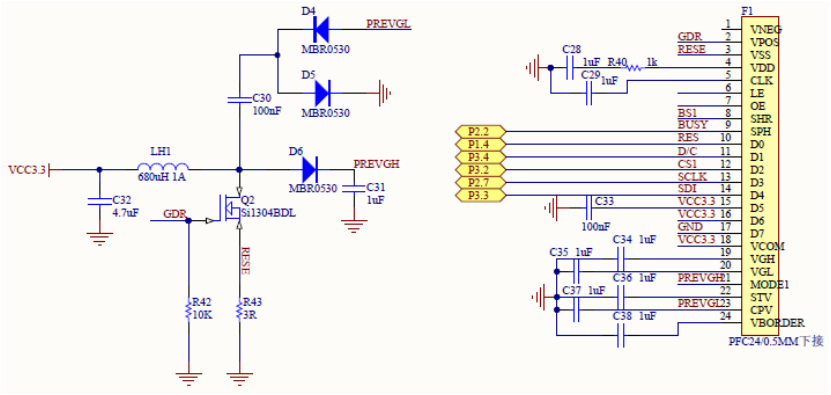


图3.5.15　电子纸屏幕接口电路

3.5.2.2 程序分析

- **编程思路**

在写程序操作 TFT 屏幕显示之前首先要配置好系统时钟，即为系统时钟配置合适的晶振，然后初始化 TFT 屏幕，初始化 TFT 过程中包括了配置 MSP430F5529 的 SPI 接口以及通过 SPI 接口对 TFT 屏幕进行读写操作初始化 TFT 屏幕。初始化 MSP430F5529 的 SPI 时序接口需要配置的寄存器有 UCA0CTL1、UCA0CTL0、UCA0BRW、P3SEL 等，寄存器详细功能可查阅文档。配置完 SPI 时序后还要通过 SPI 对 TFT 屏幕的读写操作初始化 TFT 屏幕后才算完成整个初始化过程；初始化完成后在通过 SPI 对 TFT 屏幕的读写使 TFT 屏幕上显示出图片和文字来。初始化和显示 TFT 屏幕的操作可通过调用驱动来完成，相关的驱动函数及函数功能如下：

- void display(unsigned char *str, //字符串
  unsigned int xsize, //x 方向位置
  unsigned int ysize, //y 方向位置
  unsigned int font, //字体 0,1,2
  unsigned int size, //字号 0,1
  unsigned int size, //字号 0,1
  unsigned int size,) //字号 0,1

- **实验结果**

图3.5.16　屏幕显示

3.6 通用串行通信接口----UART 模式

串行通信是与并行通信相对应的。并行通信优点是快速，但是用的 IO 多，信号线多，这是硬伤。而我们现在有办法把串行通信的速度提升到非常高的水平，例如， 现在的计算机的硬盘、光驱等高速设备与主板的连接都是用 sata 串口而不再使用并口线了。多数场合串行通信都比并行通信有用，当然，学习串行通信也是单片机学习的难点。

从最基本的原理来讲， 串并信号转换的核心单元就是数字电子技术中的移位寄存器 。 串行数据， 移位寄存后，统一送出，就完成串入转并出。并行数据经移位寄存器依次送出，也就完成了并入串出的原理。

但除了移位寄存这个简单原理外，串行通信的具体实现是大有学问的。首先，我们梳理清楚一些基本概念 ，然后分析为什么要这么干。

串行通信分为同步和异步：

同步通信是一种连续串行传送数据的通信方式，一次通信只传送一帧信息。这里的信息帧与异步通信中的字符帧不同，通常含有若干个数据字符。

它们均由同步字符、数据字符和校验字符（CRC）组成。其中同步字符位于帧开头，用于确认数据字符的开始。数据字符在同步字符之后，个数没有限制，由所需传输的数据块长度来决定；校验字符有 1 到 2 个，用于接收端对接收到的字符序列进行正确性的校验。同步通信的缺点是要求发送时钟和接收时钟保持严格的同步。

异步通信中，在异步通行中有两个比较重要的指标：字符帧格式和 波特率 。数据通常以字符或者字节为单位组成字符帧传送。字符帧由发送端逐帧发送，通过传输线被接收设备逐帧接收。发送端和接收端可以由各自的时钟来控制数据的发送和接收，这两个时钟源彼此独立，互不同步。

接收端检测到传输线上发送过来的低电平逻辑"0"（即字符帧起始位）时，确定发送端已开始发送数据，每当接收端收到字符帧中的停止位时，就知道一帧字符已经发送完毕。

串行通信的特点

1、节省传输线，这是显而易见的。尤其是在远程通信时，此特点尤为重要。这也是串行通信的主要优点。

2、数据传送效率低。与并行通信比，这也这是显而易见的。这也是串行通信的主要缺点。

在通信中，单工、全双工、半双工这几个词我们经常能看到，什么意思呢？

（1）能同时收发就是全双工，比如打电话，双方可以“对吼” 。

（2）数据能收能发，但要分时进行就是半双工，比如对讲机。一方按下按键，只能说话（发送数据） ，另一方只能收听。当一个人说完必须加一句“over” ，然后就得松开按键（接收数据），对方听到“over”知道对方讲完了，这时才能按下按键说话（发送数据）。

（3）只能单向通信就是单工了，比如广播，播音员播音时（发送数据） ，听众永远只能是听众。

通信协议：简单说，就是通信的双方要约定 1、0 序列代表什么含义，就像可以用“三长两短”代表危险一样。如果我们是自己使用两片单片机进行通信，那么我们爱怎么规定数据流的含义就怎么规定，谁也管不着。但是，自定义的通讯协议有两个缺点。

（1） 自编通讯协议的效率不高，会有 bug。成熟的通讯协议都是人类智慧的结晶。

（2） 不是通用协议的话，不能与“别人”进行通信。协议这个东西，和霸王条款差不多。

（3） 成熟的通信协议有相应的硬件支持，可以在通信时减轻 CPU 的负担，增强性

能。

UART（ Universal Asynchronous Receiver/Transmitter） 是通用异步收发器的缩写，一般称为串口。 由于不需要时钟线，且为全双工工作，所以 UART 有两根数据线，发送 Tx 和接收 Rx。

UART 通信协议的数据帧格式如图 3.6.1 所示，首先是一个起始位，然后是 7-8 位可选的数据位，0-1 位可选的地址判别位、0-1 位可选的奇偶判别位、1-2 位可选的高电平停止位。

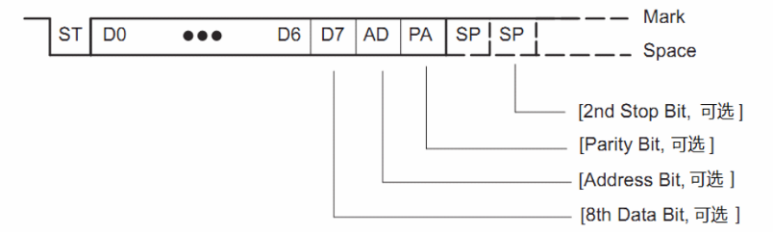


图3.6.1 UART 数据帧格式

（1） 起始位 ST 是高电平到低电平的下降沿触发的。也就是每传输 7-8 位数据都要消耗一个起始位，这是非常肉疼但又必须的。为什么要起始位呢？这是因为异步通信时，设备双方的时钟一定不相等（差别百万分之一也不行），为了消除累计误差，必须引入起始位。

（2）数据位 7-8 位。为什么会有 7 或 8 位的数据长度选择呢？我们知道 1 个字节是 8 位，按理说应该都是 8 位传输才对。其实不是所有类型数据都是 8 位的，比如 ASCII 码，7 位就够了。

（3） 地址位 Address Bit，0 表示前面的 7 或 8 位是数据，1 表示是地址，这在 1 主多从通信中有用。

（4） 奇偶校验位 Parity Bit，在数据的发送设备上预先计算本次发送数据中 1 的个数是奇数还是偶数，并在奇偶校验位标识出来。 数据接收端设备收完数据后，也算一下数据中 1 的个数，与奇偶校验位作对比，如果对不上则传输一定有错误（但即使奇偶校验位对的上，也不代表没错误。

（5）停止位 SP，数据传输完后，需至少维持高电平 1-2 位才能重新开始下一帧的传输。当然，歇个 10 天半月都是高电平再开始下一帧也是可以的，也就是说，1-2 位是最少需要的“休息”位。为什么会有 2 位的要求呢？“休息位”实际上是用来校准异步通信时钟误差的，休息位越长，对两个时钟差值容忍程度越高。 根据设备实际数据传输速率和时钟匹配程度的不同， 需要最少 1 位或者 2 位的停止位。

UART 可以地址形式多机通信。如图 3.6.2 是地址位模式的多机通信，在每帧中插入了一个地址位 Address Bit。当地址位是 1 时，说明本帧是地址帧，于是每个从机对比地址是否与自己的一样。如果一样，则意味着接下来的数据帧是给自己的。如果不是，则需等下一个地址帧的到来，这接下来的数据帧与自己无关，当然也不许发数据给主机。

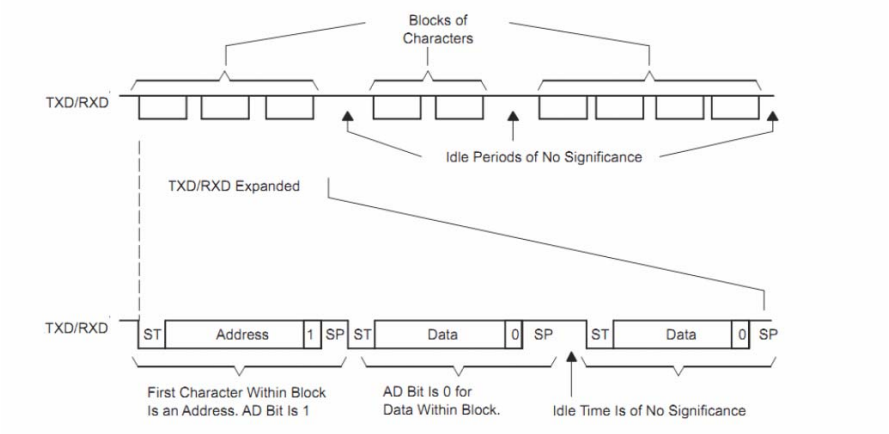


图 3.6.2　UART 地址位模式下多机通信

本节介绍通用串行通信接口（USCI）模块的另一种工作模式 UART 模式，只有 USCI_Ax 模块支持 UART 模式，因此这里介绍 USCI_Ax 模块。在 UART 模式中，USCI_Ax 模块是通过两个外部引脚连接 MSP430 到外部系统，分别是 UCAXRXD 和 UCAXTXD。当 UCSYNC 位被清零时 UART 模式被选择。

USCI_Ax 模块特点

该模块支持多种串行通信的模式，其中包括 UART 模式、带有脉冲整形的 IrDA 通信、带有自动波特率检测的 LIN 通信、SPI 模式。多种模式能满足多种不同接口的微机通信，这里主要介绍其中的最简单一种 UART 通信模式。

UART 模式下的 USCI_Ax 模块结构

其结构框图如下图所示，可分为接收、发送、红外接口和波特率发生器四个主要部分。

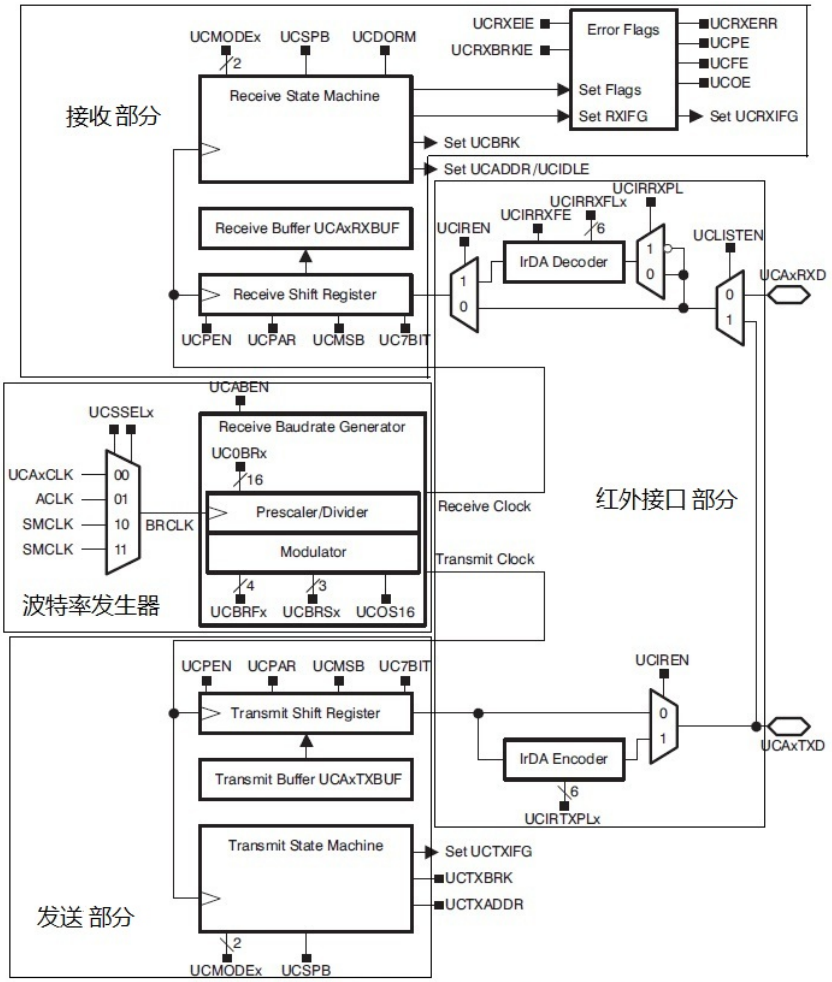


Figure 34-1. USCI_Ax Block Diagram – UART Mode (UCSYNC = 0)

图3.6.3　UART模式下USCI_Ax结构框图

- **接收部分：**接收部分将从UART外部接收串行数据，将所接收的数据放到UCAxRXBUF寄存器中存放起来，以便处理器读取。接收的格式由寄存器UCAxCTL0各个位决定，接收过程由接收状态机控制。同时接收器状态机还要对数据位的溢出出错，奇偶检验出错，帧出错,BREAK等进行检验，并根据检验结果设置状态寄存器UCAxSTAT中相应的状态位。

- **发送部分：**发送部分将从处理器接收到的数据（存放在UCAxTXBUF寄存器中），按规定的格式加上起始位，奇偶检验位和停止位后串行输出。发送数据格式由寄存器UCAxCTL0各个位来决定，发送过程由发送状态机来控制。

- **波特率发生器：**波特率发生器部分提供UART进行通信时所需时钟，还需对外部接收时钟进行同步处理,通过控制UCAxCTL1寄存器中的UCSSELx来选择输入时钟。UCAxBR0和UCAxBR1分别为控制波特率低八位和高八位数据的寄存器，还可以通过UCAxABCTL寄存器中的UCABDEN来使能波特率自动设置功能。

- **红外接口部分：**主要由编码部分和解码部分完成，编码部分完成RS232到IRDA之间的转换，解码部分完成IRDA到RS232之间的转换。UART 在使用中，通过控制

寄存器UCAxIRTCTL的UCIREN来决定采用哪种通信协议，UCIREN置1时，使能编码与解码器。

USCI_Ax 模块在 UART 模式工作方式配置相关寄存器如下图所示

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCAxCTL1	USCI_Ax Control 1	Read/write	Byte	01h	Section 34.4.2
01h	UCAxCTL0	USCI_Ax Control 0	Read/write	Byte	00h	Section 34.4.1
06h	UCAxBRW	USCI_Ax Baud Rate Control Word	Read/write	Word	0000h	
06h	UCAxBR0	USCI_Ax Baud Rate Control 0	Read/write	Byte	00h	Section 34.4.3
07h	UCAxBR1	USCI_Ax Baud Rate Control 1	Read/write	Byte	00h	Section 34.4.4
08h	UCAxMCTL	USCI_Ax Modulation Control	Read/write	Byte	00h	Section 34.4.5
09h		Reserved - reads zero	Read	Byte	00h	
0Ah	UCAxSTAT	USCI_Ax Status	Read/write	Byte	00h	Section 34.4.6
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCAxRXBUF	USCI_Ax Receive Buffer	Read/write	Byte	00h	Section 34.4.7
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCAxTXBUF	USCI_Ax Transmit Buffer	Read/write	Byte	00h	Section 34.4.8
0Fh		Reserved - reads zero	Read	Byte	00h	
10h	UCAxABCTL	USCI_Ax Auto Baud Rate Control	Read/write	Byte	00h	Section 34.4.11

图3.6.4　相关配置寄存器

3.6.1 Lab3-6-1 RS232 串口实验

3.6.1.1 实验介绍

RS232 接口标准是在串口通信中最基础最简单的通信协议，现已在微机通信接口中被广泛采用。本实验将利用 MSP430F5529 的 USCI_Ax 模块进行 RS232 串口通信，只需将 RS232 接口的 RXD 和 TXD 进行短接即可实现本机的发送以及接收，并把发送及接收的内容在 TFT 屏幕上显示出来。

3.6.1.2 实验目的

- 了解 RS232 接口标准概况
- 了解 MSP430F5529 的 USCI_Ax 模块的 UART 模式的使用
- 实现 MSP430F5529 的串口通信

3.6.1.3 实验原理

- **RS232 简介**

RS232 的通信标准中在早期是以一个 25 针的接口来定义的，并在 PC 机或 XT 机型上广泛使用，但是在 AT 机以后的机型上，实际采用了 9 针的简化版本应用，现在的 RS232 通信均默认为 9 针接口，图 3.6.5 显示了 9 针通信接口的管脚定义，在实际的应用中 RS-232 接口在 9 针的基础上可以再进行一步简化，只用其中的 2、3、5 三个管脚进行通信。

在实际的应用中 RS-232 接口在 9 针的基础上可以再进行一步简化，只用其中的 2、3、5 三个管脚进行通信，在本次的例程中用的是 P4.4 和 P4.5，即 UCA1TXD/UCA1RXD 进行通信。

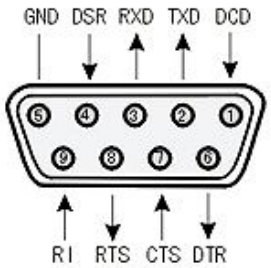


图3.6.5　RS232接口9针定义

- **MSP430F5529 串口模块的使用**

MSP430F5529 自带的 USCI_Ax 模块在软件上的通信协议由使用者来确定，章节介绍中以介绍过 USCI_Ax 模块支持的四种通信方式。在此处用到的是 UART 通信模式，即通用异步接收/发送模式，异步通讯的特性包括：

- 7 位或 8 位数据位，支持奇偶校验
- 独立的发送和接收移位寄存器
- 独立的发送接收缓存
- 可选择优先发送（接收）MSB 还是 LSB
- 空闲位多机模式和地址位多机模式
- 通过有效的起始位检测将 MSP430F5529 从低功耗中唤醒
- 状态标志检测错误或地址位
- 独立的接收和发送中断
- 可编程实现波特率调整

本实验中所用的 UART 配置为波特率 115200，8 位数据位，1 位停止位，无奇偶校验。

3.6.1.4 程序分析

- **编程思路**

本次实验使用了 USCI_Ax 模块的 UART 通信模式和定时器模块 TimerA 模块以及外围设备 TFT 屏幕，先将 UART 模式进行相应的配置，再进行初始化 TFT 屏幕，然后打开定时器，通过定时器中断方式来定时，定时 1s 通过 UART 发送出一个字符，在中断中接收数据，最后将发送和接收的数据在 TFT 屏幕上显示出来，因为这里要用到 TFT 屏幕，因此在编写代码之前先将 TFT 屏幕的驱动文件 dr_tft.h 添加到代码工程中去。

- **程序流程图**

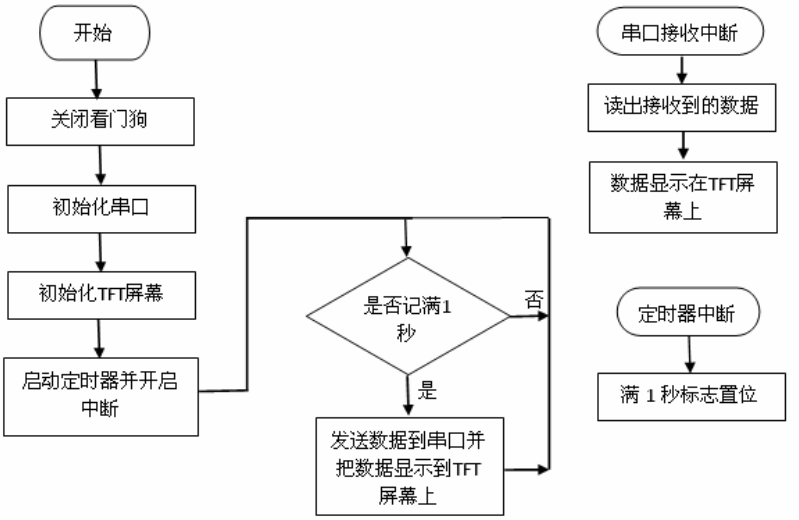


图 3.6.6　程序流程图

- **关键代码分析**

实验 RS232 通信 代码

主函数

```
int main(void)
{
    UART_RS232_Init();                    //初始化RS232接口
     TimerA_Init();                       //初始化定时器
     _EINT();                             //开启中断
```

```
    while(1)
    {
    if(flag0)                                   //flag0在定时器计时到达1s时被赋值为1
        {
        display("Send Data: ", 42, 42,0,0,1,0);          //TFT屏显示发送数据的标识
        display("Recv Data: ", 42, 60,0,0,1,0);          //TFT屏显示接收数据的标识
            flag0=0;                                     //flag0清零
            display(send_data, 130, 42,0,0,1,0);         //TFT屏上显示发送的字符
            UCA1TXBUF=send_data[0];                      //写一个字符到发送缓存发送
数据
            send_data[0]++;                              //字符加1
            if(send_data[0]>'9')                         //字符超过'9'则重新置'0'
                send_data[0]='0';
            P8OUT ^= BIT1;
        }
    }
```

USCI 中断服务函数

```
#pragma vector=USCI_A1_VECTOR                    //USCI中断服务函数
__interrupt void USCI_A1_ISR(void)
{
    switch(__even_in_range(UCA1IV,4))
    {
        case 2:                                                  //接收中断处理
        {
            if(UCTXIFG)                                  //等待完成接收
            {
            recv_data[0]=UCA1RXBUF;                          //数据读出
            tft_prints16(12,2,recv_data,WHITE,BLACK); //TFT屏幕上显示接收到的字符
            }
        }
        break;
        default:break;              //其他情况无操作
    }
}
```

定时器 TA 中断服务函数

```
#pragma vector = TIMER0_A0_VECTOR  //定时器 TA 中断服务函数
__interrupt void Timer_A (void)
{
    static unsigned int i=0;
    i++;
    if(i>=400)
    {
        i=0;
        flag0=1;                        //改变标识数据的值
    }
}
```

```
RS232 接口初始化函数
void UART_RS232_Init(void)    //RS232 接口初始化函数
{

        P4SEL |= BIT5+BIT4;                       // P4.4,5 使用外设功能  = UCA1 TXD/RXD
        UCA1CTL1 |= UCSWRST;                      //  复位 USCI
        UCA1CTL1 |= UCSSEL_2;                     //  设置 SMCLK 时钟，用于发生特定波
特率
        UCA1BR0 = 175;                            //  设置波特率，  1MHz  波特率= 115200
        UCA1BR1 = 0;
        UCA1MCTL |= UCBRS_1 + UCBRF_0;
        UCA1CTL1 &= ~UCSWRST;                     //  结束复位
        UCA1IE |= UCRXIE;                         //  使能 UCA1 接受中断

}
定时器 TA 初始化函数
void TimerA_Init(void)          //定时器 TA 初始化函数
{
      TA0CTL |= MC_1 + TASSEL_2 + TACLR; //时钟为 SMCLK,比较模式，开始时清零计数器
      TA0CCTL0 = CCIE;                   //比较器中断使能
      TA0CCR0   = 50000;                 //比较值设为 50000，相当于 50ms 的时间间隔
}
```

3.6.1.5 实验步骤与现象

- **实验步骤**

1. 完成代码的编写，并将代码烧录到开发板中
2. 使用串口调试助手通过电脑向单片机发送数据同时接收单片机发的数据
3. 观察 TFT 屏幕上显示的信息和串口调试助手显示的信息
4. 短接 MSP430F5529LP 板上 RX 与 TX 引脚（**注意不是口袋板，是 F5529LaunchPad！**），可以看到单片机自发自收现象（图 3.6.5）

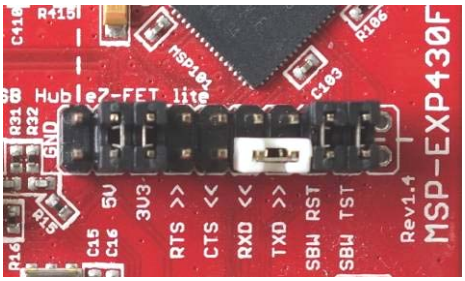


图 3.6.5　MSP430F5529LP 板上串口收发管脚短接

- **实验现象**

屏幕上显示的 RS232 端口发送与接收的数据在变化，当然我们也可以通过串口调试工具实现 F5529 与 PC 机进行串口通信。

3.7 Flash 模块

Flash 有两类 NOR Flash 和 NAND Flash， 两者最大区别就是 NOR Flash 可以按字节读取数据，所以可以直接在上面运行代码（也就是做 ROM 用） ，MCU 中的 Flash ROM 就属于 NOR Flash；NAND Flash 只能块读取数据，不能在上面运行代码，U 盘就是 NAND Flash。

处理器按存储器构架可分为普林斯顿结构（冯诺依曼结构）和哈佛结构。普林斯顿结构处理器的 RAM 和 ROM 统一编址，共用地址总线和数据总线。哈佛结构的处理器，RAM 和 ROM 分开编址，有两条地址总线和两条数据总线。这样一来处理器在一个机器周期内可以同时获得指令（来自 ROM）和操作数（来自 RAM），从而提高了执行速度。

哈佛结构更适合高速处理器，例如最新构架的 ARM 系列处理器和 DSP 处理器。MSP430 单片机属于低功耗低速类处理器，采用的是普林斯顿结构。

对 Flash 的擦写需要专门的编程电压发生器，如果 MCU 集成了该片内外设，便可在程序运行中对 Flash 的内容进行擦写，从而使 MCU 无需外接 EEPROM 芯片进行掉电不失数据存储。MSP430 全系列单片机都集成有 Flash 控制器，能够产生 Flash 写操作所需的时序以及编程电压。

MSP430F5529 芯片中集成的 128KB FLASH 存储器可以按字节（byte）、字（2-bytes）、双字（4-bytes）寻址或编程，控制器集成了编程与擦除功能。FLASH 模块包含 3 个寄存器、一个定时器以及一个产生编程与擦写电压的电压生成器。

3.7.1 Lab3-7-1 Flash 实验

3.7.1.1 实验介绍

本次实验应用了 MSP430F5529 的 Flash 读写功能，实现了在片内闪存地址中写入数据，并且在屏幕中显示 Flash 的内容。

3.7.1.2 实验目的

- 学习 Flash 的基本知识
- 熟练掌握 MSP430F5529 中 Flash 的读写操作

3.7.1.3 实验原理

Flash 闪存是非易失存储器, 全名叫 Flash EEPROM Memory，可以对称为块的存储器单元块进行擦写和再编程，它结合了 ROM 和 RAM 的长处，不仅具备电子可擦除可编程（EEPROM）的性能，还可以快速读取数据（NVRAM 的优势），使数据不会因为断电而丢失。目前 Flash 主要有两种 NORFlash 和 NANDFlash。NORFlash 的读取与 SDRAM 的读取

一样，用户可以直接运行装载在 NORFLASH 里面的代码。NANDFlash 没有采取内存的随机读取技术，它的读取是以一次读取一块的形式来进行的，通常是一次读取 512 个字节。

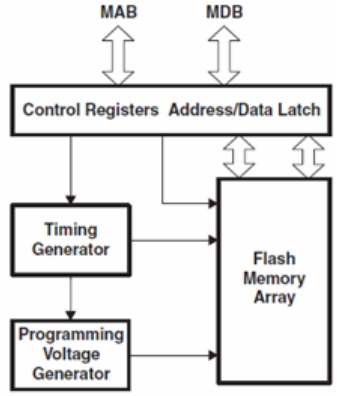


图 3.7.1 Flash 模块结构框图

控制器包括三个寄存器，一个时序发生器及一个提供编程、擦除电压的电压发生器。MSP430F5529 中的 Flash 存储器的特点有：产生内部编程电压；可位、字节、字编程，可以单个操作，也可以连续多个操作；超低功耗操作；支持段擦除和多段模块擦除。

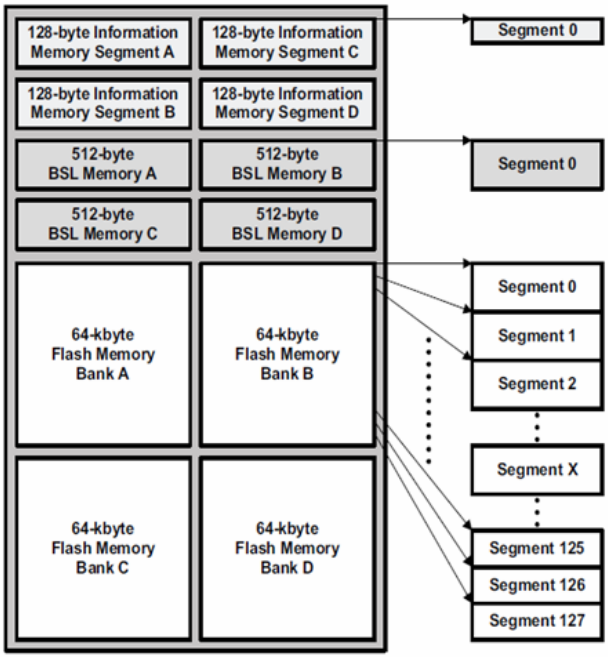


图 3.7.2 Flash 模块区段划分

Flash 存储器被分割成两部分：主存储器和信息存储器，两者在操作上区别不大；但主存储器允许整块(Bank)擦除，且整块擦除时可以继续执行其它块上的程序。MSP430F5529 中的 Flash 可以进行单个字节、字的写入，也可以进行连续多个字、字节的写入操作，但是最小的擦除单位是段。如上图所示，MSP430F5529 Flash 的信息存储器由 4 个 128 字节的段组成；主存储器由 4 个 64K 的块组成，每块又分为 128 段。

Flash 的读操作很简单，报地址就出数据，但是擦和写操作比较复杂，需要一个时钟。时钟周期短，写入不可靠，时钟周期长会降低 Flash 擦写寿命。按规定，Flash 时钟频率需为 257kHz~476kHz 之间。

如图 16.5 所示为 Flash 时钟结构，可以选择时钟来源和分频系数。由于 ACLK 时钟一般为外部低频晶振 32.768kHz 或者内部低频振荡器 12kHz，所以都是不足以提供所需的 257kHz~476kHz 频率的。MCLK 与CPU 捆绑销售，不利于低功耗，所以一般所有片内外设的时钟都不会使用 MCLK。剩下就是 SMCLK 了，SMCLK 的频率一般从 1Mhz~16Mhz 不等，需要分频后才能提供给 Flash 时钟。

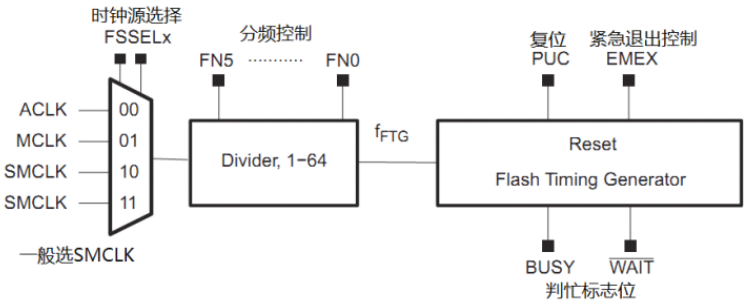


图 3.7.3 Flash 时钟控制器

Flash 存储器空白时各比特位均为 1， 写 Flash 的过程实际是将各位 1 改写为 0 的过程。当确认待写入空间为空白时，可以直接进行写操作。如果待写空间不为空白，重复写入倒不会天崩地裂，不过是重复写 0。写 Flash 有两种方式，字节/字写入，或者块写入，由 BLKWRT 控制位来选择。

Flash 字节写入模式：

如图 3.7.4 所示。

（1） BUSY=0 代表 Flash 控制器空闲。与我们一般编程习惯不同，BUSY=1 时禁止干任何事情，只能用 while(FCTL3&BUSY)语句死循环等待 BUSY=0。要是等的不耐烦了，写图 16.5 中所示的 EMEX 紧急退出控制位强行终止 Flash 操作。

（2） 写 Flash 的过程首先是打开编程电压发生器，然后写入 Flash，最后再关掉编程电压发生器。这期间 BUSY 位一直维持高电平。

（3） 此方式下，每次可写入 1 个字节（8 位）或 1 个字（16 位），每次写操作都要开关编程电压发生器。

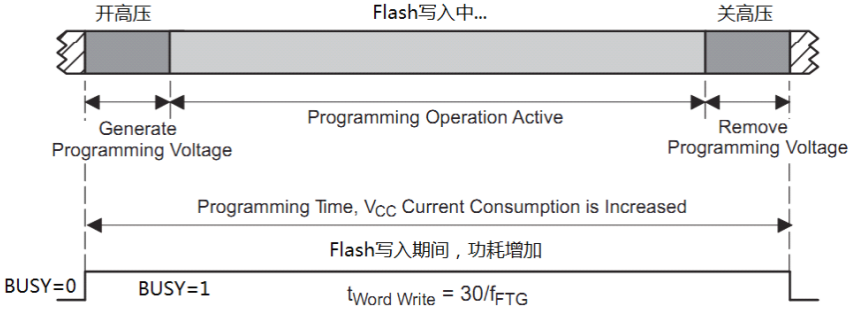


图 3.7.4 Flash 字节写入过程

Flash 块写入模式：

如图 3.7.4 所示。

（1） 与字节/字写入方式相比，除了 BUSY 用于判断 Flash 工作状态外，还多出一

个 WAIT 位。

（2） 当写完 1 字/字节时，WAIT 位会置 1，表明此时可以写下一字节。

（3）只有写完全部 1 块 64 字节，BUSY 位才会置 0，也就是块写每次最少写 64 字节。

（4） 由于只需开关一次编程电压发生器，所以块写操作要快。

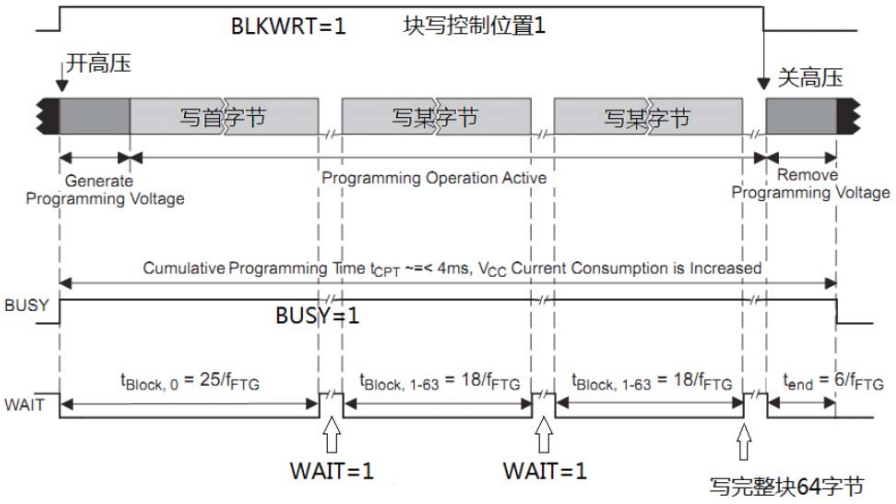


图 3.7.5 Flash 块写入过程

Flash擦除操作：

如果要对Flash数据进行改写，需要先擦除。Flash 的擦写寿命保证值是 1 万次，典型值10万次。如果Flash仅用作存放程序代码的ROM使用，那是够用的，因为完成一次下载程序不过消耗1次寿命。但是，1-10万次的寿命用于掉电不失的数据存储，是非常纠结的。原因在于，哪怕只有 1 个字节要改写，也需要擦写整个 Flash 段。

设想一下，某1字节数据需要每分钟保存一次，那么1天将保存1440遍，按10万次擦写寿命来算，可连续工作不到70天，整个数据段就毁了。有两个方法可以延长 Flash 寿命：

（1）对于那些不频繁调用仅需记录的非掉电数据，可以循环记录，递增地址计满整段后，再统一擦除。

（2）对于需要频繁调用的非掉电数据，应先使用RAM存储数据，在掉电前才将数据写入Falsh。采用这种方法要引入电池检测电池电量功能和LPM4关机功能，在将要电池耗尽和人工关机前，才将数据写入Flash。（注：MSP430单片机低功耗模式LPM4下，RAM数据不会丢失，而功耗电流小于电池漏电流，可作为等效关机使用）在Flash控制器中，MERAS和ERASE控制位负责擦写模式选择，LOCKA负责保护信息字段不被改写，对LCOKA位写 1 改变状态，写 0 没作用，默认处于保护状态。

Flash寄存器 操作：

Flash控制器的主要控制寄存器有FCTL1/3/4和SFRIE1，其中FCTL4不常用,SFRIE1不用中断也可以不用。Flash寄存器都是16位寄存器，高八位用于安全校验，必须写入0A5h否则会重启单片机。与看门狗寄存器类似，任何情况下只能用“=”进行赋值，原因见看门狗章节。Flash 控制器的控制位较多，常用的有以下几位：

（1） BLKWRT 和 WRT 配合：使能写字/字节或块写操作。

（2） MERAS、ERASE 和 LOCKA 配合：决定擦除方式。

（3） FSSELx、FNx 配合：设定 Flash 时钟及分频。

（4） EMEX：写此位可紧急退出 Flash 操作。

（5） LOCK：Flash 擦写锁定，锁定后不允许擦写。

（6） WAIT 和 BUSY：用于 Flash 擦写中判断操作结束。

（7）LOCKA：用于锁定和解锁InfoFlashA段。该控制位用法特殊，每次置1切换锁定和解锁状态，置 0无意义。

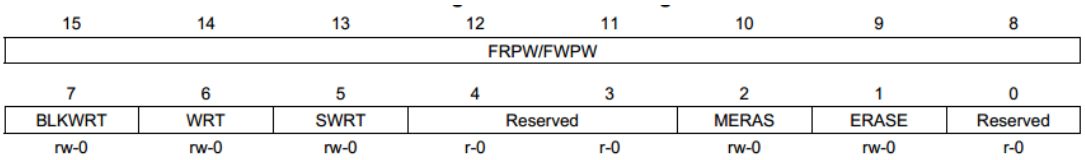


图 3.7.6 FCTL1 寄存器

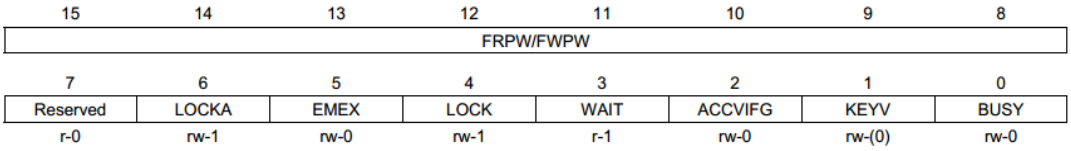


图 3.7.7 FCTL3 寄存器

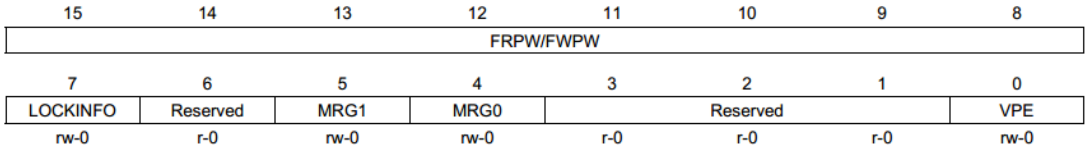


图 3.7.8 FCTL4 寄存器

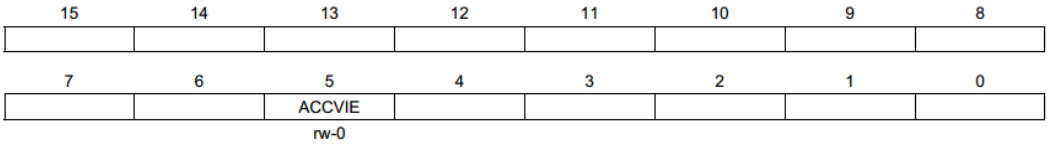


图 3.7.9 SFRIE1 寄存器

3.7.1.4 程序分析

- **编程思路**

对 Flash 的操作需要按照固定的顺序方式来执行，并且执行的结果存储在 Flash 中，为了能体现实验的可视性效果，本实验采取点亮 LED 灯的方法。当 Flash 没有写入数据时，地址内存的数据时 0xffff，我们可以先读取 Flash 内一地址数据如果是 0xffff 把 LED1 灯点亮，如果不是把 LED1 灯熄灭，并点亮 LED2。然后如果有按键按下那么就给该地址写入一任意非 0xffff 的数据。这样当按键没有按下也就是 Flash 没有写入数据时多次上电都会是 LED1 亮，LED2 灭，如果按键按下后多次上电都会是 LED2 亮，LED1 灭。

- **程序流程图**

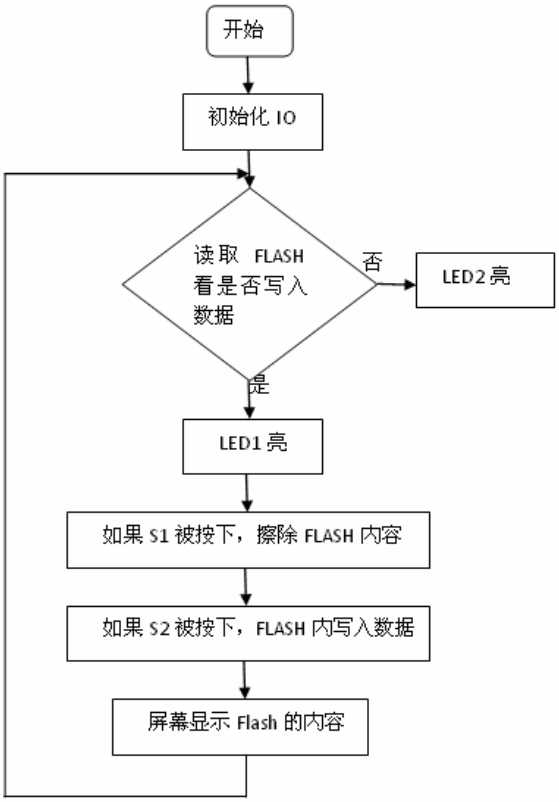


图 3.7.10　程序流程图

- 关键代码分析

主函数

```
int main(void) {
     int i=180;
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    P8DIR |= BIT1;
    P8OUT &=~ BIT1;
    initClock();
    PaperIO_Int();
    INIT_SSD1673();

    Init_buff();
    display("Huatsing Instruments", 42, 16,TimesNewRoman,size8,1,0);

     FlashErase(ADR,WORD,LEN);                    //擦除写入 Flash 中的数据
     FlashWriteWord(ADR,FlashSetBuf,LEN);         //双字节写入函数
    FlashRead(ADR,RxBuf1,WORD,LEN);               //Flash 读取函数
```

双字节写入函数

```
void FlashWriteWord(uintaddress,uint *word,uint count)
{
    FCTL3 = FWKEY;                          //Flash 解锁
    while(FCTL3 & BUSY);                    //等待忙
    FCTL1 = FWKEY + WRT;                    //选择写入模式
    while(count--)
    {
        *((uint *)address) = *(word++);     //把双字节缓冲区的数据写入对应的地址
        address = address + 2;              //写入的数据地址按顺序累加
    }
    FCTL1 = FWKEY;                          //清除写入模式
    FCTL3 = FWKEY + LOCK;                   //Flash 上锁
    while(FCTL3 & BUSY);                    //等待忙
}
```

数据读取函数

```
void FlashRead(uintaddress,uint *buf,uintmode,uint length)
{
    if(mode)                                //判断读取模式：0 表示单字节，1 表示双字节
    {
        while(length--)                     //判断读取数据的个数
        {
            *(buf++) = *((uint *)address);
                                            //把读取的双字节数据送入数据读取缓冲区
            address = address + 2;
        }
    }
    else
    {
        while(length--)
        {
            *((uchar *)buf++) = *((uchar *)address);
                                            //把读取的单字节数据送入数据读取缓冲区
            address = address + 2;
        }
    }
}
```

Flash 擦除函数

```
voidFlashErase(uintaddress,uintmode,uint length)
{
FCTL3 = FWKEY;
while(FCTL3 & BUSY);
FCTL1 = FWKEY + ERASE;
if(mode)                                    //判断数据模式
{
 while(length--)                            //判断擦除数据个数
 {
    *((uint *)address) = DELETE;            //将对应地址上的双字节数据擦除
    address = address + 2;
 }
}
else
{
 while(length--)
 {
    *((uchar *)address) = DELETE;           //将对应地址上的单字节数据擦除
    address = address + 2;
```

```
    }
  }
  FCTL1 = FWKEY;
  FCTL3 = FWKEY + LOCK;
  while(FCTL3 & BUSY);
  }
```

3.7.1.5 实验步骤与现象

- **实验步骤**

1. 实现控制代码的设计，画出流程图；
2. 完成编码，并将代码烧录到发板中；
3. 观察 LED 的状态变化，验证结果；
4. 观看液晶显示内容的变化。

- **实验现象**

按下口袋板上 S1 按键，擦除 Flash 指定区域数据；按下 S2 按键，在 Flash 指定区域写入特定数据，我们也可以通过 CCS 的 Memory Browser 来查看，应该与口袋板屏幕显示一致。

3.7.1.6 实验思考

1. Flash 分为哪几种？它们之间有什么区别？
2. 请尝试实现在 Flash 中写入数据并读出。
3. 请设计一个结果更为直观的 Flash 操作实验。

3.8 ADC 模块

模拟量--数字量转换模块（ADC）与数字量--模拟量转换模块（DAC）是非常有用的功能模块，很多单片机中只有 ADC 模块、没有 DAC 模块，MSP430F5529 也一样，只有一路 ADC，但是通过片内的多路复用模块，扩展出了 12 个通道。这里我们先介绍 F5529 内部的 ADC 模块的原理及应用，与之对应的 DAC 因为没有集成在 F5529 内部，我们把它放在下一章片外设备来讲解。下面首先简要介绍一下模拟量与数字量的概念以及他们之间的关系。

- **模拟信号**

模拟信号是指用连续变化的物理量表示的信息，其信号的幅度、频率或相位随时间作连续变化，如目前广播的声音信号，或图像信号等。当模拟信号采用连续变化的电磁波来表示时，电磁波本身既是信号载体，同时作为传输介质；而当模拟信号采用连续变化的信号电压来表示时，它一般通过传统的模拟信号传输线路（例如电话网、有线电视网）来传输。

- **数字信号**

数字信号指幅度的取值是离散的，幅值表示被限制在有限个数值之内。二进制码就是一种数字信号。它具有抗干扰能力强、无噪声积累、便于加密处理、便于存储、处理和交换等特点。

- **模拟信号与数字信号之间的相互转换**

模拟信号一般通过 PCM 脉冲编码调制(Pulse Code Modulation)方法量化为数字信号，即让模拟信号的不同幅度分别对应不同的二进制值，例如采用 8 位编码可将模拟信号量化为 2^8=256 个量级，实用中常采取 24 位或 30 位编码；数字信号一般通过对载波进行移相(Phase Shift)的方法转换为模拟信号。计算机、计算机局域网与城域网中均使用二进制数字信号，目前在计算机广域网中实际传送的则既有二进制数字信号，也有由数字信号转换而得的模拟信号。但是更具应用发展前景的是数字信号。

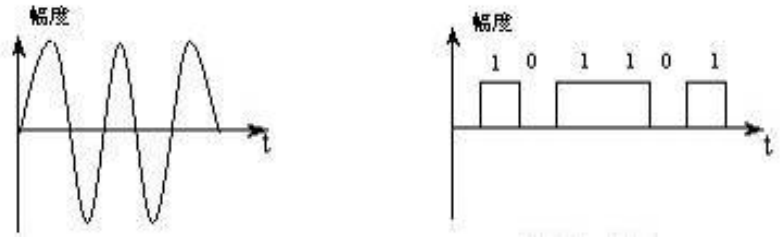


图 3.8.1　模拟信号（左）与数字信号（右）

AD 转换：AD 转换器根据其原理主要可以分为积分型、逐次逼近型和并行比较型。在获取相同的转换精度时，积分型 AD 转换器的电路最为简单、成本低、同时转换耗时最长，并行比较型则拥有最高的转换速度和最复杂的电路，逐次逼近型的性能介于另两种之间。MSP430 的 AD 模块为逐次逼近型，其转换器核心由一个比较器、一个 DA 转换器和逐次比较逻辑组成。工作时转换器会从最高位到最低位依次生成试探电压的数字值，将其送入 DA 转换器生成试探电压信号，并将这一信号与输入信号进行比较来决定每一位的取值，对于 n 位的转换器，每次转换过程需要比较 n 次。

3.8.1 Lab3-8-1 ADC 实验

3.8.1.1 实验介绍

本实验演示了 MSP430F5529 中 ADC12 模块的使用。实验结合了 ADC12 与电位器的应用，通过获取电位器的模拟电压值，实现了用电位器对 5 个不同 LED 灯的亮灭控制。

3.8.1.2 实验目的

- 了解 ADC 转换原理
- 学习 MSP430F5529 中 ADC12 的配置使用方法
- 掌握 LED 灯的控制方法
- 结合电位器与 ADC12 模块实现对 LED 灯的控制

3.8.1.3 实验原理

模数转换器（ADC）是指将连续的模拟信号转换为离散的数字信号的器件。真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发射的数字形式。在 A/D 转换中，因为输入的模拟信号在时间上是连续的，而输出的数字信号是离散量，所以进行转换时只能按一定的时间间隔对输入的模拟信号进行采样，然后再把采样值转换为输出的数字量。通常 A/D 转换需要经过采样、保持量化、编码几个步骤。MSP430F5529 中的 ADC12 模块结构如下图所示。ADC12 模块主要组成组成部分有：

- 输入端 16 路模拟开关；
- 内部电压参考源；
- ADC12 内核；
- 时钟源部分；
- 采集与保持部分；
- 数据输出部分；
- 控制寄存器。

ADC12 的模块内核是共用的，通过前端的模拟开关来分别完成采集输入。ADC12 是一个精度为 12 位的 ADC 内核，1 位非线性微分误差，1 位非线性积分误差。内核在转换时会参用到两个参考基准电压，一个是参考相对的最大输入最大值，当模拟开关输出的模拟量大于或等于最大值时 ADC 内核的输出数字量为满量程，也就是 0xfff；另一个则是最小值，当模拟开关输出的模拟变量小于或等于最大值时，ADC 内核输出的数字量为最低值，也就是 0x000。转换后的数据将存贮在 ADC12MEMx 中，Nadc=4095*（Vin-Vr-）/（Vref-Vr-），Vr-在本实验中取值为 0，因此 Nadc=4095*Vin/Vref，Vin=Nadc*Vref/4095。

ADC12 模块的主要寄存器有 ADC12CTL0（转换控制寄存器）、ADC12CTL1（终端控制寄存器）、ADC12IFG（中断标志寄存器）)、 ADC12IE（中断使能寄存器）、ADC12IV（中断向量寄存器）。

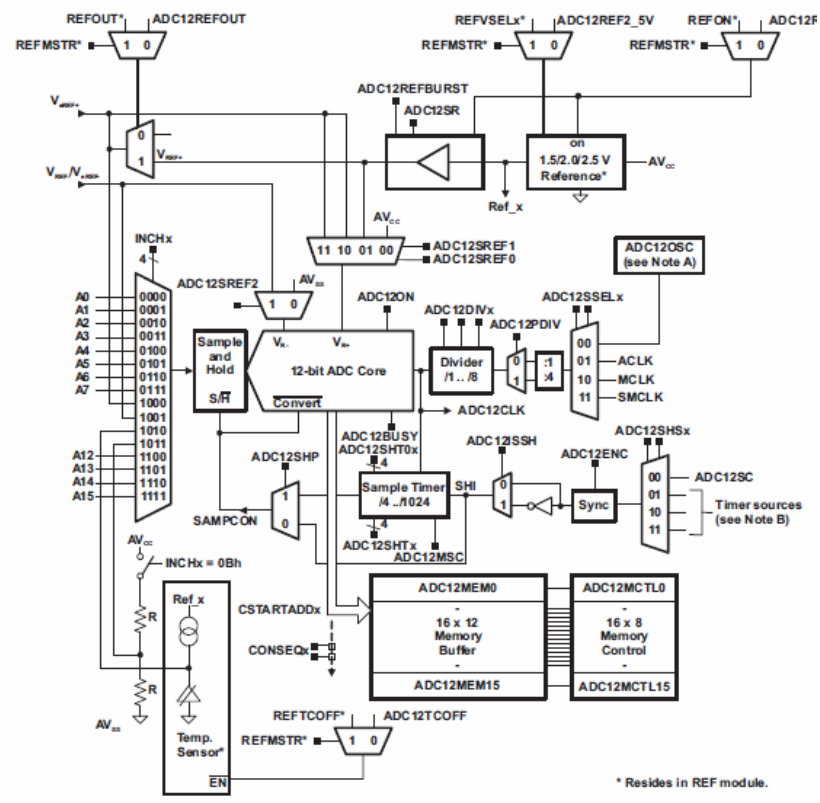


图 3.8.2　ADC12 模块结构框图

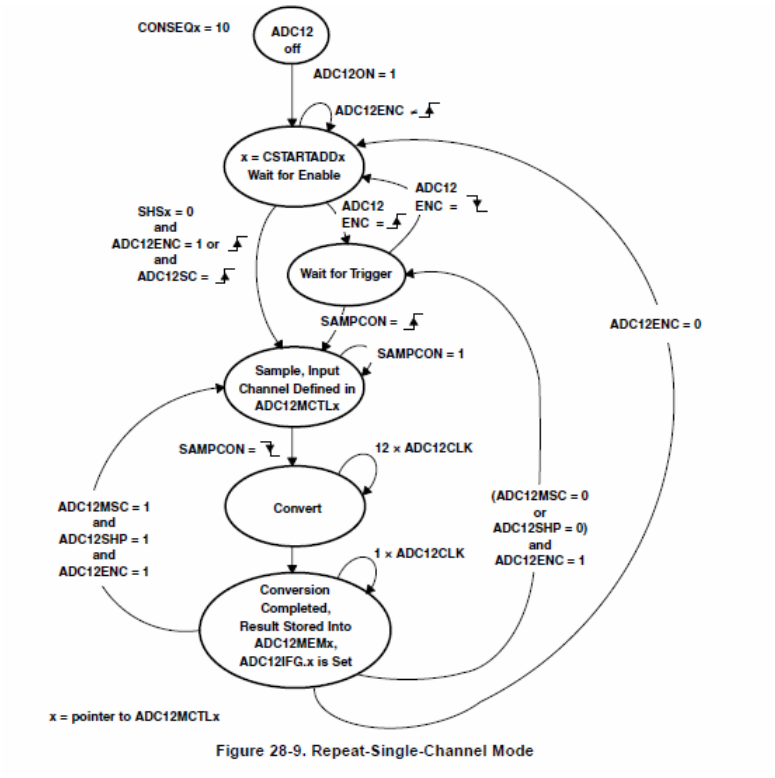


Figure 28-9. Repeat-Single-Channel Mode

图 3.8.3　ADC12 模块状态转移图

采样方式分为单通道单次采样、多通道单次采样、单通道循环采样、多通道循环采样

四种方式，我们本次实验采取的是单通道循环采样方式（上图所示）。当 ADC12ON 为高电平是，ADC 转换器启动并等待转换开始的信号；此时当 ADC12ENC 位为 1 且 ADC12SC 出现上升沿时开始转换过程，并把每次转换的数据保存在 ADC12MEN0 中。在单通道循环采样方式下转换过程会循环进行，直到将 ADC12ENC 复位。

F5529 口袋板上配置的是双路拨盘电位器（R4），其他一路用来调节 ADC（P6.5）输入端的电压（0~3.3V）；另外一路用来控制 DAC 输出电压的大小，通过这一路就可以实现对耳机插座与扬声器端输出的音频信号幅值进行控制。这个实验是改变 ADC 端输入电压，然后依据电压高低分为几档通过 LED1~LED5 显示出来。

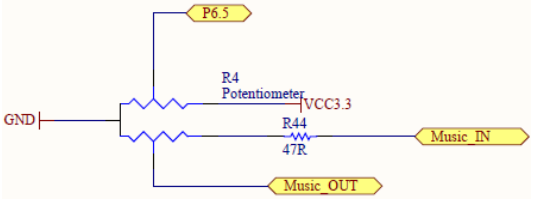


图 3.8.4　拨盘电位器模块电路原理图

3.8.1.4 程序分析

- **编程思路**

熟悉了 MSP430F5529 中的 ADC12 模块原理之后便可对其控制寄存器进行配置，设计采样模式，时刻得到电位器的输出端电压值。并通过其大小，设定范围从而来控制 LED 灯的亮灭。

- **程序流程图**

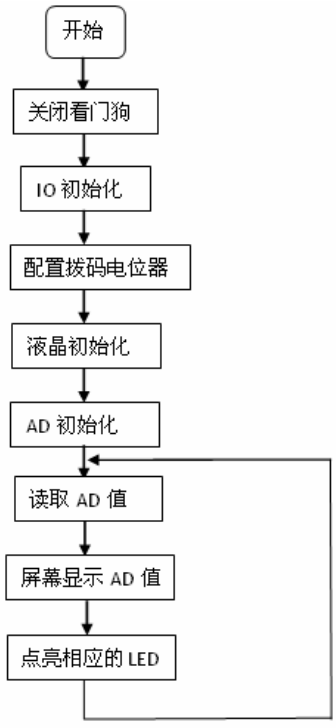


图 3.8.5　程序流程图

- 关键代码分析

实验 ADC 代码

AD 初始化函数：

```
void AD_Init()
{
    ADC12CTL0 |= ADC12MSC;
                                        //自动循环采样转换
    ADC12CTL0 |= ADC12ON;
                                        //启动 ADC12 模块
    ADC12CTL1 |= ADC12CONSEQ1 ;
                                        //选择单通道循环采样转换
    ADC12CTL1 |= ADC12SHP;
                                        //采样保持模式
    ADC12MCTL0 |= ADC12INCH_5;
    ADC12CTL0 |= ADC12ENC;              //选择通道 5，连接拨码电位器

}
```

主函数

```
int main(void) {
     int i=180;
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    P8DIR |= BIT1;
    P8OUT &=~ BIT1;
    initClock();
     ioinit();
     AD_Init();
    PaperIO_Int();
    INIT_SSD1673();
    Init_buff();
    display(" ADC INPUT ", 82, 4,TimesNewRoman,size8,1,0);
    DIS_IMG(2);
     volatile unsigned int value = 0;
          //设置判断变量
    while(1)
    {
        ADC12CTL0 |= ADC12SC;               //开始采样转换
        value = ADC12MEM0;                  //把结果赋给变量
        if(value > 5)                       //判断结果范围
            P8OUT |= BIT1;
        else
            P8OUT &= ~BIT1;
        if(value >= 800)
            P3OUT |= BIT7;
        else
            P3OUT &= ~BIT7;
        if(value >= 1600)
            P7OUT |= BIT4;
        else
```

```
                P7OUT &= ~BIT4;
            if(value >= 2400)
                P6OUT |= BIT3;
            else
                P6OUT &= ~BIT3;
            if(value >= 3200)
                P6OUT |= BIT4;
            else
                P6OUT &= ~BIT4;
            if(value >= 4000)
                P3OUT |= BIT5;
            else
                P3OUT &= ~BIT5;
            __delay_cycles(200000 );
        }
    }
```

3.8.1.5 实验步骤与现象

- **实验步骤**

1. 根据编程思路设计结构与实现方法。
2. 按照流程图实现代码编写，并在编译器上进行编译改错。
3. 将程序烧入开发板中进行调试与检测。
4. 通过调节电位器查看 LED 灯的变化是否符合设计要求。

- **实验现象**

旋转拨盘电位器，LED 灯指示出电位器输出电压的变化。

3.8.1.6 实验思考

1. 如何采集负电压信号？
2. 如何对正弦波或三角波进行模数转换实现计数功能？
3. 请尝试编写程序将采集的信号波形在 TFT 屏幕上显示。

第 4 章 I²C 扩展设备—DAC 与温度传感器

在第 3 章中全面介绍了 F5529 的片内资源，并且教大家用这些资源做了很多有趣的实验。F5529 口袋板上还有很多 F5529 之外的芯片与模块，通过这些片外资源，可以完成更加丰富有趣的实验，接下来几章我们给大家一一介绍这些片外资源。这一章中首先介绍 I²C 扩展设备：DAC 与温度传感器。

口袋板上 DAC 芯片（DAC7571）与数字温度传感器芯片（TMP421）都是通过 I²C 总线与 F5529 连接的，I²C 扩展总线也可以说是通用串行通信接口的另一种工作模式：I²C 模式。因为只有 USCI_Bx 模块才支持 I²C 模式，因此主板上两个 I²C 接口都与 MSP430F5529 中的 USCI_Bx 连接。

4.1 I²C 总线

4.1.1 I²C 总线介绍

I²C 总线是由 PHILIPS 公司开发的两线式串行通信总线，一条串行数据线 SDA 和一条串行时钟线 SCL，用于连接微控制器及其外围设备。它是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。

I²C 有两条总线线路，一条串行数据线 SDA 和一条串行时钟线 SCL；每一个连接到总线的器件都有唯一的地址；串行的 8 位双向数据传输位速率在标准模式下可达 100kbits/s，快速模式下可达 400kbits/s，高速模式下可达 3.4Mbits/s；支持多主控模块，但同一时刻只允许有一个主控。

4.1.2 I²C 总线协议

总线只有在不忙时才可以启动数据传输，其“启动”和“停止”数据传输原理图如下图所示，当 SCL 线为高电平，SDA 线由高电平变为低电平会产生“启动”条件，所有数据传输前必须有“启动”条件；当 SCL 线为高电平，SDA 线由低电平变为高电平会产生“停止”条件，所有数据传输必须以“停止”条件结束。在“停止”条件后“启动”条件前，SCL 和 SDA 线在这期间保持高电平表示空闲状态。

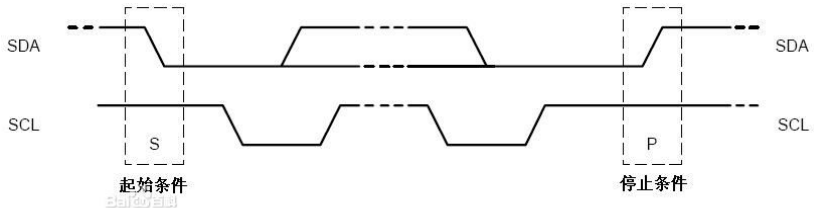


图 4.1.1 I²C 协议起始与终止条件

I²C 总线数据数据传输为字节格式，即发送到 SDA 线上的每个字节必须为 8 位，先传输最高位(MSB)，每一个被传输的字节后面都必须跟一位应答位（由从机发出）。所有主机在 SCL 线上产生它们自己的时钟来传输 I²C 总线上的报文。数据只在时钟的高电平周期有效，既在传输数据的时候，SDA 线必须在 SCL 线高电平周期保持稳定，SDA 的高或低电平状只有在 SCL 线低电平时才能改变，因此需要一个确定的时钟进行逐位仲裁。I²C 总线寻址方式有明确规定，采用 7 位寻址方式，发送的第一个字节高 7 位表示从机地址，最低位（第 8 位 LSB）决定数据的传输方向，如果是‘0’表示主机向从机写数据，如果是‘1’表示主机由从机读数据，当发送一个地址后，系统中的每个器件都在起始条件后将头 7 位与它自己的地址比较，如果一样，器件会判定它被主机寻址。

4.1.2.1 I²C 通信

如下图所示的 I2C 总线硬件连接中，一共有两条总线，串行时钟线和串行数据线。两条总线都被上拉电阻拉到 VCC，所有 I2C 设备都挂载在总线上，各设备的地位对等，都可作为主机或从机。从分配地址来看，最多挂载设备可有 1024 个，实际挂载设备数量受总线电容限制。线与逻辑的规则是，每个设备都可以把总线接到地拉低，却不许把总线电平直连 VCC 而置高。把总线电平拉低称为占用总线，总线电平为高等待被拉低则称为总线被释放。利用线与结构，I2C 制定了“神乎其技”的协议规范，仅用 2 根线就完成任意多主多从双向通信中的所有难题。

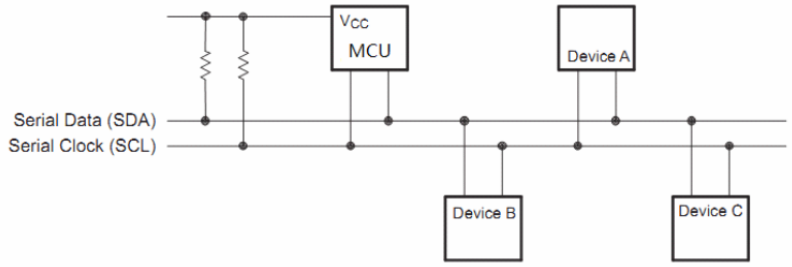


图 4.1.2 I2C 总线硬件连接

如下图所示，I2C 协议的完整帧包括起始位、地址位、读写位、应答位、数据位、应答位...数据位、应答位、停止位。

（1） 从起始位到停止位之间所有的数据位都是主机与符合地址位的从机进行通信。

（2） 从起始位开始每帧数据都是 9 位，其中第一帧是 7 位从机地址+1 位读写标识位+1 位数据接收方应答位组成。后续的每帧都是 8 位数据+1 位数据接收方应答。

（3） 如果读写标识 R/W=0，表示主机向从机发送数据，则应答位 ACK 由从机负责拉低。从机在完整收到地址或数据后拉低 SDA 数据总线，表示正确接收。不应答表示数据接收错误。

（4） 如果读写标识 R/W=1，表示主机自从机接收数据，则应答位 ACK 由主机负责拉低。

（5） 主机发出第一帧地址和读写位后，地址符合的从机拉低总线，产生 ACK 应答信号。主机开始收或发（视 R/W）下一帧，直到产生停止位。这期间，其他从机不接收数据，仅判断停止位是否出现，等待下一次比对地址通信的机会。

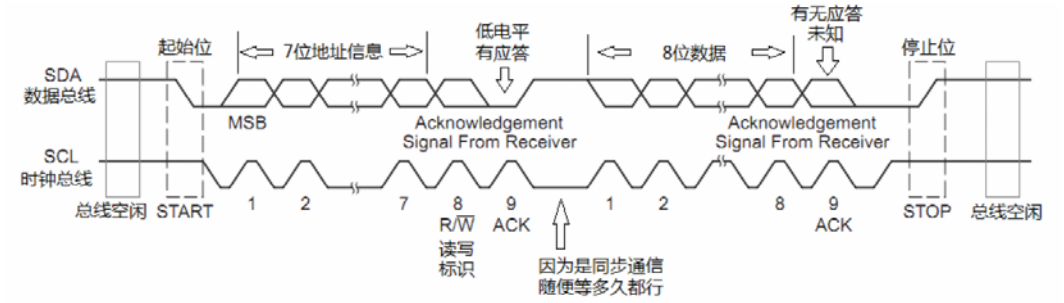


图 4.1.3 I2C 协议帧格式

4.1.2.2 I2C 的起止位

如下图所示，I2C 协议的数据线电平仅允许在时钟低电平时才允许改变。这是因为低电平时允许改变数据，高电平时读取数据，意味着数据的传输时刻在上升沿。线与逻辑是，谁都能拉低时钟总线产生下降沿，而产生上升沿却要所有设备都是 1。 这样一来，主机就不能强行给从机发数据，一定要从机应答才行，也就是如果从机没一档，就可以拉低 CLK，产生不了上升沿。

数据线电平在时钟低电平时改变是正常传输数据的状态，那么时钟线高电平时改变数据线电平就可以赋予其他含义，这是非常巧妙的设计。

（1） 在时钟线高电平时，数据线下降沿代表了起始位 START。

（2） 在时钟线高电平时，数据线上升沿代表了停止位 STOP。

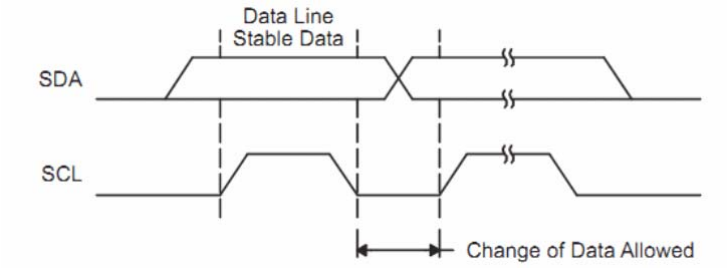


图 4.1.4 I2C 数据允许改变的情况

如果从机往主机发送信息，主机突发情况需要和其他从机通信，怎么处理呢？如下图所示，主机可以直接重新给起始信号，称为 Repeated Start。Repeated Start 后，主机发出新的地址，重新选择从机通信。

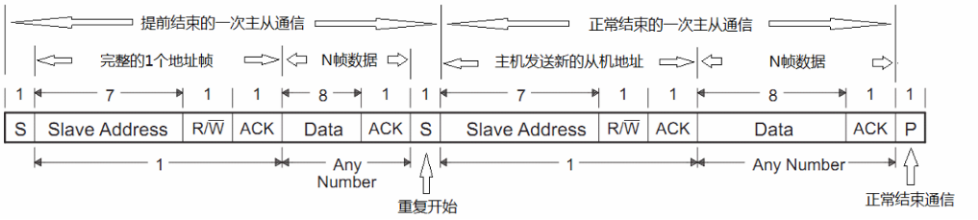


图 4.1.5 Repeated Start 模式

4.1.2.3 I2C 地址规范

I2C 总线的地址位分为 7 位和 10 位两种，实际上能够挂载的器件数量受总线最大电容 400pF 的限制。为什么地址位会有 7 位呢？7 位地址+1 位读写标识+1 位应答构成 9 位帧和普通数据帧 9 位所兼容。至于 10 位地址，则可表达 1024 位地址，按 400pF 的总线电容限制，每个器件能引入的电容仅 0.4pF，已是极限。所以 10 位地址足够。而 7 位地址已经可表示 128 个器件，能满足绝大多数需求。

7 位地址模式：如图 4.1.8 所示，起始位后的首帧为 7 位地址+1 位读写位标识位+1 位应答位，后续帧均为数据帧，直到停止位出现（或者是重复起始 repeat start） 。

10 位地址模式：如图 4.1.9 所示，起始位后的首帧中前 5 位固定为 11110（非表示地址） ，后面仅跟 2 位地址，然后是读写标识和应答。第二帧的 8 位数据作为地址的后 8 位。

这种方式可以做到 7 位地址与 10 位地址兼容， 10 位地址不过是把第 2 帧的数据继续当后续地址罢了。

地址冲突：很多芯片（非处理器）的地址是固化在芯片内部的，同一型号芯片有几种地址，以芯片后缀来区分。有的芯片则是在外部有地址引脚，依靠上拉、下拉、高阻来设定地址。

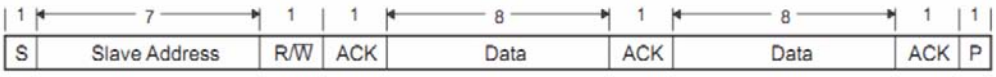


图 4.1.8　7 位地址模式

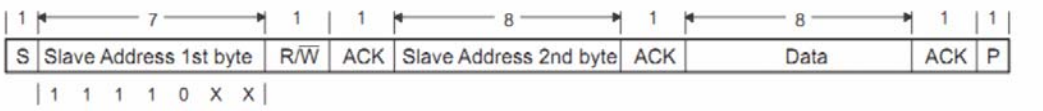


图 4.1.6　10 位地址模式

4.1.2.4 I2C 的多主机仲裁

I2C 协议是完全对称的多主机通信总线，任何一个设备都可以成为主机从而控制时钟总线。但是同一时间，只能有一个主机控制总线。当有两个或以上器件都想与别的器件进行通信时，则需要仲裁究竟由谁控制总线。

（1）只有监听到停止位，总线空闲时，才能争夺总线。这里需要强调一点，所有 I2C 设备都必须同时遵守 I2C 总线协议规范，才能正确通信，如果有一个器件不按规范 ，通信协议便会失败。例如某器件可以永久的把时钟总线和数据总线拉低，那么大家都一辈子别通信了。主机 A 和从机 B 正在通信，按规范从机 C 只能等结束位或者重新开始位到达后，才能有所争夺总线成为主机或者监听地址参与通信 。

（2）争夺总线的过程如图 4.1.10 称为仲裁，基本规则就是，谁打算与地址小的从机通信，谁占总线。器件 1 打算与 01xxxxx 地址的器件通信，而器件 2 打算与 00101xx 地址的器件通信。当两者竞争到箭头位置时，器件 1 释放总线，而总线却被器件 2 拉低，则器件 1 竞争失败，器件 1 应变为高阻状态退出总线竞争。

如果器件 1 不退出竞争怎么办？任何器件不遵守规则，通信都无从谈起,所以器件 1 不

推出竞争,那么通信就会失败。为什么是与小地址通信优先呢？一个大地址和一个小地址同时 “输出”到数据总线上，数据总线表现出来的只能是小地址。

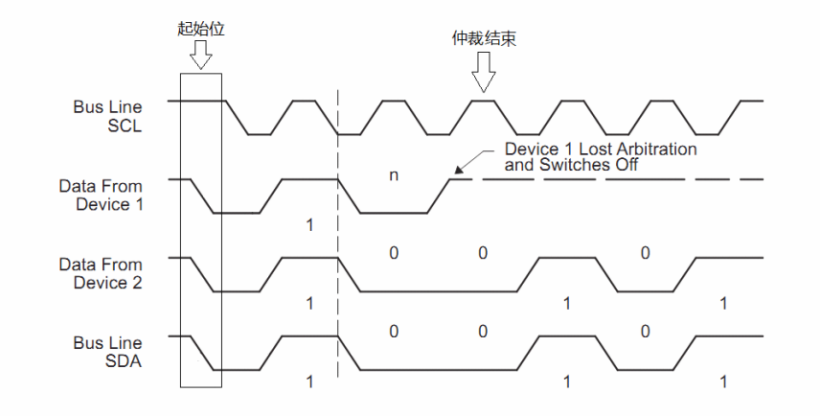


图 4.1.7　10 位总线仲裁过程

（3） 在仲裁过程中，两主机的时钟肯定不会一样，这时就有时钟同步的问题。如下图所示，器件 1 和器件 2 的时钟不同步，两者“线与”之后，才是总线时钟。所有器件都必须遵守最终的总线时钟，而不应该“只认”自己输出的时钟。如果有的从机通讯速度慢，则该从机可以拉低时钟总线，让总线等待。

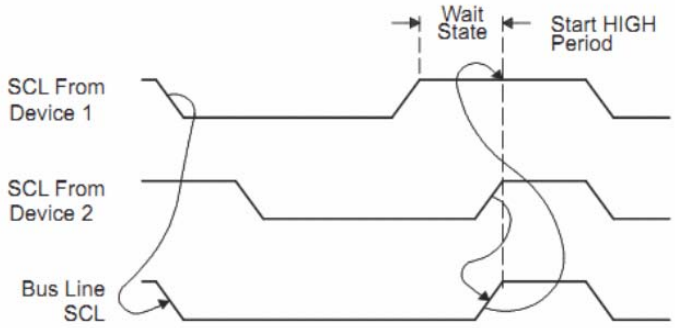


图 4.1.8　时钟总线同步过程

4.2 I²C 总线扩展设备

4.2.1 Lab4-2-1 DAC 实验

4.2.1.1 实验介绍

本实验演示了数字量转模拟量（DAC）模块的使用。MSP430F5529 通过 I2C 通信方式给 DAC 芯片 DAC7571 写入数字量，该芯片输出该数字量对应的模拟量。

4.2.1.2 实验目的

- 了解 DAC 转换原理
- 进一步熟悉 MSP430F5529 的 I2C 通信
- 能够用 DAC7571 输出 0-3.3V 之间的固定模拟电压
- 进一步熟悉 ADC。

4.2.1.3 实验原理

- **数模转换模块（DAC）**

数模转换是将数字量转换为模拟电量（电流或电压），使输出的模拟电量与输入的数字量成正比。实现这种转换功能的电路叫数模转换器（DAC）。D/A 转换器基本上由 4 个部分组成，即权电阻网络、运算放大器、基准电源和模拟开关。数字量以串行或并行方式输入、存储于数字寄存器中，数字寄存器输出的各位数码，分别控制对应位的模拟电子开关，使数码为 1 的位在位权网络上产生与其权值成正比的电流值，再由求和电路将各种权值相加，即得到数字量对应的模拟量。

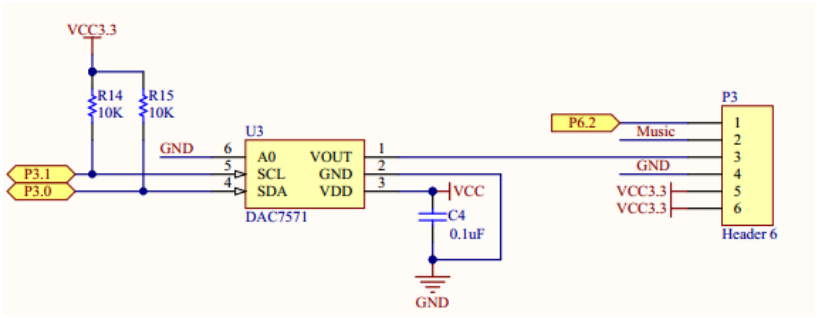


图 4.2.1 DAC7571 电路原理图

如上图所示，可以通过测量 P3 的引脚 3 的电压值得出输出的模拟量值。输出电压最大值为 VCC，最小值为 GND。

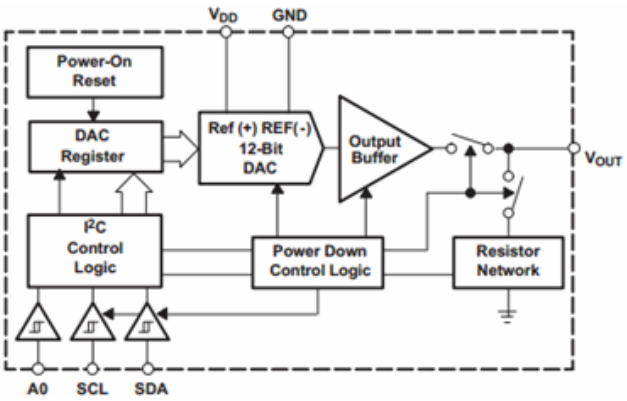


图 4.2.2 DAC7571 结构框图

- **DAC7571 介绍**

DAC7571 是低功耗,单通道 12 位 DA 转换器.DAC7571 兼容 I2C 接口,通过这两条数据线和外部通信,时钟的最高速度为 3.4Mbps.

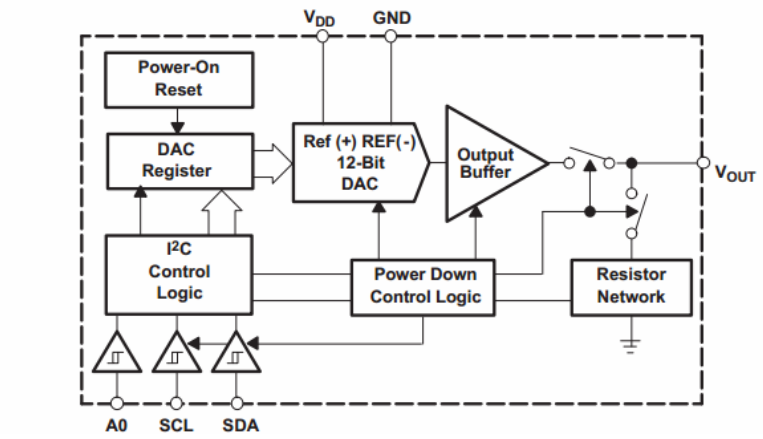
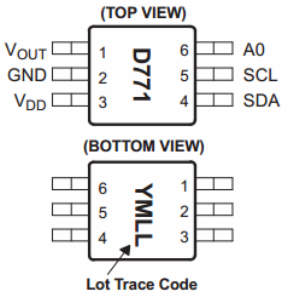


图 4.2.3　DAC7571 结构框图

如下图所示 DAC7571 的外观图和引脚定义.



PIN DESCRIPTION (SOT23-6)

PIN	NAME	DESCRIPTION
1	V_{OUT}	Analog output voltage from DAC
2	GND	Ground reference point for all circuitry on the part
3	V_{DD}	Analog Voltage Supply Input
4	SDA	Serial Data Input
5	SCL	Serial Clock Input
6	A0	Device Address Select
LOT TRACE CODE:	Year (3 = 2003); Month (1–9 = JAN–SEP; A=OCT, B=NOV, C=DEC); LL – Random code generated when assembly is requested	

图 4.2.4　引脚定义

如上图所示:

1 脚:模拟电压输出脚.

2 脚:接地.

3 脚:电源输入脚.

4 脚:串行数据输入引脚.

5 脚:串行时钟输入引脚.

6 脚:地址选择脚.

DAC7571 的数字信号转换成模拟信号是通过一个放大器进行转换，如下图所示.

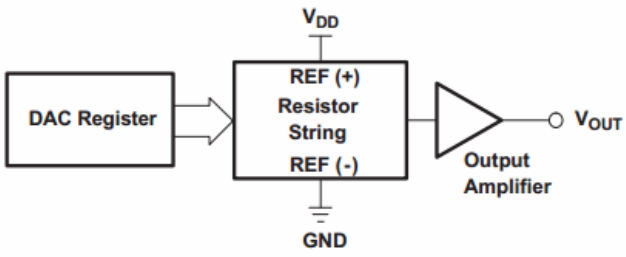


图 4.2.5　DAC 转换模块

DAC7571 的数字信号为无符号型数据.根据下面公式可以算出输出电压值和输入量的关系. $V_{OUT}= V_{DD}*(D/4096)$. V_{OUT} 代表输出电压值, V_{DD} 为电源电压值,D 为输入的数字量.

DAC7571 按照 I2C 通信规范进行通信,如下图所示:

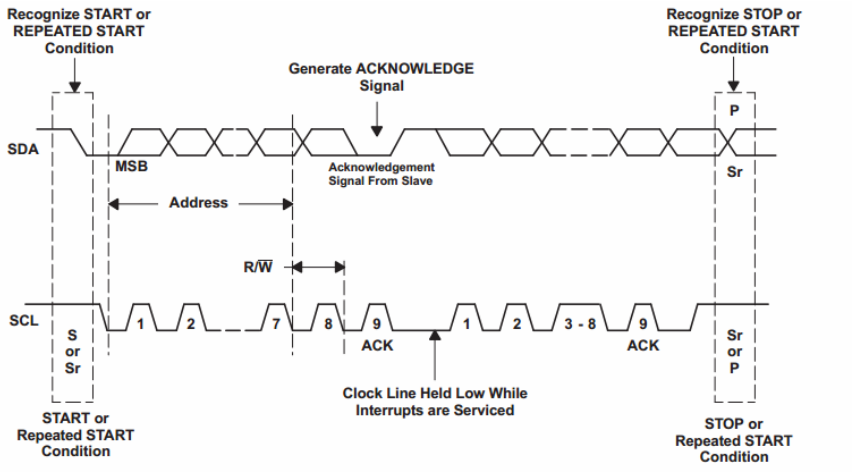


图 4.2.6　DAC7571 按照 I2C 总线通信时序

DAC7571 通信标准模式:

首先发送开始信号，然后发送从机地址字节（最后一位为读写控制位），然后等待从机应答，然后发送控制位和数据高四位组成的字节，然后等待从机应答，然后发送数据低八位，然后等待应答或者不等待应答，最后发送结束信号。

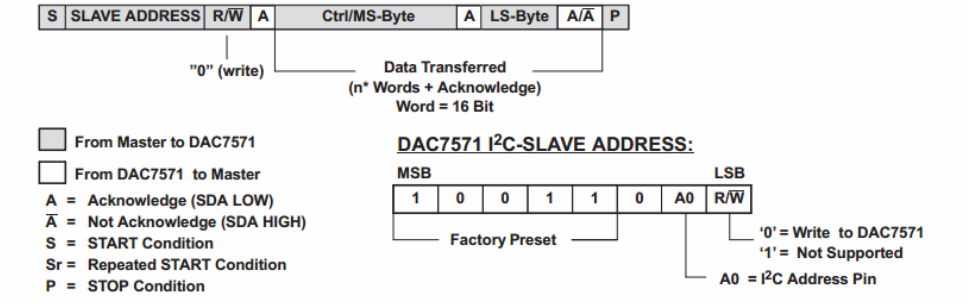


图 4.2.7　DAC7571 标准通信模式

名词解释：

A 表示应答信号,即数据线拉低。

S 表示开始信号。

Sr 表示重新发送开始信号。

P 表示停止信号。

From Master to DAC7571 表示信号是主机发送给 DAC7571。

From DAC7571 to Master 表示信号是 DAC7571 发送给主机。

DAC7571 的地址高 6 位由工厂烧录进去，最后一位是通过外部引脚确定。

地址共 7 位加上最后一个读写控制位构成一个字节。DAC7571 只支持写不支持读操作。

DAC7571 通信高速模式：

首先发送开始信号，然后发送发送高速模式控制字，然后发送一个时钟信号不用等待应答，然后重新发送开始信号，下面步骤和标准模式一样。

如下图所示

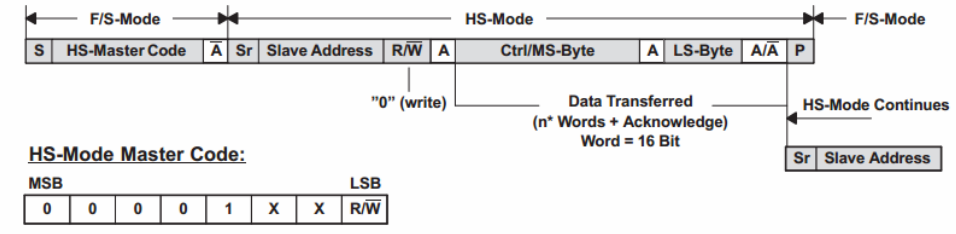


图 4.2.8 DAC7571 高速通信模式

名称解释：

HS-Mode Master Code 为告诉模式控制字

数据字节介绍：如下图所示

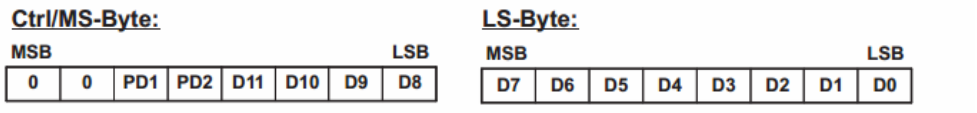


图 4.2.9 DAC7571 数据控制字

PD1 和 PD2 是操作模式选择，本实验为正常模式，所以这两个位都是 0。

D0-D11 为 12 位的数据位。

4.2.1.4 程序分析

- **编程思路**

首先通过配置基础的控制寄存器，并按照 I2C 通信原理，给 DAC7571 发送数字电压值。

- **程序流程图**

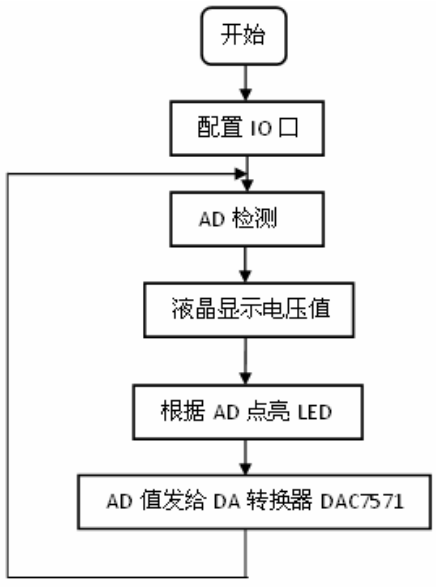


图 4.2.10 程序流程图

- **关键代码分析**

实验 DAC 代码

主函数

```
int main(void) {
    int i=180;
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    P8DIR |= BIT1;
    P8OUT &=~ BIT1;
    initClock();
    PaperIO_Int();
    INIT_SSD1673();
    Init_buff();
    display(" DAC OUTPUT ", 82, 4,TimesNewRoman,size8,1,0);
    while(1)
    {
        DACValue(0XFFF);
     P8OUT ^= BIT1;
     display("DAC: 3.3V ", 82, 42,0,0,1,0);                    //屏幕显示电压
     DIS_IMG(1);
        delay1(800);
        DACValue(0X7FF);
     P8OUT ^= BIT1;
     display("DAC: 1.65V ", 82, 42,0,0,1,0);                   //屏幕显示电压
     DIS_IMG(1);
        delay1(800);
    }

}
```

4.2.1.5 实验步骤与现象

- **实验步骤**

1. 构思好编程思路后，画出流程图；
2. 根据流程图在工程的主函数中完成代码编写；
3. 调试编译程序，完善代码，解决问题；
4. 将程序烧入开发板中测试效果，用万用表如下图测试 DAC OUT（口袋板右侧扩展插针 P3.5）输出电压。

- **实验现象**

程序将产生介于 1.65V 与 3.3V 之间跳动的方波，周期 40 秒，可以用万用表测量 DAC OUT（口袋板右侧扩展插针P3.5）输出电压，电压值同时在口袋板屏幕上显示。

4.2.2 Lab4-2-2 TMP421 数字温度传感器实验

4.2.2.1 TMP421 介绍

TMP421 是一款远程温度传感器控制芯片，内置一本地温度传感器，与 I²C 和 SMBus 串行总线兼容，实现远程控制。如下图所示，通过 DXP、DXN 连接一个三极管或二极管组成一个远程温度传感器，远程温度传感器最大精度为±1℃；本地温度传感器最大精度为±1.5℃。本次实验是通过 I²C 来控制 TMP421 的温度采集的，将格式设为扩展二进制，温度检测范围为-64～191℃。通过配置 TMP421 的配置寄存器来初始化，然后通过读取它的本地温度寄存器和远程温度寄存器来获取温度数据。

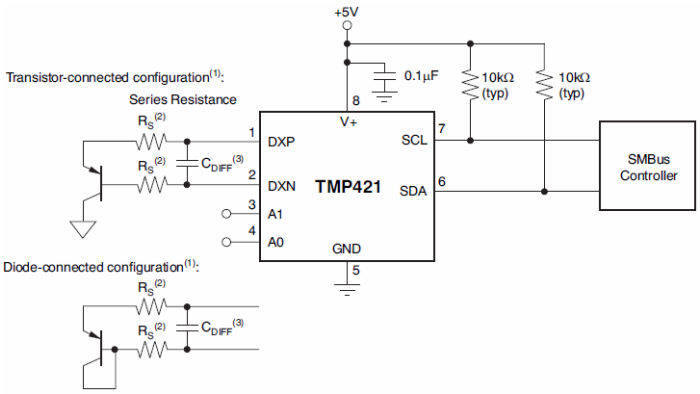


图 4.2.11　TMP421 芯片管脚连接

TMP421 可以测量本地温度（芯片的温度），还可以测量远程温度——用两根等长导线连接一个 PNP 型三极管 C9012（TO-92）即可，三极管端的温度即为远程温度。在口袋板上我们可以用 2pin 杜邦线分别连接芯片的 DXN 与 DXP 端(口袋板右侧扩展插针 P3.6 与 P3.7)，杜邦线的另一头连接 C9012 三级管的基极+集电极（b+c）与发射极（e 极），即：DXN（P3.6）连接 b 和 c，DXP（P3.7）连接 e，这种用法实际是把三极管当做一个二极管来用。

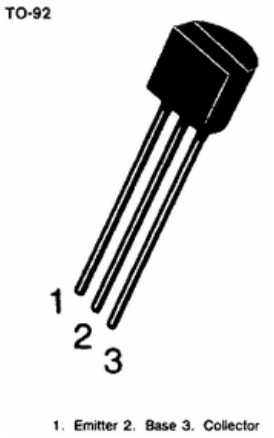


图 4.2.12　C9012 三极管管脚定义

第 5 章 H 桥驱动与电流检测

5.1 H 桥驱动电路

电机或高功率 LED 灯运性时需要的电流比较大，单片机普通 IO 口的驱动能力是不够，所以我们驱动这些器件需要加入驱动电路，典型的驱动电路最就是 H 桥驱动电路。F5529 口袋板上没有配置电机，只预留了电机扩展接口，如果你不外扩电机，只使用板上的高功率 LED 就没有必要看下面罗里吧嗦的电机控制部分了。

5.1.1 H 桥驱动器工作原理：

下图所示为一个典型的直流电机控制电路。电路得名于“H 桥驱动电路”是因为它的形状酷似字母 H。4 个三极管组成 H 的 4 条垂直腿，而电机就是 H 中的横杠。如图所示，H 桥式电机驱动电路包括 4 个三极管和一个电机。要使电机运转，必须导通对角线上的一对三极管。根据不同三极管对的导通情况，电流可能会从左至右或从右至左流过电机，从而控制电机的转向。

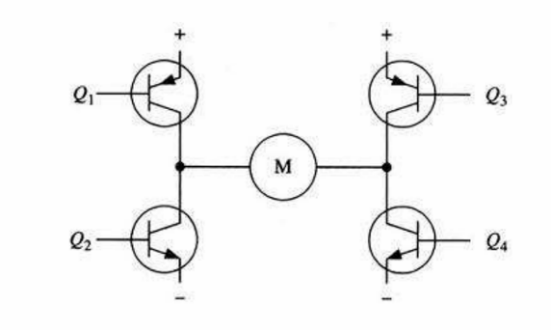


图5.1.1 H 桥驱动电路

要使电机运转，必须使对角线上的一对三极管导通。如下图所示，当 Q1 管和 Q4 管导通时，电流就从电源正极经 Q1 从左至右穿过电机，然后再经 Q4 回到电源负极。按图中电流箭头所示，该流向的电流将驱动电机顺时针转动。当三极管 Q1 和 Q4 导通时，电流将从左至右流过电机，从而驱动电机按特定方向转动。

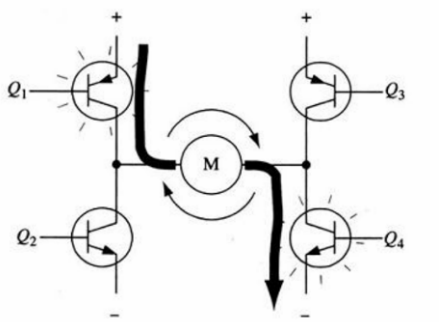


图5.1.2 H 桥驱动电机顺时针转动

下图所示为另一对三极管 Q2 和 Q3 导通的情况，电流将从右至左流过电机。当三极管 Q2 和 Q3 导通时，电流将从右至左流过电机，从而驱动电机沿另一方向转动。

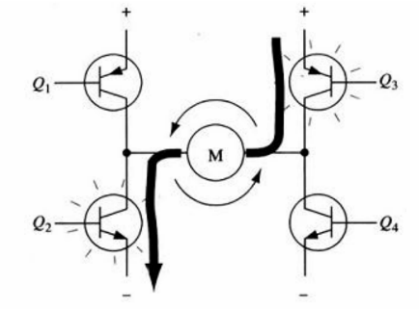


图5.1.3 H 桥驱动电机逆时针转动

驱动电机时，保证 H 桥上两个同侧的三极管不会同时导通非常重要。如果三极管 Q1 和 Q2 同时导通，那么电流就会从正极穿过两个三极管直接回到负极。此时，电路中除了三极管外没有其他任何负载，因此电路上的电流就可能达到最大值（该电流仅受电源性能限制），甚至烧坏三极管。基于上述原因，在实际驱动电路中通常要用硬件电路方便地控制三极管的开关。

下所示就是基于这种考虑的改进电路，它在基本 H 桥电路的基础上增加了 4 个与门和 2 个非门。4 个与门同一个“使能”导通信号相接，这样，用这一个信号就能控制整个电路的开关。而 2 个非门通过提供一种方向输人，可以保证任何时候在 H 桥的同侧腿上都只有一个三极管能导通。（与本节前面的示意图一样，图 5.1.4 所示也不是一个完整的电路图，特别是图中与门和三极管直接连接是不能正常工作的。）

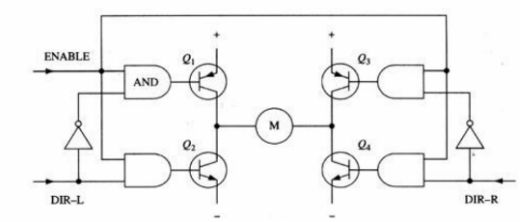


图5.1.4 具有使能控制和方向逻辑的 H 桥电路

采用以上方法，电机的运转就只需要用三个信号控制：两个方向信号和一个使能信号。如果 DIR－L 信号为 0， DIR－R 信号为 1，并且使能信号是 1，那么三极管 Q1 和 Q4 导通，电流从左至右流经电机（如 下 图所示）；如果 DIR－L 信号变为 1，而 DIR－R 信号变为 0，那么 Q2 和 Q3 将导通，电流则反向流过电机。

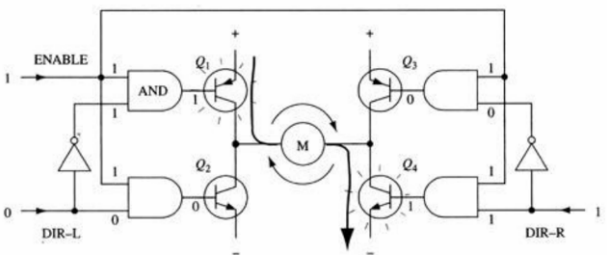


图5.1.5 使能信号与方向信号的使用

实际使用的时候，用分立元件制作 H 桥是很麻烦的，好在现在市面上有很多封装好的 H 桥集成电路，接上电源、电机和控制信号就可以使用了，在额定的电压和电流内使用非常方便可靠。

实验系统的电机模块包含直流电机与步进电机各一个，MSP430F5529 芯片 GPIO 口输出 PWM 波形对这两个电机的转速、转向、步长等进行控制。

5.1.2 低压 H 桥控制器 DRV8837：

口袋板上使用了 TI DRV8837 低电压电机驱动芯片，芯片的基础构造就是用的前面介绍的 H 桥。DRV8837 的主要特性与优势：

●延长电池使用寿命：280 毫欧低 RDS(ON)与仅为 35 nA 的超低睡眠电流可为玩具、智能气表或水表、电子锁、微型打印机以及摄像机等应用提高散热性能，延长电池使用寿命；

●提高性能：逻辑与电机的分开供电可提高启动及停止条件下电池供电应用的性能；

●宽/低电压工作范围：1.8 V 至 11 V 宽泛工作电压支持多达 6 节碱性电池或 2 节锂离子电池组应用；

●减少板级空间：微小型 2 毫米 x 2 毫米 WSON 封装支持更小、更时尚的设计；

●高级片上保护：过流、过温、贯通以及欠压保护可提高系统可靠性，降低设计复杂性。

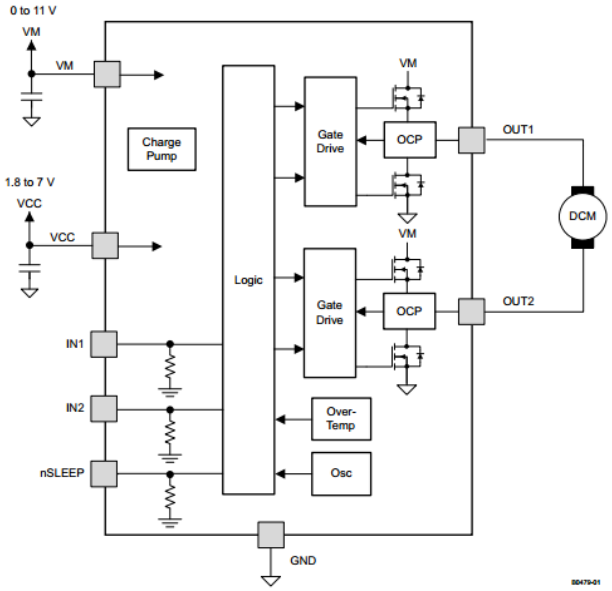


图5.1.6 TI DRV8837 芯片内部结构图

5.1.3 Lab5-1-1 PWM 控制高亮 LED 实验

5.1.3.1 实验介绍

功率型 LED 发光时需要电流比较大,所以需要外加驱动电路进行驱动，DRV8837 是一块

大功率驱动器在前面已经介绍过这块驱动芯片。本实验中把 LED 驱动电路和电机驱动共用一个模块，在口袋板上有跳线进行功能选择。

5.1.3.2 实验原理

- LED 驱动：

  驱动电路采用的是电机驱动电路，详细原理请看前面章节。

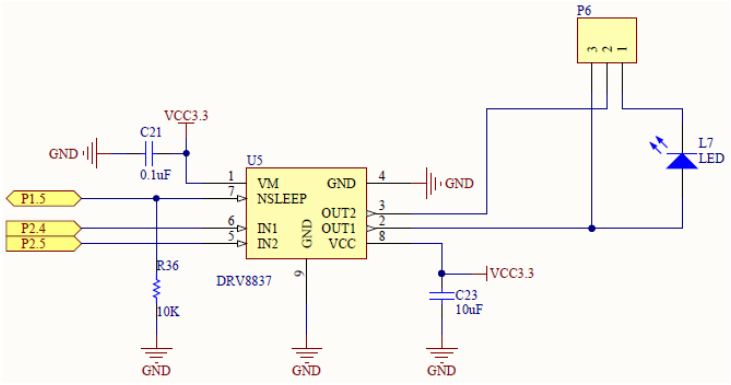


图 5.1.7　LED 驱动电路

- 功率调节：

  LED 正常供电的情况下功率固定，发光也是固定的，我们想要降低它的平均功率只有一个方法，那就是通过开断供电电源的方式来调低 LED 的功率。如果开端的频率比较低那么我们就会发现灯亮一下灭一下，非常影响使用，所以我们要非常快速的进行开端切换，这样就两全其美了，既能降低功耗，又能保持人眼看到常亮状态。一定频率的开断，开断的比例可调节这就是我们常用的 PWM。DRV8837 有个控制引脚 NSLEEP，把该引脚置高驱动电路生效，把该引脚置低，驱动电路失效。我们把 PWM 信号输出引脚接到 NSLEEP 引脚上，就可以随意调节功耗了。

5.1.3.3 程序分析

- **编程思路**

  首先按照驱动电机的原理，让 LED 满额长亮。然后用设置 PWM 占空比，用 PWM 信号控制电机驱动电路工作和停止。通过调节占空比就可以控制功耗了。

- **实验现象**

  高功率 LED 灯点亮。

5.2 电流检测

电流检测有很多种方法，比如使用运放检测精密电阻两端的电压差值，使用霍尔型电流传感器等等。口袋板上用的是 TI INA210 电流检测芯片，工作原理就是用运放测量精密电阻

两端的压降，然后输出一个电压值送到 F5529 的 ADC 端口，完成电流的定量检测。

5.2.1 口袋板电流检测模块：

5.2.1.1 INA210 电流检测芯片

口袋板上采用的电流检测芯片是 TI INA210，其内部结构如下图所示：

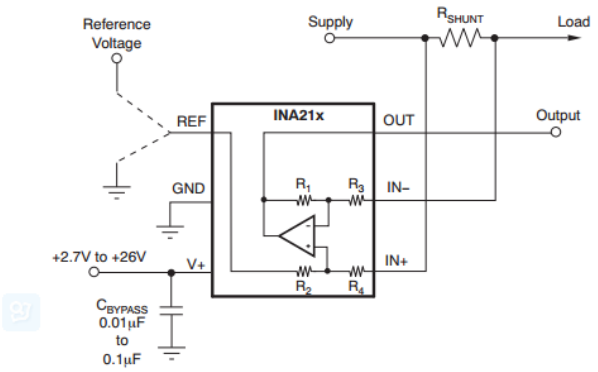


图 5.2.1　INA210 结构框图

INA210 输入引脚-IN 和+IN，分别连接到口袋板左侧扩展插针 P4.1 与 P4.2 上。本实验时采用 INA210 单项测量方式（REF 管脚接地），INA210 能够测量从一个方向流经一个阻性分路的电流。

5.2.1.2 电流检测电路

口袋板上使用的 0.01 欧姆的精密电阻，能够检测的电流范围是 0~300mA，超过这个电流，可能会对器件造成损坏。转换关系大约是流经 INA210 的电流是 1mA，V_OUT 端电压是 10mV。

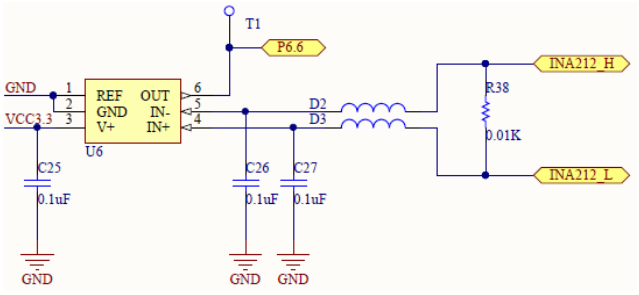


图 5.2.2　电流检测电路

5.2.2 Lab5-2-1 电流检测实验：

5.2.2.1 程序分析

- **编程思路**

把电流测量模块接入电路中后，开启 AD 转换，读出电压值，然后经过计算，算出电流值，最后把该值在液晶上显示出来。

- **关键代码分析**

选择AD通道6进行AD采样。

```
void ADInit(void)
{
    ADC12CTL0 |= ADC12MSC;                  //自动循环采样转换
    ADC12CTL0 |= ADC12ON;                   //启动ADC12模块
    ADC12CTL1 |= ADC12CONSEQ1 ;             //选择单通道循环采样转换
    ADC12CTL1 |= ADC12SHP;                  //采样保持模式
    ADC12MCTL0 |= ADC12INCH_6;              //选择通道6，连接拨码电位器
    ADC12CTL0 |= ADC12ENC;
    ADC12CTL0 |= ADC12SC;
}
```

读取AD值并计算，得到对应电压值，电压值/10,就会得到对应的电流值。

```
ADvalue = ADC12MEM0;
V1=ADvalue;
V1=V1/4095*3300;
V2=V1;
```

改变电路中的电流，通过上个实验功率型LED，把该模块接入到电流测量电路中，改变LED亮度实现电流变化。

```
while(1)
{
 ADvalue = ADC12MEM0;                       //读取AD值
 V1=ADvalue;
    V2=(V1/4095*3300)*0.98;                 //计算电压值
    t[0]=disp[V2/1000];
    t[1]=disp[V2/100%10];
    t[2]=disp[V2%100/10];
    t[3]='\n';
    display(" I =   ", 42, 48,TimesNewRoman,size8,1,0);
    display(t, 80, 48,TimesNewRoman,size8,1,0);
    display(" mA   ", 104, 48,TimesNewRoman,size8,1,0);
```

5.2.2.2 实验现象

- 高功率 LED 灯点亮，流经 LED 的电流值在屏幕上显示。

第 6 章　音频信号处理与 D 类放大器

音频信号是一种重要的信息载体，将声音无失真的获取，保存，传送，再现等等技术是众多电子工程自 19 世纪起就不断研究，直至今天仍有不断创新与突破。由于音频信号容易获取，感知与判别，我们在 F5529 口袋实验板上设置了声音采集，处理与回放模块，构成一个完整的音频处理系统并设计了一系列实验，通过对音频信号的处理，让同学们学习掌握相关的知识与技术。音频实验大多可以听到动听的声音------当然，做错了除外。因此，相信这些实验会比只能看到 LED 闪烁或是只能用示波器观察实验结果的实验项目更能引起同学们的兴趣。

对于音频信号的处理可以粗略的分为模拟与数字两大类，这一章侧重模拟处理，下一章侧重数字处理。通过这一章的学习，可以学到小信号放大，滤波，功率放大等等重要的模拟电子技术。如果你是音乐发烧友，同时又具有一定的动手能力，认真学习这一章，你就可以自己动手设计组装一台音频放大器，将动听的音乐传播到整个房间，怎么样？赶快努力吧，加油！

6.1 音频信号放大流程

声音信号作为一种振动信号要经过麦克风这样的换能器件将动能转化为电信号，然后将这个微弱的电信号经过一系列处理与放大，才能最终推动扬声器，整个处理过程请看下面框图。

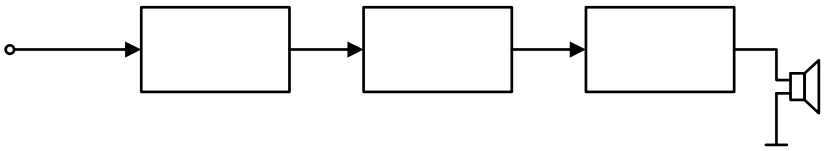


图 6.1.1　声音信号放大流程

从图上我们可以看到有前置放大、音调控制（滤波）、功率放大三个环节，下面我们一一介绍。

6.2 音频信号前置放大

同学们看到图 6.1 是不是感到有些困惑------这个前置放大与功率放大有啥区别啊？家里的音响功放机只有一台，没见分什么前置放大器与功率放大器？其实声音源的种类有多种，如传声器（话筒）、电唱机、CD 唱机及线路传输等，这些声音源的输出信号的电压差别很大，从零点几毫伏到几百毫伏。一般功率放大器的输入灵敏度是一定的，这些不同的声音源信号如果直接输入到功率放大器中的话，对于输入过低的信号，功率放大器输出功率不足，不能充分发挥功放的作用；如果输入信号的幅值过大，功率放大器的输出信号将严重过载失真。所以一个实用的音频功率放大系统必须设置前置放大器，以便使放大器适应不同的的输

入信号，或放大，或衰减，或进行阻抗变换，使其与功率放大器的输入灵敏度相匹配。

前置放大器的主要功能一是使话筒的输出阻抗与前置放大器的输入阻抗相匹配；二是使前置放大器的输出电压幅度与功率放大器的输入灵敏度相匹配。由于话筒输出信号非常微弱，一般只有 100 微伏~几毫伏，所以前置放大器输入级的噪声对整个放大器的信噪比影响很大。前置放大器的输入级首先采用低噪声电路，对于由晶体管组成的分立元件组成的前置放大器，首先要选择低噪声的晶体管，另外还要设置合适的静态工作点。由于场效应管的噪声系数一般比晶体管小，而且它几乎与静态工作点无关，在要求高输入阻抗的前置放大器的情况下，采用低噪声场效应管组成放大器是合理的选择。如果采用集成运算放大器构成前置放大器，一定要选择低噪声、低漂移的集成运算放大器。对于前置放大器的另外一要求是要有足够宽的频带，以保证音频信号进行不失真的放大。下图就是一个典型的麦克风用前置放大电路，放大器件使用的是 NE5532 集成运放芯片。

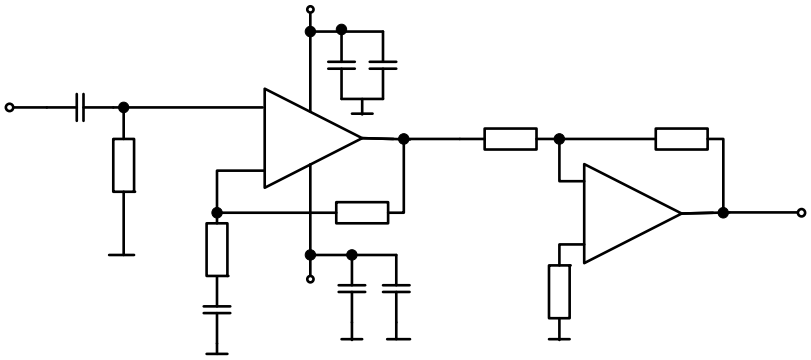
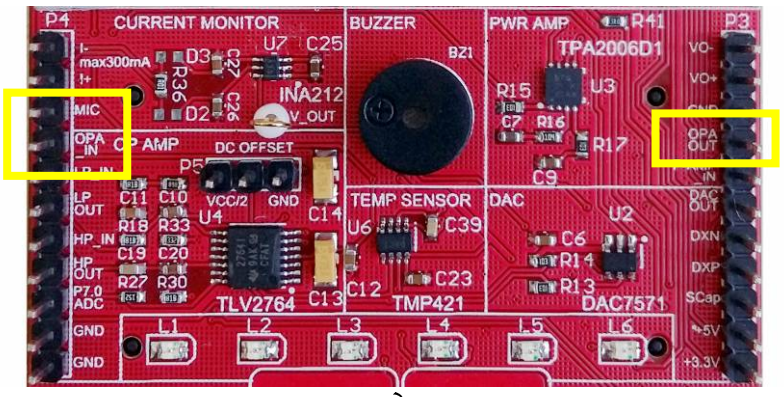


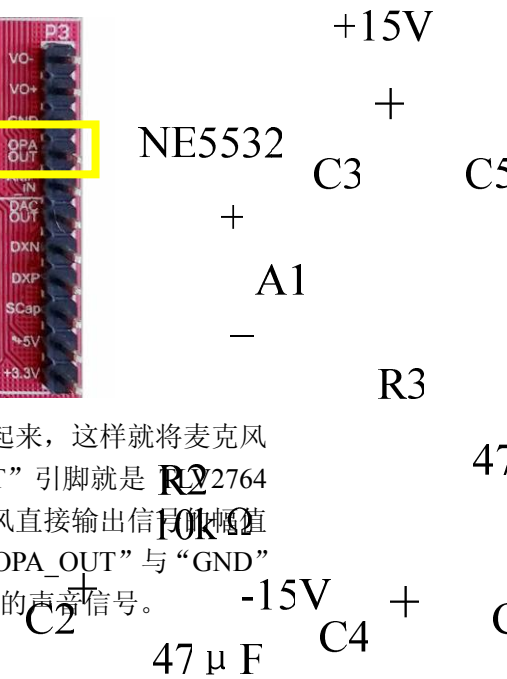
图 6.2.1　典型前置放大电路

在 F5529 口袋实验板上使用 TLV2764 集成运放作为麦克风信号的前置放大器，有兴趣的同学可以通过示波器观察一下麦克风输出信号的幅值与经过 TLV2764 放大后信号的幅值有什么区别。

Lab 6-2-1：麦克风信号的前置放大实验



将 F5529 口袋板左侧“MIC”与“OPA_IN”引脚用跳线帽短接起来，这样就将麦克风的输出信号与 TLV2764 运放的输入端连接起来，右侧的“OPA_OUT”引脚就是 TLV2764 运放的输出，我们可以用示波器观察此引脚的输出信号波形并与麦克风直接输出信号的幅值做对比。更简单的实验办法是用杜邦线将蜂鸣器的管脚分别连接在“OPA_OUT”与“GND”引脚上，这时我们应该能从蜂鸣器中听到麦克风采集经过前置放大后的声音信号。



6.3 音调控制（模拟滤波）

音调控制电路的主要功能是通过对放音频带内放大器的频率响应曲线的形状进行控制，从而达到控制放音音色的目的，以适应不同听众对音色的不同爱好。此外还能补偿信号中所欠缺的频率分量，使音质得到改善，从而提高放音系统的放音效果，音调控制本质是各种形式的滤波。在高保真放音电路中，一般采用的是高、低音分别可调的音调控制电路。一个良好的音调控制电路，要求有足够的高、低音调节范围，同时有要求在高、低音从最强调到最弱的整个过程中，中音信号（一般指 1kHz）不发生明显的幅值变化，以保证音量在音调控制过程中不至于有太大的变化。音调控制电路大多由 RC 元件组成，利用 RC 电路的传输特性，提升或衰减某一频段的音频信音调控制电路一般可分为衰减式和负反馈式两大类，衰减式音调控制电路的调节范围可以做得较宽，但由于中音电平也要作很大的衰减，并且在调节过程中整个电路的阻抗也在变化，所以噪声和失真较大。负反馈式音调控制电路的噪音和失真较小，并且在调节音调时，其转折频率保持固定不变，而特性曲线的斜率却随之改变。

由于集成运算放大器具有电压增益高、输入阻抗高等优点，用它构成有源低通与高通滤波器作为音调控制电路具有电路结构简单、工作稳定等优点，典型的电路结构如下图所示。其中电位器 Rp1 是高音调节电位器，Rp2 是低音调节电位器，电容 C 是音频信号输入耦合电容，电容 C1、C2 是低音提升和衰减电容，一般选择 C1=C2，电容 C3 起到高音提升和衰减作用，要求 C3 的值远远小于 C1。电路中各元件一般要满足的关系为：Rp1=Rp2，R1=R2=R3，C1=C2，Rp1=9R1。

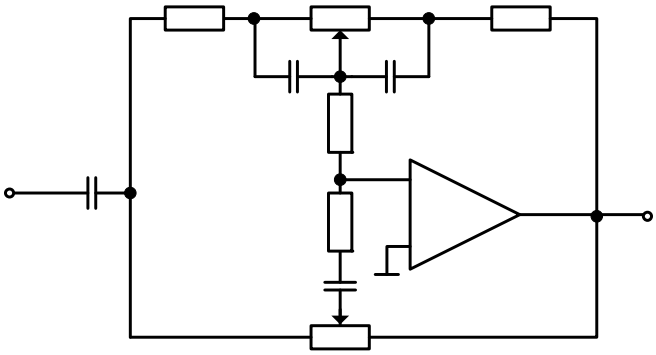
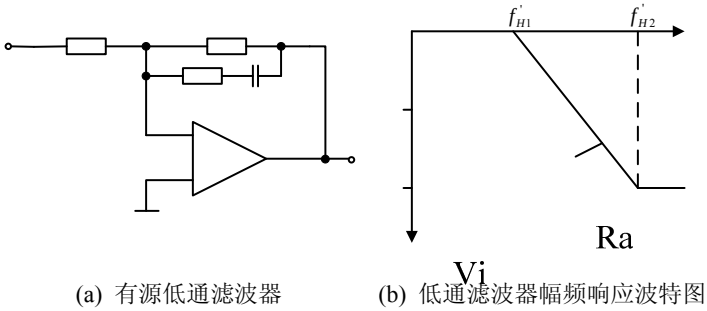


图 6.3.1　有源滤波器构成的音调控制器电路

6.3.1 有源低通滤波器

在上图中，对于低音信号来说，由于 C3 的容抗很大，相当于开路，此时高音调节电位器 Rp1 在任何位置对低音都不会影响。当低音调节电位器 Rp2 滑动端调到最左端时，C1 被短路，此时电路图可简化为下图。

R1



(a) 有源低通滤波器　　(b) 低通滤波器幅频响应波特图

图 6.3.2　低通滤波器电路图及幅频响应曲线

口袋板上的二阶有源低通滤波器，截止频率 8KHz。低通滤波器的偏置电压可以通过跳线在 0V 与 1/2VCC 间进行切换，有无直流偏置的信号都可以输入。滤波器输入输出接口分别是口袋板左侧扩展插针 P4.5 LP_IN 与 P4.6_LP OUT。

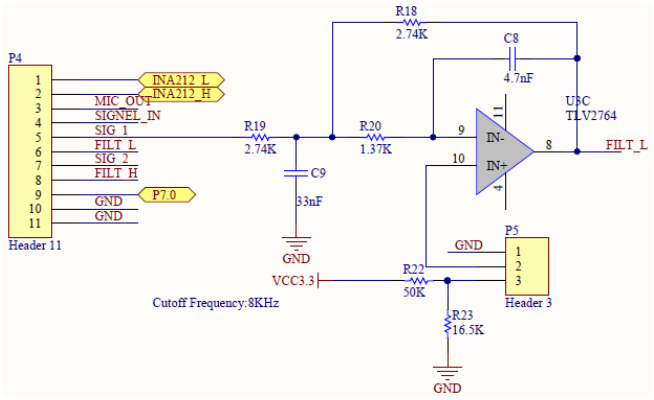
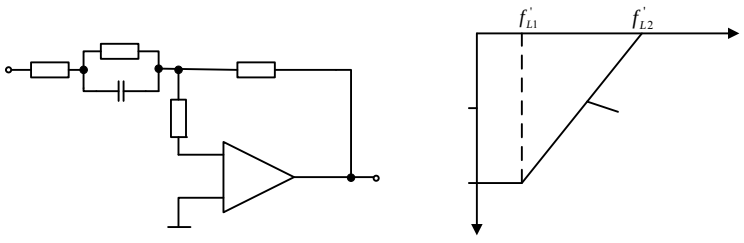


图 6.3.3　低通滤波器电路

6.3.2 有源高通滤波器

将图 6.3.2 的电路拓扑结构做一定改变就可以将低通滤波器改为高通滤波器。



(a) 有源高通滤波器　　(b) 高通滤波器幅频响应波特图

图 6.3.4　高通滤波器电路图及幅频响应曲线

口袋板上的二阶有源高通滤波器，截止频率 500Hz，滤波器输入输出接口分别是口袋板左侧扩展插针 P4.7 HP_IN 与 P4.8_HP OUT。

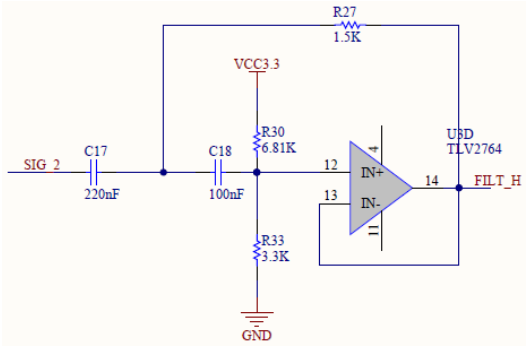
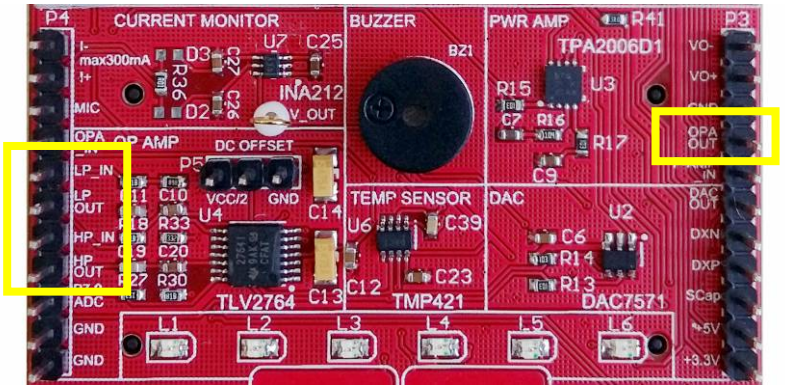


图 6.3.5　高通滤波器电路

Lab 6-3-1：音频信号的音调调理实验



F5529 口袋板上的 TLV2764 还组成了 2 个二阶有源滤波器，一个低通，一个高通，如果同学们想尝试音调调整的效果，可以将“OPA_OUT”与左侧低通滤波器的输入引脚“LP_IN”连接或与高通滤波器的输入引脚“HP_IN”连接，相应的滤波器输出分别为“LP OUT”与“HP OUT”引脚，同学可以比较一下经过滤波与未经过滤波的信号听上去有什么不同，做这个使用时一定要使用无源音箱或耳机，不能使用蜂鸣器代替喇叭，因为蜂鸣器的频响带宽太低。

6.4 音频信号功率放大

经过前置放大与音调调整，声音信号可能已经很好听了，但是这个好听的信号幅值比较小，推动耳机等小功率的设备还可以，但很多时候我们希望将好听的音乐通过音箱播放出来与小伙伴们一同分享------独乐了不如众乐乐嘛！这种情况下我们就需要对音频信号进一步放大，也就是我们常说的功率放大，使其有足够的能量推动落地音箱等大家伙。好的音频放大机是音乐发烧友眼中的宝贝，让我们也来一睹它的芳容，看看其内部构造。

图 6.4.1　Marantz 立体声音频放大机

上图所示的是一款每声道额定输出功率 100W 的立体声前置放大与功率放大一体机，可以对 CD 机、唱机输出的小信号进行放大，推动承载功率在 200W 以内的大型落地的音箱------这个功率在家里开派对足够了，前提还得是你的邻居没有意见！有意思的是，这台机器还可以将仅仅经过前置放大与音调调理后的“半成品信号”通过“前置放大”端子输出。输出一中间信号干啥呢？这是为了给更加发烧的发烧友用来接驳档次更高，功率更大的功率放大器用的！超级音乐发烧友的想法我们永远不懂，我们这么细致的将也不是为了让你也去发高烧（那可以很费钱的哟），而是证明在音响系统中的确有前置放大器与功率放大器之分，不信的话我们可以看看这台 Marantz 放大机的后面板。

图 6.4.2　Marantz 放大机后面板

图中黄色框标示的“PRE-OUT”端子就是前置放大器（包含音调调理电路）输出端子，蓝色框标示的“SPEAKER”端子就是用于接驳扬声器的功率信号输出端子，由于功率信号电压高（能到几十伏特），电流大（能到几个安培），为了更好连接与散热，端子体积大很多。接下来再来看看这台放大机内容构造：

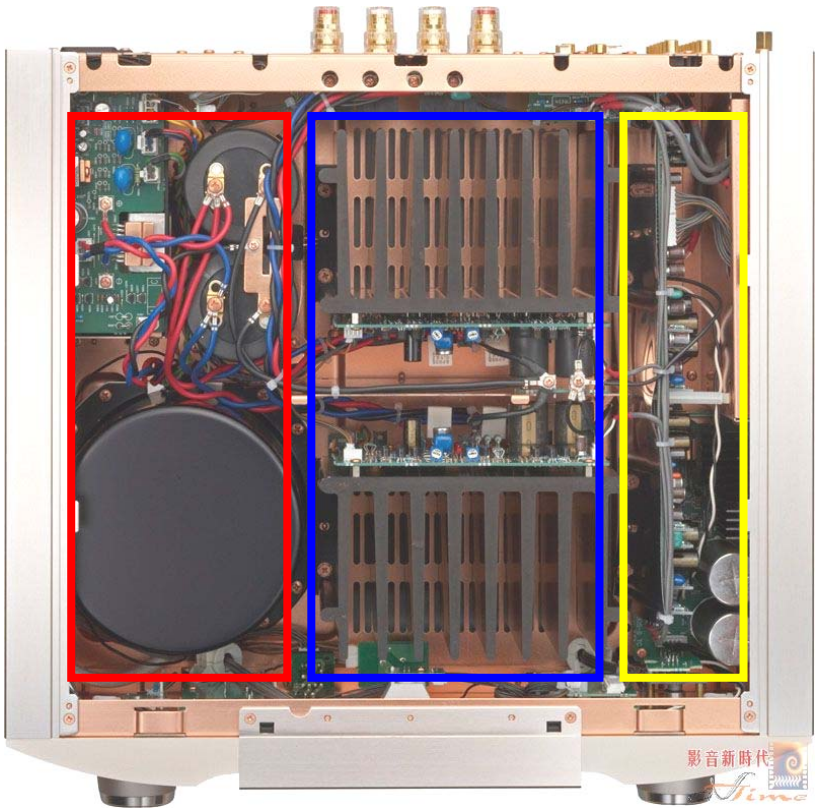


图 6.4.3　Marantz 放大机内部构造

图中黄色框标示的区域是前置放大与音调调理电路，蓝色框标示的区域是功率放大电路，红色框标示的区域是电源电路。

根据工作点不同，传统的音频功率放大器有 A 类（CLASS-A，甲类），B 类（CLASS-B，乙类），AB 类（CLASS-AB，甲乙类）三种，这三种形态的功放各种有其特点，下面我们加以简单介绍。

6.4.1 甲类功放

甲类 放大器 的输出晶体管(或电子管)的工作点在其线性部分中点，不论信号电平如何变化，它从电源取出的电流总是恒定不变，甲类功放的工作方式使得晶体管始终工作在线性区，因而具有最佳的线性，每个输出晶体管均放大讯号全波，完全不存在交越 失真（SwitchingDistortion），即使不施用 负反馈 ，它的开环路失真十分低，因此被称为是声音最理想的放大线路设计。甲类功放是重播音乐的理想选择，它能提供非常平滑的 音质 ，音色 圆润温暖，听感上质感特别好，尤其是小信号时，整个声音通透细节丰富。

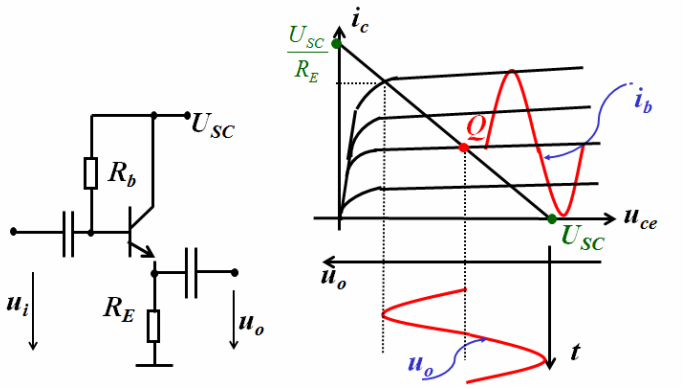


图 6.4.4 甲类功放示意图

甲类功放音色优美，失真极低，但也存在着很大的缺点，那就是发热量很大，效率极低。这是因为甲类功放输出级中的晶体管永远处于导电状态，也就是说不管有无讯号输入它们都保持传导电流，这些电能全部转为高热量。当讯号 电平 增加时，有些功率可进入负载，但许多仍转变为热量，甲类功放实际效率不超过 25%。

6.4.2 乙类功放

乙类功放由一对晶体管组成，放大的工作方式是当无讯号输入时，输出晶体管不导电，所以不消耗功率。当有讯号时，每对输出管各放大一半波形，彼此一开一关轮流工作完成一个全波放大，在两个输出 晶体管 轮换工作时便发生 交越失真 ，因此形成非线性。

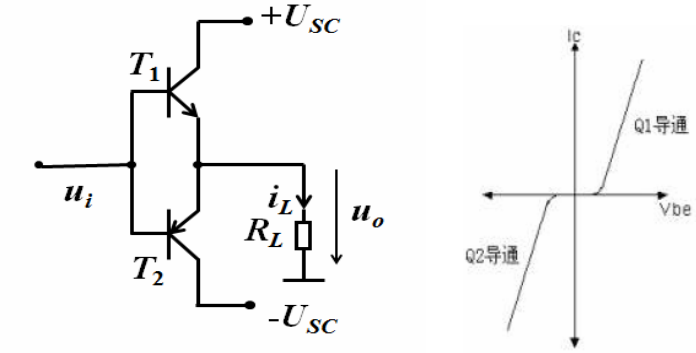


图 6.4.5　乙类功放及交越失真

乙类功放在音频放大中较少使用，因为在讯号非常低时失真十分严重，交越失真令声音变得粗糙。但是乙类功放的效率平均约为75%，产生的热量较甲类机低很多。

6.4.3 甲乙类功放

甲乙类放大器在低电平驱动时，放大器为甲类工作，当提高驱动电平时，转为乙类工作。甲乙类放大器的长处在于它比甲类提高了小信号输入时的效率，随着输出功率的增大，效率也增高，虽然失真比甲类大，然而至今仍是应用最广泛的晶体管功率放大器模式，趋向是越来越多的采用高偏流的甲乙类，减少低电平信号的失真。

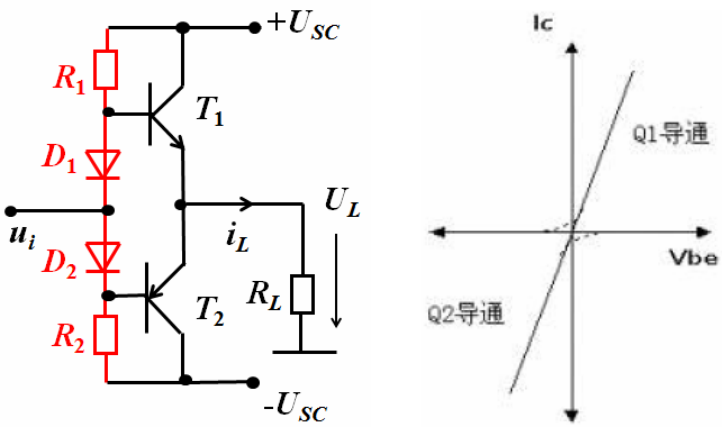


图 6.4.6　甲乙类功放及交越失真

可以看出，甲乙类功放实际就是甲类功放与乙类功放的综合，中和了二者的优缺点，因此甲乙类功放具有较好的音质与较高的效率。但是人们总是不知足，希望能够得到更好的音质与更高的效率，于是又逼着电子工程师开发出了 D 类（CLASS-D）功率放大器。

6.5 D 类音频信号功率放大器

6.5.1 D 类功放原理

D 类功放是一种不同于传统甲类、乙类、甲乙类功放的全新放大器形式。D 类功放使用脉宽调制技术技术（Pulse Width Modulation，PWM），将输入信号经过比较器装换为脉冲信号，然后利用开关管对脉冲信号进行放大，放大后的脉冲信号再经过低通滤波器恢复为模拟信号，因此 D 类放大器也可以称作是数字放大器。

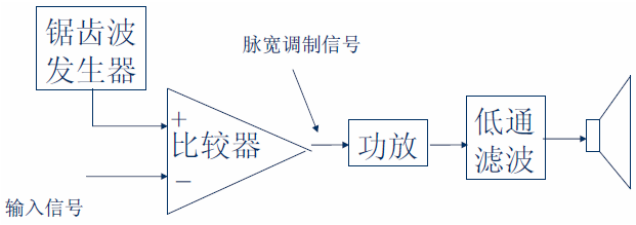


图6.5.1 D类放大器原理框图

从图中可以看出 D 类功放实际上只具有开关功能，早期仅用于继电器和电机等执行元件的开关控制电路中。然而开关功能（也就是产生数字信号的功能）随着数字音频技术研究的不断深入，用与 Hi-Fi 音频放大的道路却日益畅通。20 世纪 60 年代，设计人员开始研究 D 类功放用于音频的放大技术，70 年代 Bose 公司就开始生产 D 类汽车功放。一方面汽车用蓄电池供电需要更高的效率，另一方面空间小无法放入有大散热板结构的功放，两者都希望有 D 类这样高效的放大器来放大音频信号。其中关键的一步就是对音频信号的调制。

D 类放大器第一部分为调制器，最简单的只需用一只运放构成比较器即可完成。把原始音频信号加上一定直流偏置后放在运放的正输入端，另通过自激振荡生成一个三角形波加到运放的负输入端。当正端上的电位高于负端三角波电位时，比较器输出为高电平，反之则输出低电平。若音频输入信号为零、直流偏置三角波峰值的 1/2，则比较器输出的高低电平持续的时间一样，输出就是一个占空比为 1：1 的方波。当有音频信号输入时，正半周期间，比较器输出高电平的时间比低电平长，方波的占空比大于 1：1；负半周期间，由于还有直流偏置，所以比较器正输入端的电平还是大于零，但音频信号幅度高于三角波幅度的时间却大为减少，方波占空比小于 1：1。这样，比较器输出的波形就是一个脉冲宽度被音频信号幅度调制后的波形，称为 PWM（Pulse Width Modulation 脉宽调制）或 PDM（Pulse Duration Modulation 脉冲持续时间调制）波形。音频信息被调制到脉冲波形中。

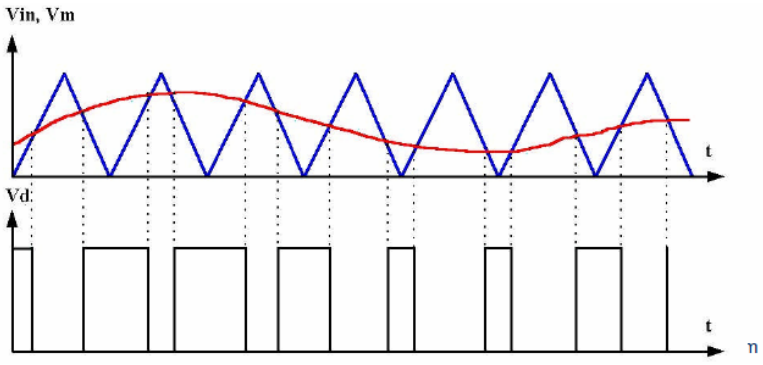


图6.5.2 PWM（脉宽调制）原理示意图

第二部分就是开关放大器，这是一个脉冲控制的大电流开关放大器，把比较器输出的PWM信号变成高电压、大电流的大功率PWM信号。能够输出的最大功率有负载、电源电压和晶体管允许流过的电流来决定。

第三部分需把大功率PWM波形中的声音信息还原出来。方法很简单，只需要用一个低通滤波器。但由于此时电流很大，RC结构的低通滤波器电阻会耗能，不能采用，必须使用LC低通滤波器。当占空比大于1：1的脉冲到来时，C的充电时间大于放电时间，输出电平上升；窄脉冲到来时，放电时间长，输出电平下降，正好与原音频信号的幅度变化相一致，所以原音频信号被恢复出来。

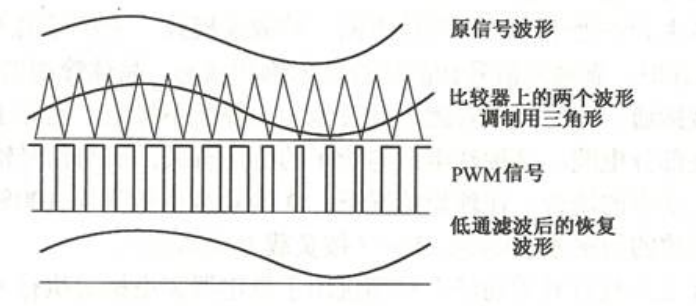


图6.5.3 PWM（脉宽调制）信号的波形恢复

6.5.2 D 类功放特点与效率分析

6.5.2.1 D 类功放电路特点

D 类功放设计考虑的角度与 AB 类功放完全不同。此时功放管的线性已没有太大意义，更重要的开关响应和饱和压降。由于功放管处理的脉冲频率是音频信号的几十倍，且要求保持良好的脉冲前后沿，所以管子的开关响应要好。另外，整机的效率全在于管子饱和压降引起的管耗。所以，饱和管压降小不但效率高，功放管的散热结构也能得到简化。若干年前，这种高频大功率管的价格昂贵，在一定程度上限制了 D 类功放的发展。现在小电流控制大电流的 MOSFET 已普遍运用于工业领域，特别是近年来 UHC MOSFET 已在 Hi-Fi 功放上应用，器件的障碍已经消除。

调制电路也是D类功放的一个特殊环节。要把20KHz以下的音频调制成PWM信号，三角波的频率至少要达到200KHz。频率过低达到同样要求的THD标准，对无源LC低通滤波器的元件要求就高，结构复杂。频率高，输出波形的锯齿小，更加接近原波形，THD小，而且可以用低数值、小体积和精度要求相对差一些的电感和电容来制成滤波器，造价相应降低。但此时晶体管的开关损耗会随频率上升而上升，无源器件中的高频损耗、谢频的取肤效应都会使整机效率下降。更高的调制频率还会出现射频干扰，所以调制频率也不能高于1MHz。同时三角波形的形状、频率的准确性和时钟信号的抖晃都会影响到以后复原的信号与原信号不同而产生失真。所以要实现高保真，出现了很多与数字音响保真相同的考虑。

还有一个与音质有很大关系的因数就是位于驱动输出与负载之间的无源滤波器。该低通滤波器工作在大电流下，负载就是音箱。严格地讲，设计时应把音箱阻抗的变化一起考虑进去，但作为一个功放产品指定音箱是行不通的，所以D类功放与音箱的搭配中更有发烧友驰骋的天地。实际证明，当失真要求在0.5%以下时，用二阶Butterworth最平坦响应低通滤波器就能达到要求。如要求更高则需用四阶滤波器，这时成本和匹配等问题都必须加

以考虑。

近年来一般应用的D类功放已有集成电路芯片，用户只需按要求设计低通滤波器即可。

6.5.2.2 D 类功放效率分析

在保证音质的前提下，传统放大电路中 AB 类电路的效率最高，对于不同种类的信号，效率能够达到 10-50%左右，与之相比，D 类放大器的效率又有了更高的效率。

音频信号源	峰值因素（dB）	实际平均输出功率（瓦）	效率 (%)	
			AB类放大器	D类放大器
正弦波	3	1	45	80
语音信号	9	0.25	24	80
音乐信号	15	0.063	12	75

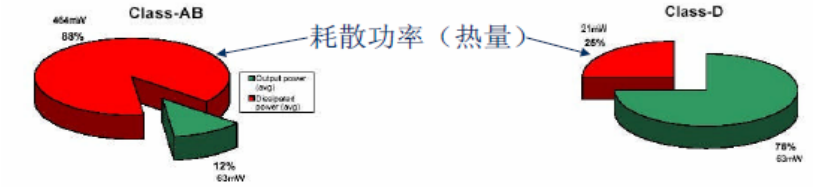


图6.5.4 D类放大器与AB类放大器效率比较

从图中我们可以看到D类功放与AB类功放比较效率上有了2-3倍提升，这对于电池供电的便携音频设备或是散热空间小的音频播放设备来说是再理想不过的了。

6.5.3 TI TPA2006D1 D 类音频功率放大器

F5529口袋板上使用了一颗TI公司的D类音频功放芯片，型号是TPA2006D1，该芯片具有1.45W的额定输出功率，最高效率能够解决90%左右，适合低功耗设备上使用。

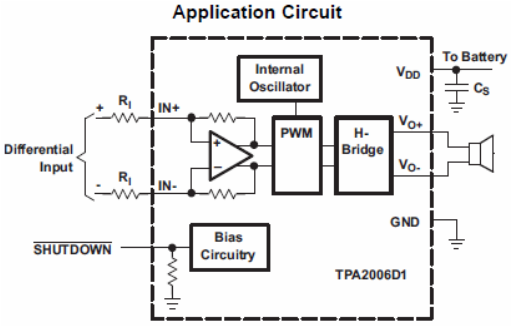


图6.5.5 TPA2006D1芯片结构框图

这颗芯片采用“无输出滤波器设计”，也就是输出端“VO+”与“VO-”可以直接连接4-8欧姆阻抗的扬声器而无需任何辅助元器件，因此电路上非常简单。F5529口袋板上也为TPA2006D1引出了输出管脚，有兴趣的同学可以用杜邦线连接一个4欧姆3W左右的无源小音箱，在距离1米左右的播放距离，可以听到清晰的声音。

还有一点值得注意，这颗运放芯片的输出采用的是桥接式负载输出（Bridge-Tied-load，BTL），也就是说负载的两端分别接在芯片内部两个放大器的输出端。这两个放大器的

互为镜像输出，也就是说加在负载两端的信号仅在相位上相差180°。负载上将得到原来单端输出的2倍电压。从理论上来讲电路的输出功率增加了3倍。BTL电路能充分利用系统电压，因此这种结构常应用于低电压系统或电池供电系统中。

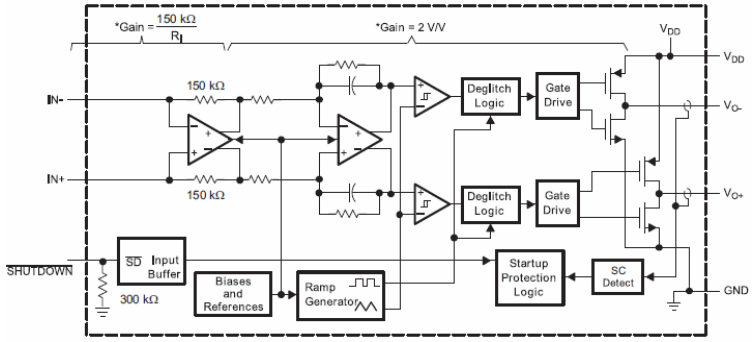
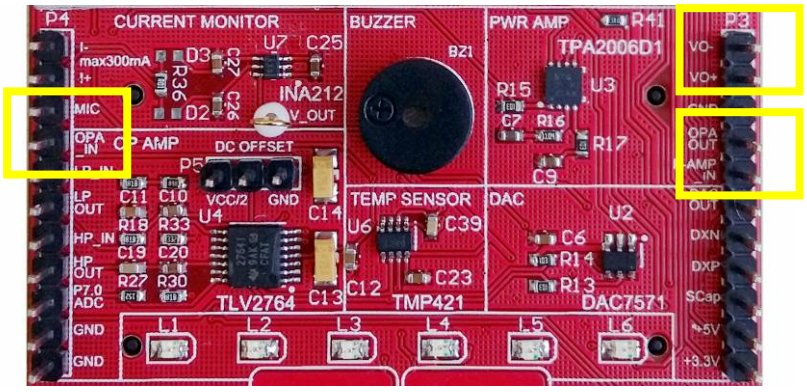


图6.5.6　TPA2006D1功放芯片采用BTL输出

从图中我们可以看到TPA2006D1内部有2个镜像的放大器，输出信号VO+与VO-形成镜像关系，采用BTL输出，能够使功放的驱动能力提升到非BTL形式的4倍。但是我们使用BTL输出的器件时要特别注意，只能将扬声器等负载接到BTL输出端VO+与VO-上，决不能将负载接到VO+与GND之间或VO-与GND之间，因为这样会有一个直流偏置电流一直加载在负载上，不但引起失真，时间长了，还会造成负载与TPA2006D1芯片发烫乃至烧毁。

Lab 6-5-1：麦克风信号的前置放大与功率放大实验



在 Lab 6-2-1 与 Lab 6-3-1 实验中，同学们已经掌握了对音频信号进行前置放大与音调调理的技术，我们通过耳机能够听到前置放大后的声音。但是如果我们想将声音放的更大，用一只音箱播放出来，仅仅通过前置放大就不够了，我们还要将信号进一部进行功率放大。

将 F5529 口袋板右侧的运放输出脚“OPA_OUT”与 TPA2006D1 功放输入引脚“P-AMP_IN”用跳线短接起来，这样经过前置放大的信号就进一步进行功率放大，放大后的信号可以推动无源蜂鸣器或是一个阻抗 4-8 欧姆，功率在 4W 以内的扬声器。由于麦克风与蜂鸣器的位置比较靠近，会形成自激，听到的是刺耳的噪声。因此请用 1 米左右的杜邦线将连接在功放两个输出引脚“VO+”与“VO-”上，使蜂鸣器或扬声器远离麦克风，避免自激发生。

有的同学可能会想，如果不经过前置放大，直接将麦克风输出信号接到 TPA2006D1 输入端会怎样呢？有这样的想法说明你认真思考了，非常好。F5529 口袋板开放出这么多信号引出脚就是为了让你进行各种“折腾”的，你可以用杜邦线将麦克风输出引脚“MIC”直接

连接到功放输入引脚“P-AMP_IN”上，看看你能听到什么。我猜想你什么都听不到，连刺耳的自激声都听不到，这不是因为器件或电路坏了，那是为什么呢？请再仔细思考一下，答案就在本章教材中。

Lab 6-5-2：耳机音乐播放实验

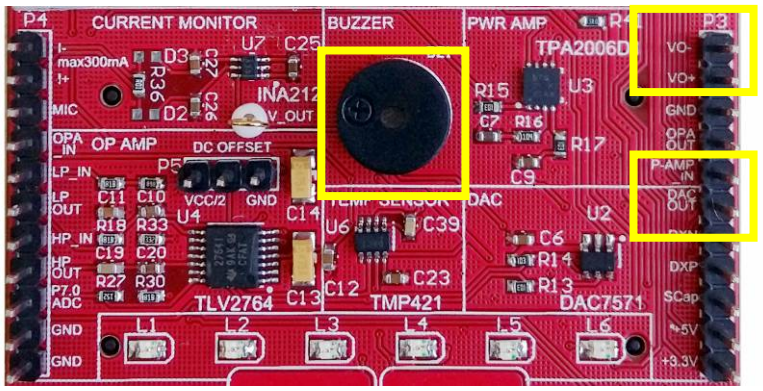
在 Lab 6-2-1、Lab 6-3-1 与 Lab 6-5-1 实验中，同学们估计已经听够了刺耳的自激声和并不算动听的“喂，喂，喂……”的对着麦克风的喊话，从这个实验开始我们就能听到动听的音乐了。

首先我们将音乐文件考入 TF 卡，运行音乐播放程序（这些请参考下一节内容），然后我们找一个耳机插入 F5529 口袋板的耳机插座，不需要任何连线，我们就能听到动听的音乐了！怎么样？很简单吧。之所以这样，是因为 F5529 口袋板的 DAC 输出端与耳机插座在电路上是直接连接的，这也就意味着板子右侧“DAC OUT”引脚与耳机输出端是相通的，都能得到 DAC 输出信号，并且我们可以通过拨盘电位器调节音量大小（模拟调节）。

除了使用耳机，我们也可以用蜂鸣器来听听 DAC 直接输出的音乐，这样做我们只需要用杜邦线将蜂鸣器的两个引脚分别接到板子右侧“DAC OUT”引脚与“GND”引脚上即可。

爱思考的同学可能又有问题了：DAC 输出的信号为啥不用经过前置放大就能驱动耳机能？这是因为 DAC 内部的输出缓冲级已经起到了前置放大作用，输出信号已经具备了一定的驱动能力，但这个驱动能力只局限于驱动耳机等 1W 以下小功率负载。

Lab 6-5-3：扬声器音乐播放实验



在刚才的实验中我们用耳机已经听到的优美动听的音乐，接下来我们将 DAC 输出的信号经过 TPA2006D1 进行功率放大，听听声音有什么不同。首先，我们还是用蜂鸣器做扬声器，在这个实验中不会出现自激，所以蜂鸣器不必用延长线延伸至远端，插在板子上即可。怎么样，声音是不是比刚才 DAC 信号直接推蜂鸣器时大了很多，清晰了很多？接下来，我们找一个无源小音箱再试试。什么？找不到？亲，花 11 元淘宝上买一对电脑小音箱，稍加改造，就能做成一个无源音箱外加一个有源音箱，就买下面这种音箱就行，很划算的哟！

图 6.5.7　小音箱（4 欧姆 3W）

将这个音箱用杜邦线连接在 F5529 口袋板右侧“VO+”与“VO-”引脚上，整个房间的小伙伴就能一同收听美妙的音乐了，怎么样，是不是很爽？

别光顾着高兴，我们还能让声音听上去更有趣一些，怎么做？很简单，结合 EX6-2 音调调节实验啊。具体说来就是将“DAC OUT”与“P-AMP_IN”之间的跳线帽去掉，将 DAC 输出的信号用杜邦线连接到低通或高通滤波器输入引脚上（LP_IN / HP_IN），然后将滤波器输出信号（LP_OUT / HP_OUT）连接到功放输入引脚“P-AMP_IN”上，怎么样，音调是不是的确出现了很大改变？

现在问题又来了，我们能否既经过低通又通过高通滤波呢？如果这样做我们是要先经过低通还是先经过高通？这些问题的答案还是留给爱动手与爱思考的同学们吧。

6.6 WAV 音频播放

6.6.1 WAV 文件

在实验 Lab 6-5-2 与 Lab 6-5-3 的时候我们已经用到了 WAV 文件的播放，但为了完整的将音频信号模拟处理技术讲完，我们将有关 WAV 音频播放的知识放在这一节来讲解。

WAV 为微软公司开发的一种声音 文件格式，被 Windows 平台及其 应用程序 所广泛支持，也是其音乐发烧友中常用的指定规格之一。由于此音频格式未经过压缩，所以在音质方面不会出现失真的情况，但档案的体积因而在众多音频格式中较为大。

我们通常使用三个 参数 来表示声音：量化位数（8~16bits），取样频率（8~44.1KS/s）和声道数（1~2Ch）。WAV 文件大小的计算方式为：文件所占容量=取样频率×量化位数×声道×时间，1 分钟 WAV 格式的 CD 音频文件（16bits，44.1KS/s，2Ch）的大小为 10.3MB，一张 CD 盘的容量是 700MB，因此每张音乐 CD 最多可以容纳 70 分钟左右的节目。

所有的 WAV 都有一个文件头，这个文件头音频流的编码参数。如下表所示：

地址	字节 数	类型	内容
00H~03H	4	字符	资源交换文件标志（RIFF）
04H~07H	4	长整数	从下个地址开始到文件尾的总字节数
08H~0BH	4	字符	WAV 文件标志（WAVE）
0CH~0FH	4	字符	波形格式标志（fmt）
10H~13H	4	整数	过滤字节（一般为 00000010H）
14H~15H	2	整数	格式种类（值为 1 时，表示数据为线性 PCM 编码）
16H~17H	2	整数	通道数，单声道为 1，双声道为 2
18H~1BH	4	长整数	采样频率
1CH~1FH	4	长整数	波形数据传输速率（每秒平均字节数）
20H~21H	2	整数	数据的调整数（按字节计算）
22H~23H	2	整数	样本数据位数
24H~28H	4	字符	“fact”,该部分一下是可选部分，即可能有，可能没有,一般到 WAV 文件由某些软件转换而成时，包含这部分。
29H~33H	4	长整数	size,数值为 4
34H~38H	4	长整数	data

表6.6.1　WAV 文件的文件头

WAV 音频格式的优点在于简单的编/解码（几乎直接存储来自 ADC 的信号）、普遍的认同与支持以及无损耗存储。WAV 格式的主要缺点是需要音频 存储空间 。对于小的存储限制或小带宽应用而言，这可能是一个重要的问题。WAV 格式的另外一个潜在缺陷是在 32 位 WAV 文件中的 2G 限制，这种限制已在为 SoundForge 开发的 W64 格式中得到了改善。

6.6.2 SD 卡中 WAV 文件操作

关于 SD 卡的操作可以参考 3.5.1 节相关内容。播放时需要使用 FAT 文件系统程序从 SD 卡中读出待播放的 wav 文件，解析其格式并将解码得到的数据按照其采样间隔送至 DAC 进行输出。WAV 文件是 RIFF 格式的一种，其开头为 RIFF 文件头，格式为：

```
struct RIFF_HEADER
{
charszRiffID[4]; // 'R','I','F','F'
uint32_tdwRiffSize;
charszRiffFormat[4]; // 'W','A','V','E'
} audio_riff_header;
```

其中 dwRiffSize 是去掉 szRiffID 和 dwRiffSize 之后的整个文件长度，也就是文件长度减 8。

RIFF 文件头之后则是若干 chunk，每个 chunk 的开头都是 4 个字节的 chunk id，然后是 4 个字节的 chunk size，size 同样是该 chunk 的总长减去 8. 一般来说，RIFF 文件头后应该是 format chunk，chunk 体可能是 16 或者 18 字节，定义如下

```
struct WAV_FMT_CHUNK
{
//RIFF_CHUNK_HEADER chHeader; // 'f','m','t',' '
uint16_twFormatTag;
uint16_twChannels;
uint32_tdwSamplesPerSec;
uint32_tdwAvgBytesPerSec;
uint16_twBlockAlign;
uint16_twBitsPerSample;
uint16_twExtInfo;
} audio_wav_fmt;
```

其中 wExtInfo 就是可选区段，是否存在可由 size 判断。

Format chunk 后可能有 Fact chunk，这一段对于本例的播放没有用处，可以直接跳过。

文件的最后一个 chunk 应当是 data chunk，在其 chunk 头后保存有音频数据，数据格式如下：

- 对于 8 位单声道，每个样本数据由 8 位(bit)表示；
- 对于 8 位立体声，每个声道的数据由一个 8 位(bit)数据表示，且第一个 8 位(bit)数据表示 0 声道(左)数据，紧随其后的 8 位(bit)数据表示 1 声道(右)数据；
- 对于 16 位单声道，每个样本数据由 16 位(bit)表示；其中低字节存放高位，高字节存放低位；
- 对于 16 位立体声，每个声道的数据由一个 16 位(bit)数据表示，且第一个 16 位(bit)数据表示 0 声道(左)数据，紧随其后的 16 位(bit)数据表示 1 声道(右)数据。