第1章 实验套件应用快速入门

杭州艾研信息技术有限公司

2014 年 11 月

申明

杭州艾研信息技术有限公司保留随时对其产品进行修正、改进和完善的权利，同时也保留在不作任何通告的情况下，终止其任何一款产品的供应的权利。用户在下订单前应及时获取相关信息的最新版本，并验证这些信息是当前的和完整的。

可通过如下方式获取最新信息、技术资料和技术支持：

技术支持电话：0571-86134572

技术支持邮箱：support@hpati.com

产品&资料下载中心：http://www.hpati.com/products/

互动论坛：http://www.hpati.com/bbs/forum.php

公司地址：浙江省杭州市西湖区留和路16号新峰商务楼B306

第1章 实验套件应用快速入门

本章以图表形式介绍实验套件的软硬件资源及其使用方式，并通过引导用户来做一个简单的验证性实验，实现对套件的工作模式有粗线条的概念。图表能更直观表达硬件组装和软件安装、配置的过程，提供读者按图索骥的方式,帮助用户快速入门,但图表也会因文字解释少而缺乏理论性。本章只在 CCS 工程的头文件引用以及 TivaWare 库中有关寄存器宏定义的部分规则作了解释，但仍请大家理解这些内容，它们是理解后续代码和在 Tiva Launchpad 上开发的基础。

实验套件特点介绍

本套件以Tiva Cortex M4 LaunchPad为核心模块，包含多个模拟应用模块，兼顾MCU的开发学习和模拟知识的理解与应用。套件中所有模块采用BoosterPack为统一硬件接口标准，并以此接口为总线的母板。这种设计方案为实验套件提供灵活的连接方式和无限制的可扩展性。套件设计时仔细考虑每个模块的功率，使得整个实验套件工作时只需通过标准USB端口取电。由于Launchpad自带调试器电路，MCU的程序调试和下载也通过同一USB端口进行。这样就简化实验过程的联线并降低对其它资源的依赖。每个模拟模块提供大量的跳线配置和测试点，便于学习模拟芯片的多种应用方式和电路参数测试。

操作接口简洁
模板轻便易携带

套件使用USB供电，板载仿真器，使得模块界面小巧，便于携带。

功能强大
覆盖大量模拟知识点

模块覆盖DC/DC、AD/DA转换
大量种类运放
及其应用方式

与Tiva更好结合

基于Launchpad的Boosterpack接口

数据测量方便

每个模拟模块都提供多种电路组成方式的配置选择，同时为知识点学习设计大量测试点。

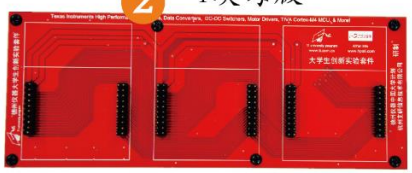
易扩展和
不断扩充的模块

灵活的连接方式为后续模块的扩展提供了方便，近期将陆续推出高速信号模块组和医疗电子模块组

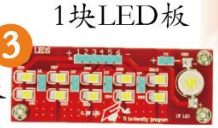
套件组成

1 8个基本模拟模块

2 1块母版



3 1块LED板



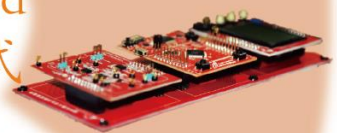
1块Tiva LaunchPad 开发板

5 干跳线帽

模块与Launchpad的结合使用方式

1 launchpad与LCD模块

2 三个模块共用联接方式

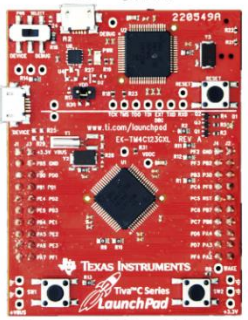


3 四个模块共用联接方式

注意

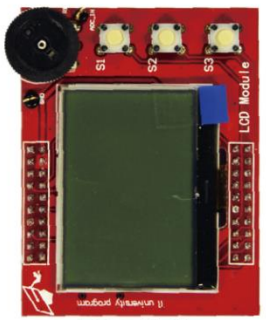
不同的模似模块之间可能存在BoosterPack兼容性问题，当两个模块同时使用一个引脚时，可能会发生冲突。LCD模块可与任何其它模块兼容，但除LCD外的模块是否兼容，要仔细比对本章的“BoosterPack引脚排序表”

Tiva Cortex M4最简系统，
USB接口调试器，
USB通信接口



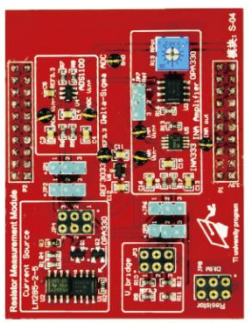
LaunchPad模块

提供实验时需要的
显示、按键、
SD卡存储和蜂鸣器



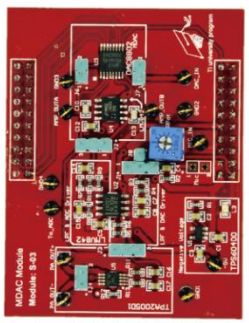
LCD模块

恒流源，电桥，
仅用运放，Δ-∑型ADC，
电荷泵式负电源



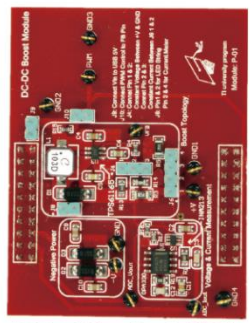
电阻测量模块

音频程控增益，
音频输入滤波，
音频输出D类功放电路



程控增益模块

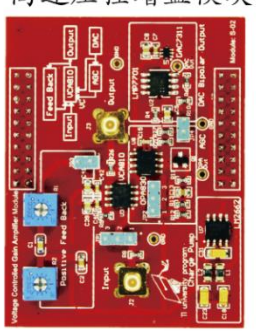
异步 Boost 拓扑，
恒流 LED 驱动电路，
无反馈升压负压拓扑。



DC-DC- 升压模块

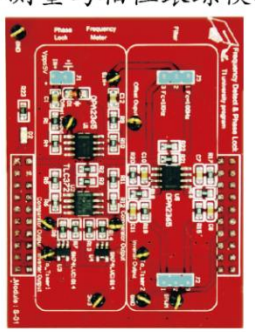
套件的基本模块配置

高速压控增益模块



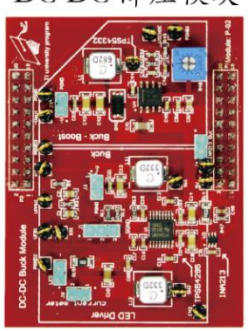
宽带压控增益放大器，
闭环自稳幅电路，
高速有源 RC 振荡器，
非直流信号作电压控制源，
R-String DAC 的应用

频率测量与相位跟踪模块



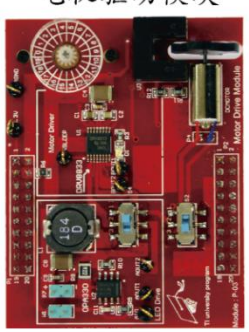
单电源运放，有源滤波器，
波形产生与变换，
基于 DDS 原理的
相位和频率跟踪算法

DC-DC 降压模块



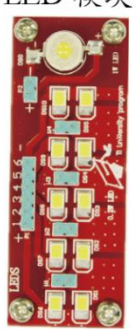
同步 Buck 拓扑，
降压恒流 LED 驱动，
多种输出调节方式，
Buck 到 Boost 拓扑的转变。

电机驱动模块



步进电机驱动，
直流电机驱动，
电机测速，电机驱动芯片
用作 DC-DC 电源芯片。

LED 模块



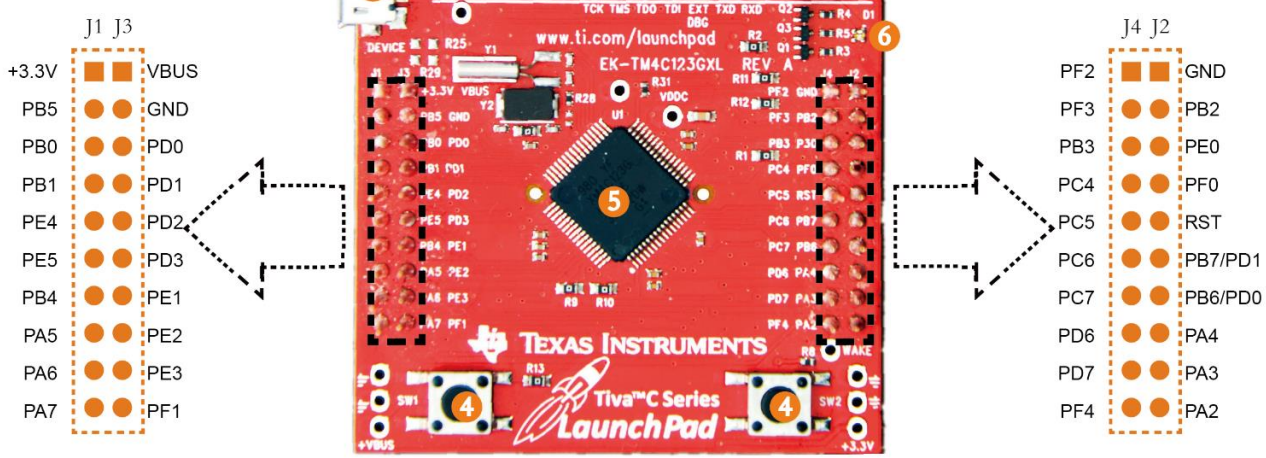
做为 DC-DC 升压、
DC-DC 降压
和电机驱动模块
的负载

TIVA Launchpad

TI的Launchpad是分别以MSP430,C2000,Cortex M4等MCU为核心组成小系统的一系列板卡，它们以低成本，易于调试，便于携带，强大的可扩展性等多方面优点而广受好评。Launchpad为对外扩展提供了一组接口，这组接口被命名为BoosterPack。TIVA Launchpad的BoosterPack是一共40个引脚，分别被命名为J1,J2,J3,J4. Launchpad和BoosterPack在英文里分别是火箭发射平台和火箭助推器。TI之所以这么命名其产品,估计也是希望用户对MCU系统及外围模拟电路的认识和开发经验就会如同火箭发射平台（LaunchPad）上加了助推器（Booster）的火箭，迅猛升空，这也是我们这套实验设备的愿景。

Tiva C系列具有以下特性

1. 支持USB方式的供电及调试
2. 输入电源可选：从USB调试端口取电
   或从USB通信端口取电
3. 支持USB 2.0的主、从、OTG数据通信
4. 两个通用按键，一个复位按键
5. ARM Cotex-M4F处理器，
   最高支持80MHz, 32KB SRAM, 256KB Flash
6. 配备RGB三色LED

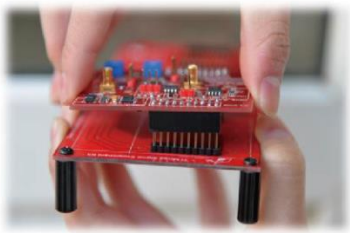


更多有关Launchpad BoosterPack的资料，请查阅Tiva™ C Series EK-TM4C123GXL LaunchPad: BoosterPack Development Guide，文档编号：SPMU288A – August 2012 – Revised April 2013

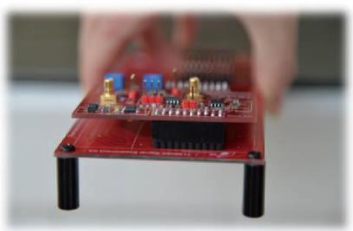
模块的安装与卸载

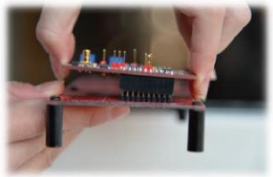
在母板上安装模块

1 设法释放手上的静电，用手指抓紧模块两端，对齐模块排母与母板排针的位置，错位的安装可能带来不可预料的错误与损坏。



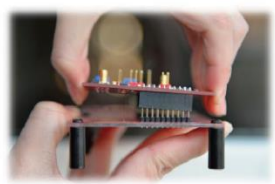
2 对齐后两端同时用力下压，不平衡野蛮用力会引起接插件损坏。





如图所示，因为直接把模块竖直拔下难度很大，卸载时需要将模块沿排针方向前后依次抬升，逐渐分离排针与排母。注意不可大力强行拔下，可能会引起接口损坏。

从母板上卸下模块

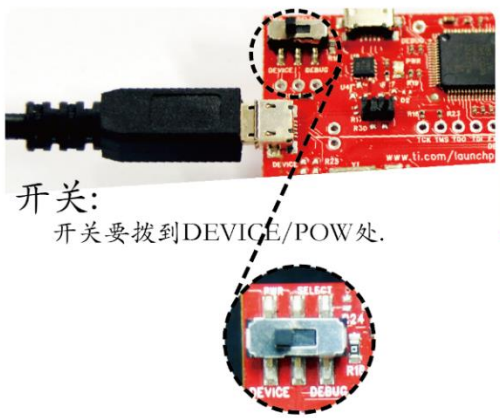


USB的使用

1 联接在板子左侧DEVICE接口
开关拨向左侧DEVICE标识
处于USB通信模式

2 联接顶端DEBUG接口
开关拨向右侧DEBUG标识
处于调试模式

套件中硬件可配置的环节



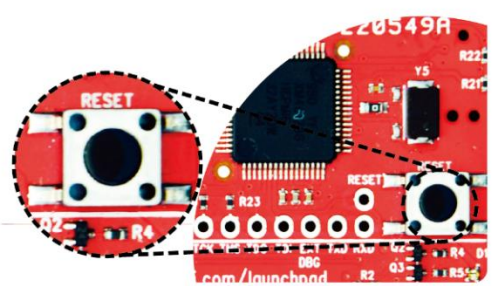
开关:

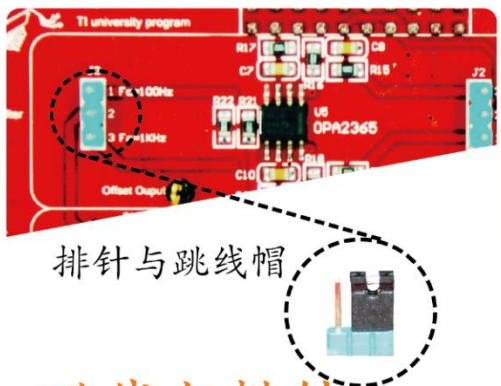
开关要拨到DEVICE/POW处.

开关与按钮

RESET开关:

RESET开关即为复位开关。

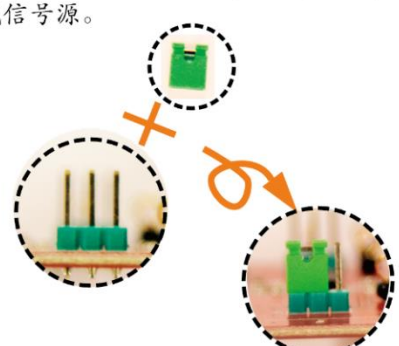


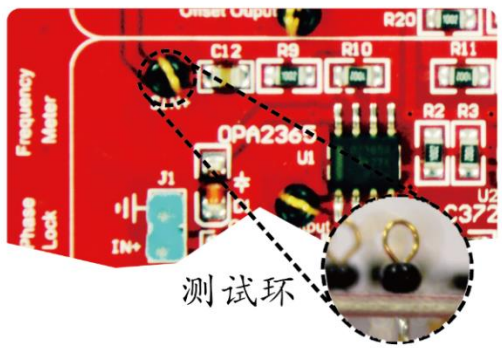


排针与跳线帽

跳线与排针

跳线帽与排针的组合使用是用来配置不同的电路连接方式。跳线帽上的小缺口也可以用来做测试点勾住表笔。排针有时候也通过杜邦线连接外接的负载或信号源。

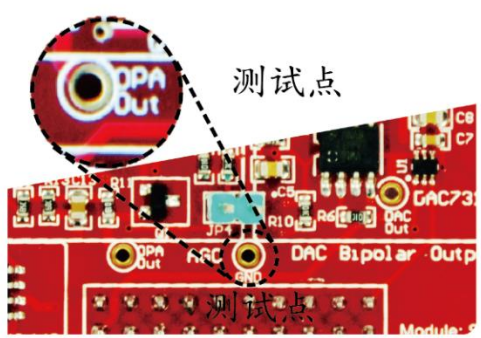




测试环

测试环和测试点提供表笔的接入接口，通常测试点是用来接万用表的表笔，测试环是用来勾住示波器表笔。

数据测量



测试点

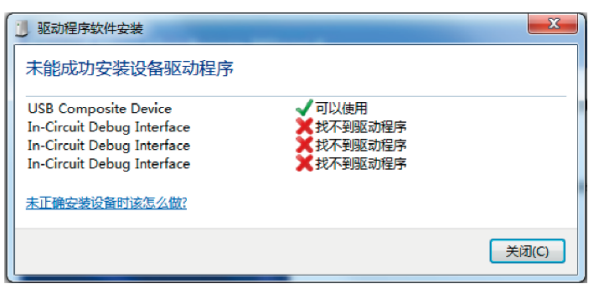
测试点

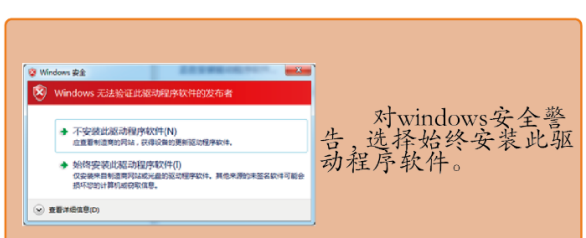
软件安装—Stellaris ICDI驱动安装

Tiva Launchpad主要特点之一就是提供ICDI，它的使用需在计算机上安装Stellaris ICDI驱动,主要包括以下三个部分: 1.Stellaris V--irtual Serial Port 2.Stellaris ICDI JTAG/SWD 3.Stellaris ICDI DFU。下面就以MS Windows 7系统下的安装为例,详述ICDI驱动的安装步骤,其他操作系统用户可参考 *http://www.ti.com.cn/tool/cn/Stellaris_icdi_drivers* 网站上的"Stellaris Driver Installation Guide"文档进行操作。

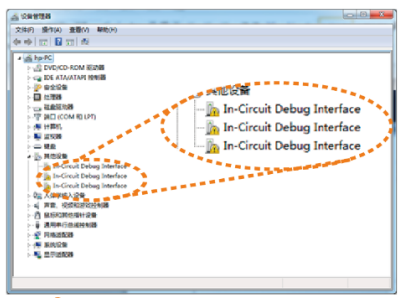
1.解压缩驱动包。

通过ICDI USB连接口(ICDI)连接计算机和开发板，随后打开电源开关到DEBUG位置。单击任务栏上驱动程序安装图标 ，如下图可看到ICDI并未正确安装。

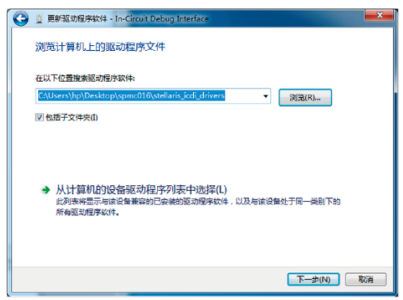




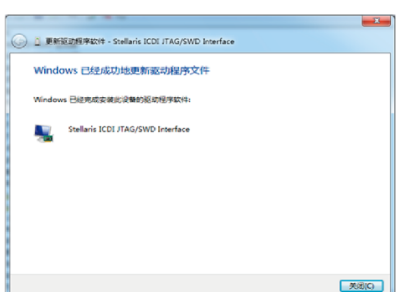
对windows安全警告，选择始终安装此驱动程序软件。



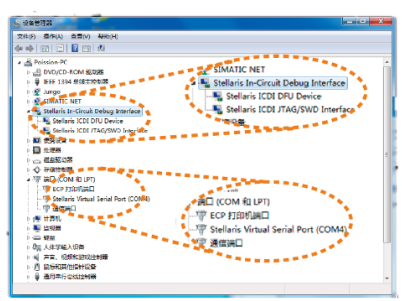
2.右键单击桌面的"计算机"图标，选择属性->设备管理器，在设备管理器的其他设备处可看到未正确安装的驱动设备。



3.选择其中一个In-Circuit Debug Interface右键单击,选择更新驱动程序软件，出现上图界面，点击浏览选择驱动文件所在位置，选中包含子文件夹，点击下一步。



4.安装完成后，出上图界面。



5.对其余未安装的In-Circuit Debug Interface进行同样的操作，即可完成ICDI驱动安装. 如上图所示，安装完成后，设备管理器中可出现Stellaris ICDI和Stellaris com端口

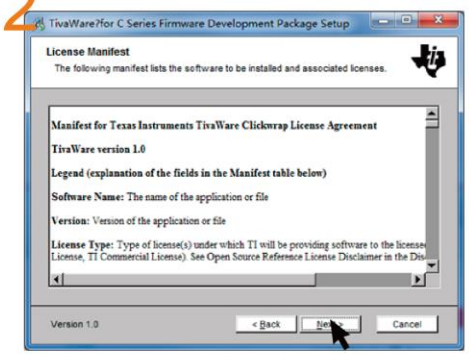
软件安装—TivaWare安装

从http://www.ti.com/tool/sw-tm4c上下载最新版本的TivaWare软件包后，双击下载文件，开始如下安装。

1

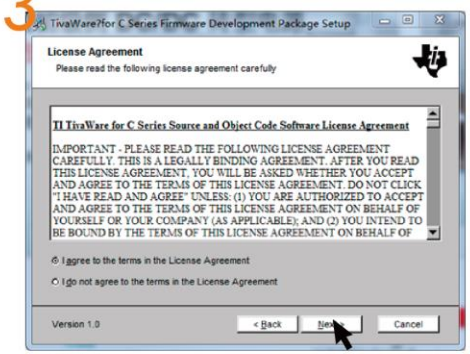
进入TivaWare安装界面

2



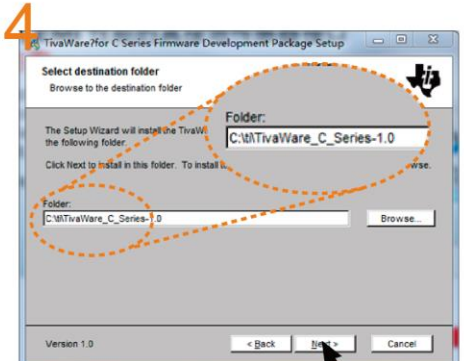
出现TivaWare的版本信息,,点击next

3



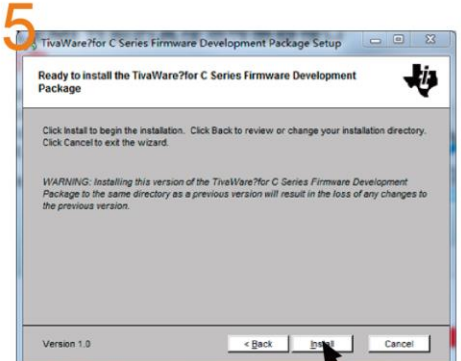
选择I agree to the terms in the License Agreement,点击next

4



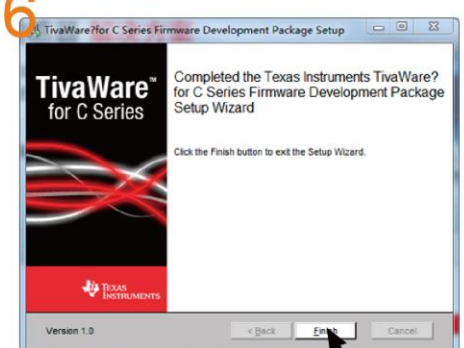
选择软件包安装目录,点击next

5



最后一步点击install,点击next

6

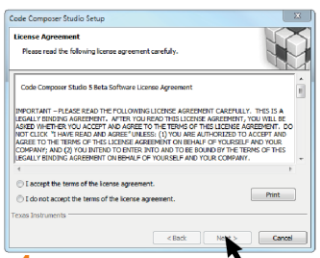


点击Finish完成安装

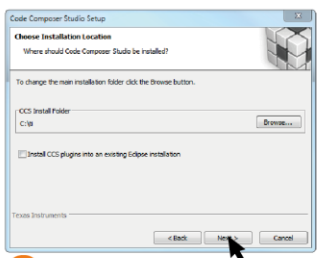
注意 你可以选择将TivaWare安装到你自定义的目录下,但请记住你的安装路径,在后续的编译属性和链接属性设置过程中，引用头文件和库文件的路径需要到此安装目录下查找。

软件安装—Code Composer Studio安装

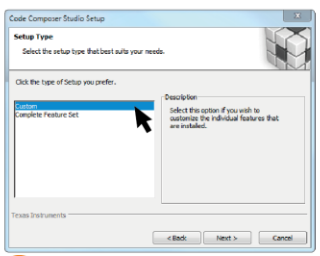
Code Composer Studio™ (CCStudio，很多时候也简称CCS) 是TI 嵌入式处理器系列的集成开发环境 (IDE)。 CCStudio包含一整套用于开发和调试嵌入式应用的工具。它包含适用于每个 TI 器件系列的编译器、源码编辑器、项目构建环境、调试器、描述器、仿真器、实时操作系统以及多种其他功能。直观的 IDE 提供了单个用户界面,可帮助您完成应用开发流程的每个步骤。更多资料请登陆http://processors.wiki.ti.com/index.php/GSG:CCSv5_installation。



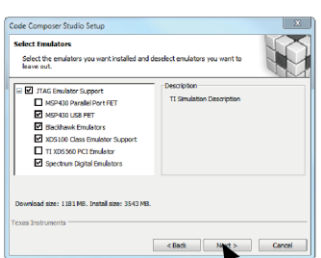
1 进入安装界面出现版本信息,选择I accept the term of the license agreement后点击next.



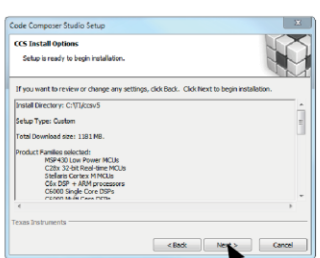
2 选择安装位置，完成后点击next



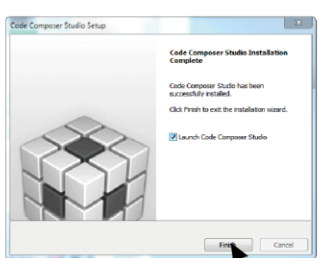
3 确认所需安装类型,建议选择Custom进行完整安装。



4 对仿真器进行选择

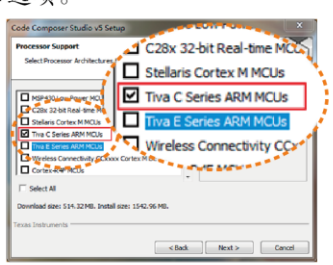


5 显示安装选项的摘要

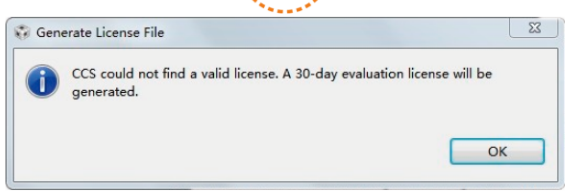


6 安装完成

注意 如果你以前安装过5.4以下的版本,只通过升级是无法实现TIVA支持,应当先卸载原来的程序，安装最新的5.4或更高版本 并在出现下图选择时至少选中所示选项。



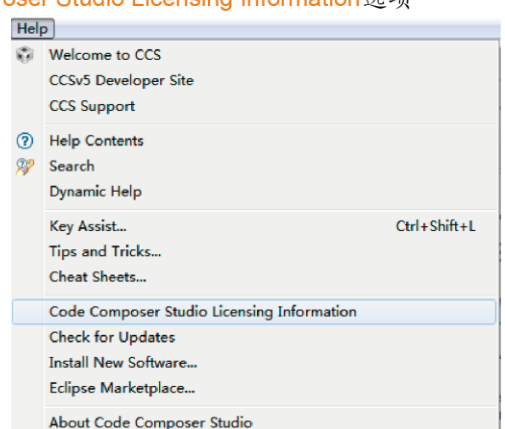
软件安装—CCS license的安装



未安装License打开CCS会弹出有关的提示信息，表明你正在用软件的评估版。如果要继续使用评估版，可以忽略这些信息。评估版最多可以用30天。

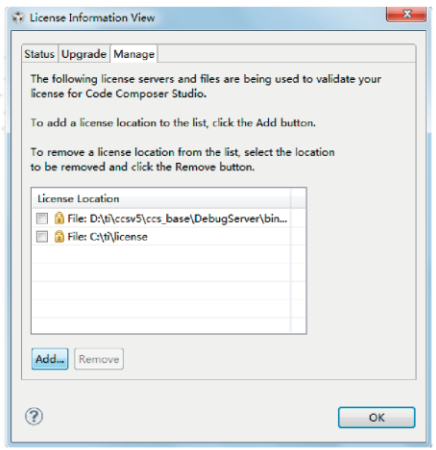
1.license状态对话框

点击菜单栏的Help选择其中的Code Composer Studio Licensing Information选项



2.安装license

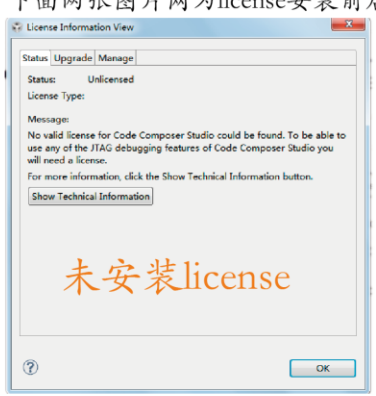
在弹出对话框中，选择Manage菜单，在License Location中点击下面Add按键来添加你的license。

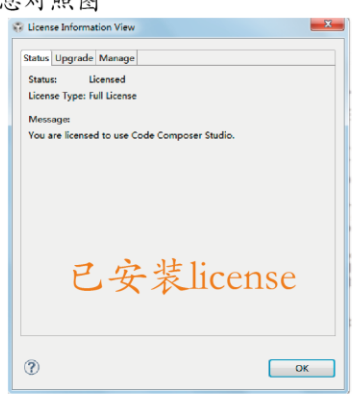


3.查看license

在License Information View对话框中，选中Status就可查看license信息

下面两张图片网为license安装前后的信息对照图

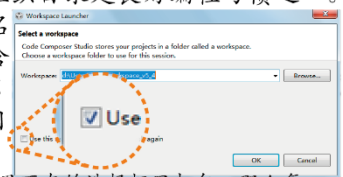




实验一、CCS工程的创建—创建流程

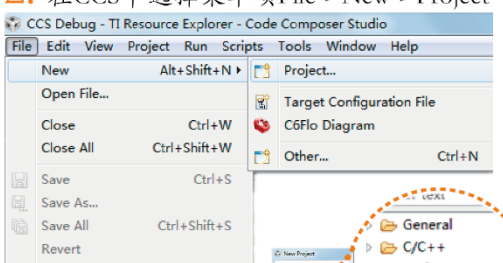
CCS中的开发是以工程(Project)为单元来组织管理的,理解工程的概念对初学者来说有一定的难度,但至少可以先把工程理解为一个以工程名命名的文件夹,这个文件夹包含用户编写的一个或都多个源代码文件、工程的配置信息文件、调试过程生成的中间文件、生成的可用于下载到MCU中的输出文件以及其它与编码、调试、生成有关的其它文件。以工程的方式组织管理方便用户修改,编译,生成,调试,移植等工作都在一种工具里完成,所以我们称CCS为IDE(集成开发环境)。下面为大家介绍如何新建一个以Tiva TM4C123G为目标设备的工程。

1. 首次运行CCS时会要求用户定义工作目录。新建的工程文件都会放到这个工作目录下。作本书的练习时建议每次个工程都单独放到一个目录里，有序组织目录是良好编程习惯之一。格外注意目录名及文件不能包含中文名，CCS不支持亚洲语言目录。

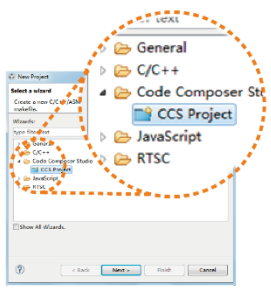


如果不在的选择框里打勾，那么每次运行CCS时都会再次弹出这个对话框

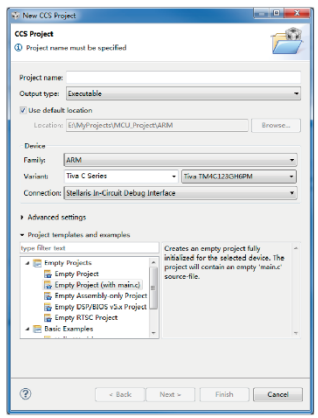
2. 在CCS中选择菜单项File->New->Project



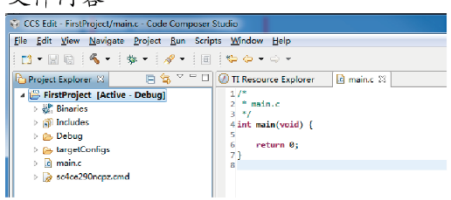
3. 在弹出对话框中选Code Composer Studio菜单下CCS Project。



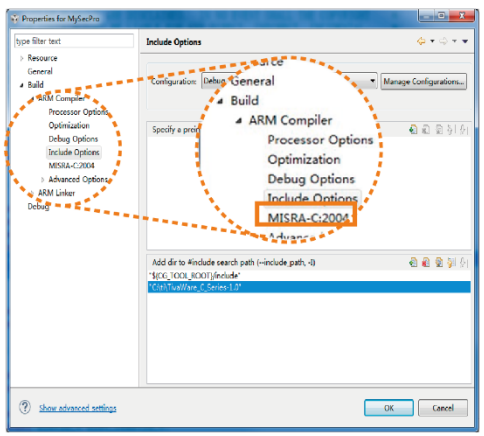
4. 在弹出的新建CCSv5工程对话框中对工程属性进行设定(详情参见下页)



5. 工程创建后CCS进入如下图所示窗口。窗口左侧显示为项目的文件组成,右侧为文件内容



6. 在Project Explorer窗口中的工程名上右键单击弹出菜单，选择Properties， 出现下图所示窗口。在此窗口中选择Build->ARM Compiler-> Include Options，设置工程的编译属性。



在Add dir to #include search path中添加路径"C:\ti\TivaWare_C_Series-1.0"(注意:请按您安装TivaWare的路径的路径来设置)

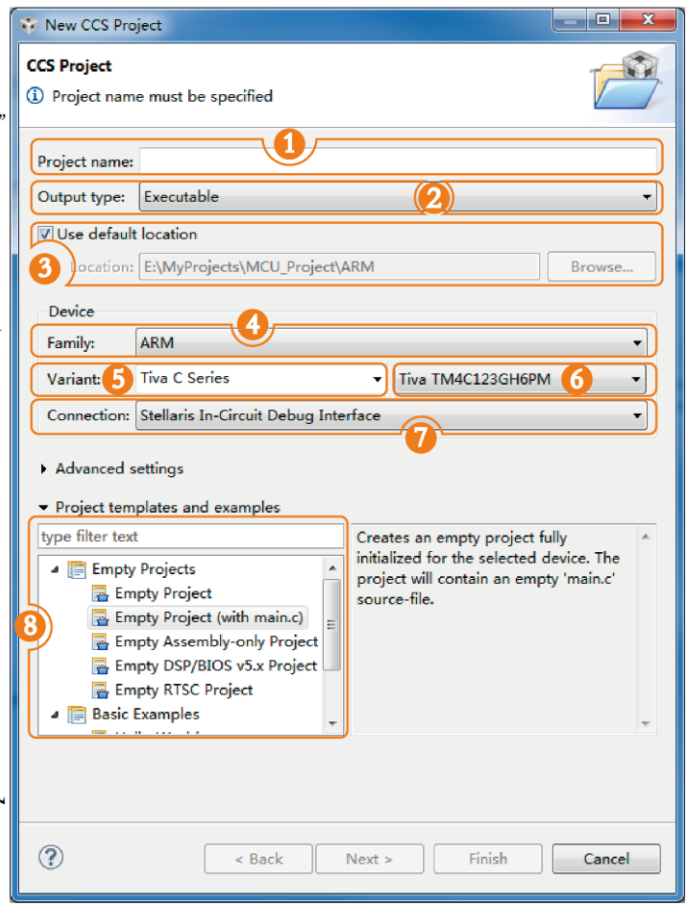
实验一、CCS工程的创建—新建工程属性配置

1 工程名
与工程相关的文件都会保存在以工程名命名的文件夹中，请在Project Name中输入新建工程的名称,此处用“FirstProject”为例，请勿用中文!

3 工程文件夹存放路径
通常我们会将工程存放在某一固定的文件夹，并在此文件夹下按目标设备（器件类型）分类存放。便于日后管理。

5 标设备系列
Variant选择Tiva C Series选项

7 目标设备与计算机的联接方式
选择Stellaris In-Circuit Debug Interface



2 工程输出类型
Output Type是指工程最终生成的结果的类型，可以是可执行代码（Executable），它是下载到设备上执行的代码；也可以是静态库(Static library)，它通常作为功能模块被其它工程使用。在此我们先选择：Executable。

4 目标设备类型
图中的Device部分是用来选择工程的目标设备(器件种类)和调试设备的,因为CCS支持TI的全系列处理器，所以目标设备的列表会非常长，为了方便用户选择，按器件的大类(Family)和系列(Variant)进行分类,本例中在Family选择ARM选项

6 芯片型号
此处芯片选择Tiva TM4C123GH6PM

8 项目模板和实例。
选择空工程(带main.c),然后单击Finish完成新工程的创建

CCS工程的创建—包含头文件的说明

在我们下面要做的工程中使用了两类头文件，第一类头文件在叫“inc”的子目录里，
它们是根据芯片名字命名的，它们定义了与芯片相关的所有寄存器的宏，
比如有头文件“tm4c123gh6gm.h”就定义了与我们这实验套件里MCU所有的寄存器的宏，
我们如果要直接引用这些宏时，就必须包含这个头文件，
有关寄存器的宏的概念，请见下一段落的内容。

另一类头文件是TI对C语言的数据类型的重定义，
C语言中的数据类型比如int，long等，它们的名字和数据位数的对应关系不直观不明确，
很多用户对C语言中的数据类型所表达的数据位数的认识又不深入，
基于MCU的硬件编程对操作数的数据位数又很敏感，
所以TI为了让用户能更方便更直接地使用数据类型，
根据不同的类型的目标设备定义了一套自己的标准数据类型。
这些数据类型的定义被放在"stdint.h"和"stdbool.h"这两个头文件中，
与ARM处理器相关这个两头文件位于CCS安装目录下的“/tools/complier/arm_5.0.4/include”。
请在写Tiva系列的程序时务必包括这两个头文件。
如下面代码框中所示重定义，带符号的32位整型和无符号的32位整型分别被定义为int32_t 和uint32_t。
这样把数据位数和是否带符号信息直接呈现在变量类型上，降低了代码数据类型出错的可能。

```
typedef          int    int32_t;
typedef unsigned int    uint32_t;
```

实验一、CCS工程的创建

—寄存器的宏定义

我们写代码时对芯片内部的寄存器进行读写访问，其本质是对某一地址的存储器进行读写，但记住所有寄存器地址太困难，所以用#define定义了一系列的宏，如果你打开tm4c123gh6pm.h这个头文件，你会发现大量类似如下的定义：

```
#define GPIO_PORTA_DATA_R      (*((volatile uint32_t *)0x400043FC))
```

GPIO_PORTA_DATA_R就是宏名，0x400043FC是寄存器地址，通过这个宏定义，我们在程序中写代码时就可以直接用GPIO_PORTA_DATA_R来表示0x400043FC这个寄存器。

为了方便记忆，宏的名字采用了匈牙利命名法，

GPIO表示寄存器所在的外设功能模块类型，

PORTA表示寄存器所在的部分(除GPIO外其它模块没有这一项)，

DATA表示这个寄存器本身的功能(即它是数据寄存器)，

R表示这个宏名表示的是寄存器(有些宏不是用来定义寄存器的，不以_R结尾)，

也就是说寄存器的宏名定义按照右侧规则来定义，其中括号表示有些功能模块的宏名没有这部分。

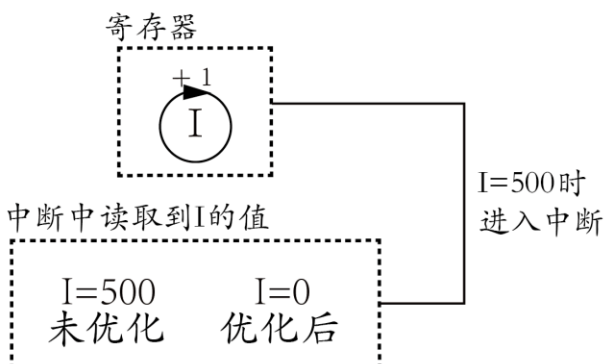
功能模块类型名_（所在部分名）_寄存器本身的功能_R

实验一、CCS工程的创建

—寄存器的Volatile变量

还要注意宏定义中的“volatile”关键字，内存变量最终参与内核运算单位的运算时总要被加载到内核的寄存器上，运算完的结果回写到内存。但有时一个变量会在短时间内连续调用多次，这样会引起不断地加载和回写操作，比如说一个循环程序里的循环变量。现代编译器通常会自动优化这部分代码，使得变量在这一段时间内都不回写，当这段过程结束后再回写变量。这种优化对单线程程序没有问题，但我们的MCU程序经常应用大量的中断功能。

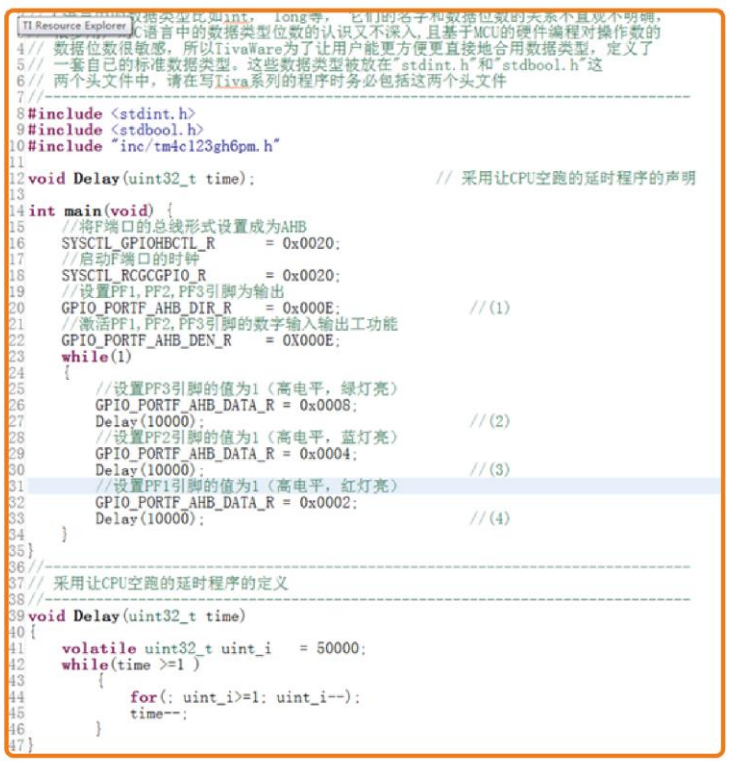
假设我们在一段循环程序里用了一个循环变量I，整个循环要做50000次，循环开始的时候内存变量I等于 0 ，I被加载到寄存器(假设被加载到寄存器R5)开始做循环计数，如里每循环一次I增加1，当循环了500次后，这时寄存器R5内的值已经变成 500 ，但因为优化过的程序不回写I，所以内存的那个I还是保持原来的初始值0 , 只有当循环结束时发生回写 , 内存的I才会更新为50000。我们现在假设在循环到 500 次时程序刚好发生了一个硬件中断 , 程序会停止当前的循环转去中断服务程序处理。如果中断服务程序里也希望引用I的值 ，这时错误就发生了,因为 I 的值没有被回写，但实现R5的值已经是500，中断服务程序不会去R5取I的值，因为它只认识在内存的I， 所以它会取回I值为0。



如果我们把变量都注明为volatile，那么编译器就不会帮你做上述的代码优化。我们得到的结论就是当在多个线程中共享的数据，它们都要被注明是volatile的类型，不然就发生意想不到的错误。

实验一、CCS工程的创建—mani.c的代码

在按上述步骤建好的工程中打开main.c文件，输入如图所示代码，或者直接从网站http://www.hpati.com上下载第一章的代码，并将代码直接拷贝到main.c中。




```
//数据类型比如int，long等，它们的名字和数据位数的关系不直观不明确，
//C语言中的数据类型位数的认识又不深入，且基于MCU的硬件编程对操作数的
// 数据位数很敏感，所以TivaWare为了让用户能更方便更直接地合用数据类型，定义了
// 一套自己的标准数据类型。这些数据类型被放在"stdint.h"和"stdbool.h"这
// 两个头文件中，请在写Tiva系列的程序时务必包括这两个头文件
//*****************************************************************************
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"

void Delay(uint32_t time);                          // 采用让CPU空跑的延时程序的声明

int main(void) {
    //将F端口的总线形式设置成为AHB
    SYSCTL_GPIOHBCTL_R      = 0x0020;
    //启动F端口的时钟
    SYSCTL_RCGCGPIO_R       = 0x0020;
    //设置PF1,PF2,PF3引脚为输出
    GPIO_PORTF_AHB_DIR_R    = 0x000E;                 //(1)
    //激活PF1,PF2,PF3引脚的数字输入输出工功能
    GPIO_PORTF_AHB_DEN_R    = 0X000E;
    while(1)
    {
        //设置PF3引脚的值为1（高电平，绿灯亮）
        GPIO_PORTF_AHB_DATA_R = 0x0008;
        Delay(10000);                               //(2)
        //设置PF2引脚的值为1（高电平，蓝灯亮）
        GPIO_PORTF_AHB_DATA_R = 0x0004;
        Delay(10000);                               //(3)
        //设置PF1引脚的值为1（高电平，红灯亮）
        GPIO_PORTF_AHB_DATA_R = 0x0002;
        Delay(10000);                               //(4)
    }
}
//*****************************************************************************
// 采用让CPU空跑的延时程序的定义
//*****************************************************************************
void Delay(uint32_t time)
{
    volatile uint32_t uint_i    = 50000;
    while(time >=1 )
        {
            for(; uint_i>=1; uint_i--);
            time--;
        }
}
```

❶ 在建好的工程中找开 main.c文件，用本页所示的代码替代原来的内容。或者直接从http://www.hpati.com上下载本章对应的代码，直接拷贝到文件中。

❷ 保存文件内容，可以用快捷键 ctrl + s 或者工具栏上的保存按键。

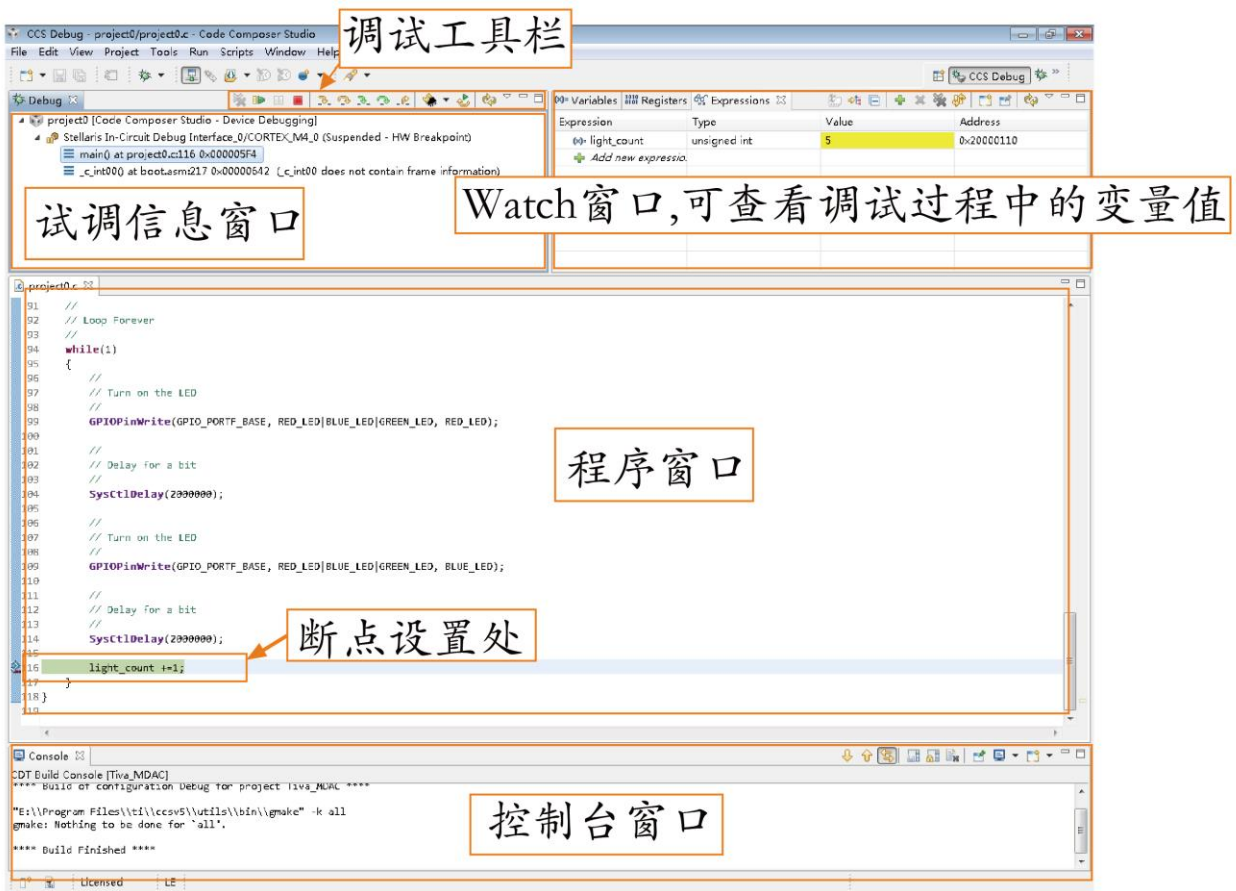
❸ 用Launchpad带的USB联接线，联接计算机和板上的标注为DEBUG的USB接口。

❹ 把开关拔到 DEBUG 一侧，可以看到绿色电源灯点亮。

❺ 点击 CCS里工具栏上的绿虫子按键，并等待 CCS处理完成。如果发生错误，请根据提示仔细检查代码和属性设置。

❻ 再点击 CCS调试工具栏上的运行按键，观察程序的运行结果：三色LED灯交替闪烁。

实验一、CCS工程的创建—程序的调试



CCS快捷菜单

可以在Window->Preferences->General->Keys目录下自定义快捷键

图标	操作命令项	快捷键	介绍
	Build		编译链接
	Debug	F11	编译链接并进入调试界面
	运行(Rusume)	F8	开始运行
	暂停(Suspend)	Alt+F8	暂停运行
	退出调试(Terminate)	ctrl+F2	停止调试并退出调试界面回到编辑界面
	单步步入(Step Into)	F5	运行程序当前行(调试界面程序窗口箭头所在行),运行结束,箭头指向下一行,暂停调试,若当前行所在为函数调用行,调试时其会跳进函数内部,函数执行完成后再跳出
	单步步过(Step Over)	F6	运行程序当前行(调试界面程序窗口箭头所在行),运行结束,箭头指向下一行,暂停调试.若当前行所在为函数调用行,调试时不会进入函数内部,但会执行函数内部程序
	单步步出(Step Return)	F7	当前在子函数内部执行时，若执行单步步出则会跳出子函数，箭头指向下一行，并暂停调试
	内核复位(Core Reset)	ctrl+Shift+R	内核复位会使M4的内核完成复位。
	重新开始(Restart)	ctrl+Shift+B	在调试过程中操作“重新开始”,程序会回到主函数第一行(箭头会指向第一行)
	断点 (Toggle Breakpoint)		设置或者移除断点
	移除所有断点 (Remove All Breakpoint)		移除工程中所有的断点
	移除选中的断点 (Remove Select Breakpoint)		移除当前选中的断点
	查看当前环境快捷键	ctrl+Shift+L	调出当前快捷键查看窗口

TIVA M4 Launchpad 管脚的定义				各模拟模块使用的管脚（电源和地线未标注在此表中）							
Pin	GPIO	Analog	GPIOPCTL Setting	DC-DC-升压	LCD模块	MDAC	频率与相位	DC-DC-降压	电机驱动	电阻测量	高速压控
1.01			3.3V					AIN11			
1.02	PB5	AIN11	SSI2FSS/M0PWM3/T1CCP1/CAN0Tx					GPIO			
1.03	PB0	USB0ID	T2CCP0						GPIO		
1.04	PB1	USB0VBUS	T2CCP1		GPIO						
1.05	PE4	AIN9	U5Rx/I2C2SCL/M0PWM4/M1PWM2/CAN0Rx					GPIO			
1.06	PE5	AIN8	U5Tx/I2C2SDA/M0PWM5/M1PWM3/CAN0Tx		GPIO						
1.07	PB4	AIN10	M0PWM2/T1CCP0/CAN0Rx				T1CCP0		M0PWM2	ADC	
1.08	PA5	\	SSI0Tx		SSI0Tx						
1.09	PA6	\	I2C1SCL/M1PWM2					M1PWM2		I2C1SCL	
1.10	PA7	\	I2C1SDA/M1PWM3							I2C1SDA	
2.01			GND								
2.02	PB2	\	I2C0SCL/T3CCP0						T3CCP0(Fb)		
2.03	PE0	AIN3	U7Rx								
2.04	PF0	\	U1RTS/SSI1Rx/CAN0Rx/M1PWM4/PhA0/T0CCP0/NMI/C0								
2.05			RESET								
2.06	PB7	\	M0PWM1/T0CCP1	M0PWM1					M0PWM1		
	PD1	AIN6	SSI3Fss/SSI1Fss/M0PWM7/M1PWM1/T0CCP0								
2.07	PB6	\	M0PWM0/T0CCP0								
	PD0	AIN7	SSI3Clk/SSI1Clk/M0PWM6/M1PWM0/WT2CCP0								
2.08	PA4	\	SSI0Rx		SSI0Rx						
2.09	PA3	\	SSI0Fss		SSI0Fss						
2.10	PA2	\	SSI0Clk		SSI0Clk						
3.01			5.0V								
3.02			GND								
3.03	PD0	AIN7	SSI3CLK/SSI1CLK/I2C3SCL/M0PWM6/M1PWM0/WT2CCP0		SSI3CLK						
	PB6	\	SSI2Rx/M0PWM0/T0CCP0								
3.04	PD1	AIN6	SSI3Fss/SSI1Fss/I2C3SDA/M0PWM7/M1PWM1/WT2CCP1		GPIO						
	PB7	\	SSI2Tx/M0PWM1/T0CCP1								
3.05	PD2	AIN5	SSI3Rx/SSI1Rx/M0FAULT0/WT3CCP0/USB0EPEN								
3.06	PD3	AIN4	SSI3TX/SSI1Tx/WT3CCP1/USB0PFLT		SSI3TC						
3.07	PE1	AIN2	U7Tx						AIN2		
3.08	PE2	AIN1	\	AIN1							
3.09	PE3	AIN0	\	AIN0							
3.10	PF1	\	U1CTS\SSI1Tx\M1PWM5\T0CCP1\C1O\TRD1						M1PWM5		SSI1TX
4.01	PF2	\	SSI1CLK/M0FAULT0/M1PWM6/T1CCP0/TRD0						M1PWM6		SSI1CLK
4.02	PF3	\	SSI1FSS/CAN0Tx/M1PWM7/T1CCP1/TRCLK								GPIO(SYNC)
4.03	PB3	\	I2C0SDA/T3CCP1								
4.04	PC4	C1-	U4Rx/U1Rx/M0PWM6/IDX1/WT0CCP0/U1RTS					P0PWM6			
4.05	PC5	C1+	U4Tx/U1Tx/M0PWM7/PhA1/WT0CCP1/U1CTS				P0PWM7				
4.06	PC6	C0+	U3Rx/PhB1/WT1CCP0/USB0EPEN		GPIO						
4.07	PC7	C0-	U3Tx/WT1CCP1/USB0PFLT		GPIO						
4.08	PD6	\	U2Rx/PhA0/WT5CCP0		GPIO						
4.09	PD7	\	U2Tx/PhB0/WT5CCP1/NMI		GPIO						
4.10	PF4	\	M1FAULT0/IDX0/T2CCP0/USB0EPEN					T2CCP0			

第2章 TivaWare 应用入门

杭州艾研信息技术有限公司

2014 年 11 月

申明

杭州艾研信息技术有限公司保留随时对其产品进行修正、改进和完善的权利，同时也保留在不作任何通告的情况下，终止其任何一款产品的供应的权利。用户在下订单前应及时获取相关信息的最新版本，并验证这些信息是当前的和完整的。

可通过如下方式获取最新信息、技术资料和技术支持：

技术支持电话：0571-86134572

技术支持邮箱：support@hpati.com

产品&资料下载中心：http://www.hpati.com/products/

互动论坛：http://www.hpati.com/bbs/forum.php

公司地址：浙江省杭州市西湖区留和路16号新峰商务楼B306

第2章 TivaWare 应用入门

CCS 是 TI 的一个集成开发环境，它提供开发所需的编辑、编译、生成、下载并调试的功能，并支持 TI 的所有处理器的开发。上一章我们用 CCS 写了一个简单的程序，并**编译**、**生成**、**下载**并**调试运行**。 我们可以把 CCS 想象成一个有灶台、炉子、锅、铲、刀和案板等基本工具的厨房。在具备这些工具的厨房里，如果你是娴熟的厨师，不需要借助其它工具，只须再有些原料就可以做出各式菜品。同样，有了 CCS，你只要具备足够强的开发能力，不需要借助其它软件工具，只需拥有开发板之类的硬件，就可以设计出各种功能的系统。 但你会发现用刀切肉未一定不如绞肉机做得更快更好，于是，你在厨房里不断添置各种工具，比如微波炉，面包机等等，它们加快我们做菜的效率；类似的，TI 也不断在 CCS 中添加各种工具，让用户能更快更有效地开发产品。TivaWare 就是其中的一种工具，它是一个函数代码库的集合，它作为一套完整的基于 Tiva 系列 MCU 的库软件，简化并加速开发人员在 Tiva 系列 MCU 上的应用开发。如果想把厨房里的新工具用得得心应手，我们就得花一点时间了解它。用好 TivaWare 也需要一定的时间。 为了使初学者能够更快地入门、理解并掌握 TivaWare，我们以相对简单的 GPIO 操作为例。首先根据硬件的数据手册对 GPIO 的描述，通过对寄存器直接读写编写一小段代码实现对 GPIO 的基本控制。然后在此代码的基础上逐步修改扩展，直到实现一个自建的函数库，并最终将自己实现的函数库与 TivaWare 已有的函数进行比对。希望借此能让读者理解 TivaWare。

1 TivaWare 简介

如果你对预编译、编译、链接、库、接口、实现这些计算机软件的术语没有概念，建议您阅读本章的时候先按第二节的说明试做一遍实验，回过头来再来读第一节的内容会更容易理解。对有那些在其它语言中有过库函数调用的经验的读者，第一节有关 TivaWare 的介绍可以帮助你快速了解 TivaWare 的概况，对有经验的开发人员，了解概况之后再参考相关的库函数说明手册就可以着手开发了。

1.1 TI ARM Cortex MCU 的开发方式

这里所说的 TI ARM Cortex MCU 的开发方式是指下载并运行在 MCU 上的软件的开发方式。 开发方式的含义之一是开发语言的选择。MCU 上的软件通常有两类开发语言，一类是汇编指令，另一类是高级语言（通常是 C 或者 C++）。汇编指令会因 MCU 种类不同而不同，程序逻辑控制困难，代码可读性及可维护性都不佳，而现在编译器的效率越来越高，所以采用汇编指令作为开发工具的机会越来越少，只有少数程序或者程序的某个模块对代码的效率要求极高时才会采用汇编指令作为程序开发语言。C 语言已经成为当前大部分 MCU 软件开发的主要语言。随着 C++的编译器的进步和 MCU 性能的提高，在项目复杂度和 MCU 性能都较高的应用越来越多地利用面向对象的 C++语言作为开发工具，以获得更优的项目管理特性。鉴于当前仍以 C 语言为主的开发方式，本书的例程和所有讲解仍以 C 语言作为开发语言。

开发方式的另一含义是指利用 TI 提供的哪些**库（Library）**，或者利用库的哪些部分。所谓的库，它是一组功能函数的集合，这个集合里函数的功能都已实现，并有一个易于调用的接口[1]。库为软件开发人员方便使用其它开发人员已完成的功能（代码）提供了一种载体，使代码具有可重用性。也就是说你也可以将你实现的一系列代码做成库的形式，来供其它人员使用，当然也包括你自己将来使用这个库，称之谓可重用性。这么解释显然很苍白，这一章的实验二的预备知识和实验过程会让大家更进一步理解“库”的概念。

其实 TivaWare 就是由多个库组合成的库的集合，如本章的引言中所述，它像 CCS 标准配置外的额外工具， 所以我们有时也形象地得把这种库的集合叫**工具包，**也经常被叫作 SDK（Software Develop Kit）。既然 TivaWare 是库的集合，就有必要了解一下它到底由

[1] 维基对计算机软件中的库的定义。

哪些库组成。

如图 2-1 所示，图的左半部为 TivaWare 库的结构，其中的开源协议栈和开源 RTOS 严格意义上不属于 TI 开发的库。库的结构从下至上的排序是按照库与用户的最终应用程序的“远近”来排序的，越上层的离应用程序越“近”。离应用程序“近”的库是专门为某种功能或某类功能而设计的，它的功能很专一，在这个功能领域的开发效率也很高，但只有需要这种功能时才会使用它；离应用程序“远”的库通常都具有通用性和基础性， 通常大部分应用程序都会使用它。就好像厨房的刀既可以切菜也可以切肉，还能切出各种形状，而绞肉机只能把肉绞成肉末，刀的通用性很强成为厨房的必备工具，而绞肉机在绞肉时可以大大提升做肉末的效率。

TivaWare 是一个不断扩展的库，TI 公司、第三方以及开源社区都为此做出贡献。TivaWare 为用户使用它提供了大量的说明文档和示例代码。如图 2-1 右侧所示，它们是学习 TivaWare 最准确最有效的资源。你安装完 TivaWare 之后，这些文档及示例代码也随之安装完毕。本章 **TivaWare 安装目录下的结构**这一小节会告诉大家这些文档的存放位置。

图 2-1 TivaWare 的结构

1.2 BootLoader 简介

(1) BootLoader 的概念

BootLoader，顾名思义，是 Boot（启动）的时候起到 Loader（加载器）作用的代码，即系统刚一上电启动，最先被执行的一段代码，根据这段代码的安排，系统的其余部分会被它加载（Load）并运行。

上述对 BootLoader 的解释对从来没听说过它的读者来说估计仍然是难以理解的。 你可能还会问：知道 BootLoader 又能怎样？有什么用？ 它跟我学 MCU 有什么关系吗？我

从你最熟悉的手机开始讲。 相信你一定有手机，而且很可能是智能手机（它们安装了不同的操作系统，比如是 Android, iOS, Windows Phone, Symbian）。如果你“与时俱进”，你可能还经常升级手机上的系统，就是我们常说的“刷机”。手机上的系统升级就是要用到 BootLoader, 如果你有刷机经验，最明显的感受是你要先下载一个新版本的系统，通常那个玩意被简称为固件（firmware）或者映像(Image), 再把新版系统通过某种方式**加载**到你的手机里才能更新原来的系统，而且这个过程一定会伴随着手机的重新启动。当旧版操作系统正在运行时，你的手机里的一切活动由旧版系统掌控，一山不容二虎，新版系统不能和旧版系统同时存在于你的手机上。新版系统覆盖旧版系统的那一段时间内，旧版系统不可能掌控手机，因为它正逐步被覆盖，新版系统也不可能掌控手机，因为它还没有完全更新到手机上。必须有人站出来处理这事，那就是 BootLoader。BootLoader 被放在系统的某一个特定的地方，每次系统一上电，它都会比任何其它程序先运行。如果没有系统更新的任务，它就老老实实加载旧版本系统运行；如果有新的系统要更新，它就会加载新系统，覆盖旧系统，这个过程中它掌握了一切。当新系统加载完，它把掌控权转交给新系统。总而言之，BootLoader 就是一个加载器，它不是加载已有的程序运行，就加载新的程序覆盖已有的程序。它的出现让手机用户更新手机系统可以自己完成，而无需去找厂商（要知道十多年前手机换系统必须去找厂家或其代理点）。

如果你熟悉个人计算机（PC）并有过 PC 的 DIY 经验（现在用笔记本电脑的人占多数，只有用台式 PC 才会有更多的 DIY 机会），也许听说过“BIOS”这个名字，它的部分功能就是 BootLoader，PC 一上电就会先执行 BIOS 程序，然后由它来加载系统（比如 Windows[2]）。它是打开电源后最新执行的程序，最明显的证据是当你拿到一个新的机器，没有安装任何系统前也能进入 BIOS 设置窗口。

(2) Tiva 的 BootLoader

在 PC 里，BIOS 通常存放在主板上的一块 ROM 里，PC 一上电默认找到这块 ROM 内的程序执行；在 Tiva 系列 MCU 系统里，起到类似 PC 里 ROM 的功能的是闪存（Flash 存储器，简称 Flash）， BootLoader 是存放在 Flash 起始地址处的一小段代码，占据默认大小为 2K 字节的空间。Tiva 的 BootLoader 有两种可配置的加载功能，其一是加载用户应用程序（Application）；其次是与其它设备建立标准通信通路，并从该设备上加载新的系统软件来更新（Update）原有的系统。更新的过程通过对固件（Firmware）的烧写来实现。烧写

[2] 学过操作系统的人会认为引导程序才是 Windows 的 BootLoader。没错，对 Windows 系统来说是这样。但对于整个 PC 来说，操作系统的引导程序也是由 BIOS 来调用，所以我认为 BIOS 是 PC 的 BootLoader。

固件也是那些经常更新手机系统的用户比较熟悉的术语。系统上电后 BootLoader 内的启动代码（Start-up Code）将先被执行，进行一系列的初始化操作后，根据预先设定的条件，选择执行用户应用程序或更新控制程序（Updater）。

我们在 PC 上安装新系统时会通常要从光盘或者 U 盘把这个新系统加载到系统的硬盘上。Tiva BootLoader 的 Updater 在升级的过程中，是通过某种通信方式联接到别的设备上，把那个设备上的系统加载到 Tiva 的 Flash 中。Tiva C 系列的芯片它可能采用的通信有 UART0，SSI0，I2C0, CAN, USB 或以太网端口。数据通信必须确保无差错，为了保证数据的无差错传输，BootLoader 采用了控制传输的通信协议。当采用 UART0，SSI0，I2C0 和 CAN 这些端口通信时，BootLoader 要求用户自定义串行通信协议，来确保更新过程中的通信可靠性，将要更新到 Flash 上的系统软件集合也叫固件映像（Firmware Image），为了减少数据通信量，Firmware Image 通常某种数据压缩形式被压缩成一个包）。 而采用以太网或 USB 时，因为这两种接口都有现成的支持更新功能的协议，所以无需用户自定义协议。用以太网为接口时采用了标准 BOOTP（Standard Bootstrap Protocol）；而 USB 接口则采用标准 DFU（Standard Device Firmware Upgrade）。对接收到的格式正确且校验成功的数据包，Updater 能够将其解包，并将得到的加载命令转化为对 Flash 底层寄存器的操作。

由于 TivaWare 提供了 BootLoader 的全部源代码，用户也可自行修改通信端口、通信协议等相关组件，使其更好地符合用户需求。

注意： *没有 BootLoader 时，系统也能执行用户应用程序。详见下一小节的内容。*

(3) Tiva C 系列的存储空间分布和启动机制

为了使同一系列的处理器有更好的兼容性，Cortex M4F 的内核存储器映射是按表 2-1 固定分配的。

Cortex M4F 的地址空间中，0~0.5G 之间的空间被分配（一般称为映射）为 Flash 空间，0.5G~1G 被映射为 SRAM。因为 SRAM 是易失性存储器，系统刚上电时，SRAM 中没有内容，因此系统必须从 Flash 开始启动。

Flash 空间起始地址处必须存放**异常向量表**（Exception Vector）。异常向量表与其它单片机中的中断向量表功能类同，只是异常比中断的含义更广。向量表是在内存指定位置上的一张由多个单元组成的表，表中的每个单元都对应着某一种异常，也事先按约定的顺序排好，这样，每个异常对应的向量表的位置在硬件上是固定的。当某个异常产生时，我们需要程序自动跳转到异常处理程序所在的位置。 但系统不知道这个位置，所以我们将这个地址放在异常向量表里，系统知道异常的类型，根据类型找到向量表中与此类型对应的异常向量单元。向量单元在 Tiva 中由 4 个字节组成，用来存放程序地址信息。

地址范围	存储区功能	描　　述
0x0000.0000 ~ 0x1FFF.FFFF	代码区，数据区	这一部分是 FLASH 存储，通常用来存放程序代码，可以存储数据。
0x2000.0000 ~ 0x3FFF.FFFF	SRAM	数据和代码的运行的区域。断电后数据消失
0x4000.0000 ~ 0x5FFF.FFFF	片内外设	片内外设模块中的各类寄存器和缓存区域
0x6000.0000 ~ 0x9FFF.FFFF	外部 RAM	内部 SRAM 不够用，外加的 RAM
0xA000.0000 ~ 0xDFFF.FFFF	片外外设	请关注芯片手册的说明
0xE000.0000 ~ 0xE00F.FFFF	专用外设总线	请关注芯片手册的说明
0xE010.0000 ~ 0xFFFF.FFFF	保留区	-

表 2-1 Cortex M4F **存储器映射表**

系统复位时，向量表的地址是固定在 0x0000.0000 的，具有系统级权限的软件可以在程序代码开始运行后修改向量表的起始地址。修改的方式是通过软件设置 NVIC 中的向量表偏移寄存器（NVIC_VTABLE，0xE000ED08），向量表的地址可以修改为 0x0000.0400 到 0x3FFF.FC00 之间的位置。

注意：*修改后的起始地址是 4 的倍数，用于 32 位齐，同时留下 1024 字节的表空间。*

Cortex M4F 系列单片机硬件启动原理如下：硬件复位时，NVIC_VTABLE 复位为 0，向量表默认位于 Flash 空间起始地址处（0x00000000）。内核读取向量表第 1 个单元设置主堆栈（SP_main），读取第 2 个单元设置 PC 指针，之后跳转到复位处理函数中运行。复位处理函数获得系统的控制权，复位处理函数通常就是系统的用户应用程序。这就是没有 BootLoader 时系统的调用应用程序的方式。

(4) BootLoader 的启动代码（Startup Code）

BootLoader的启动代码是一小段精简代码，它要完成（1）配置向量表；（2）初始化存储空间的分布；（3）复制BootLoader代码到SRAM；（4）从SRAM中执行代码等一系列操作。

由于配置向量表和初始化存储空间的分布与用户使用的开发工具的编译器和链接器有关，所以每种工具都有它自己对应的启动代码实现和链接配置文档。与CCS相关的启动

代码和链接配置脚本分别在：

```
bl_startup_ccs.s          （TI CCS的启动代码）
bl_Link_ccs.com           （配置文档）
```

配置完向量表并初始存储空间后，启动代码复制 BootLoader 代码到 SRAM。由于 BootLoader 的作用之一，是提供运行时修改 Flash 的功能。而由于 Tiva 系列单片机具有单周期的 Flash 读写能力，因此默认的代码段本身就位于 Flash 中。这样，如果内核直接从 Flash 中加载修改其自身的指令，则既容易造成时序上的混乱，又有可能因 Flash 中某些关键指令被修改而导致整个系统崩溃。针对这个问题，Tiva 系列单片机采取的方案为：在 SRAM 中建立 BootLoader 的映像，即把 BootLoader 复制到 SRAM 中，然后从 SRAM 运行代码来修改在 Flash 中的代码。图 2-2 所示。

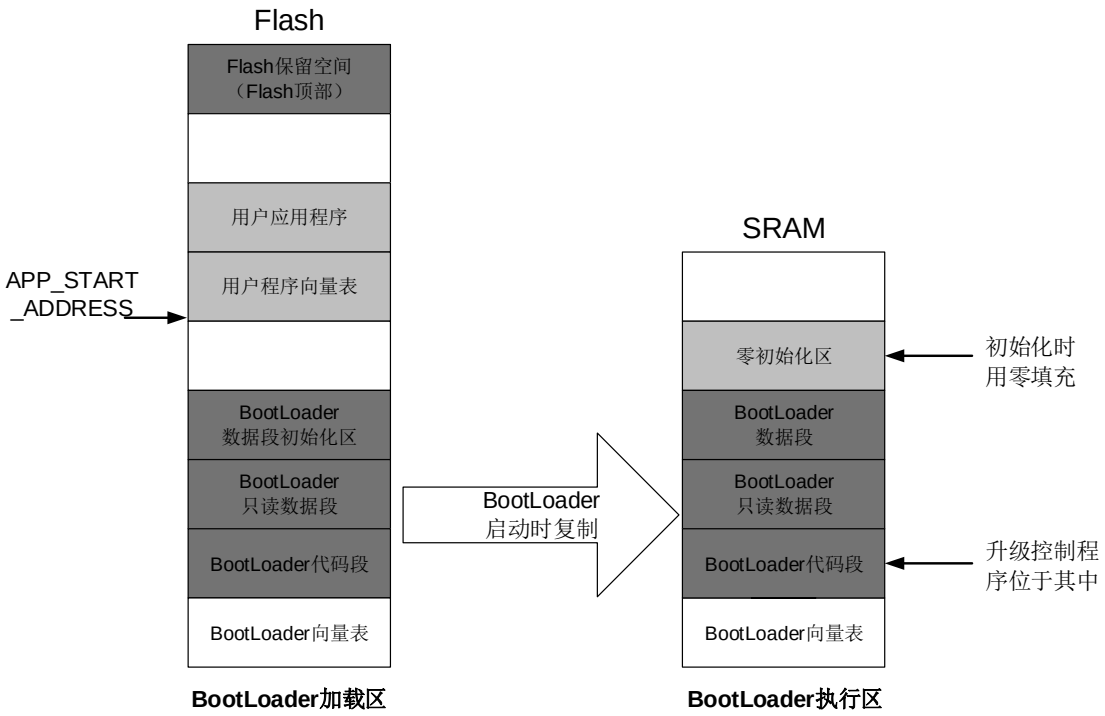


图 2-2 BootLoader 内存示意图

这样，执行代码的位置（SRAM）与修改操作的目标（Flash）相分离，一定程度上保证了软件升级的可靠性和安全性。这样的存储器映射允许修改 Flash 的全部代码。此外，BootLoader 也提供了保护机制，用于保护 Flash 中的 BootLoader 代码本身，以及 Flash 空

间顶部的一段存储区（保存即使 Flash 升级也不需要擦除的代码）。除非相应的配置选项使能，否则，这两段代码不可随意修改。

在 SRAM 中映像建立完毕并开始执行后，BootLoader 将调用 *CheckForceUpdate()*来决定运行 Application 还是 Updater。 *CheckForceUpdate()*会检测应用程序是否有效并检测用户配置的专用于升级的 GPIO 端口是否出现规定的电平特性。应用程序是否有效的判断依据是：（1）堆栈指针是否指向一个有效的位置（这个位置应当在 SRAM 里，即指针值应当是 0x2xxx.xxxx）；（2）复位处理程序的地址是否是一个有效位置（这个位置应当在 Flash 里，地址应当是 0x000x.xxxx，而且必需是偶数）。 上述两个条件都满足，表示应用程序有效。如果用户没有在 BootLoader 的配置文件 *bl_config.h* 里配置检测 GPIO 管脚（通过 ENABLE_UPDATE_CHECK 来配置），且应用程序有效，BootLoader 就选择执行用户应用程序。反之，只要应用程序无效， 或用户要求检测 GPIO 某管脚（通常那个管脚接一个按钮）且此管脚电平符合要求（一般设计成按钮按下）， BootLoader 执行升级程序。（默认复位时 PB4 引脚为低则运行 Updater，为高则运行 Application），如图 2-3 所示。引脚的选择及其极性的意义可在配置文件中修改。

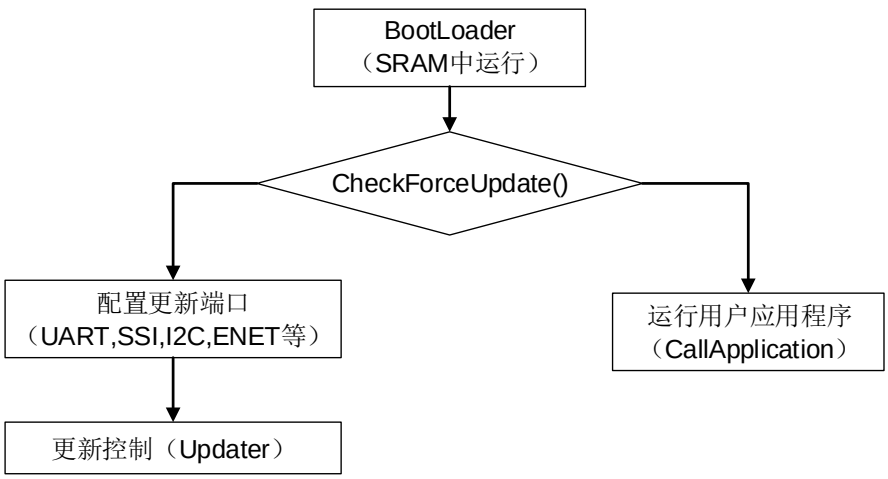


图 2-3 BootLoader 调用流程

由 BootLoader 加载 Application，与系统复位后直接加载 Application 有所不同。区别在于，在运行 BootLoader 的系统中，Application 的向量表并不位于 Flash 起始地址处。Application 有自己的向量表，且不与 BootLoader 共用向量表。这样做的目的，就是将 BootLoader 本身与 Application 分离开来，使两者具有一定的独立性。这也是由实际应用决

定的：BootLoader 需在系统投入使用之前编译完成并通过 JTAG 口或其它方式烧写进单片机，而 Application 作为 Flash 的升级程序，则通常在系统运行一段时间后才编写完成。

向量表能够分离，关键在于向量表偏移寄存器（NVIC_VTABLE）的使用。在硬件启动时，该寄存器会被复位为全 0，即向量表位于 Flash 空间起始地址处；当 BootLoader 在 SRAM 中建立映像之后，该寄存器被设置为 0x20000000，表示向量表位于 SRAM 空间起始地址处；如果选择执行 Application，那么该寄存器又被设置为 APP_START_ADDRESS（默认为 0x800，即 2K），表示用户应用程序的向量表位于 Flash 空间 BootLoader 以上的某一地址处（BootLoader 的代码量不能超过 APP_START_ADDRESS）。

我们这里对 BootLoader 概念和启动代码的部分基本知识作了不是很深入的介绍，有关 BootLoader 细节配置请参阅 TivaWare 安装后的文档目录里的“SW-BOOTLDR-UG-1.0.pdf”文档。

1.3 TivaWare 中各种库的介绍

本节只对 TivaWare 驱动库做基本介绍，目的是让读者了解各个驱动库的基本功能。如果要深入了解驱动库，要通过后续章节的实验并全面阅读 TivaWare 的驱动库的文档。这些文档都在安装时放在安装目录下的“/docs”的目录下。

(1) 外设驱动程序库（Tiva Peripheral Driver Library）

Tiva 外设驱动程序库是一组外设驱动程序的集合，这里的外设是针对 TI ARM® Cortex™-M 类 TivaTM 系列微处理器芯片内部自带的外部功能模块（相对于芯片内部的内核而言）；而驱动程序也不是在“操作系统”严格意义上的驱动，因为这里的驱动程序既没有面向操作系统的统一接口，也没有可以融入全局驱动程序架构（global device driver infrastructure）。 这里的驱动程序应当只能符合广义的驱动程序概念，它是为方便用户使用外设而设计的。 有了外设驱动程序，用户只需要了解外设操作的基本过程来调用相关的函数，无需记住或者查阅与这些操作相关的寄存器的详细信息和寄存器中位的安排，大大降低了开发的难度。

Tiva 外设驱动程序库支持**两种**使用它的**编程方式：直接寄存器访问方式和软件驱动方式**。根据开发的需要和开发人员的喜好，两种编程方式可以单独使用，也可以混合使用。每种模式都各有所特点。

直接寄存器访问方式顾名思义是通过直接对外设的寄存器进行读写来控制外设的行为，它只需包括可用于待开发的 Tiva MCU 命名的头文件（比如，TM4C123GH6PM 的头

文件是 inc/tm4c123gh6pm.h， 这个头文件里有寄存器的宏定义，第一章的实验就是直接寄存器访问方式的，它只包含了与 MCU 名相关的头文件）。这种方式可以生成效率更高、代码占用空间更小的可执行文件，但用户要了解各种寄存器的细节。有关头文件中的细节定义内容，请见本章实验一的部分。

软件驱动方式无需了解芯片内部的寄存器细节内容，通过直接调用硬件的 API 函数便可以实现快速的系统开发。当然软件驱动模式有一定的限制性，因为 TivaWare 的外设驱动库是面向 Tiva 系列的所有 MCU，对于某些 MCU 的部分硬件功能，未能做到全面支持。所以绝大部分应用可以直接采用软件驱动方式进行开发，但某些功能必须通过自行添加或修改库的代码来实现。在小型化的应用中，如果库的函数不能实现的功能，我们的策略一般是用直接寄存器访问方式来实现这部分功能，这就是**混合使用**。本实验套件的大部分实难都要依赖外设驱动库来开发，我们在后继的实验内容中会逐步加深我们对这个库的了解。

(2) 图形库（Tiva Graphics Library）

图形库是一组为了方便用户开发图形化显示（通常是在点阵 LCD 上）的函数的集合。它提供画线条、圆、长方形等各种图形基本单元的函数和一些快速实现显示控件（比如下拉列表、按钮、选择框等）的工具集，以便用户能快速实现这些部分的显示。

图形库并非只为 Tiva 服务，它也能用在 TI 的其它 ARM 处理器上，它的目的就是要降低显示器的电路板上创建图形用户界面。图形库由下至上由三个功能组成：**显示驱动层**（Display Driver Layer）、**基本图元层**（Graphics Primitives Layer）、**小控件层**（Widget Layer）。

显示驱动层：针对使用显示设备（点阵 LCD），移植相关的驱动程序;

基本图元层：可是实现无抖动的动态显示缓冲来绘制点、线、矩形、圆、字体、位图和文本等等图形图像。

小控件层：支持复选框、按钮、单选按钮、滑块、列表框等控件。

当 Tiva MCU 连接有显示设备并能够良好通信的状态下，通过外设的初始化能够使得 MCU 在显示设备中显示做多样的图形和文字。如下图 2-4 所示：

图 2-4 grlib 图像库的应用

(3) USB 库（Tiva USB Library）

TI 的 USB 库是一组与 USB 应用开发相关的数据类型和函数的集合。借用它们可以快速实现 USB 主设备（Host）、USB 从设备（Device）和 USB OTG（On-The-Go）的各类应用程序。USB 库的内容和与之相关的头文件分成四类：通用的功能函数；从设备的函数；主设备相关的函数以及模式检测和控制函数。

通用功能函数用于各种 USB 设备，包括主设备，从设备或两种功能都具备的设备。它主要包含了用于解析 USB 描述子和配置 USB 功能的函数。 从设备相关的函数主要功能是响应主设备的联接信号、应答主设备的描述子请求等等。主设备相关的函数提供从设备检测、总线上的从设备扫描、终结端子管理等功能。有关 USB 的应用，我们在第三章有详细解释，这里不再赘述。

(4) IQMath 库（IQmath Library）

当我们在开发一些实时的应用，这些应用对计算的速度和计算的精度都提出较高的要求时，我们要就考虑采用 IQMath 库了。它是高精度的数学运算库，并对代码的效率做过优化，调用库里的函数，我们可以将浮点数的运算融入（seamlessly port）定点数的算法代码。

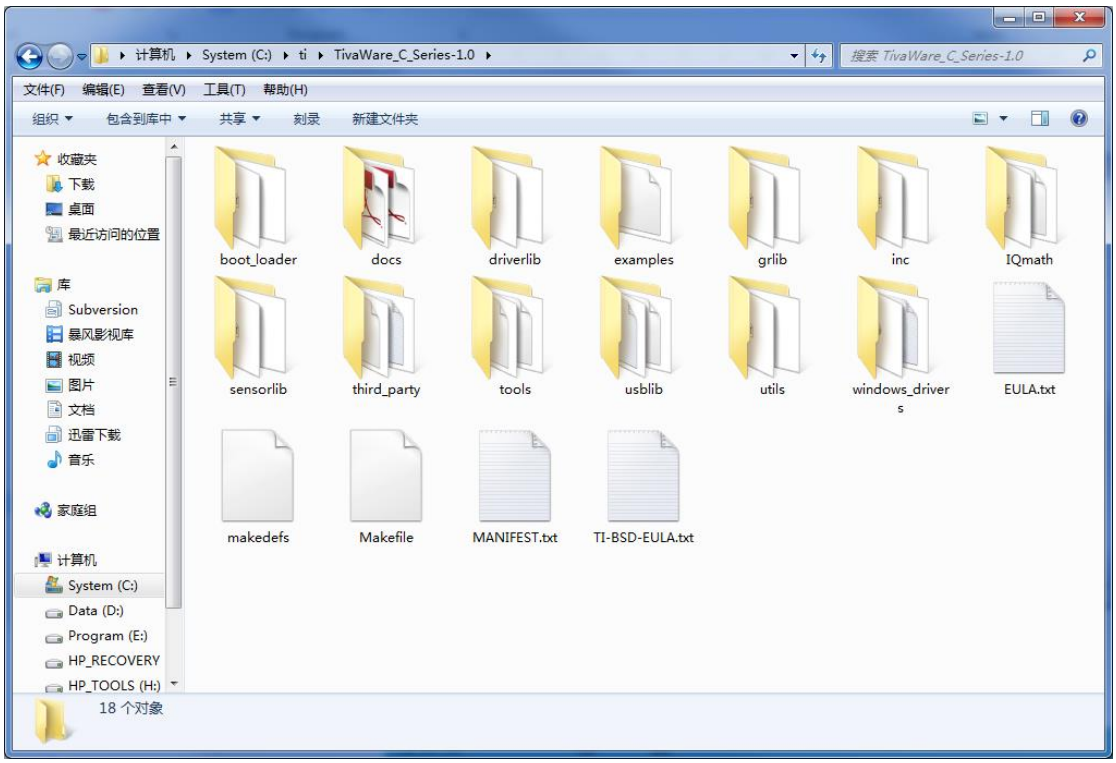


图 2-5 TivaWare 目录构成

(5) 传感器库（Sensor Library）

传感器库的函数让我们更容易实现对各类传感器的控制，这些传感器主要是环境参数传感器，包括加速度传感器、角度传感器（陀螺仪）、磁场传感器、温度传感器、压力传感器、亮度传感器、相对湿度传感器、位移传感器等。这些传感器与数据通信接口都采用了I2C，库函数主要实现对传感器的配置和数据传送。

1.4 TivaWare 安装目录下的结构

TivaWare 安装完成后，默认位于 C:\ti\TivaWare_C_Series-1.0。TivaWare 文件夹中包含 boot_loader、docs、driverlib、examples、grlib、inc、usblib、IQmath、sensorlib、third_party、tools 等文件夹，如图 2-5 所示。

下面就每个文件夹的作用作简单的介绍它的内容, 建议读者也打开自己 PC 上已安装的 TivaWare 目录，并依次浏览每个文件夹里的内容，尝试打开里面的文件进行查阅，以增加认识。每个文件夹的内容如表 2-2 所述。

表 2-2 TivaWare 目录结构说明

文件夹	文件夹内文档的简介
boot_loader	该目录包含引导加载程序的源代码
docs	TivaWare 文档
driverlib	外设驱动程序库
example	例子程序
grlib	该目录包含 TivaWare 图形库
Inc	该目录保持了直接寄存器访问编程方式的部分头文件
IQMath	该目录包含高精度数学运算函数库
sensorlib	该目录包含环境传感器函数库
usblib	该目录包含 Tiva USB 驱动程序库
third_party	该目录包含 Tiva 微控制器家族已使用（ported）的第三方软件包
utils	该目录包含一组实用程序函数，供示范应用使用

1.5 TivaWare 的特性说明

(1) 开发 TivaWare 之初定下的目标：

- 驱动程序全部用 C 编写，除非有些驱动程序无法用 C 语言来实现；
- 驱动程序演示了如何在常用的操作模式下使用 Cortex M；
- 驱动程序很容易理解，包括代码和程序的结构；
- 从内存和处理器使用的角度，驱动程序都很高效；
- 驱动程序尽可能具有良好的封装特性（self-contained）；
- 只要可能，可以在编译中处理的计算都在编译过程中完成，不占用运行时间；提高代码的可调试特性。
- 它们可以支持多种开发工具。

(2) 到当前版本为止，TivaWare 库的设计结果：

- 站在代码大小或执行速度的角度，驱动程序没有达到它们所能实现的最高效率。原因在于，虽然执行外设操作的最高效率的代码都可用汇编编写，

然后进行裁减来满足应用的特殊要求，但过度优化驱动程序的大小会使它们变得更难理解；

- 驱动程序不支持某些外设硬件的全部功能。尽管现有的代码可以作为一个参考，在它们的基础上增加对附加功能的支持，但是一些外设提供的复杂功能是库中的驱动程序不能使用的；
- API 中所有的错误检查代码都被移走。由于错误代码通常只在初始程序开发的过程中使用，所以可以把它移走来改善代码大小和速度。

对于许多应用来说，驱动程序可以直接使用。但是，在某些情况下，为了满足应用的功能、内存或处理要求，必须增加驱动程序的功能或改写驱动程序。如果这样，现有的驱动程序就只能用作如何操作外设的一个参考。TivaWare 支持的开发工具有：

- KeilTMRealView 微处理器开发工具；
- Tiva EABI 的 CodeSourcery Sourcery G++；
- IAR Embedded Workbench；
- Code Red Technologies tools。

所以在很多的文件夹下我们会看到 ccs、awarm、gcc、rvmdk 四个文件夹，实际上就是存放对应的不同编译工具的连接库。

2 理解 TivaWare 的外设驱动程序库

TivaWare 虽然在设计时就以易于理解为目的，但对于刚学过 C 语言而没有开发经验的读者来说，仍然难以入手。本节试图通过一系列实验，让读者能够理解 TivaWare 的外设驱动程序库的设计由来和结构特点。由于所有的库的设计方式相似，对外设驱动程序库的理解将大大帮助读者理解 TivaWare 的其它库。

注意：*实验步骤的内容可以保证你能依葫芦画瓢做出结果，但那只能确保你会画个瓢出来，如果要锅碗瓢盘都会画的，还必须仔细看实验原理的部分，并保证能亲自动手在理解的基础上完全过程的操作，包括代码的编写，数据测试。在实验阶段，应该牢记于心的是自己是一名坚忍不拔、实践至上的工程师，普通的技术工人和纯粹的学者都不能单独胜任设计 MCU 混合信号系统的实际工作。写一篇关于 MCU 或者模拟电子的漂亮的文章是一回事，而制作一套能正常运行各项指标都能满足要求的 MCU 及模拟电子的混合信号系统是另一回事。在认识到这一点的基础上，本书将通过大量的实验实践来突出理论基础，从而为读者提供更有效的指导。我可能不厌其烦得不断提到实践和理论的并重，因为确实很*

重要。

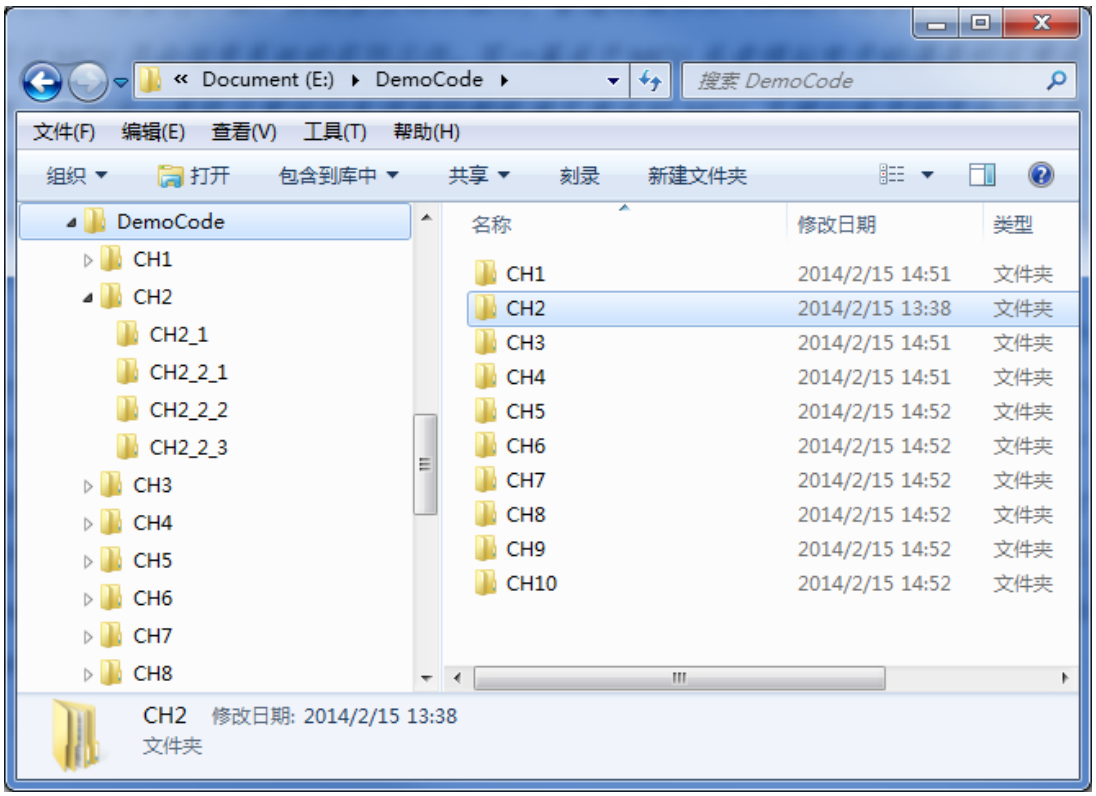


图 2-6 工程代码存放目录结构

建议：*为了更好学习本套件的配套的代码内容，并能与后续实验的操作保持一致的命名方式。建议在计算机上的某盘（比如 D 盘）根目录上建了一个目录一个叫“DemoCode”的文件夹，并在这个文件夹里按章节建好子文件夹，每个章节对应的文件夹按顺序叫“CH1”、“CH2”、“CH3”……，每个章节文件下再按实验需要的工程文件夹，可以取名为“CH2_1”，表示为第二章第一个实验的工程，或者某个实验需要多个工程，文件夹可以命名为“CH2_2_2”，表示为第二章第二个实验的第二个工程文件夹。形成如图 2-6 所示的目录结构，这样便于文档管理。*

2.1 采用直接寄存器访问模式的 GPIO 操作（实验一）

实验内容

在实验套件中的 Tiva LaunchPad 模块上实现如下功能：Tiva LaunchPad 模块上有一个 RGB 三色 LED，写代码实现让 LED 每种颜色依次循环点亮一小会儿，最终看到的结果是三种颜色轮流依次闪现。

实验目的

(1) 学习在 MCU 的 GPIO 管脚上外接 LED 的方式；

(2) 了解 GPIO 的初始化过程和各种寄存器的功能和使用方法。

(3) 学习使用 TivaWare 外设驱动程序库的其中一种编程方式：**直接寄存器访问方式**。

实验设备：

硬件： Tiva LanuchPad 开发板

软件： Code Composer Studio 5.4（或更新的版本）

实验原理

- **硬件电路原理**

在 Tiva LaunchPad 模块上，有一个三色 LED（红绿蓝，D1，靠近 Reset 按键），它与 MCU 的联接方式如图 2-7 所示。三色 LED 是由三个不同颜色的发光二极管封装在一起形成，它们通常有一个公共端（如果公共端是三个二极管的正端，称为共阳，反之称为共阴）。三个发光二极管分别通过其驱动电路连接到 MCU 的 PF1、PF2、PF3 的引脚上。

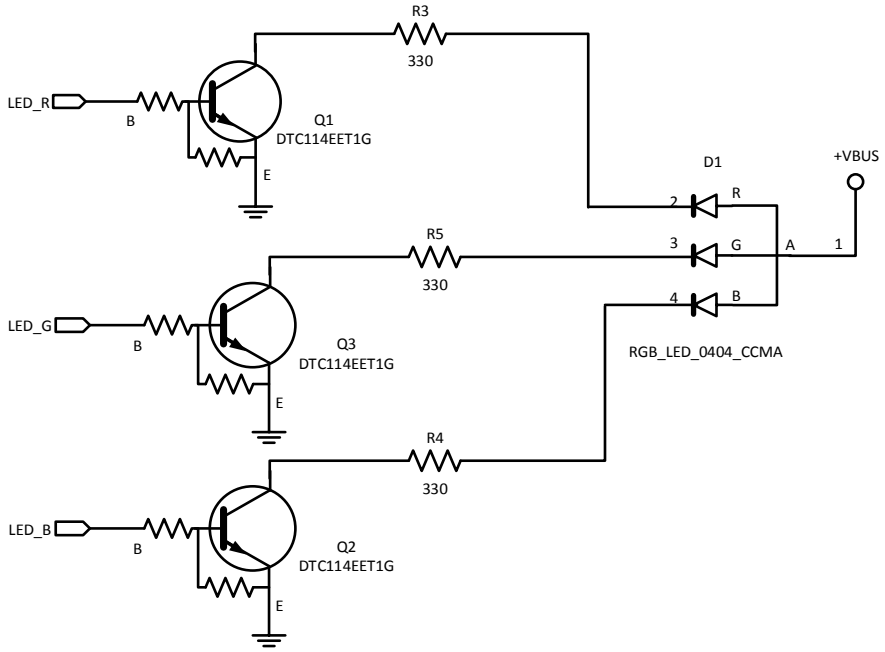


图 2-7 三色 LED 灯电路图

这里所谓的“LED 驱动电路”就是指一个型号为 DTC114EE 的三极管和一个电阻组成的电路。先来看三极管的作用，我们的目标是用 MCU 的 GPIO 管脚来控制 LED 的亮灭。所有的 GPIO 的管脚都有一定的驱动能力[3]（输出电流的能力和吸入电流的能力），通常用电流的大小表示，我们所使用的 TM4C123GH6PM 的 MCU，GPIO 管脚的最大输出电流为 8mA，最大吸入电流为 8mA，只有 4 个接口的吸入电流可以达到 18mA[4]。而我们实验套件里的三色 LED 中红光 LED 的点亮需要 10mA、绿光为 8mA、蓝光为 5mA[5]。所以直接用 GPIO 的管脚联接 LED 难以驱动 LED。我们知道三极管有电流放大作用，处于正常放大工

[3] 注：管脚的**驱动能力**是指这个管脚带负载的能力，或者更通俗地说是这个管脚最大能提供多少能量。单位时间内的能量表示为功率，而功率等于电流和电压的乘积，对于数字电路来说，管脚信号只有高电平和低电平两种状态，管脚为高电平时就会向外提供电流，低电平时就从外部吸入电流。因为对于特定的数字芯片，高电平和低电平的值都保持恒定值，所以管脚的驱动能力就用输出电流和吸入电流能力来表示。我们在日常生活中很经常听到“高压危险”之类的话，常常会误以为引起触电死亡的是电压，也会误认为只要是电压高就有很强的驱动能力。如果你用过电苍蝇拍就会明白，电苍蝇拍的电压可以高达 8000 伏以上，可以电死苍蝇，但不能使人体致死；而家庭内部的供电电源插座只有 220 伏，却足以使人体触电致死，是因为它们提供的电流大小不一样。

[4] 《Tiva™ TM4C123GH6PM Microcontroller Data Sheet》July 09, 2013, p1354, Table24-6

[5] Tiva Launchpad 的原理图和手册里没有提供三色 LED 的具体型号，这里的驱动电流数据是根据常用的 0404 大小的三色 LED 来说明的。

作方式的三极管集电极电流和基极电流的关系为：$I_c = I_b \times \beta$，其中 β 为三极管的电流放大倍数。以图 2-7 中红色 LED 一路为例，当 LED_R 输入点处为高电平（MCU 的对应 LED_R 的引脚为高电平）。三极管 Q1 的 C、E 两极导通，LED 正端接 VBUS（USB 接口提供的电压，电压值为 5V），负端通过三极管 CE 接地，形成电流通路，LED 点亮。根据 DTC114EE 的数据手册，三极管的集电极电流最大可以 50mA，足以驱动 LED；当 LED_R 处为低电平时，Q1 的 C、E 两极截止，LED 熄灭。

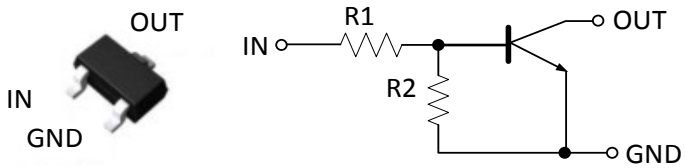


图 2-8 DTC114EE 的外形和内部电路示意图

DTC114EE 是内部自带偏置电阻的 NPN 型三极管，图 2-8 左侧为它的外形，右侧为它的内部电路。其内置的电阻 R1、R2 ，R1 = R2 = 10K。 它们起到三极管偏置电路的它们是薄膜电阻（Thin-film Resistor），相比于厚薄电阻（Thick-film Resistor）， 它具有精度高、温度系数小、寄生电感和等效电阻小等特点，因此它在电路中表现出稳定的特性，尤其在高频电路中（数字电路就是高频电路）。所以内置的薄膜偏置电阻即省去了外部的电阻，又提高了电路开关的可靠性。

图 2-7 电路中与 LED 串联的电阻 R3/R4/R5 为限流电阻。因为 LED 的特性本质上是一个二极管，它导通（点亮）时，导通电压是跟 LED 制作材料有关的一个恒定值，导通电流I_f随外加电压变大而快速变大，过大的I_f使 LED 很快损坏，所以要在驱动电路中加入限流电阻。这个限流电阻的阻值计数方式可以参考第五章有关 LED 驱动的描述。

- **软件逻辑**

有了电路的原理，我们还需要知道如何让 MCU 的 GPIO 端口产生我们想要的信号。

注意：*毕竟本书只是一本实验指南，我尽可能把实验套件的电路原理描述清楚，对于这些特定的电路，读者很难找到其它参考资料，而 MCU 的内容一方面很丰富，另一方面参考资料也很多，所以就不作详细说明。有关 TivaTM TM4C123GH6PM 的细节内容请参考 TI 提供的数据手册或相关的资料，类似的，后续有关 MCU 内容的原理细节都不在本书陈述，还请读者自己查找相关资料学习。这里只简要描述 GPIO 的操作顺序，要真正理解这*

些操作顺序，你必须先仔细地阅读数据手册或者相关资料。当然为了实现快速应用开发，你也可以只知道顺序而不必了解为什么是这个顺序，只有开发过程中出了问题再去有目的得仔细看资料，我个人倾向于这种学习方式，因为这样效率更高。但值得强调的是，当多份资料提供不同的说明时，我们必须以数据手册和 TI 提供的技术手册为准。

Tiva™ TM4C123GH6PM 的 GPIO 端口的操作过程如下：

a) 配置系统时钟

b) 打开与此 GPIO 端口对应的端口时钟。

c) 将端口配置为数字输入输出的 GPIO 功能，而不是端口的其它复用功能

d) 配置 GPIO 端口驱动电流大小和上下拉电阻的模式

e) 设置管脚的输入输出方向。

f) 设置管脚输出是高电平还是低电平。

对 GPIO 端口的操作有赖于对 MCU 众多寄存器的控制，借助 TivaWare 我们可提高我们的开发效率，降低开发难度，增加代码的可维护性。先来说明 TivaWare 的第一种使用方式：**直接寄存器访问模式**。

- 直接寄存器访问模式（Direct Register Access Model）

第一章的实验例子程序就是采用直接寄存器访问模式的。在那里我解释了寄存器宏定义的规则，如果你对寄存器的宏定义还不清楚，你需要返回到那一部分再仔细读一遍。事实上，那个程序并不是纯正意义上的直接寄存器访问模式的程序，之所以这么说是因为它只用到了 TivaWare 有关这种模式的部分宏定义。下面我们来解释其它的宏定义。

当我们要对 Tiva 的某个外设模块进行操作时，我们总是先去看它的数据手册的相关内容。比如我们要对同步串行接口 SSI 进行操作，我们会先去翻看数据手册中有关 SSI 的章节，当我们第一次用 SSI 时，我们会通看这个章节的全部内容。从功能描述、端口信号特点、操作流程以及各个寄存器的功能，如果我们对 Tiva MCU 的 SSI 已经熟悉，我们只会关注寄存器的细节定义的章节（图 2-9），很少有人能记住这些寄存器的定义细节（图 2-10），所以每次编程时总要去看它们，并根据细节定义来编程。

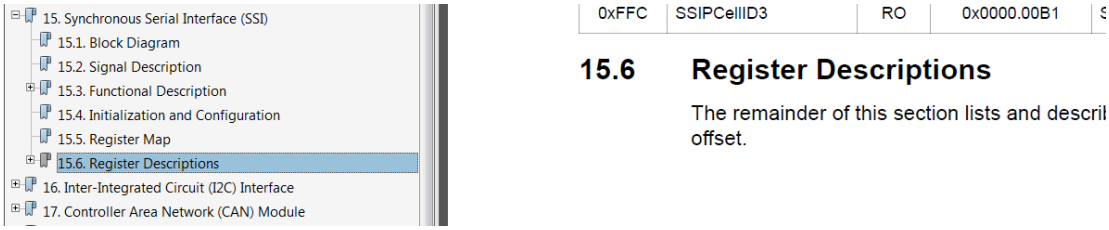


图 2-9 SSI 寄存器描述一节在数据手册中的位置

寄存器的宏定义（第一章例程中用到的宏定义）免除我们写代码时要写**寄存器地址**（图 2-10 中所示）的苦恼。

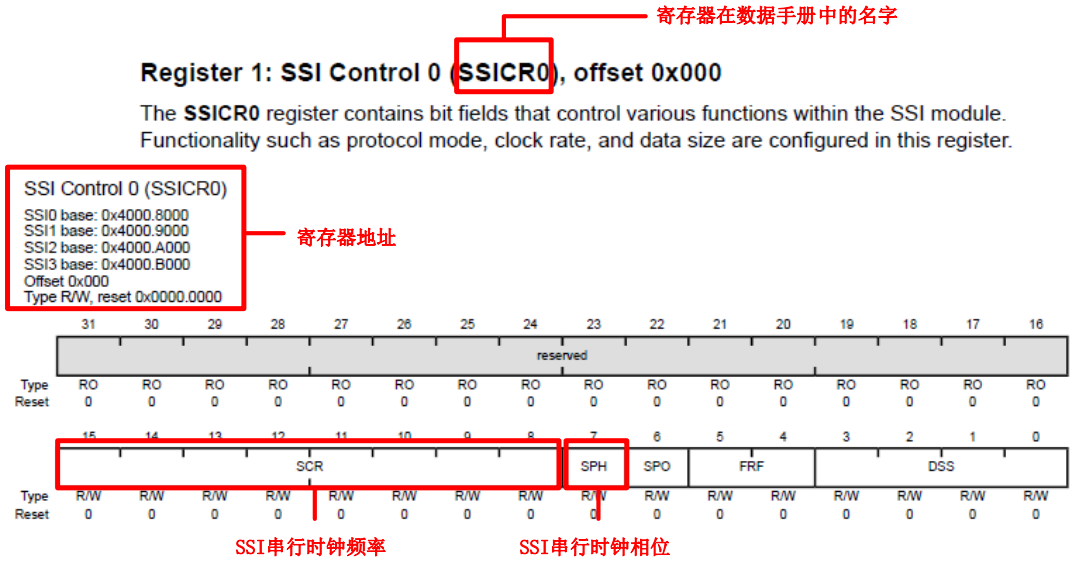


图 2-10 SSI CR0 在数据手册中的数据位定义表

TivaWare 的寄存器宏定义在以芯片名命名的头文件中，针对我们的实验套件，它的名字为“tm4c123gh6pm.h”，在 TivaWare 安装目录下的 inc 文件夹内。**寄存器的宏总是以“_R”结尾；以功能模块的名字开头；以寄存器在数据手册中的名字（图 2-10 中所示）作为第二个字段。**比如“SSI0_CR0_R”就代表 SSI0 模块中的 CR0 寄存器。GPIO 的稍有例外，因为有多个 GPIO 接口，定义成：GPIO_PORTA_DIR_R。用 CCS 打开这个头文件，可以看这些定义。

当只有对寄存器的宏定义时，写程序代码时还必须根据需要设定的功能查**寄存器数据位定义表**确认给寄存器赋什么值。寄存器中不同的位用来定义不同的功能，有些功能只需一个位来定义，比如 **SSI 串行时钟相位** SPH（图 2-10），有些功能需要多个位来定义，比

如 **SSI0 串行时钟频率** SCR（图 2-10）。不论是一个位的还是多个位的功能区块称之为“**功能域**”。每个功能域在数据手册中都有一个名字。现在假设我们要根据需求，要在 SCR 中设置值“0x08”和 SPH 中设置值“1”。这时我们要根据在整个 32 位的寄存器中 SCR 和 SPH 的位置先计算好将要设定的值为：0x00000880，然后写出如下程序代码：

```
SSI0_CR0_R = 0x00000880;
```

显然，每次修改寄存器的设置或理解其他人写的类似代码都必须对照数据手册中的寄存器数据位定义表并计算，这使得代码的可读性和可维护性很差。因此，TivaWare 定义了一组对寄存器设置的宏，这些宏的定义规则如下：

- 这些宏的定义也在以 MCU 型号命名的头文件中（本实验套件对应着 tm4c123gh6pm.h 文件）。
- 当某个功能域有多个位组成时，宏定义中必然存在一个以“_M”结尾的宏。比如 SCR 域，就用一个名为 SSI0_CR0_SCR_M 寄存器，这个宏称为**功能域掩码**。它被定义为一个只在此功能域对应的位上为“1”而其它位为“0”的数。比如 SCR 功能域，根据图 2-10，它位于整个 CR0 寄存器的第 8 位到第 15 位之间，因此它的功能域掩码就是这些位都是“1”，其它位都为“0”。功能域掩码的定义如下面代码所示，可以在 tm4c123gh6pm.h 文件中找到：

```
#define SSI_CR0_SCR_M           0x0000FF00
```

功能域掩码在读取某个功能域的值的时侯很有用，它“遮掩”了其它功能域的影响，让你只关心这个功能域，比如我们要从 CR0 读取 SCR 的内容，就可以写成：

```
//通过掩码"与"运算实现读取功能域内容
ulValue = SSI0_CR0_R & SSI_CRO_SCR_M;
```

其实光有掩码还不够，因为取得的数据还会处在整个 32 位数的中间部分，我们通常还需要把这部分数据移位到数据位的最低位那一头。为了实现这个功能，TivaWare 定义了另一个宏，它总是以“_S”结尾， 可以称其为**功能域偏移量**。它定义一个多个位数组成的功能域的最低位和整个 32 位寄存器最低位之间的位数差。还是以 SCR 功能域为例，它的最低位是 CR0 寄存器的第 8 位，SCR 功能域偏移量宏就定义为：

```
#define SSI_CR0_SCR_S           8    //SCR功能域偏移量宏
```

有了偏移量宏，上面读取功能域值的方式可以改为：

```
//通过掩码"与"运算后，再移位偏移量宏所定义的位数，读取某一功能域的值。
```

```
ulValue = (SSI0_CR0_R & SSI_CRO_SCR_M)>>SSI_CR0_SCR_S
```

功能域偏移量宏也可以用于设置值：

```
SSI0_CR0_R = 5 << SSI_CR0_SCR_S;   //与下面语句的设置等效
SSIO_CR0_R = 0x00000500;
```

因此，由多个位组成的功能域的设定和读取，经常会用到功能域掩码和偏移量这两个宏。

- 如果功能域只有一个数据位构成，TivaWare 就不会有对应的掩码和偏移量，通常只定义在这一位所在的位置上值为“1”而其它位为零的宏。这种宏的作用是方便对这个位进行读写。比如 SPH 域有如下定义：

```
#define SSI_CR0_SPH             0x00000080
```

有了这个宏，我们在 SCR 中设置值“0x08”和 SPH 中设置值“1”的代码变为：

```
SSI0_CR0_R = (0x08 << SSI_CR0_SCR_S) | SSI_CR0_SPH;
//上语句等效于：
SSI0_CR0_R = 0x00000880;
```

- 有关功能域设定和读取的宏都采用统一的命名方式，即“模块名_寄存器名_功能域名”，而且模块名、寄存器名、功能域名都保持与数据手册中一致，程序的可读性和可维性得以提高。

注意：*同在 TivaWare 安装目录中的 inc 文件夹下，还有大量以“hw_*.h”命名的头文件，它们是用来使用软件驱动模式编程时要包含的头文件，这些文件里也有很多与上述宏同样的定义，所以不要在一个源文件里同时包含这些文件和我们上面提到的以 MCU 型号命名名的头文件（tm4c123gh6pm.h），那样会引起 CCS 报警（重复定义）。*

实验步骤

(1) 将 Tiva LaunchPad 模块通过 USB（ ICDI/Power）接口连到 PC，并将模块上的 Power Select Switch 开关拔到 DEBUG 位置。

(2) 在 CCS 中创建一个空的工程，工程的名称为“CH2_1”,并将工程存放到一个新的目录“CH2_1”。

(3) 在程序代码编辑窗口里开始写代码，因为我们选择了**直接寄存器访问方式**，因此要包含的头文件有 1）“tm4c123gh6pm.h”，它有我们要用到的宏定义；2）

“stdint.h”，它有 int 数据类型的重定义，如果你的程序中还使用了别的数据类型，比如 bool 型，那你还要包含“stdbool.h”。

(4) 对 GPIO 端口做初始化设置；根据 GPIO 初始化的要求，我们需要 F 端口的总线形式、启动 F 端口的时钟、将 F 端口设置工作在输出模式并使能该模式；在主函数中添加如下代码：

```
#include <stdint.h>
#include <inc\tm4c123gh6pm.h>
void main(void)
    {
    volatile uint32_t ulLoop;
    //将F端口的总线形式设置为AHB总线形式
    SYSCTL_GPIOHBCTL_R = SYSCTL_GPIOHBCTL_PORTF;
    //启动F端口时钟
    SYSCTL_RCGCGPIO_R = SYSCTL_RCGCGPIO_R5;
    //设置PF3、PF2、PF1引脚为输出
    //GPIO的某些寄存器位是直接与端口管脚号对应，因此没有掩码之类的寄存器
    GPIO_PORTF_AHB_DIR_R = 0x0000000E;
    //激活PF3、PF2、PF1引脚的数字输入输出功能
    GPIO_PORTF_AHB_DEN_R = 0x0000000E;
    while(1)
    {
        GPIO_PORTF_AHB_DATA_R = 0x00000002; //绿灯亮
        for(ulLoop = 0; ulLoop < 200000; ulLoop++){}
        GPIO_PORTF_AHB_DATA_R = 0x00000004; //蓝灯亮
        for(ulLoop = 0; ulLoop < 200000; ulLoop++){}
        GPIO_PORTF_AHB_DATA_R = 0x00000008; //红灯亮
        for(ulLoop = 0; ulLoop < 200000; ulLoop++){}
    }
    }
```

我们采用直接寄存器访问方式修改 LED 端口的电平状态来驱动 LED 的亮和灭。其中 For 循环内多次连续的空操作起到延时的效果。

(5) 写完代码之后，没有做工程编译属性设置前，你会发现如图 2-11 中标识（1）所示的 CCS 中桔黄色？和对应代码内容下面有桔黄色波浪线。CCS 的内置自动包含文件路径检查功能，不能通过已有编译设置找到的头文件都会被标识。如果不做任何设置，就直接点击绿虫子，编译器会报告头文件找不到的错误。根据的提示，完成工程编译属性设置，再回到 CCS 的 main.c 的代码里，可以观察到原来的黄色的？标记已经消失，表示头文件包含路径设置完成了。

(6) 点击绿虫子，实现编译、链接和下载可执行代码到目标设备。

(7) 完成编译和下载过后，程序会停留在 main 函数的第一句话上，等待下一步的调试动作。通过 run-resume、F8 或者 Debug 调试框中的开始按钮都能够执行后续代码，也可以通过单步调试观察 GPIO 的寄存器的被修改状态，如图 2-12。

(8) 执行后可以观察到三色 LED 灯的闪烁。

❶ 编译器在未找到正确路径时会以“？”提示。

❷ 黑色加粗字体表示当前 Active 的工程，右击工程名得到快捷菜单，单击其中“Properties”选项得到图示“Properties for CH2_1”窗口。

❸ ARM Compiler为编译特性设置。

❹ ARM Linker为链接特性设置。

❺ Include Options为头文件包含路径设置。

❻ “Add dir to #include search path”中的“ ”单击后可添加新的头文件路径。如对话框“Add directory path”所示。

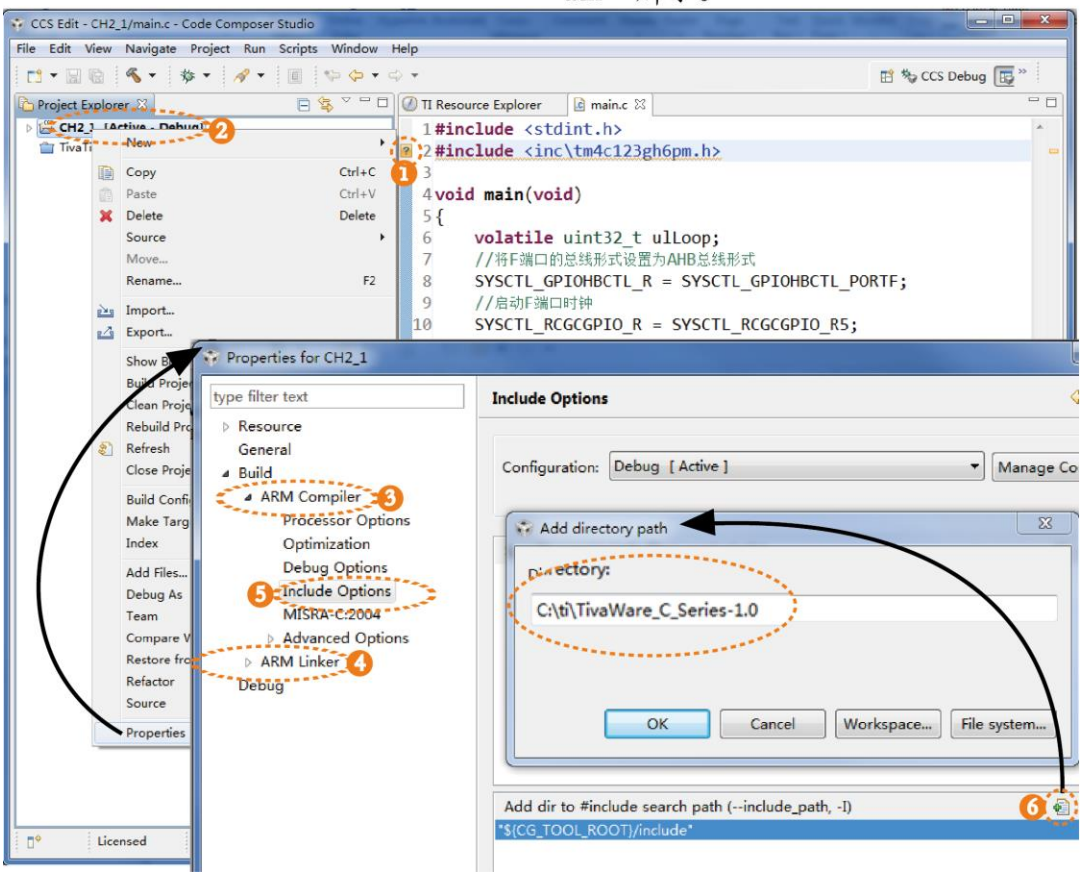


图 2-11 编译包含路径的设置

、

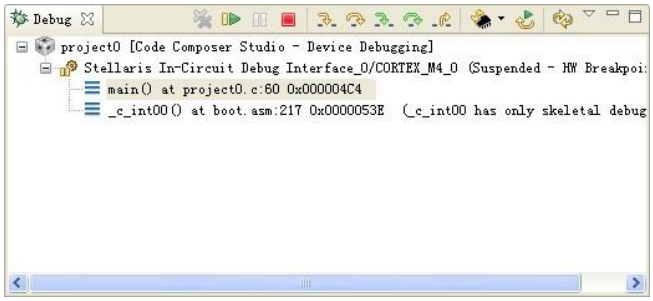


图 2-12 CCS Debug 功能框

实验结论

实验结果：当 Launch Pad 开始正常的运行时，能够看到红绿蓝三色 LED 灯依次点亮/熄灭。

总结：在整个程序中，我们没有使用任何的函数调用，全部都是通过寄存器的直接修改来改变 GPIO 的工作模式和状态的。虽然各种宏定义提升了程序的可读性和可维护性，**直接寄存器访问方式编程**仍然要了解寄存器的操作细节，编程效率相对较低。但这种方式写的代码执行效率高。这种开发方式，在简单逻辑和小量代码情况；或者在代码性能要求很高的情况下常被采用。

2.2 库的实现（实验二）

预备知识

虽然这一章要解释的是 TivaWare 的应用方式，做完外设驱动程序库的**直接寄存器访问方式**的实验，按理应当轮到**软件驱动方式**的实验了。但我仍然坚持安排这一小节的内容，来介绍由多个源文件组成的工程和有关库的一些概念。一直习惯在一个源文件（通常是 main.c）里完成所有工作的，从来就不曾了解编译、链接和库的概念，在我所认识的人中这样的 MCU 爱好者并不占少数，他们中确有不少能够在 CCS 里快速做出一个能跑并能用的程序，这也是他们爱上 MCU 的原因之一：快速成功的成就感。但是一旦程序变长并出了非语法问题，其实也没那么长，也许 500 行，他们可以会花很久去调试，如果不能成功，常常会情绪失控。没有库的基本知识，直接引入 TivaWare 的**软件驱动方式**编程，是否真的能理解，我一直对此持怀疑态度。想必不理解的结果就是不得其髓，空有其皮肉，对于手上的代码没有主自权。

绿虫子后隐藏的过程

现在的大部分集成开发工具中，为实现某一目的并相互有关联关系的文档通常被组织成一个叫**工程**（project）的单元里。CCS 也是如此，它以工程为管理项目的基本组织单元，于是 CCS 的主菜单里会专门有一项叫“**project**”的内容。

CCS 里的工具栏上有一个绿虫子（  ），写完代码，大家习惯性地按一下绿虫子来开始调试[6]程序。你可能知道按下绿虫子之后，CCS 后把我们写的代码转换成芯片可以执行的指令并下载芯片中。但这个过程不像我们想象得那么简单，接下来我会描述绿虫子背后的过程。

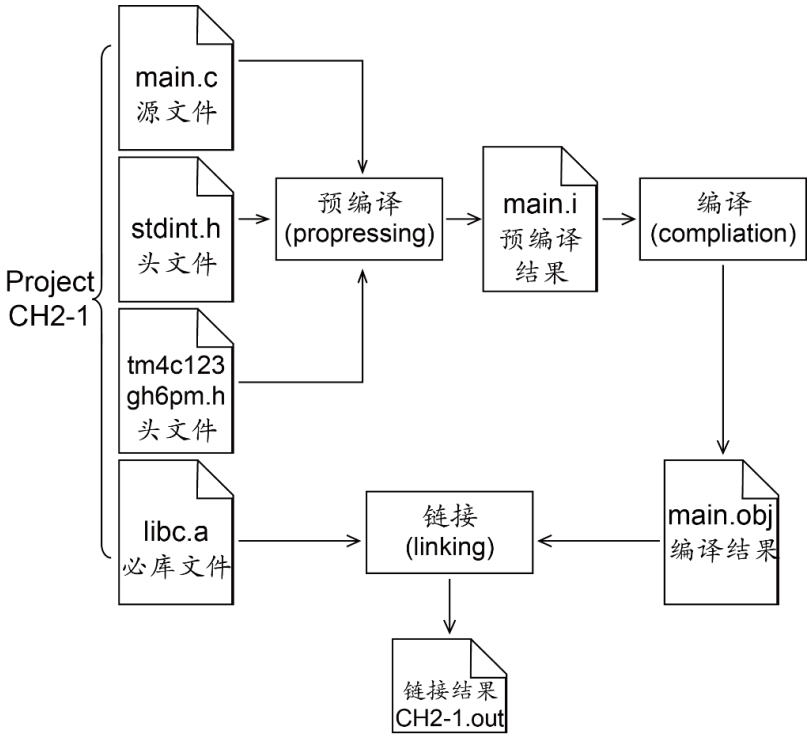


图 2-13 预编译、编译及链接

事实上，这个过程分为 4 个步骤，分别是预编译（Prepressing），编译（Compilation），链接（Linking）和装载（Loading）。以本章的实验一为例，如图 2-13 所示。通常以“.C”为扩展名的文件称为源码文件，以“.h”为扩展名的文件称为头文件。一个工程（Project）内

[6] 因为程序中存在的问题叫 bug, bug 的英文原意就是虫子；调试 debug，就是把虫子打死。所以 CCS 用一个绿虫子作为这个功能的按键。

部可能包含多个源文件、头文件以及库文件，后续我会说明工程的概念。CCS 会以源文件为组织单位，每个源文件和它包含的头文件会先被送到预编译器中处理，处理结果生成一个与源文件的文件名相同而扩展名为 “.i” 的预编译结果。实验一中只有一个源文件 main.c，所以只生成一个预编译结果 main.i。 main.i 只是个中间结果，它不最终保留在硬盘上，它作为编译过程的输入，编译完后会生成一个与源文件名相同扩展名为 “.obj” 的编译结果，通常它被称为目标文件。如果只有一个源文件那就只生成一个目标文件，如果有多个文件就会生成多个目标文件。将多个目标文件和这些目标文件用到的库 “组装” 在一起的过程称为链接，后续我会介绍链接和库的概念。链接生成的结果是以工程名为文件名，“.out”为扩展名的文件，它已经是最终的可执行代码，这个可执行代码要在 MCU 上执行，它还要被装载到 MCU 里。我们如果仔细比对第一次点击绿虫子之前和之后工程文件的文件夹的文件名和文件创建时间，可以更清楚得认识到绿虫子所做的事远比我们想象得丰富。

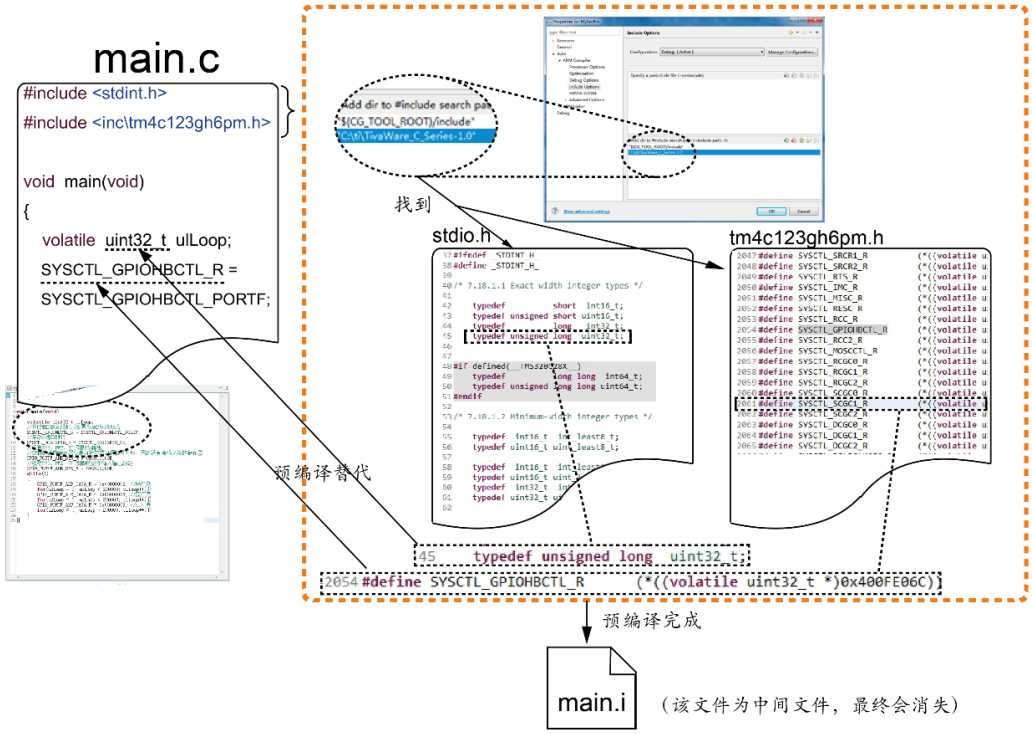


图 2-14 预编译过程

预编译

预编译过程主要处理那些源代码中以“#”开始的预编译指令，比如“#define”，“#include” “#ifdef”, “#endif”等等。首先是源码文件和相关的头文件输入到预编译器中，前一章我们说明了如何在 CCS 里设置工程的属性，主要是为在按下绿虫子之后 CCS 能按工程属性设

定的路径找到相应的文件。 我们前面说 CCS 是以源文件为单位进行预编译，因此 CCS 会先找到工程中的源文件载入预编译器，预编译器看到源文件“main.c”的头两行文件包含指令，就会沿 CCS 工程属性设定的路径去查找相应的头文件，注意这个过程是递归进行的，如果找到的文件里又包含其它文件，就会继续找，找到的文件的内容被插入到源代码文档中。然后所有的“#define”被展开，如图 2-14 所示。预编译的结果会形成一个中间文件“main.i”，它作为下一步**编译**的输入，编译结束后它就消失。

编译

编译是一个很复杂的过程，详细描述这个过程是计算机专业的一门课程，我只能简单得说这个过程就是**编译器**把预编译形成的结果进行各种分析，最终生成目标文件的过程。如果这个过程有错，编译器会报告错误，所谓的错误就是编译器认为不合语法规范的代码。在实验一中，编译过程可以理解为输入“main.i”，输出目标文件“main.obj”，目标文件不像中间文件会消失，它会保留在我们的计算上。如图 2-15 所示，如果你试图看一下 main.obj 这个文件的内容，你可以强行用 Notepad 之类的软件打开它，但你会发现都是看不明白的乱码，因为它已经是机器指令。但你可以用 Ultraedit 之类的软件以十六进制形式呈现目标代码的内容，如果你对机器指令有一定的了解，在十六进制形式的内容呈现中会看到某些熟悉的内容。

链接

链接通常是一个让人比较困惑的过程，都已经编译完了，为什么不直接输出可执行的文件？其实链接也是随软件规模发展而产生的。最早在纸带上穿孔的程序即不编译也不链接，很难想象那时的程序员是怎么检查长达几米的纸带程序，那个黑暗的时代程序代码不可能太长。随后汇编指令和 C 语言出现了，程序的规模迅速扩大，大规模的程序代码量动辄上百万行。如果把所有的代码都写到一个文件里，维护的难度可想而知，为了看懂程序，你很可能不断地按“Page Down”“Page Up”键以便浏览整个源文件。

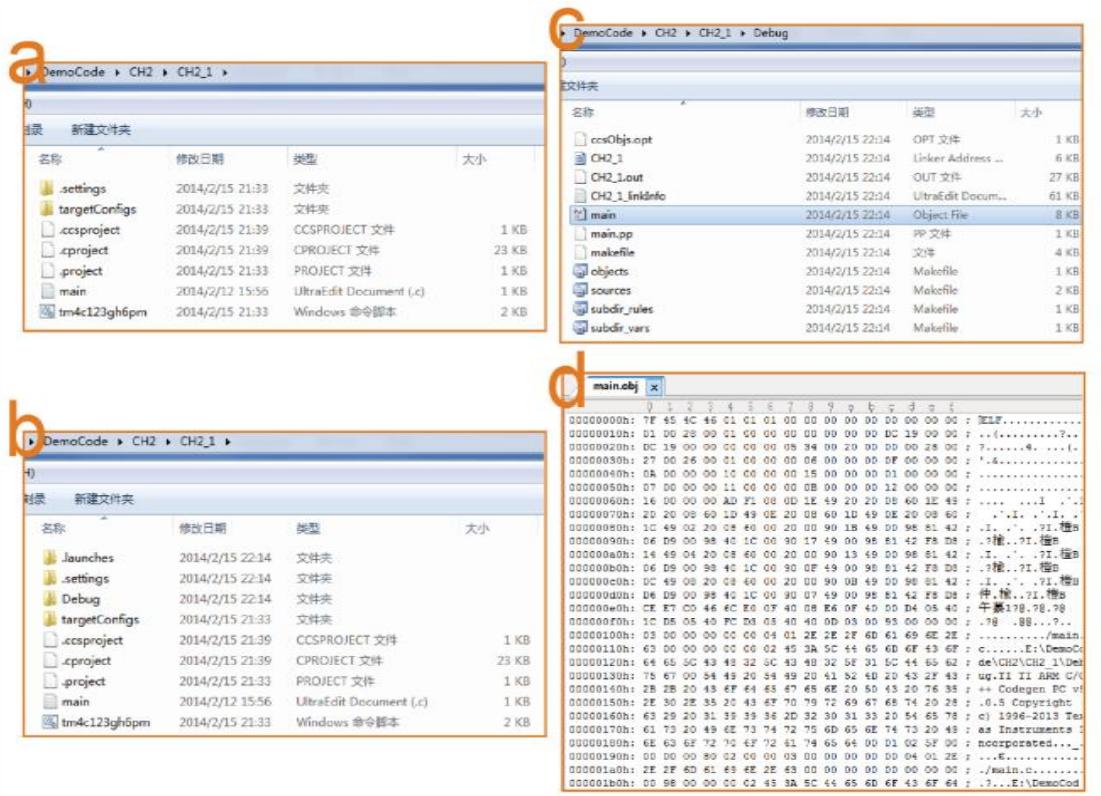


图 2-15 编译前后工程目录的变化

系统变复杂之后，程序设计的模块化就成为必然，因为必须将一个复杂的系统逐步分割成小的模块以达到各个突破的目的。将系统分割成小模块的最基本方式就是把原来过长的文档分成多个文档来管理。结构化的程序通常是由多个功能函数组成，把类似功能的函数集中到一个文档里管理是最直接想到的分割方式，当然如何分割模块也有很多讲究，这不是我们要讨论的内容。分割后的多个文档最终还是要被组装到一起，我们把多个文档“组装”起来，这个组装的过程就是链接（Linking）。随着系统变复杂，发现每次代码被修改后将系统的所有代码都合在一起编译一次要花很长的时间，这个时间长得不可接受。设想一个程序员发现程序中有一处小错误，然后他花了三分钟修改了某一个源文件的某几行代码，然后就想试试作了这样的修改后，错误是不是不见了，然后他要编译系统所有的代码，一个小规模系统的编译时间就常常超过半小时，等了半小时后他终于等到编译完成，试运行之后发现错误还在，然后他又花了三分钟改程序，花了半小时等编译完成……如果都这样的调试节奏程序们早就疯了。所以分割好的文件模块在第一次编译时就完成生成一个目标文件，如果这个模块的代码在系统修改时没有改动，它就不会再编译，编译系统只把修改

过的文档模块进行再编译，这样就快很多。所以链接是发生在编译之后，它把多个目标文件进行组装，而不是发生在编译前把多个源文件进行剪切粘贴式机械式地组装。有些目标文件里的功能很通用，不仅在这个系统里有用，在另一个系统里也有用。我们把这些目标文件包装在一起，形成一个所谓的“库”，这样下次我们用到这些功能的时候，只要在我们设计的系统中加载库文件就可以了。链接的主要内容就是把各个目标文件和库文件间相互引用的部分处理好，使得各个文档之间能够正确地衔接，这个过程是很有技术含量的，但我们现在只需要知道它是把目标文件和库文件进行组装的过程。

库里的各种功能对外只提供**接口**规范，而不展示**实现**的细节。接口和实现这两个术语含义有时不太容易理解，我举个简单的例子来让大家认识它们的主要含义，但远不是这个术语的全面描述。众所周知，C 语言里函数的定义和声明是两个不同的概念。函数的声明是为了告诉编译器这个函数被调用的时候叫什么名字；有多少个调用参数；各个参数的类型及排列顺序；函数的返回类型等信息。函数被调用前都要先声明，因为编译器要知道函数的上述信息才能判断程序员写的函数调用是否正确。函数的定义是函数功能的实现代码。当一个函数功能实现完成后，我们去用它的时候只需在意它的声明，而不必在关心它的实现。想必你在用 C 语言调用 printf 这个函数时从未关心过它是怎么实现的，而只在乎用要以什么样的参数来调用。我们的教课书上也是这么告诉大家的，调用 printf 函数时，要以双引号把第一个参数包起来，那是因为 printf 的声明中要求第一个参数就是字符串型的。事实上函数的声明就是它的**接口**规范， 函数的定义就是它的**实现**细节。

另外，调用 printf 时要先用#define 预编译指令去包含 “stdio.h”这个头文件，这个头文件叫标准输入输出头文件（stdio = standard input and output）。stdio.h 中就有 printf 函数的声明，但没有定义，你可以打开你机器上的 stdio.h 文件来看个究竟。包含的头文件在预编译时就被加入到源文件中（图 2-14）。所以编译时 printf 函数的声明已经存在了。编译只对语法进行分析，所以此时即使没有 printf 的定义也能通过编译。紧接着的链接过程才会真正用到 printf 的定义，链接过程就是解决函数调用的代码引用问题。但我们通常不看也看不到 printf 的定义，因为最早的程序员发现大家都要用 printf，于是把这个功能的定义代码（也就是实现）编译完的目标文件放到一个叫标准库的库文件里。链接程序扫描目标文件，当它发现有一个函数的调用，就会去找函数的定义，函数的定义可能在这个目标文件内、另一个目标文件内或者库文件内，不管定义哪里，链接的任务就是保证顺利引用到定义。所以当我们要引用库文件的内容时，要在工程的链接设置中注明库文件的位置及名字。

现在我把多个文件组成的工程的生成过程描述成图 2-16 所示的样子，大家应当有更深入的理解了。

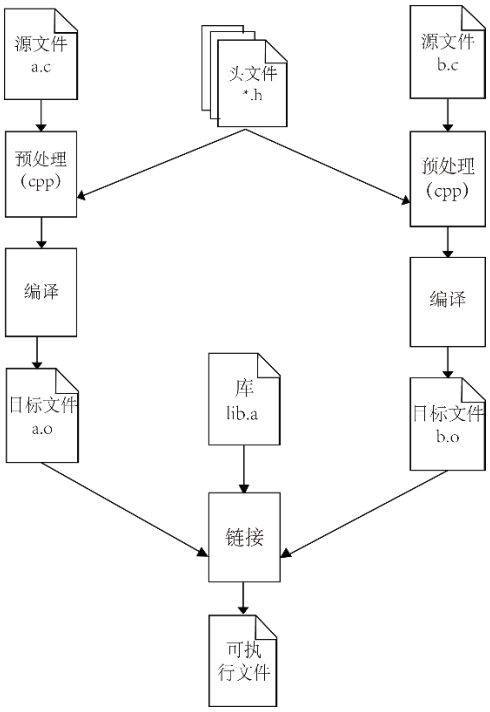


图 2-16 多个文件组成的工程的生成

实验内容

修改实验一的工程，将源程序中有关端口初始化、IO 端口读写的部分及延时功能单独分割到另一个文档中，使其变成有多个源文件组成，随后将其变成一个可供调用的函数库，并在一个新建的工程里调用这个函数库。工程仍然实现与实验一一致的功能。

实验目的

理解由多个源文件组成的工程的预编译、编译及链接及装载过程，理解库的概念。从而理解 CCS 的部分编译，链接设置，以便在将来自主开发过程中更好掌握 CCS 的使用。

实验设备

硬件： Tiva LanuchPad 开发板

软件： Code Composer Studio 5.4(或更高版本)

实验原理

在实现实验一的功能的前提下，修改整个 main 函数中的代码，将零散的关于 GPIO 寄存器的操作加以函数包装。

根据实验一的代码分析，可以将其内部的代码根据实现的功能分为：

- 初始化 GPIO F 端口；
- 修改 GPIO F 的某个特定端口；
- 一定时间长度的延时；

整个程序的逻辑流程实际上就是初始化后循环执行修改端口、延时、修改端口、延时…。

不妨将这三种逻辑块的代码都分别放在特定的函数中，而在 main 函数中直接调用该函数即可。我们把三个功能独立到一个新的文件中，以“Aiyan_gpio.c”来命名这个文件。它的代码如下所示：

```
#include <stdint.h>
#include <inc\tm4c123gh6pm.h>
#include "Aiyan_gpio.h"              //包含三个函数定义的头文件

void main(void)
    {
   Aiyan_GPIOInit();
   while(1)
   {
       Aiyan_GPIOPinWrite(0x00000002); //红灯亮
       Aiyan_Delay();
       Aiyan_GPIOPinWrite(0x00000004); //绿灯亮
       Aiyan_Delay();
       Aiyan_GPIOPinWrite(0x00000008); //蓝灯亮
       Aiyan_Delay();
    }
    }
```

由于我们需要在 main.c 文件中调用这些函数，而调用函数需要先声明函数，但不需要这个函数的实现。所以我们再在工程中加入一个头文件，它只对这些函数进行声明。这个头文件命名为“Aiyan.h”，它的代码如下：

```
#ifndef __AIYAN_GPIO_H__
#define __AIYAN_GPIO_H__
```

```
void Aiyan_GPIOPinWrite(uint32_t data);
void Aiyan_Delay();
void Aiyan_GPIOInit();

#endif
```

除了三个函数的声明外，我还加入了几行预编译指令，现在先不要关心他们的作用，我后面会解释。有了这个头文件之后，我们的 main.c 文件可以采用一条预编译包含指令#include 把这个头文件包含进来，就相当于在 main.c 文件里对这个三个函数进行声明，因为所有的#include 指令在预编译时就会将被 include 的文件完整得插入到这条指令所在的那一行。于是 main.c 文件相比于实验一就“瘦身”了，代码如下所示：

```
#include <stdint.h>
#include <inc\tm4c123gh6pm.h>

void main(void)
    {
   Aiyan_GPIOInit();
   while(1)
   {
       Aiyan_GPIOPinWrite(0x00000002); //红灯亮
       Aiyan_Delay();
       Aiyan_GPIOPinWrite(0x00000004); //绿灯亮
       Aiyan_Delay();
       Aiyan_GPIOPinWrite(0x00000008); //蓝灯亮
       Aiyan_Delay();
    }
    }
```

实验步骤

在这个实验中要陆续创建三个 project 来完整地理解多文档工程和库的概念。我把实验步骤分成两个部分，需要依次完成。如果你对**编译，链接，库**等术语没有概念，请务必先阅读本实验的预备知识部分。

第一部分：理解多文件形成的工程

(1) 保持与本章实验一一致的硬件连接方式不变。

(2) 创建一个新的 CCS 工程，将工程放到一个文件夹“Ch2_2_1”中。工程的名称为“CH2_2_1”。并设置工程的编译属性，即 TivaWare 库头文件的包含路径。

(3) 在工程中添加入两个新的文件，一个源文件名为 Aiyan_gpio.c，另一个头文件名为 Aiyan_gpio.h。在两个文件中写入上述实验原理处所示的代码。

(4) 修改 main.c 使基与上述实验原理处所示的代码一致，保存工程的文件。去工程文件夹查看文件，可以看到上述三个新加入的源代码。

(5) 点击菜单栏上的绿虫子

注意： *当绿虫子按键边上有个下拉菜单，当你的 CCS 里有多个工程同时存在时，它是让你选你的目标工程的，一定要确保你选对了目标工程。*

(6) 查看实验结果，观察 LED 的亮灯效果。

(7) 再次查看工程所在的文件夹内容，比较点击绿虫子之后和之前的差异，尤其要注意观察 Debug 文件夹的内容。查看方式可以借鉴图 2-15。

(8) 找到 Debug 文件夹中的 Aiyan_gpio.obj 和 main.obj，查看文件的创建时间。回到 CCS 中对 main.c 文件进行简单的修改（比如在一个空行里加入一个空语句，即加入一个分号“;”），再次点击绿虫子。然后再去观察 main.obj 和 Aiyan_gpio.obj 的修改时间。

注意： *理解多文件工程编译是以源文件为单位进行的，如果源文件没有变化，对这个源文件的编译不重复进行，只对改动过的源文件进行编译。*

第二部分：学习如何创建库

第一部分创建了一个新的模块，它由一个源文件和一个头文件组成。如果我想把这个模块交给其它程序人员使用，就必须把这两个文件都交给他，他要把这两个文件加入到他所创建的工程才能使用，这就是代码重用最原始的方式。如果模块很多，又希望重复利用这些代码，这样每次都要传递源文件和头文件会很麻烦。更重要的是源文件的实现细节，很多时候是有版权问题（比如某种处理算法），我们不希望其它人看到我们的实现方式，即使是没有版机问题，其它程序人员不小心改动了源文件中的实现，也会是件麻烦事。解决这个问题的方式就是把模块变成一个库文件，下次只要传递头文件和不能修改也看不到细节的库文件。这样即使有很多个模块，也只生成一个库文件。

下面来利用第一部分分离出来的文档模块来建一个库，并学习调用这个库。

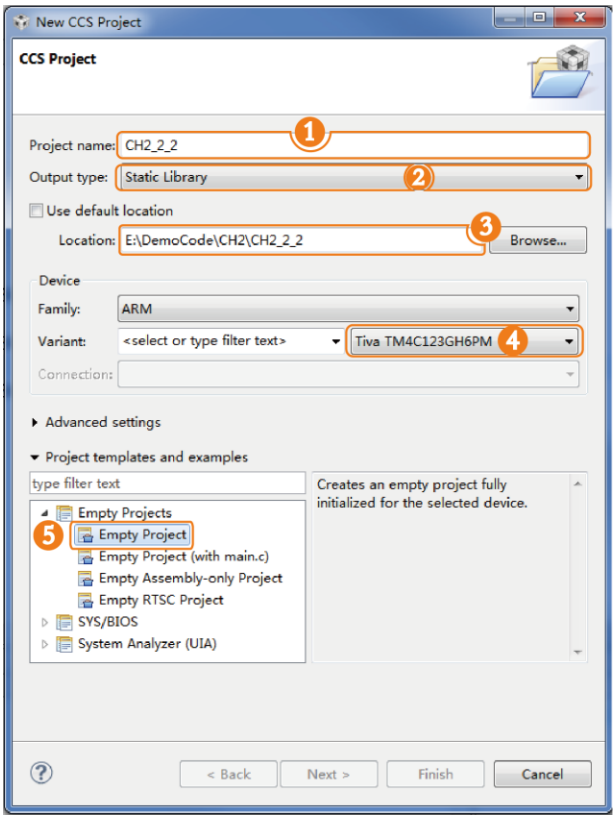
(1) 创建一个新 CCS 工程。如图 2-17 所示。工程名为 CH2_2_2，工程放到入文件目录“CH2\CH2_2_2”。

(2) 在这个新的工程中加入两个文件，一个源文件名为 Aiyan_gpio.c，另一个头文件名为 Aiyan_gpio.h。这两个文件的内容与第一部分保持完全一致。

(3) 在 CCS 的“Project Explorer”窗口中，鼠标右击 CH2_2_2 工程，在弹出菜单中选择“ Build Project”。

注意：*Build Project 和点击绿虫子的差别在于点击绿虫子会直接下载链接完的结果到目标设备，而 Build Project 只完成预编译、编译和链接三个步骤。通常把 Build 称为**生成**，因为要建的是一个库文件，它不是可执行文件，不能下载到目标设备中。*

(4) 如果没有错误，这个过程结束后，请到工程所在的文件夹中 Debug 目录下查找是否有一个新生成的库文件，这个文件的文件名和工程名一致，扩展名为“.lib”， 即“**CH2_2_2.lib**”。



1 工程的名称。

2 将工程设成静态库模式，将生成库文件，而不是生成可执行可下载到目标机的文件。

3 放到一个新的文件夹目录。

4 选定目标类型。

5 选一个没有任何文件的模板。

图 2-17 创建一个库的配置方式

(5) 到上一步结束，我们已经生成了一个新的库文件。为了确认库文件是否可用以及了解如何使用库，需要建一个新的工程来验证。在 CCS 中新建一个工程，工程的输出类型（Output type）为 Executable，工程名为“CH2_2_3”，工程放倒新建文件夹“CH2_2_3”中。

(6) 在新建的工程中加入名为“main.c”的源文件，将本实验第一部分中工程 CH2_2_1 中的 main.c 的代码都拷贝到这个工程 CH2_2_3 中的 main.c 文件里。

(7) 在 Project Explorer 中右击工程名 CH2_2_3，弹出快捷菜单，点击菜单中的最后一项“Properties”,弹出属性窗口。按如图 2-18 所示的方式进行编译属性和链接属性的设置。图中上半部分表示编译置性设置，因为在程序中引用了前一个工程的头文件 Aiyan_gpio.h，需要告诉编译器这个头文件的位置。下半部分表示链接置性设置，它指明了链接过程中要加入的库文件，就是我们前一个工程生成的库文件。

(8) 设置完后，点击绿虫子来看新的程序在 TIVA 上的运行效果。请仔细比对 CH2_2_1 比对 CH2_2_3 两个工程的不同。

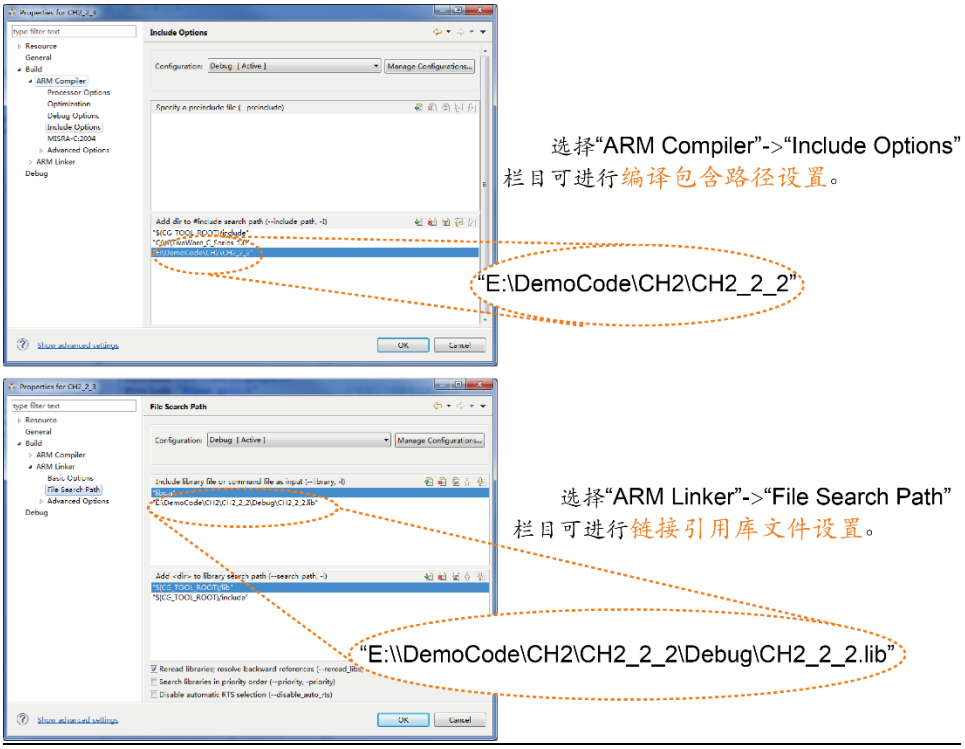


图 2-18 编译链接属性设置

2.3 TivaWare 库的调用实现（实验三）

实验目的：

通过调用 TivaWare 的函数库了解它的运行和使用机制；

实验设备：

硬件： Tiva LanuchPad 开发板

软件： Code Composer Studio 5.3.0

实验内容：

前面的两个实验，我们用了一些 TivaWare 中的宏定义，这些宏定义帮助我们更方便引用及读写 MCU 上的寄存器，但我们没有引用 TivaWare 库中的任何函数，为了简化主函数并更好操作硬件，还自己实现了一些函数，并把它们放到库里调用。事实上 TivaWare 提供了丰富的函数，我们前面要实现的库函数，TivaWare 都提供了。所以前面实验要实现的 LED 三色闪烁的功能，我们用 TivaWare 的库函数很容易就实现了。

这个实验要完成的任务就是应用 TivaWare 的库函数，依旧实现 LED 三色闪烁的功能。

实验步骤：

(1) 完成硬件的连接，连接方式和实验一一致；

(2) 建立新的 CCS 工程，工程名为“CH2_3”, 将工程放入到新建的文件夹“CH2_3”中。

(3) 设置工程的编译属性中的包含路径，第一章和前面的实验都有相关描述。

(4) 设置工程的链接属性。将 TivaWare 的外设驱动库加入到工程中，以便引用。做过实验二并能理解其过程后，加入 TivaWare 外设驱动库的过程就是找到这个库的位置的过程。在工程的属性对话框中选“ARM Linker”下的“File Search Path”, 在“Include library file or command file as input”中加入对 TivaWare 外设驱动库引用，如图 2-19 所示。

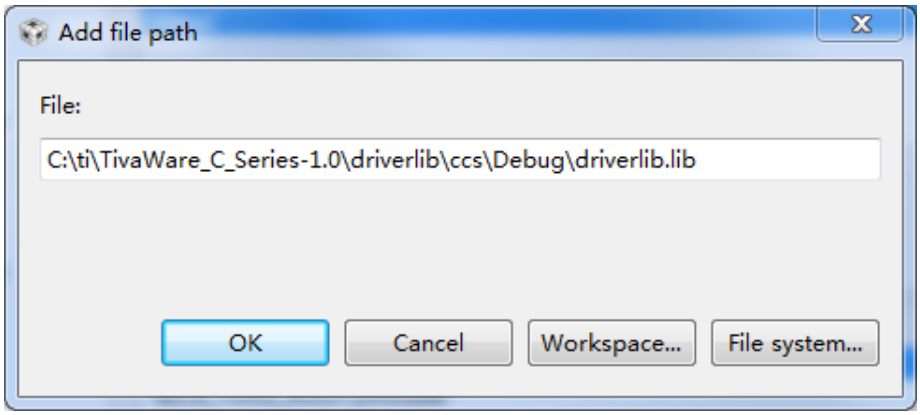


图 2-19 TivaWare 外设驱动库的引用

完成添加后，如图 2-20 所示，在包含库文件列表中看到对应的驱动库文件。

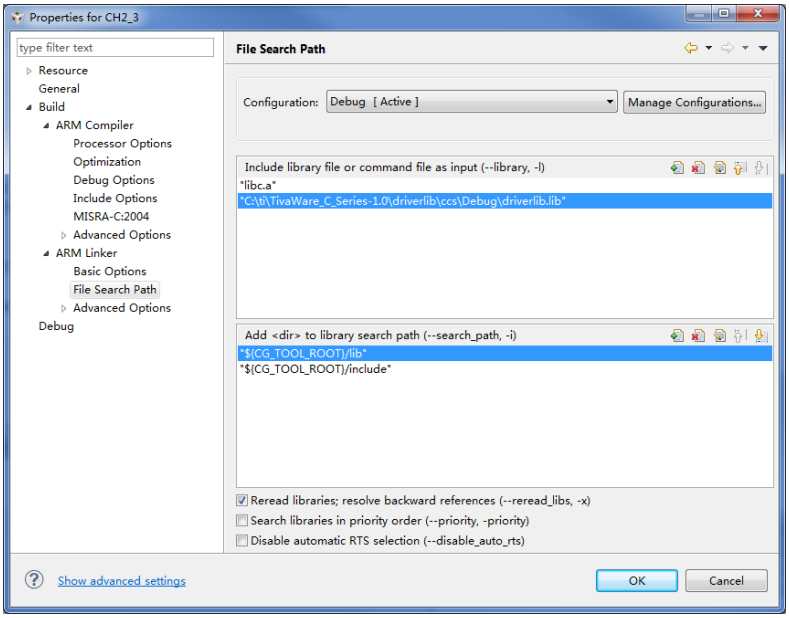


图 2-20 引用 TivaWare 外设驱动库的链接属性窗口

(5) 在 main.c 中写入如下代码

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
```

```
//*****************************************************************
**
// Define pin to LED color mapping.
//*****************************************************************
**
#define RED_LED   GPIO_PIN_1
#define BLUE_LED  GPIO_PIN_2
#define GREEN_LED GPIO_PIN_3

int main(void)
    {
    // Setup the system clock to run at 50 Mhz from PLL with crystal
reference

SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYS
CTL_OSC_MAIN);
    // Enable and configure the GPIO port for the LED operation.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);


GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,RED_LED|BLUE_LED|GREEN_LED);
    // Loop Forever
    while(1)
    {
        // Turn on the Red LED
        GPIOPinWrite(GPIO_PORTF_BASE,
                     RED_LED|BLUE_LED|GREEN_LED, RED_LED);
        // Delay for a bit
        SysCtlDelay(200000);
        // Turn on the Blue LED
        GPIOPinWrite(GPIO_PORTF_BASE,
                     RED_LED|BLUE_LED|GREEN_LED, BLUE_LED);
        // Delay for a bit
        SysCtlDelay(200000);
        // Turn on the Green LED
        GPIOPinWrite(GPIO_PORTF_BASE,
```

```
                RED_LED|BLUE_LED|GREEN_LED, GREEN_LED);
        // Delay for a bit
        SysCtlDelay(200000);
    }
    }
```

(6) 点击绿虫子，调试程序并运行查看 LED 的效果。

实验结果和总结：

实验效果是目标板上的三色 LED 闪。我实验一、二、三的效果一致。

通过上面三个实验的比较。我们能够看出实现同样的运行效果有很多不同程序构成方式。从第一章的例子到本章实验三的演化，就是从最底层的寄存器编程到 TivaWare 驱动库编程的一个演化过程。通过这种演化过程的实践操作，我们能更加全面的感受 TivaWare 的作用和使用方式。结合本章的第一节有关 TivaWare 的介绍，希望大家对其有一个较系统的了解。

第3章 LCD 模块与 USB

杭州艾研信息技术有限公司

2014 年 11 月

申明

杭州艾研信息技术有限公司保留随时对其产品进行修正、改进和完善的权利，同时也保留在不作任何通告的情况下，终止其任何一款产品的供应的权利。用户在下订单前应及时获取相关信息的最新版本，并验证这些信息是当前的和完整的。

可通过如下方式获取最新信息、技术资料和技术支持：

技术支持电话：0571-86134572

技术支持邮箱：support@hpati.com

产品&资料下载中心：http://www.hpati.com/products/

互动论坛：http://www.hpati.com/bbs/forum.php

公司地址：浙江省杭州市西湖区留和路16号新峰商务楼B306

第3章 LCD 模块与 USB

Tiva Launchpad 是 Cortex M4 的最简系统，作为 Cortex M4 学习，它还不能完全胜任，因为它板上资源很有限，即没有足够多的输入方式来控制，也没有合适的输出呈现实验结果。Launchpad 设计时已经为输入输出设备的搭载留好了两条通路，其一是通过 USB 与 PC 机相联，用户通过在 PC 上编写程序，在 PC 上实现对 Launchpad 控制和结果显示；其二为通过 BoosterPack 搭载其它模块来实现，实验套件中的 LCD 模块就是为此目的而设计的。LCD 模块包含：一块像素为 128X64 点阵式液晶，三个按键，一个滚轮，一个 Micro SD 卡槽，一个蜂鸣器，基本大部分的实验输入输出要求。

本章介绍 USB 和 LCD 的基本知识，随后通过实验帮助大家掌握 LCD 液晶显示的工作原理；USB 与上位机通信的工作原理及应用；读写 Micro SD 卡资源和 Fat 文件系统的使用。

1 预备知识

1.1 USB 简介

USB 的内容非常多，我只介绍 USB 的基本物理结构以及 USB2.0 通信协议有关的基本知识点和术语，以便大家理解实验的程序代码。

(1) USB 的物理结构简介

USB 接口有标准 A 型、标准 B 型、Mini USB、Micro USB 四种。如图 3-1 USB 接口所示。

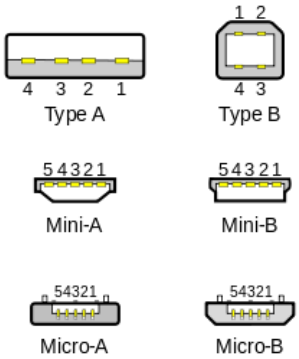


图 3-1 USB 接口

- 标准 USB 接口

一般标准 A 型是扁平的，电脑主机上的 USB 接口一般都是标准 A 型，USB 打印机上的一般是标准 B 型接口。如图 3-2 所示。

图 3-2 标准 USB 接口

标准 USB 接口为 4 个引脚：两个电源引脚，两个信号引脚。

表 3-1 标准 USB 引脚

引 脚	功 能	颜 色	备 注
1	VBUS	红色	电源正 5V
2	D-	白色	数据-
3	D+	绿色	数据+
4	GND	黑色	电源地

- Mini USB

Mini USB 是为了减小接口尺寸而设计的，一般在移动设备如手机、数码相机上有，Mini USB 接口分成 Mini A 型和 Mini B 型。如图 3-3 所示

图 3-3 Mini 接口（左：A 型,：右 B 型）

Mini USB 接口为 5 个引脚：两个电源引脚，两个信号引脚，一个 ID 引脚。

表 3-2 Mini USB 引脚

引脚	功能	颜色	备注
1	VBUS	红色	电源正 5V
2	D-	白色	数据-
3	D+	绿色	数据+
4	ID	-	A 型：接地 B 型：空置

5	GND	黑色	电源地

- Micro USB

Micro USB 是 USB 2.0 标准的一个便携版本，Micro USB 比 Mini USB 接口更小，Micro-USB 是 Mini-USB 的下一代规格，由 USB 标准化组织美国 USB Implementers Forum（USB-IF）于 2007 年 1 月 4 日制定完成。Micro USB 分为 Micro A 型，Micro B 型和 Micro AB。Tiva LaunchPad 中使用的是 Micro USB 接口，其有两个 USB 接口，仔细观察即可发现该两个接口是不同。其中标记为 DEBUG 的 USB 接口是 Micro B 型接口，而标记为 DEVCIE 的 USB 接口是 Micro AB 型接口，Micro AB 型接口可以插入 Micro A 和 Micro B 的插头。Micro AB 接口多应用于 OTG(On-The-Go)设备中，我随后解释 OTG 的概念。

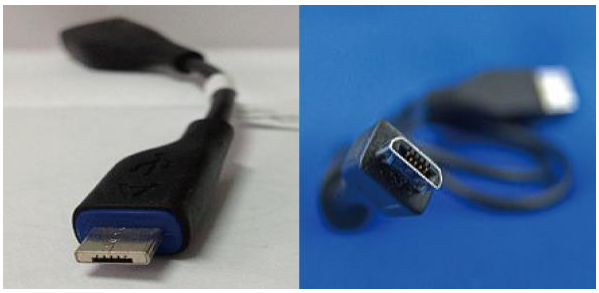


图 3-4 Micro USB 接口（左：A 型，右：B 型）

Micro USB 接口为 5 个引脚：两个电源引脚，两个信号引脚，一个 ID 引脚。

表 3-3 Micro USB 引脚

引脚	功能	颜色	备注
1	VBUS	红色	电源正 5V
2	D-	白色	数据-
3	D+	绿色	数据+
4	ID	-	A 型：接地；B 型：空置
5	GND	黑色	电源地

Mini USB 和 Micro USB 接口比标准 USB 接口多出一个引脚，即 ID 引脚。该引脚在 OTG 功能下使用。**OTG** 是 On-The-Go 的缩写。因为 USB 最早是 PC 上的一个接口，其它设备联接到 USB 时，PC 是导演，而联接的 USB 设备是演员，它们在 PC 的指挥下工作。按专业术语，PC 是 Host，其它设备是 Device。没有导演戏就没法进行，这种垄断的方式不利于市场发展，有能力的演员即当演员也能当导演。OTG 就是因这样市场需求而生，它让 USB 设备可以不依赖于 Host 互联。在 OTG 中，两 USB 设备互联时，其中如果有一方设备的 ID 引脚接地，则此设备默认为主机，否则为外设。同时，在设备连接使用过程中，通过主机协商协议(NHP)，允许主机和外设角色互换。OTG 主要应用于各种不同的设备或移动设备间的联接，进行数据交换。特别是 PDA、移动电话、消费类设备。

(2) USB 传输

USB 通信采用主从结构，通过 USB 相联的两个或者多个设备，其中有一方为主机(Host)，另外的为从设备(Device)，如果采用 OTG，Host 和 Device 的角色可以通过协议来更换。为发送和接收数据，主机会发起 **USB 传输**。每个传输使用定义好的格式来发送数据、地址信息、错误检测以及状态和控制信息。USB 通信均发生在主机和从设备之间，主机负责管理总线上的传输，而设备响应来自主机的通信。**端点**是设备的缓冲区，用来存储接收的数据和待发送的数据。每一个 USB 传输由一个或者多个**事务**组成，这些事务可将数据载入端点或者从端点取出。而**管道**就是设备和主机之间通信的一条通道。

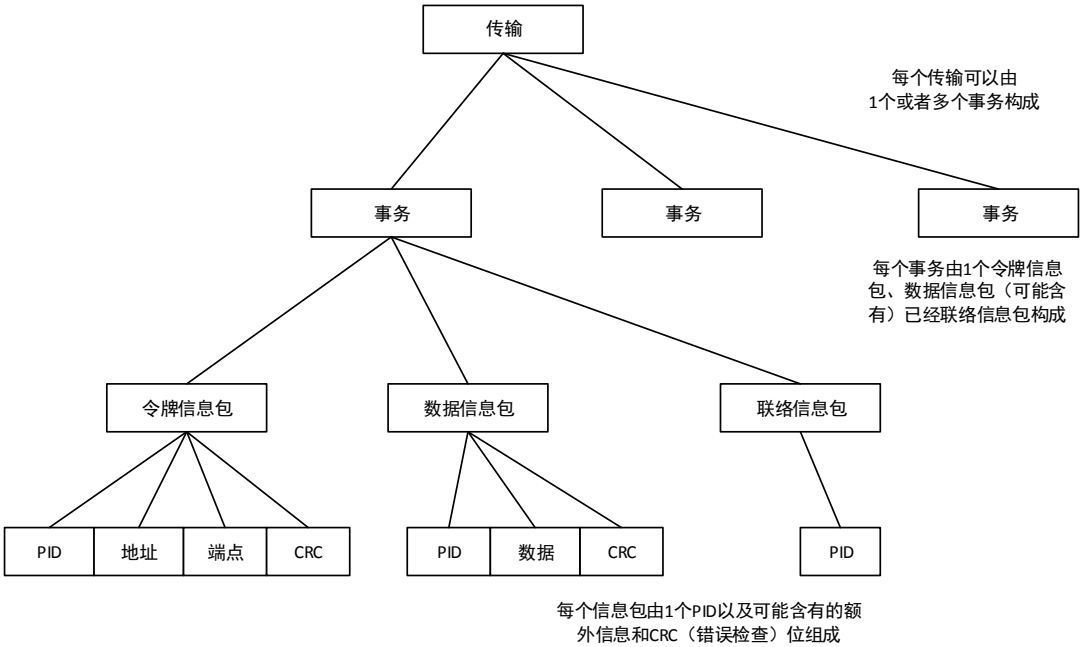


图 3-5 由事务组成的 USB2.0 传输

USB 事务开始于主机在总线上发送的**令牌信息包**，令牌信息包含有目标端点号和方向。令牌信息包可分为 IN 令牌包、OUT 令牌包、SOF 令牌包以及 SETUP 令牌包。其中 IN 令牌信息包表示向端点请求数据信息包，OUT 令牌信息包则是主机派发数据信息包的先行信息。除了数据，每个数据包还包含有错误检查位和一个带有数据顺序值的信息包 ID（PID）。许多事务还包含有联络信息包，数据的接收端用它来报告事务成或失败。图 3-5 给出了一个典型的 USB2.0 传输的组成。USB 传输包括控制传输、批量传输、中断传输和等时传输，每一个传输都由事务组成。

- **控制传输**有两个用途：一是对于所有设备，控制传输携带主机用以了解和配置设备的标准请求。二是控制传输还可以携带任何由类或者厂商定义的请求。控制传输通过端点 0 完成，并且每个 USB 设备都必须支持通过端点 0 进行传输的管道。
- **批量传输**用来在时间要求不严格的情况下传输数据。批量传输可发送大量数据而不阻塞总线，因为传输会首先遵从其他的传输类型，直到时间可用时才恢复批量传输。批量传输的用途包括向打印机发送数据、

从扫描仪接收数据以及向驱动器读写信息。若总线空闲批量传输是最快的传输类型。

- **中断传输**用在数据无延时传输的情况下，其典型的应用包括键盘、鼠标、游戏控制器以及集线器状态报告等。
- **等时传输**是实时的流传输，可在数据必须以恒定的速率传输或在特定的时间限制内到达，且允许偶尔出错的情况下使用，在发生错误的情况下不支持数据的自动重传。其例子包括实时解码的语音和乐曲。

(3) USB 的枚举

当 USB 设备连接到主机（PC 机）时，主机会通过控制传输了解设备的相关信息，这个过程就是 USB **枚举**。说得更加通俗一点，枚举是 USB 主机对 USB 总线的一次扫描过程，如果枚举不成功，设备就无法被识别，当然也就不能工作。熟悉 USB 的枚举过程是编程的第一步，我个人认为一旦理解了枚举并学会了通过编程来让主机识别设备，事实上就已入门 USB 的开发了。

USB 的软件的开发，要同时完成主机和从设备的软件开发，才能实现最终两者互联的通信。如果主机和从设备的软件都是白手起家，那么 USB 的开发工作量会变得很巨大，幸好 TI 帮我们完成了大量的工作, Launchpad 与 PC 相联时，PC 充当主机，Launchpad 充当从设备。在主机上，TI 为 Launchpad 提供了驱动，我们开发时只要用少量的几个参数来调用驱动包内的函数就可以实现枚举、数据通信等功能，而不必一一写出每个传输的数据包格式。在从设备上，TivaWare 准备了 USB 的数据包格式生成及分析、发送及读取的大量函数，借助 TivaWare，我们也可以用比较简单的几个函数参数实现 USB 的通信。但这些参数的理解还是需要我们理解 USB 的通信协议，不然我们的程序出现 bug 时，这些不理解的部分会让我们忧心忡忡。

在 USB 的枚举过程，主机是通过获取接入设备相关的**描述符**来了解并识别设备的。描述符是用来主机获取设备描述符是通过发送 USB 设备请求命令（Get Descriptor）获得的。在 C 语言的程序代码中，这些描述符的内容是由一些结构体来表示的。每个请求命令数据包包含 8 个字节（5 个字段）组成。其数据包格式如下。

表 3-4 请求数据包格式

偏移量	字段	大小（字节）	描述
0	bmRequestType	1	请求特征
1	bRequest	1	请求命令
2	wValue	2	请求不同含义不同
4	wIndex	2	请求不同含义不同
6	wLength	2	数据传输阶段，为数据字节数

USB 枚举过程中主机会请求获取多个描述符，一般为设备描述符、配置描述符、接口描述符、字符串描述符、端点描述符等。

设备描述符是在设备连接时主机读取的第一个描述符，帮助主机获取设备的相关额外信息。主机通过发送 Get Descriptor 请求，并设置 wValue 的高字节为 01H 获取设备描述符。设置描述符包含如下信息：

表 3-5 设备描述符数据包格式

偏移量	字 段	大小（字节）	描述
0	bLength	1	描述符长度：12H
1	bDescriptorType	1	描述符类型：01H
2	bcdUSB	2	USB 规范版本号
4	bDeviceClass	1	类代码
5	bDeviceSubClass	1	子类代码
6	bDeviceProtocol	1	协议代码
7	bMaxPacketSize0	1	端点 0 的最大信息包长度
8	idVendor	2	厂商 ID
10	idProduct	2	产品 ID
12	bcdDevice	2	设备版本号
14	iManufacturer	1	制造商字符串描述符索引

15	iProduct	1	产品字符串描述符索引
16	iSerialNumber	1	序列号字符串描述字索引
17	vNumConfigurations	1	可能配置的数目

配置描述符规定了设备的能力和特征，每个设备描述符都有附属描述符，包括一个或多个接口描述符已经可选的端点描述符。主机通过发送 Get Descriptor 请求，设置 wValue 高字节为 02H，获取配置描述符及其附属描述符。

表 3-6 配置描述符数据包格式

偏移量	字段	大小（字节）	描述
0	bLength	1	描述符长度：09H
1	bDescriptorType	1	描述符类型：02H
2	sTotalLength	2	配置描述符以及全部附属描述符的长度
4	bNumInterfaces	1	配置中接口的数据
5	bConfigurationValue	1	请求标识符
6	iConfiguration	1	配置字符串描述符的索引
7	bmAttributes	1	供电方式和远程唤醒设置
8	bMaxPower	1	总线功耗

接口描述符提供了关于设备所实现的功能和特性信息。

表 3-7 接口描述符数据包格式

偏移量	字段	大小（字节）	描述
0	bLength	1	描述符长度：09H
1	bDescriptorType	1	描述符类型：04H
2	bInterfaceNmber	1	用来确认接口编号
3	bAlternateSetting	1	用来确认 bInterfaceNmber 替代

			设置编号
4	bNumEndpoints	1	所支持的端点数目（不包括端点 0）
5	bInterfaceClass	1	类代码
6	bInterfaceSubClass	1	子类代码
7	bInterfaceProcotol	1	协议代码
8	iInterface	1	接口字符串描述符的索引

接口描述符中规定的端点都有一个**端点描述符**，而端点 0 不具有描述符，因为所以设备必须包含端点 0。端点描述符作为附属描述符以响应配置描述符的请求。

表 3-8 端点描述符数据包格式

偏移量	字段	大小（字节）	描述
0	bLength	1	描述符长度：07H
1	bDescriptorType	1	描述符类型：05H
2	bEndpointAddress	1	端点数目和方向
3	bmAttributes	1	传输类型和补充信息
4	wMaxPacketSize	2	所支持的最大信息包尺寸
6	bInterval	1	服务时距或 NAK 率

字符串描述符含有描述性文字，有些描述符可能含有指向描述制造商、产品、序列号、配置和接口的字符串索引，此时就可以通过字符串描述符来获取这些信息。主机通过发送 Get Descriptor 请求，设置 wValue 的高字节为 03H 来获取。

表 3-9 字符串表示符数据包格式

偏移量	字段	大小（字节）	描述
0	bLength	1	描述符长度（可变）
1	bDescriptorType	1	描述符类型：

			03H
2	bSTRING 或 wLANGD	可变	字符串

USB 枚举一般过程

① 首先，USB 主机检测到有 USB 设备插入，会对设备复位。USB 设备复位后其地址都为 0，主机就可以和刚刚插入的设备通过 0 地址端点 0 进行通信。

② USB 主机对设备发送获取设备描述符的标准请求，设备收到请求后，将设备描述符发给主机。

③ 主机对总线进行复位，之后发送 Set Address 请求，设置设备地址。

④ 主机发送请求到新的 USB 地址，并获取设备描述符的标准请求。

⑤ 主机获取描述符，包括配置描述符、接口描述符、设备描述符、端点描述符等（顺序不分先后）。

⑥ 主机根据已获取的描述符，请求相关字符串描述符。

⑦ 主机指定并加载设备驱动程序。

图 3-6 为 Bus Hound 抓取到的 USB 枚举过程。Bus Hound 为 USB 抓包软件，该软件将在之后介绍。

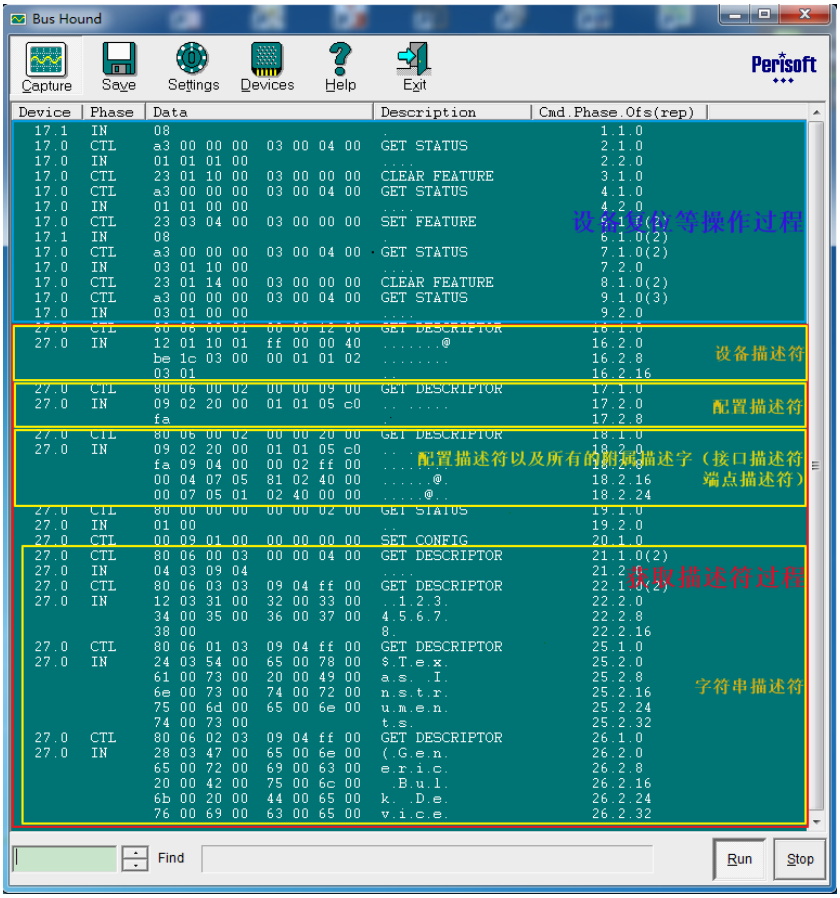


图 3-6 USB 枚举过程

1.2 LCD 液晶简介

LCD（Liquid Crystal Display 的简称）液晶显示器。LCD 的构造是在两片平行的玻璃基板当中放置液晶盒，下基板玻璃上设置 TFT（薄膜晶体管），上基板玻璃上设置彩色滤光片，通过 TFT 上的信号与电压改变来控制液晶分子的转动方向，从而达到控制每个像素点偏振光出射与否而达到显示目的。现在 LCD 已经替代 CRT 成为主流，并已充分的普及。

(1) LCD 分类

液晶显示器按照控制方式不同可分为被动矩阵式 LCD 及主动矩阵式 LCD 两种。段码式显示和点阵式显示。段码是最早最普通的显示方式，比如计算器，电子表这些。Tiva 试验箱套件中的 LCD 模块就是采用了点阵

式的液晶，点阵大小为 128*64。而点阵式的液晶又分为如下两种类型：被动矩阵式和主动矩阵式。

被动矩阵式 LCD 在亮度及可视角方面受到较大的限制，反应速度也较慢。由于画面质量方面的问题，使得这种显示设备不利于发展为桌面型显示器，但由于成本低廉的因素，市场上仍有部分的显示器采用被动矩阵式 LCD。

目前应用比较广泛的主动矩阵式 LCD，也称 TFT-LCD(Thin Film Transistor-LCD，薄膜晶体管 LCD)。TFT 液晶显示器是在画面中的每个像素内建晶体管，可使亮度更明亮、色彩更丰富及更宽广的可视面积。

(2) LCD 技术参数

可视面积：液晶显示器所标示的尺寸就是实际可以使用的屏幕范围。

可视角度：液晶显示器的可视角度左右对称，而上下则不一定对称。举个例子，当背光源的入射光通过偏光板、液晶及取向膜后，输出光便具备了特定的方向特性，也就是说，大多数从屏幕射出的光具备了垂直方向。假如从一个非常斜的角度观看一个全白的画面，我们可能会看到黑色或是色彩失真。一般来说，上下角度要小于或等于左右角度。如果可视角度为左右 80 度，表示在始于屏幕法线 80 度的位置时可以清晰地看见屏幕图像。但是，由于人的视力范围不同，如果没有站在最佳的可视角度内，所看到的颜色和亮度将会有误差。

点距：举例来说一般 14 英寸 LCD 的可视面积为 285.7mm×214.3mm，它的最大分辨率为 1024×768，那么点距就等于：可视宽度/水平像素(或者可视高度/垂直像素)，即 285.7mm/1024=0.279mm(或者是 214.3mm/768=0.279mm)。

色彩度：LCD 重要的当然是的色彩表现度。LCD 面板上是由 1024×768 个像素点组成显像的，每个独立的像素色彩是由红、绿、蓝(R、G、B)三种基本色来控制。大部分厂商生产出来的液晶显示器，每个基本色(R、G、B)达到 6 位，即 64 种表现度，那么每个独立的像素就有 64×64×64=262144 种色彩。Tiva 试验箱采用的是黑白两色的 LCD。所以就谈不上色彩度这一参数。

对比值：对比值是定义最大亮度值(全白)除以最小亮度值(全黑)的比值。对 LCD 来说，由冷阴极射线管所构成的背光源是很难去做快速地开关动

作，因此背光源始终处于点亮的状态。为了要得到全黑画面，液晶模块必须完全把由背光源而来的光全部阻挡，但在物理特性上，这些元件并无法完全达到这样的要求，总是会有一些漏光发生。一般来说，人眼可以接受的对比值约为 250:1。

亮度值：液晶显示器的最大亮度，通常由冷阴极射线管(背光源)来决定，亮度值一般都在 200～250 cd/m2 间。液晶显示器的亮度略低，会觉得屏幕发暗。

响应时间：响应时间是指液晶显示器各像素点对输入信号反应的速度，此值当然是越小越好。如果响应时间太长了，就有可能使液晶显示器在显示动态图像时，有尾影拖曳的感觉。一般的液晶显示器的响应时间在 20～30ms 之间。

(3) LCD 工作原理

如图 3-7 所示，从 LCD 显示器的结构来看，LCD 显示屏都是由不同部分组成的分层结构。中间由两块玻璃板构成，厚约 1mm，其间由包含有液晶材料的 5μm 均匀间隔隔开。因为液晶材料本身并不发光，所以在显示屏底板都设有光源，也叫背光板。背光板是由荧光物质组成的可以发射光线，其作用主要是提供均匀的背景光源。

背光板的光线在穿过第一层偏振过滤层之后进入包含成千上万液晶液滴的液晶层。液晶层中的液滴都被包含在细小的单元格结构中，一个或多个单元格构成屏幕上的一个像素。在玻璃板与液晶材料之间是透明的电极，电极分为行和列，在行与列的交叉点上，通过改变电压而改变液晶的旋光状态，液晶材料的作用类似于一个个小的光阀。在液晶材料周边是控制电路部分和驱动电路部分。当 LCD 中的电极产生电场时，液晶分子就会产生扭曲，从而将穿越其中的光线进行有规则的折射，然后经过第二层过滤层的过滤在屏幕上显示出来。

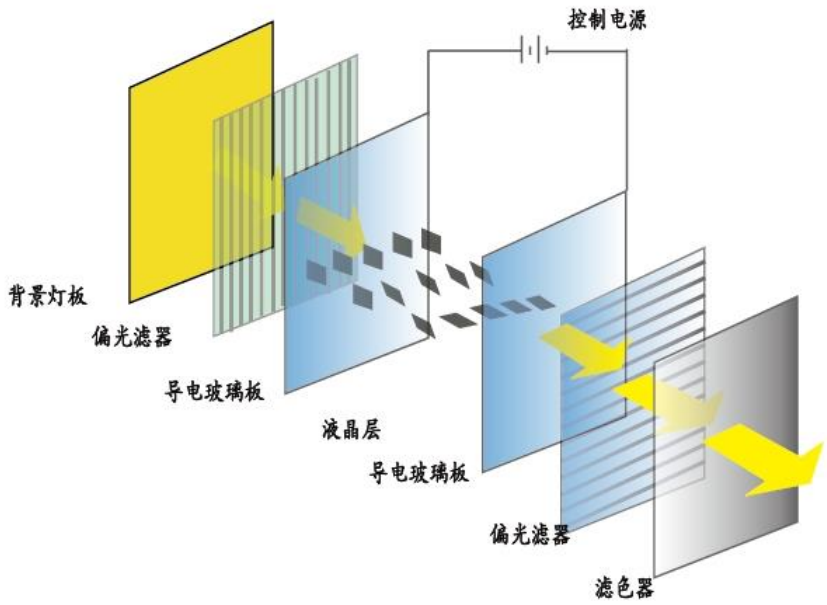


图 3-7 液晶结构

液晶显示技术也存在弱点和技术瓶颈，与 CRT 显示器相比亮度、画面均匀度、可视角度和反应时间上都存在明显的差距。其中反应时间和可视角度均取决于液晶面板的质量，画面均匀度和辅助光学模块有很大关系。

信号反应时间也就是液晶显示器的液晶单元响应延迟。实际上就是指的液晶单元从一种分子排列状态转变成另外一种分子排列状态所需要的时间，响应时间愈小愈好，它反应了液晶显示器各像素点对输入信号反应的速度，即屏幕由暗转亮或由亮转暗的速度。响应时间越小则使用者在看运动画面时不会出现尾影拖拽的感觉。

液晶显示内容的发送如图 3-8 显示和数据的关系所示，液晶点阵中通过有无加电来区分亮与暗来达到显示字符的目的。当没有加电压是，液晶粒子能够将通过横向偏振片的偏振光转向到纵向上通过纵向偏振片。在上方观察时，显示点阵亮起。反之，如果加电在电极作用下液晶粒子发生同向偏转。而通过横向偏振片的偏振光无法再次通过纵向偏振片。在上方看来，改点呈现暗点。假如要显示一个“H”的字符过程如下所示：

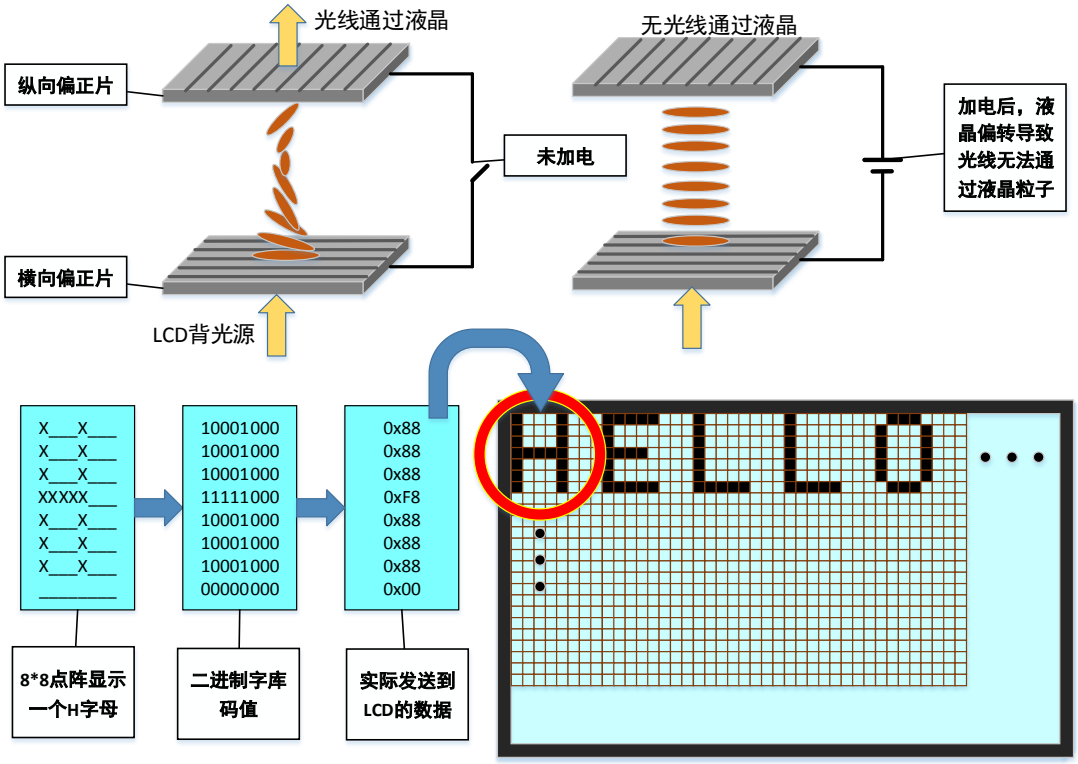


图 3-8 显示和数据的关系

2 实验模块介绍

2.1 硬件电路介绍

LCD 模块硬件示意如图 3-9 所示，图中没有画出蜂鸣器与 LaunchPad 的连接位方式，可以参考本书附的原理图或者网上资源来查看蜂鸣器的连接。

LCD 的工作需要若干的控制信号和通信线路来维持正常的工作状态，它与 Tiva LaunchPad 的连接如图 3-9 所示。液晶驱动需要 Tiva LaunchPad 提供端口使能控制信号。其中：PB1 为片选使能信号，低电平使能；PC6 为寄存器选择使能信号，低电平有效；PE5 为 LCD 复位信号线,负责 LCD 显示的控制信号。配置 Tiva LaunchPad 与 LCD 的通信协议，PD0、PD3 为同

步串行端口（SSI）。PD0 设置为 SSI 的通信时钟端口，PD3 为 SSI 的数据交换端口。SSI 负责将 Tiva LaunchPad 需要显示的数据传输到 LCD。

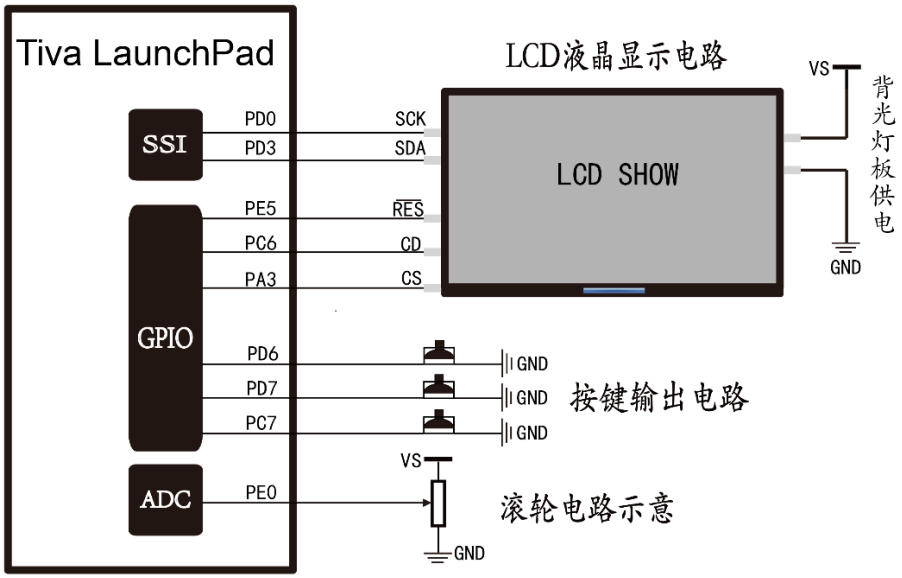


图 3-9 硬件结构图

2.2 软件设计总体说明介绍

(1) TivaWare USB 驱动库

Tiva USB 开发的驱动库由 TI 提供。其驱动库程序在 usblib 文件夹中。对于完成不同的功能又加以区别。故具体的程序可从 usblib 目录、usblib\device 目录、usblib\host 目录查看。

usblib：含有 USB 开发所需使用的源文件、头文件以及一些数据类型的定义。该目录下的文件是配置成 USB 设备模式应用或者是配置成 USB 主机模式应用所通过的。

usblib\device：该目录下的文件是配置成 USB 设备（device）模式应用使用的。

usblib\host：该目录下的文件是配置成 USB 主机（host）模式应用使用的。

USB 库分为三个层次：USB 驱动 API、设备类驱动、设备类 API。对应 USB 的开发有如图 3-10 所示的几种模式（图中的数字 1,2,3,4 表示不同的模式）。应用程序表示我们要写的代码，不用的模式只是调用不同的 USB 库，比如 1 号表示直接调用 USB 驱动底层的内容，而 4 号调用设备类 API。越是调用高层的库，代码越简单，但灵活性越差。下面我来说明一下我们实验中要用的模式：Bulk 模式。

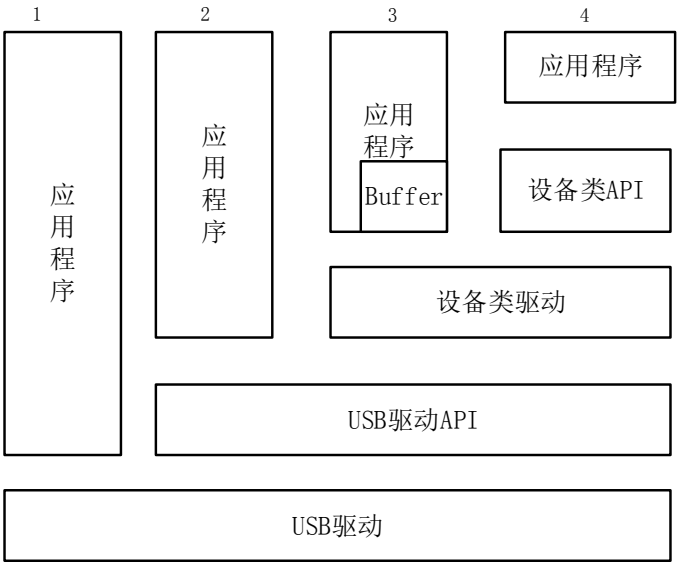


图 3-10 USB 开发框架

(2) USB Bulk 模式开发

对于 USB 的应用软件的开发，主要通过 Bulk 模式（批量传输模式）的开发来说明。批量传输（Bulk）采用的是流状态传输，批量传输的特点是支持不确定时间内进行大量数据传输，能够保证数据一定可以传输，但是不能保证传输的带宽和延迟，而且批量传输是一种单向的传输，要进行双向传输必须使用两个通道。Tiva 能够完成双向传输，因为其具有两个 Bulk 端点，一个为 IN 端点，一个为 OUT 端点。BULK 模式的应用程序程序开发框架如图所示。

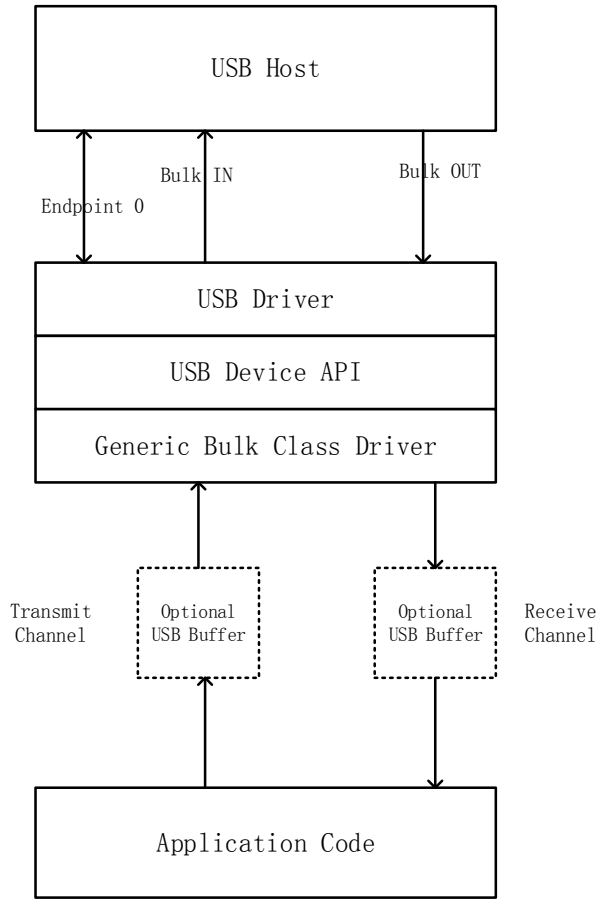


图 3-11 Bulk 模式开发

USB Host，就是主机端。

Endpoint 0：端点 0，完成控制传输。USB 枚举使用的通道。对应传输过程我们只需要完成相应的结构体定义即可：tUSBDBulkDevice、tUSBBuffer。而 Bulk IN 和 Bulk OUT 是 Bulk 模式数据传输的通道。tUSBDBulkDevice 完成 BULK 设备注册，包括各种描述符信息以及数据传输过程中使用的 Callback 函数等。而 tUSBBuffer 分成 RxBuffer 和 TxBuffer，完成底层数据和应用层数据的传输。

Application Code 是运行在 Tiva M4 上的顶层应用程序，该应用程序需要使用 USB Buffer 的相关 API 函数完成数据传输（传输到底层 USB 驱动）以及 Bulk 的相关 API 函数完成 Bulk 模式的初始化等操作。

(3) LCD 函数库构成

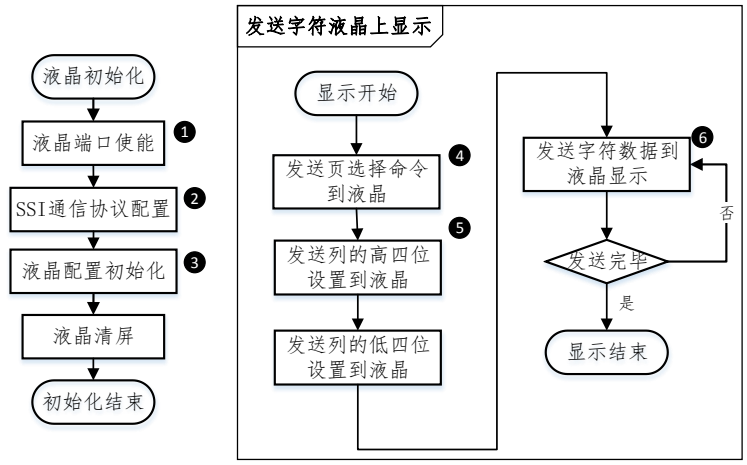
LCD 的函数库由 LCD_Matrix.h、LCDDriver.c、LCDDriver.h 构成，其中：

LCD_Matrix.h：点阵式 LCD 的显示通过控制 LCD 中的点阵的亮灭来显示内容，每一个字母、汉字或标点等都是通过一个 8*8 或者 8*16 大小的点阵来显示。因此该头文件就是存储各种字符的显示点阵库信息，又称作字模库。例如：当需要显示一个 8*8 大小的“8”的数字时，只需要将“8”对应的字模库数据发送给 LCD 来控制 LCD 的导电玻璃板上的加电点阵的电压状态。这样就能控制液晶旋转状态，最终控制光的通过与否。这样就可在 LCD 上显示出数字“8”。

LCDDriver.h：为 LCD 驱动的头文件，其中存放了 LCD 初始化的设置和显示字符函数，包括了信号使能，通信使能、显示函数等功能函数。

LCDDriver.c：实现 LCDDriver.h 中定义的驱动函数。当需要在 LCD 中显示时，调用其中的显示函数即可。

(4) LCD 开发流程与步骤



图x、液晶显示流程图

❶ 液晶（LCD）驱动需要TIVA LaunchPad提供端口使能控制信号。其中：PB1为片选使能信号，低电平使能；PC6为寄存器选择使能信号，低电平有效；PE5为LCD复位信号线,负责LCD显示的控制信号。

❷ 配置TIVA LaunchPad与LCD的通信协议，PD0、PD3为同步串行端口（SSI）。其中：PD0设置为SSI的通信时钟端口，PD3为SSI的数据交换端口。SSI负责将TIVA LaunchPad需要显示的数据传输到LCD。

❸ 初始化LCD配置，首先PE5的复位信号置低后再置高表示LCD上电启动；随后配置LCD对比度参数、行扫描列扫描的顺序以及正反显示等LCD显示参数。最后发送命令LCD开始显示接收到的数据内容。

❹ LCD显示字符，首先需要通过SSI发送页选择命令到液晶上以决定需要刷新显示的页位置。过程是：低电平使能片选信号，再低电平使能寄存器选择信号。等待SSI闲置后发送命令指令到LCD上。最后等待SSI传输结束后完成发送命令。

❺ 参照步骤❹，分别发送列的高四位和低四位设置命令到LCD。

❻ 与发送命令到液晶类似，将需要显示的字符的字模码值以两字节为单位依次地发送到LCD上显示。

图 3-12 液晶显示流程图

3 实验项目

3.1 概述

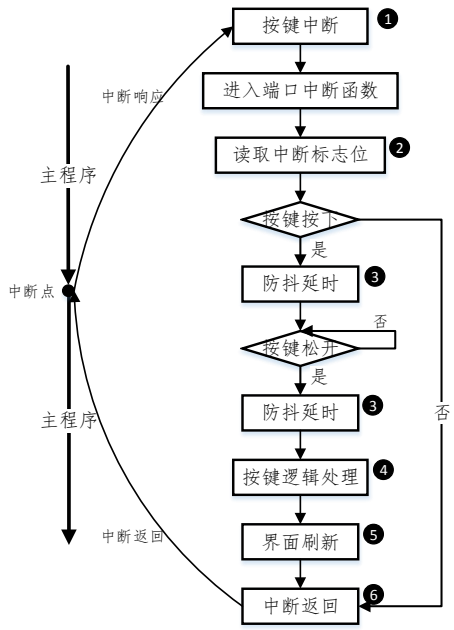
本模块和 USB 部分能完成的实验项目：（1）实验一：按键和滚轮控制液晶显示内容；主要目的是帮助体会和学习应用按键和滚轮的这两种不同输入控制方式，它们分别应用了 MCU 的不同外设，一个是采用了 GPIO，另一个则利用了 AD。 此外也让用户学习使用点阵式液晶的应用，同时学习 I^2C 串行通信的软件编程。（2）实验二：通过 USB 数据线与 PC 通信；通过实验学习 USB Bulk 模式通信，学习如何利用多种软件工具调试，学习 TivaWare 中 USB 函数库的应用。（3）实验三、SD 卡数据读写，通过实验学习 SPI 串行通信的软件编程，同时理解存储设备上的文件存储格式 FAT。

3.2 按键和滚轮控制液晶显示内容

实验内容

创建一个在 LCD 中显示闪烁光标的程序，并能够实现的通过按键和调节滚轮的操作变换光标的位置。如按键可以变换光标所在横坐标位置，滚轮调节可以调节光标所在的纵坐标的位置，使得光标可以在 LCD 整屏的任意位置自由的移动。

软件流程图及代码解析



❶ 主程序启动后会执行按键中断初始化，配置了LCD模块上的三个端口PC7、PC6、PC7的硬件中断。包括使能外设端口、开放端口输入、端口软件上拉、中断响应电平等。当端口电平探测到下降沿时，触发按键中断。

❷ 模块中三个按键的中断来自不同的外设端口和管脚号。读取响应中断的中断标志位，确认被按下的按键。

❸ 在按键被按下和抬起后都做一定时间的延时，来防止中断响应端口电平在按下和抬起按键时候的电平抖动。

❹ 确认了模块上被按下松开的键值后，根据给定按键的逻辑做响应的逻辑处理。例如：在主菜单界面按下模块最贴近滚轮的按键，表示菜单选中光标向上移动。则在逻辑处理中，认定主菜单选中光标向上移动。

❺ 如步骤❹所述，完成了对应的按键响应逻辑处理后，需要刷新对应的逻辑显示。刷新主菜单，将选中光标上移。

❻ 完成了所有的按键中断处理后，返回中断响应前的位置继续执行主程序代码。

图 3-13 按键中断响应流程图

(1) 按键代码解析

实现按键中断响应实际上就是触发 Tiva 某个端口的外部硬件中断。需要如下三步操作：

1、在 startup_ccs.c 中声明中断响应函数；

```
void Int_GPIO_C_Handler(void);
```

2、在 startup_ccs.c 的中断向量表中添加对应硬件中断端口的中断响应函数；

```
    /*************************************************
************
    *The vector table.  Note that the proper constructs
must be
    *placed on this to ensure that it ends up at
physical address
    *0x0000.0000 or at the start of the program if
located at a
```

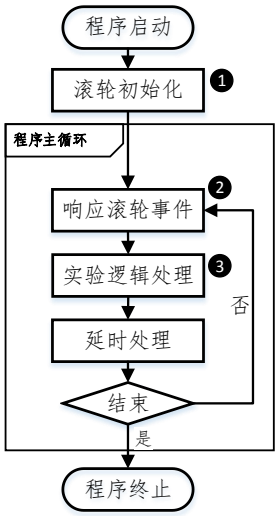
```
    *start address other than 0.
    ***************************************************
***********/
    #pragma DATA_SECTION(g_pfnVectors, ".intvecs")
    void (* const g_pfnVectors[])(void) =
    {
        (void (*)(void))((unsigned long)&__STACK_TOP),
                                        // The initial
stack pointer
        ResetISR,                       // The reset
handler
        NmiSR,                          // The NMI handler
        FaultISR,                       // The hard fault
handler
        IntDefaultHandler,              // The MPU fault
handler
        IntDefaultHandler,              // The bus fault
handler
        IntDefaultHandler,              // The usage fault
handler
        0,                              // Reserved
        0,                              // Reserved
        0,                              // Reserved
        0,                              // Reserved
        IntDefaultHandler,              // SVCall handler
        IntDefaultHandler,              // Debug monitor
handler
        0,                              // Reserved
        IntDefaultHandler,              // The PendSV
handler
        IntDefaultHandler,              // The SysTick
handler
        IntDefaultHandler,              // GPIO Port A
        IntDefaultHandler,              // GPIO Port B
        Int_GPIO_C_Handler,             // GPIO Port C
    }
```

3、在程序中实现中断响应函数；

```
    /*****************************************************
************
     * @brief   LCD模块按键响应中断 PC7 对应的为按键S1
     * @param   null
     * @return  null
    *****************************************************
***********/
    extern uint8_t VCA_BUTTON_UP_DOWM;
    //中断处理子函数
    void Int_GPIO_C_Handler(void)
    {
        unsigned long ulStatus;
        // 读取中断状态
        ulStatus = GPIOIntStatus(GPIO_PORTC_BASE, true);
        // 清除中断状态
        GPIOIntClear(GPIO_PORTC_BASE, ulStatus);
        // 如果KEY的中断状态有效
        if (ulStatus & GPIO_PIN_7)
        {
            // 延时约10ms，消除按键抖动
            SysCtlDelay(10 * (SysCtlClockGet() / 3000));
            // 等待KEY抬起
            while (GPIOPinRead(GPIO_PORTC_BASE,
GPIO_PIN_7) == 0x00);
            // 延时约10ms，消除松键抖动
            SysCtlDelay(10 * (SysCtlClockGet()  / 3000));
            //TODO 逻辑处理
            Handler_KEY(ENUM_KEY_UP);
        }
    }
```

当完成上述三部的设定后，假如设定中断后能够进入后面的逻辑处理说明按键的中断处理完成了。而后面需要做就就是实现按键多对应的逻辑事务了，如 LCD 光标的移动显示等等。

(2) 滚轮流程图



❶ 滚轮是通过调节一个划变电阻的阻值改变其两端的电压值，并通过TIVA LaunchPad端口的模数转化（ADC）获得一个0~4096的数字量。ADC都有一个参考电平，参考电平的数字量即为4096。所以滚轮的初始化就是滚轮对应端口PE0的ADC初始化配置。最终得到端口电平相对于参考电平的数字量。

❷ 在程序的主循环中，不断的根据自身需要获取滚轮ADC采样值。获取流程是：

1、触发对应ADC基址和序列号的ADC响应；
2、等待ADC模块完成模拟量到数字量的转换；
3、完成后清除其中断标志位；
4、根据ADC基址和序列号获取结合端口电压的转化数字量。

❸ 根据获取的滚轮位置可以对实验中的参数做一定量的调节。例如：改变某个端口输出PWM信号的占空比等。

图 3-14 滚轮响应流程图

(3) 滚轮代码解析

在启动程序时，对连接滚轮的 Tiva 端口初始化操作。将该端口设置为外部 ADC 采样端口。并使用软件触发获取端口电压的方式得到 ADC 采样值。

初始化流程如下代码所示：

```
    /*****************************************************
*************
     * @brief   初始化ADC获取滚轮电压值
     *      ————————|
     *      TIVA        |
     *      M4  PE0|<--ADC      模数转换信号源
     *      ————————|
    *****************************************************
**********/
    #define ADC_BASE        ADC0_BASE       // 使用ADC0
    #define SequenceNum     3               // 使用序列3
    void Init_ADCWheel()
    {
        // 使能ADC0外设
```

```
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
        // 使能外设端口E
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
        // 选择PE0为外部模拟数字转换功能
        ROM_GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0);
        // 使能采样序列号为触发处理获取模式
        ROM_ADCSequenceConfigure(ADC_BASE, SequenceNum,
ADC_TRIGGER_PROCESSOR, 0);
        // ADC配置
        ROM_ADCSequenceStepConfigure(ADC_BASE,
SequenceNum, 0, ADC_CTL_CH3 | ADC_CTL_IE |ADC_CTL_END);
        // 使能ADC配置
        ROM_ADCSequenceEnable(ADC_BASE, SequenceNum);
        // 清除ADC中断标志位
        ROM_ADCIntClear(ADC_BASE, SequenceNum);
    }
```

当在程序中需要获取 ADC 端口的 ADC 采样值时，使用 ADC 处理器软件触发信号获取数据（ADCProcessorTrigger），等待 ADC 触发中断返回数据并可以存储在一个 uint32_t 数据结构中。ADC_ValueGet 函数实现的就是调取特定 ADC 基址（ui32Base）和序列号（ui32SequenceNum）中的 ADC 采样数据，如下代码所示：

```
    /*************************************************
************
     * @brief   获取特定ADC的模数转化采样值。
     * @param   ui32Base          ADC采样基地址
     * @param   ui32Peripheral   ADC启动的外设端口
    * @return                     ADC采样值
    *************************************************
***********/
    unsigned long ADC_ValueGet(uint32_t ui32Base,
uint32_t ui32SequenceNum)
    {
        unsigned long value = 0;
        // 保存ADC采样值
        uint32_t ADCValue[1];
```

```
        // 触发获取端口采样
        ADCProcessorTrigger(ui32Base, ui32SequenceNum);
        //等待采样结束
        while(!ADCIntStatus(ui32Base, ui32SequenceNum,
false))
        {
        }
        // 清除ADC采样中断标志
        ADCIntClear(ui32Base, ui32SequenceNum);
        // 读取ADC采样值
        ADCSequenceDataGet(ui32Base, ui32SequenceNum,
ADCValue);
        value = ADCValue[0];
        return value;
    }
```

当得到 ADC 采样值浮动变化值超过了设定的最小变化量，表明滚轮有了人为的操作，这样可以根据 ADC 采样值做后续的逻辑事务处理。如根据 ADC 采样值的 0~4095 的变化量等比例的将光标在 LCD 上做横向 的左右移动等。

实验步骤

(1) 在 Tiva LaunchPad 实验套件母版上插上 Tiva 模块和 LCD 显示模块；

注意：*在母版上插 Tiva 模块和 LCD 模块的时候要注意模块的方向不要插反了而且也要小心管脚出现错位没有插好的情况出现。这会导致所有的管脚链接错位导致信号不一致而致使实验失败。*

(2) 将 Tiva LaunchPad 模块通过 USB（ ICDI/Power）接口连到 PC，并将模块上的 Power Select Switch 开关拔到 DEBUG 位置。

(3) 按照前一章节中介绍的方式添加创建一个新的 CCS 工程 CH3_1。

(4) 参照第二章实验 CH2_1 中图 2-11 所述，添加工程包含路径：（..\TI\TivaWare_C_Series-1.0）其中..\代表实验中本地电脑中 TI 的安装路径。

(5) 参照第二章实验 CH2_2_2 中图 2-18 所述，添加工程编译链接属性设置。将（..\TI\TivaWare_C_Series-1.0\driverlib\ccs\Debug\dirverlib.lib）链接库添加到工程配置中。

(6) 从 CCS 安装目录中（..\Program Files\TI\TivaWare_C_Series-1.0\examples\boards\ek-tm4c123gxl\project0）拷贝 startup_ccs.c 文件到 CH3_1 工程所在目录中。

注意：*这个 startup_ccs.c 文件定义了系统堆栈、中断向量表、默认系统异常中断等重要信息。这些设置是编译工程时必不可少的重要内容。其中我们本实验中的按键中断函数就必须添加到中断向量表中才能得到系统响应。在没有必要的情况下不要轻易的修改本文件内的任何内容。*

(7) 添加对应配套程序包中匹配章节 CH3_1 源码 main.c、LCDDriver.h、LCDDriver.c、LCD_Matrix.h 四份源代码文件到工程目录下。其中：

- main.c：主程序入口，它对程序做了必要的初始化设置、监听了按键响应事件和滚轮状态变化的处理。
- LCDDriver.h：定义了 LCD 驱动显示的一些常用状态变量和主要函数；
- LCDDriver.c：实现了 LCD 驱动的初始化、显示字符内容、刷新等功能；
- LCD_Matrix.h：储存了常用的 8*8 点阵显示字库，用于 LCD 的字符显示；

(8) 编译工程文件，查找是否有问题或编译错误存在并加以纠正。

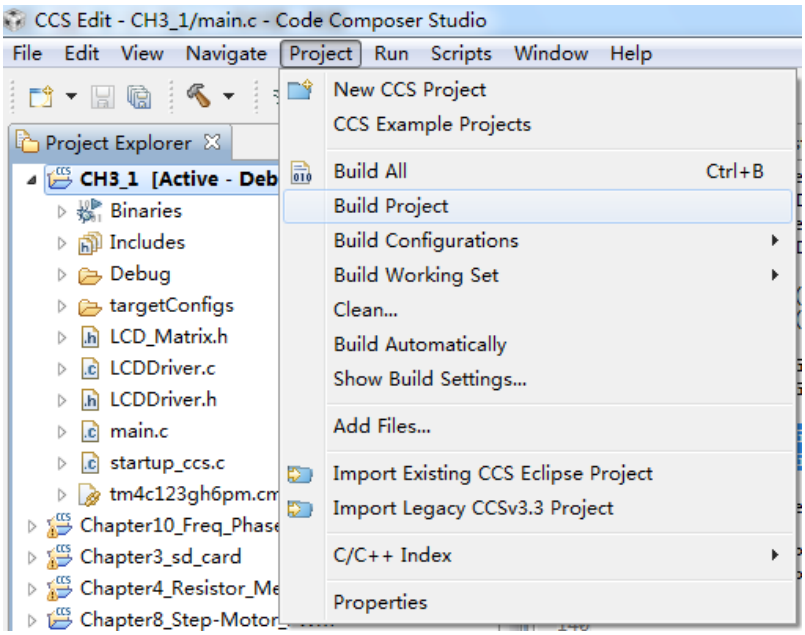


图 3-15 编译工程

(9) 成功通过编译后，执行 CCS 菜单项“RUN”中的“DEBUG”烧写到 Tiva LaunchPad 上调试运行（也可按快捷运行键 F11 直接运行）。

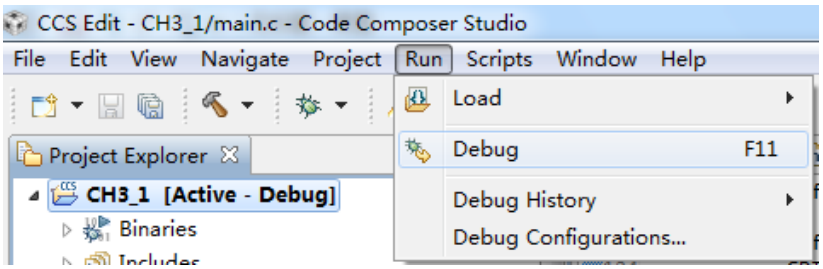


图 3-16 烧写工程到 Tiva LaunchPad

(10) 调整 LCD 模块上滚轮位置或操作按键看 LCD 显示内容的变化情况。

3.3 通过 USB 数据线与 PC 通信

实验内容

通过 USB 数据线，实现与 PC 的 USB BULK 模式传输（批量传输），

接收主机发送的信息并回发给主机，在检测到有数据传输完成时点亮 Launchpad 上的 LED 灯。

软件流程图及代码解析

实验程序使用 TivaWare 提供的 usb_dev_bulk 实例。该实例完成了实验内容的要求。该程序对应的软件流程图如图 3-17 所示。

USB 设备连接到主机后，主机会进行 USB 枚举，在该实验中首先需要完成 USB 枚举过程中的相关变量定义。这些变量通过 tUSBDBulkDevice、tUSBBuffer 结构体完成定义。

tUSBDBulkDevice 结构体 g_sBulkDevice 变量定义代码如下：

```
tUSBDBulkDevice g_sBulkDevice =
{
    USB_VID_TI_1CBE,           //厂商ID
    USB_PID_BULK,              //产品ID
    500,                       //最大电流
    USB_CONF_ATTR_SELF_PWR,     //供电模式，自供电
    USBBufferEventCallback,     //Callback函数，处理接收事件
    (void *)&g_sRxBuffer,       //传入上述Callback函数的首个参数
    USBBufferEventCallback,     //Callback函数，处理发送事件
    (void *)&g_sTxBuffer,       //传入上述Callback函数的首个参数
    g_ppui8StringDescriptors,  //字符串指针数组，包括相应的字符串描述符内容
    NUM_STRING_DESCRIPTORS
};
```

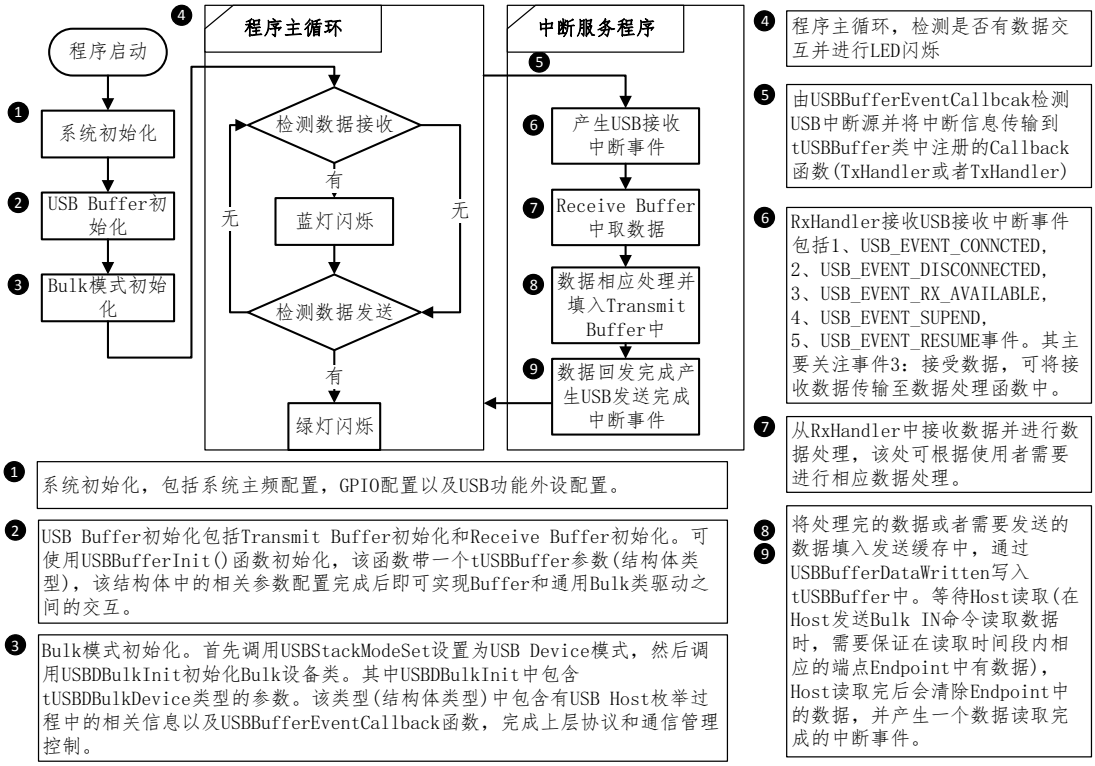


图 3-17 Bulk 模式软件流程图

g_sBulkDevice 变量中的 USBBufferEventCallback 函数由 TivaWare 提供, g_sRxBuffer 和 g_sTxBuffer 是 tUSBBuffer 结构体变量，该两个变量将在之后介绍。此外变量中需要自行完成的是字符串指针数组内容，其包括相应的字符串描述符信息。字符串指针数组定义如下：

```
    /*****************************************************************
******
    *字符串描述信息
    *****************************************************************
*****/
    const uint8_t * const g_ppui8StringDescriptors[] =
    {
        g_pui8LangDescriptor,                   //设备支持的语言字符串
        g_pui8ManufacturerString,              //厂商信息字符串
        g_pui8ProductString,                    //产品信息字符串
```

```
        g_pui8SerialNumberString,           //序列号字符串
        g_pui8DataInterfaceString,          //接口信息字符串
        g_pui8ConfigString                  //配置信息字符串
    };
```

指针数组中的变量都是数组类型，这些变量定义了相关的 USB 枚举过程中的字符串描述符信息，变量可按照如下模式定义：

```
    const uint8_t g_pui8DataInterfaceString[] =
    {
        (19 + 1) * 2,                   //该字符串描述符长度
        USB_DTYPE_STRING,               //描述符类型：0x03
        'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0,
't', 0,
        'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0,
'f', 0,
        'a', 0, 'c', 0, 'e', 0
    }
```

tUSBBuffer 结构体需要两个，一个是 g_sRxBuffer，另一个是 g_sTxBuffer。此两个结构体变量定义如下

```
    /*************************************************************
******
    * 接收缓存区
    *************************************************************
*****/
    uint8_t g_pui8USBRxBuffer[BULK_BUFFER_SIZE];
    uint8_t
g_pui8RxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
    const tUSBBuffer g_sRxBuffer =
    {
        false,                              // false表示接收
        RxHandler,                         // Callback函数
        (void *)&g_sBulkDevice,            // 作为Callback函数的参数.
        USBDBulkPacketRead,                // 函数，数据包读取
        USBDBulkRxPacketAvailable,         // 函数，检测端点状态
```

```
        (void *)&g_sBulkDevice,         // 作为上述两个函数的参数
        g_pui8USBRxBuffer,              // 数据存储区
        BULK_BUFFER_SIZE,               // 数据存储区大小
        g_pui8RxBufferWorkspace
    };

    //*************************************************************
******
    *发送缓存区
    //*************************************************************
*****/
    uint8_t g_pui8USBTxBuffer[BULK_BUFFER_SIZE];
    uint8_t
g_pui8TxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
    const tUSBBuffer g_sTxBuffer =
    {
        true,                           // true表示发送
        TxHandler,                       // Callback函数
        (void *)&g_sBulkDevice,         // 作为Callback函数的参数.
        USBDBulkPacketWrite,             // 函数，数据包发送
        USBDBulkTxPacketAvailable,       // 函数，检测端点状态
        (void *)&g_sBulkDevice,         // 作为上述两个函数的参数
        g_pui8USBTxBuffer,              // 数据存储区
        BULK_BUFFER_SIZE,               // 数据存储区大小
        g_pui8TxBufferWorkspace
    };
```

上述三个结构体是管理 Bulk 设备的主要部分，该三个结构体之间的联系如图 3-18。

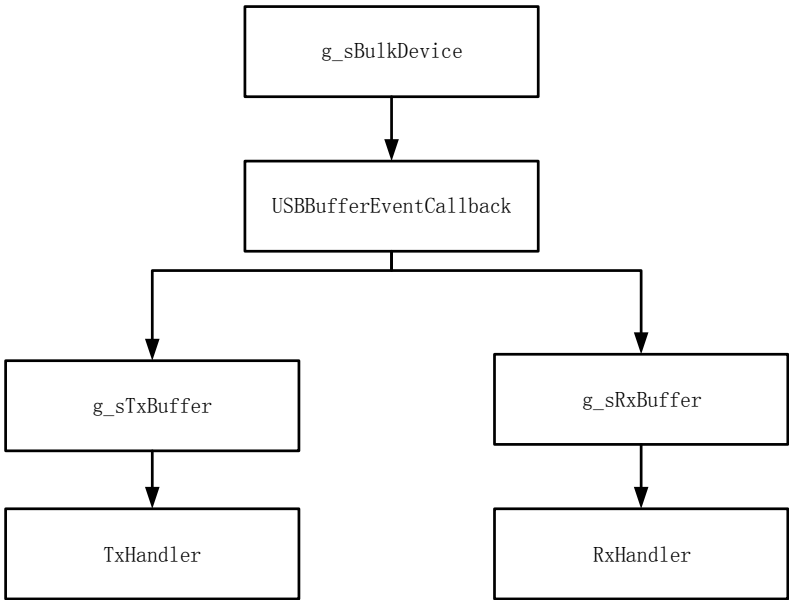


图 3-18 结构体关系图

g_sBulkDevice 进行上层协议与通信管理，通过 USBBufferEventCallback 函数处理 Buffer 数据，其通过 g_sTxBuffer 中的 USBDBulkPacketWrite 和 USBDBulkTxPacketAvailable 实现顶层数据发送，并通过 TxHandler 发送处理结果供应用程序使用；通过 g_sRxBuffer 中的 USBDBulkPacketRead 和 USBDBulkRxPacketAvailable 实现顶层数据发送，并通过 RxHandler 发送处理结果供应用程序使用。

RxHandler 接收上层 USB 事件进行处理，其需要处理的事件如下 USB_EVENT_CONNECTED,

USB_EVENT_DISCONNECTED,USB_EVNET_RX_AVAILABLE,USB_EVENT_SUSPEND,USB_EVENT_RESUME,USB_EVENT_ERROR。其代码如下

```
uint32_t RxHandler(void *pvCBData, uint32_t ui32Event,
            uint32_t ui32MsgValue, void *pvMsgData)
{

    switch(ui32Event)
    {
        //USB设备已经连接主机
        case USB_EVENT_CONNECTED:
        {
```

```
                break;
            }
            //USB设备已经与主机断开
            case USB_EVENT_DISCONNECTED:
            {
                break;
            }
            //有接收数据
            case USB_EVENT_RX_AVAILABLE:
            {
                //通过EchoNewDataToHost完成实验要求的回发功能
                tUSBDBulkDevice *psDevice;
                psDevice = (tUSBDBulkDevice *)pvCBData;
                return(EchoNewDataToHost(psDevice, pvMsgData,
ui32MsgValue));
            }
            case USB_EVENT_SUSPEND:
            case USB_EVENT_RESUME:
             {
                break;
            }
            default:
            {
                break;
            }
        }
        return(0);
    }
```

在本次实验中 TxHandler 函数可主要关注 USB_EVENT_RX_AVAILABLE 事件，因为该事件表示 USB 通信有数据接收，在检测到该事件后可通过 EchoNewDataToHost 函数完成回发，并记录数据接收个数以完成 LaunchPad 亮灯操作。回发完成后会产生 USB_EVENT_TX_COMPLETE 事件，该事件通过 TxHandler 响应处理，并记录发送数据个数以完成 LaunchPad 亮灯操作。

实验步骤

(1) 完成硬件的连接，将 LanuchPad 通过 USB 接口连接至 PC 机，此次实验需要将两个 USB 接口都通过 USB 连接线连接至 PC。其中 DEBUG 所标记的是程序烧录和调试的端口，DEVICE 所标记的是 USB 通信端口。完成连接后，将模块上的 Power Select Switch 开关拨到 DEBUG 位置。

(2) 建立新的 CCS 工程，工程名为“CH3_2”,将工程放入到新建的文件夹“CH3_2”中。

(3) 设置工程的编译属性中的包含路径，第一章和前面的实验都有描述。

(4) 设置工程的链接属性。将 TivaWare 的外设驱动库（driverlib.lin）以及 USB 库（usblib.lib）加入到工程中，以便引用。加入 TivaWare 外设驱动库和 USB 库的过程就是找到这两个库的位置的过程。在工程的属性对话框中选“ARM Linker”下的“File Search Path”， 在“Include library file or command file as input” 中加入对 TivaWare 外设驱动库引用以及 USB 库引用。添加过程需要分别操作，如图 3-19 和图 3-20 所示。

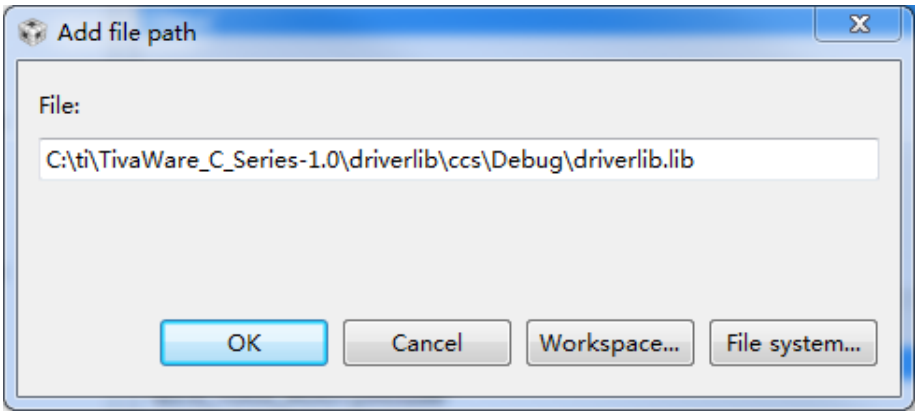


图 3-19 TivaWare 外设驱动库的引用

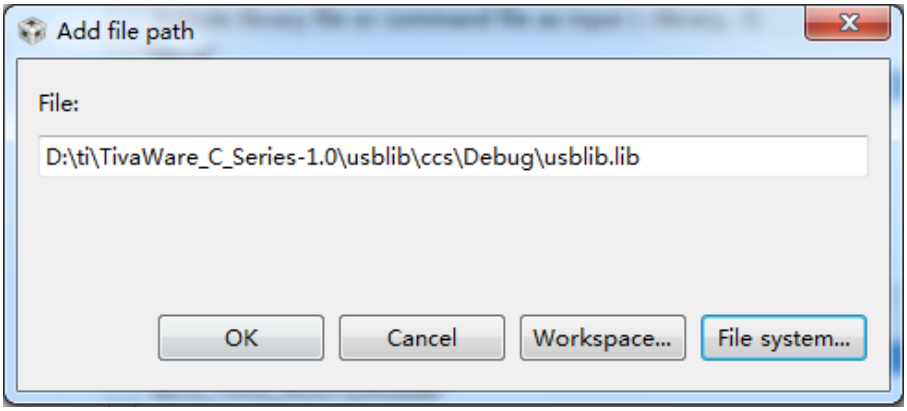


图 3-20 TivaWare USB 库的引用

添加完成后，如图 3-21 所示，在包含库文件列表中看到对应的库文件。

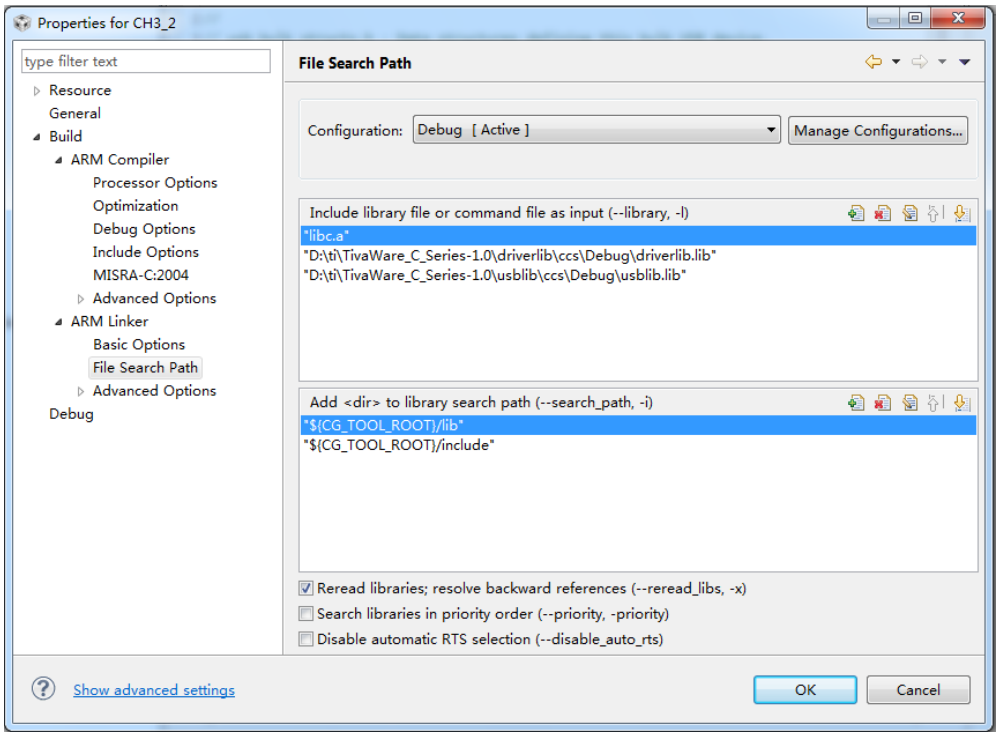


图 3-21 引用 TivaWare 库的链接属性窗口

(5) 向建好的工程“CH3_2”中添加指导书提供的程序代码文件。文件包括 usb_bulk_structs.h、usb_bulk_structs.c、usb_dev_bulk.c 以及启动文件 startup_ccs.c。

(6) 编译工程（Bulid Project），会提示“#35#Error Unrecognized COMPILER !”表示无法确认编译器，如图 3-22 所示。此时需要在工程属性对话框中 Build->Advanced Options->Predefined Symbols->Pre-define NAME(--define,-D)中加入“ccs="ccs"”，添加方式如图 3-23 所示。

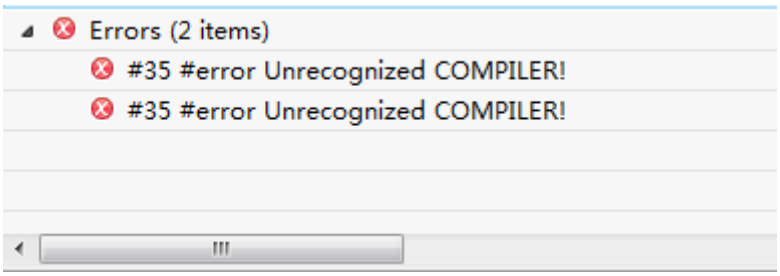


图 3-22 编译错误

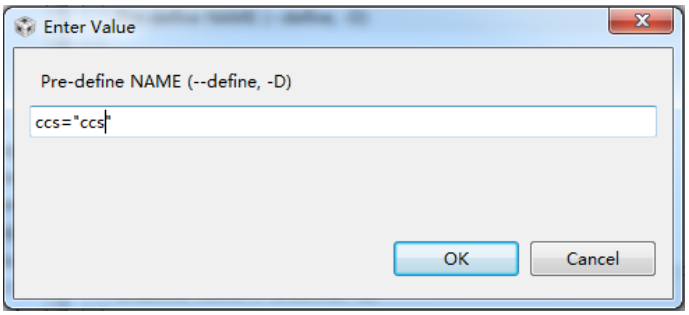


图 3-23 添加 ccs=“ccs”

(7) 添加完成后，再次编译工程，仍然会有有错误提示。如图 3-24 所示（截取了部分错误）。提示的错误表示无法处理以“ROM_”开头的函数，而非未定义，说明这些函数在函数库中是被定义过的，但是在编译的过程中无法确定这些函数。

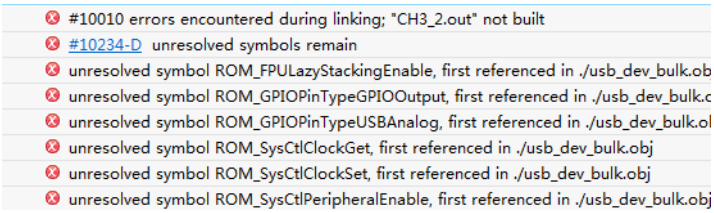


图 3-24 编译错误

可以打开定义这些函数的库文件（rom.h，位于 driverlib 文件夹下）查看，函数定义方式如图 3-25。表示在使用这些函数时需要预定义

TARGET_IS_BLIZZARD_RA1 或者
TARGET_IS_BLIZZARD_RA3 或者 TARGET_IS_BLIZZARD_RB1，而实验中使用的是芯片是属于 TARGET_IS_BLIZZARD_RB1 系列，所以在预定义中完成 TARGET_IS_BLIZZARD_RB1 定义即可。

```
#if defined(TARGET_IS_BLIZZARD_RA1) ||                                  \
    defined(TARGET_IS_BLIZZARD_RA3) ||                                  \
    defined(TARGET_IS_BLIZZARD_RB1)
#define ROM_SysCtlClockSet                                              \
        ((void (*)(uint32_t ui32Config))ROM_SYSCTLTABLE[23])
#endif
```

图 3-25 ROM 函数定义

预定义方式有两种：1、在程序中写入预定义语句。2、在 CCS 开发环境中直接设置。

1、在程序中写入预定义语句

一般会在使用到以“ROM_”开头函数的源文件中（.c 文件）的开头部分写上如下代码。

```
#ifndef TARGET_IS_BLIZZARD_RB1
#define TARGET_IS_BLIZZARD_RB1
#endif
```

2、在 CCS 开发环境中直接设置

进入工程属性对话框，在 Build->Advanced Options->Predefined Symbols->Pre-define NAME(--define,-D) 中加入“TARGET_IS_BLIZZARD_RB1”即可。如图 3-26 所示

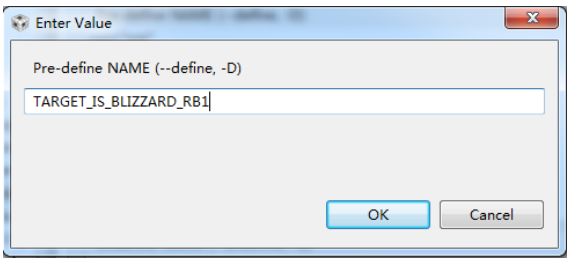


图 3-26 添加 TARGET_IS_BLIZZARD_RB1

(8) 再次编译工程（Bulid Project）。此时会出现错误。如图 3-27 所示。

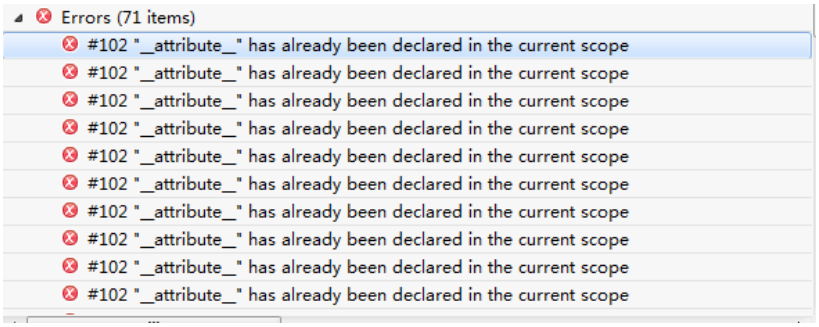


图 3-27 编译错误

(9) 出现上述错误时，可以右击工程，进入工程属性对话框，在 Build->ARM Compiler->

Advacnced Options->Language Options 中勾选上“Enable support for GC extensions(--gcc)”即可，如图 3-28 所示。或者在工程属性对话框中的 Build->ARM Compiler 上单击，进入 AMR Compiler 对话框，找到“Summary of flags set:”右下角的“Set Additional Flags…”单击进入后添加“—gcc”即可，如图 3-29 所示。

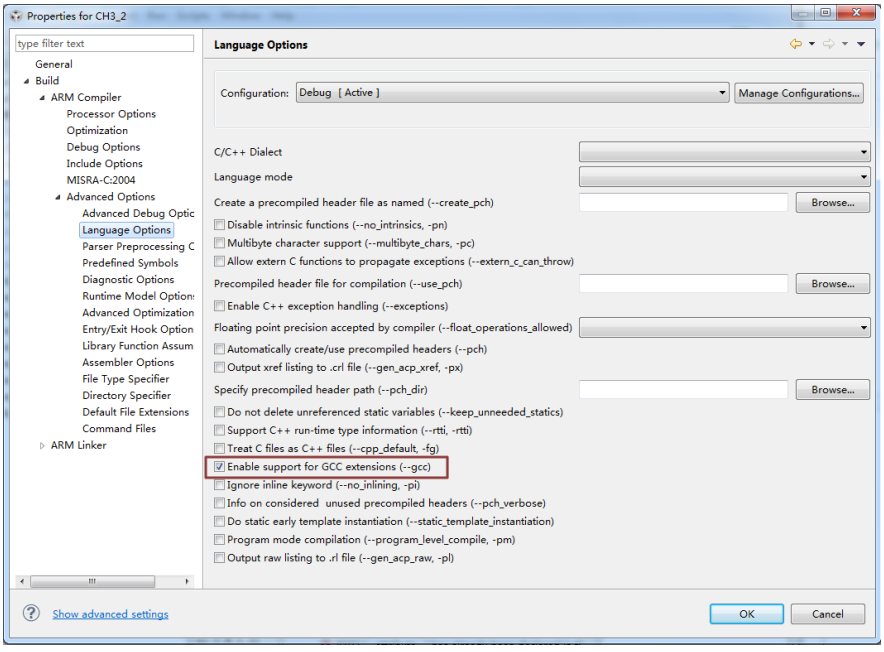


图 3-28 勾选 GCC

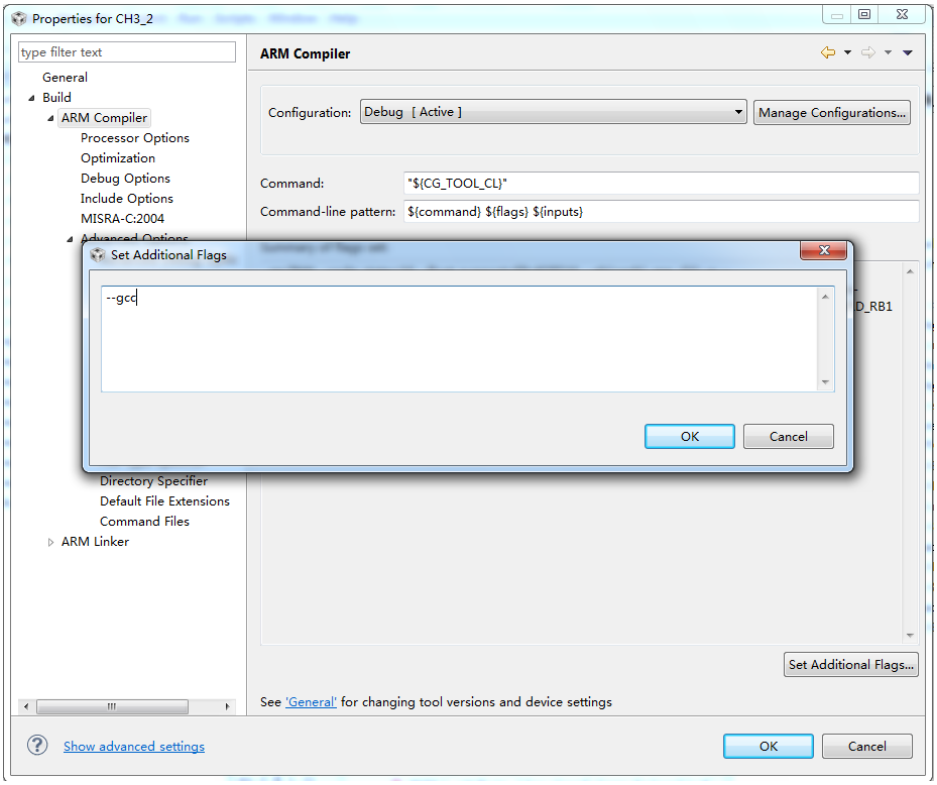


图 3-29 添加—gcc

(10)重新编译工程（Rebuild Project），此时编译通过。

(11)点击绿虫子，烧录程序，并进入调试（DEBUG）界面。

(12) 将 Power Select Switch 开关拨到 DEVICE 位置，运行程序（单机 Resume 按钮或者按 F8 快捷键）。首次运行程序时会提示安装相应驱动。如图 3-30 所示。此时操作系统会去相应目录下查找合适的驱动程序。

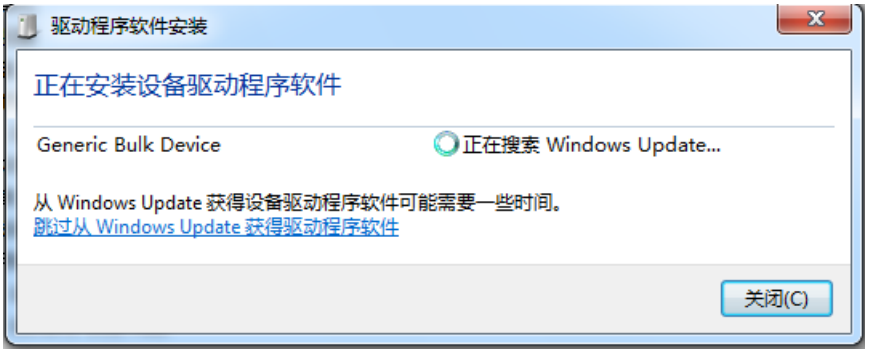


图 3-30 查找驱动程序

(13) 实验使用的驱动程序是配合 Tiva LanuchPad 的，首次运行该程序时，操作系统的相应目录下是不存在该驱动的，故系统自动安装驱动程序会失败，出现如图 3-31 所示情况。

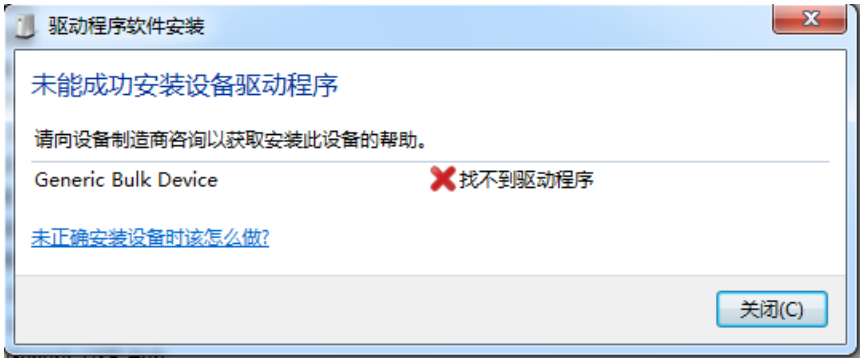


图 3-31 驱动程序安装失败

(14) 手动安装驱动程序，进入设备管理器，（右键单击计算机，单击属性，找到弹出对话框中的设备管理器（位于左侧））。在设备管理器界面中找到其他设备，右键单击有黄色叹号标记的设备，选择更新驱动程序软件。如图 3-32 所示。进入更新驱动程序软件对话框后选择“浏览计算机以查找驱动软件（R）”，进入对话框后左键单击浏览选择知道书提供的 CH3_2 目录下的驱动文件即可。驱动安装完成后可在设备管理中看到,如图 3-33 所示。

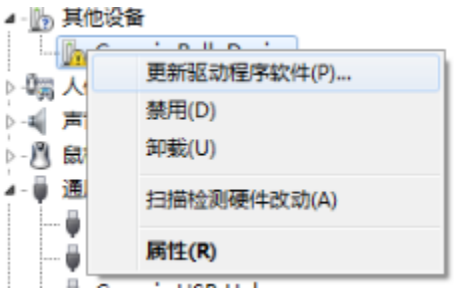


图 3-32 更新驱动程序

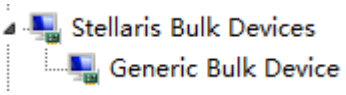


图 3-33 驱动安装完成

(15) 打开 USB Bulk Example 软件，该软件为 USB Bulk 通信过程中上位机配套使用软件，可到 TI 官网下载。在软件界面中输入任何字符，按回车（Enter）发送，该软件会收到下位机回发的数据，此时可以观察下位机闪灯情况。

USB 通信的上位机调试工具的应用

(a) USB Bulk Example

USB Bulk Example 是 TI 提供的在 Bulk 模式通信中配合下位机使用的软件。该软件可以发送数据并且接收下位机回发的数据，可以用于观察实验结果。例如在进行实验步骤(15)过程中，使用该软件来观察结果：在软件中键入“Tiva USB”并按回车发送至下位机，下位机接送数据、处理并回发，此时 USB Bulk Example 接收到回发的数据并显示在界面上，界面上呈现为“tIVA usb”，这个大小写的处理是在下位机完成的。软件还显示了 USB 通信过程中发送的数据个数和接收的数据个数。图 3-34 为实验结果展示。

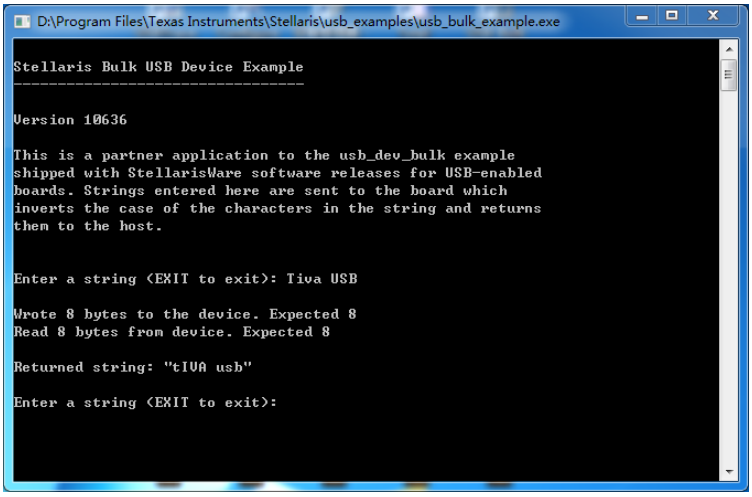


图 3-34 实验结果

(b) Bus Hound

BusHound 软件是由美国 perisoft 公司研制的一种专用于 PC 机各种总线数据包监视和控制的开发工具软件，其名“hound”的中文意思为“猎犬”，即指其能敏锐地感知到总线的丝毫变化。Bus Hound 是一个超级软件总线协议分析器，用于捕捉来自设备的协议包和输入输出操作。在 USB 开发工程中可以使用该软件进行 USB 数据包抓取。

整个软件界面如下。

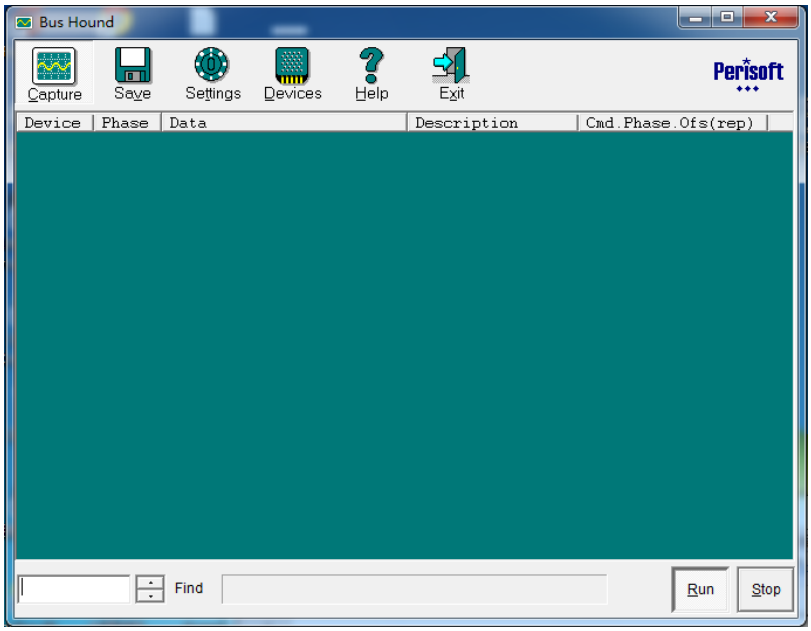


图 3-35 Bus Bound 软件界面

Bus Hound 捕获显示窗口中会在捕获数据列表中显示通信过程的详细信息。以下将描述捕获窗口中每一列的功能：

1、Device 列 ：

Bus Hound 对每个设备分配了一个数字 ID，第 1 个设备从 0 开始，第 2 个设备为 1，依此类推，这项功能对于软件同时捕获多个并行设备非常有用，设备的 ID 分配在 Device 窗口中完成。对于 USB 设备，设备的端点同时显示出来（例如图 3-37 表示 ID 为 36 的设备的 0 号端点（控制传输对应的端点）以及 1 号端点（批量传输对应的端点）） 。

2、Phase 列：

表示通信过程中的处于的阶段。在 USB 通信中主要会出现以下几个阶段。

表 3-10 USB 通信阶段

阶段	功能描述
CTL	控制传输中，Get Descriptor 请求
IN	数据传输（设备到 PC）
OUT	数据传输（PC 到设备）

3、Data 列：

与每个阶段对应的数据显示在此列中，例如命令字节，数据传输字节，和状态字节都能在此窗口中显示出来。

4、Description 列：

每个阶段对应的文本说明给你带来极大的便利 。

5、Cmd.Phase.Ofs(rep)例：

1 个数据组合体用来的标识当前捕获的数据的确切阶段，所有值均为十进制。包括 Cmd、Phase、Ofs 和 Rep 四个不同参数。

Cmd 是命令数，命令计数从 1 开始，当有新命令发送到设备时进行加 1 操作

Phase 是命令中的阶段数，1 个命令通常由许多阶段组成，例如数据传输阶段和状态反馈阶段，阶段计数值从 1 开始，在命令中当有新的阶段产生时进行加 1 操作

Ofs 每个阶段中字节的偏移量，计数值从 0 开始，数据传输过程中的每个新的数据字节时进行加 1 操作。

Rep 指示命令重复发布数，重复计数特性能在系统设定窗口中进行开启和关闭设定。

在安装完驱动后打开 Bus Hound 进入 Devices 选项，在 Standard Enhanced PCI to USB Host Controller 下找到 Generic Bulk Device，在小方框上打上勾，然后进入 Capture 选项，左键单击 Run，就可以对该 USB 设备进行数据包抓取。

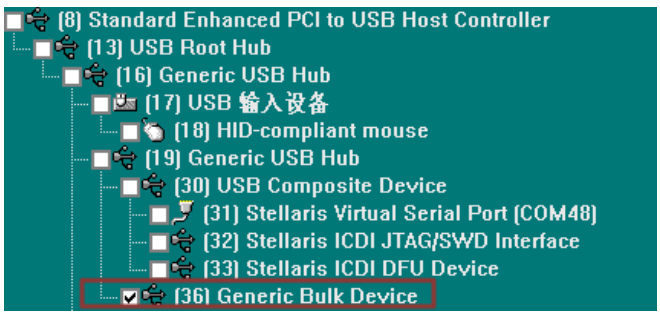


图 3-36 Devices 设置

在完成设置后，重新运行 Tiva LaunchPad 程序（由于已经安装了驱动

程序所以操作系统会自动判别驱动程序，无需再次手动安装驱动程序）可在 Bus Hound 上观察到枚举过程。在 USB Bulk Example 软件上发送“Tiva USB”并且 Tiva LaunchPad 回发的“tIVA usb”都可以在 Bus Hound 中观察到。图 3-37 展现的就是在完整的实验过程中

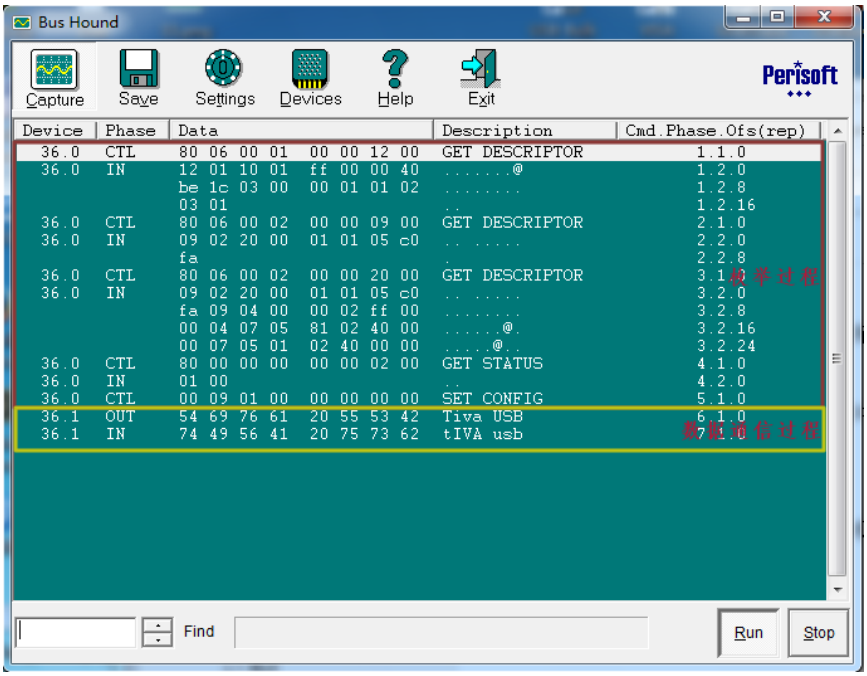


图 3-37 抓包过程

3.4 SD 卡数据读写

实验内容

在本实验中需要更为高效的输入输出方式来体验 Micro SD 卡的数据读写过程，利用串口通信程序连接上位机 PC 和 Tiva LaunchPad。 将 SD Card 读写程序烧写到 Tiva LaunchPad 中，上电后，打开串口通信程序实现下位机与 Tiva LaunchPad 的通信。将 Micro SD 卡插到 LCD 模块上的 Micro SD 卡卡槽中，在上位机串口通信程序控制台界面输入命令行来对 SD 卡数据进行读写操作。使用 ftp 文件系统查看 Micro SD 中的数据内容，并创建修改保存一个文本文档。如下表所示，可以通过一些基本的命令行完成对于 Micro SD 卡数据的读写。

表 3-11 可用命令行列表

命令行	命令含义
help	显示可用的命令行帮助信息
h	同上
?	同上
ls	显示文件列表
chdir	改变当前目录
cd	改变当前目录
pwd	显示当前工作路径
cat	查看文本文件内容
mkdir	创建文件夹
creat	创建文件
write	修改文件内容

执行的效果如下图所示：

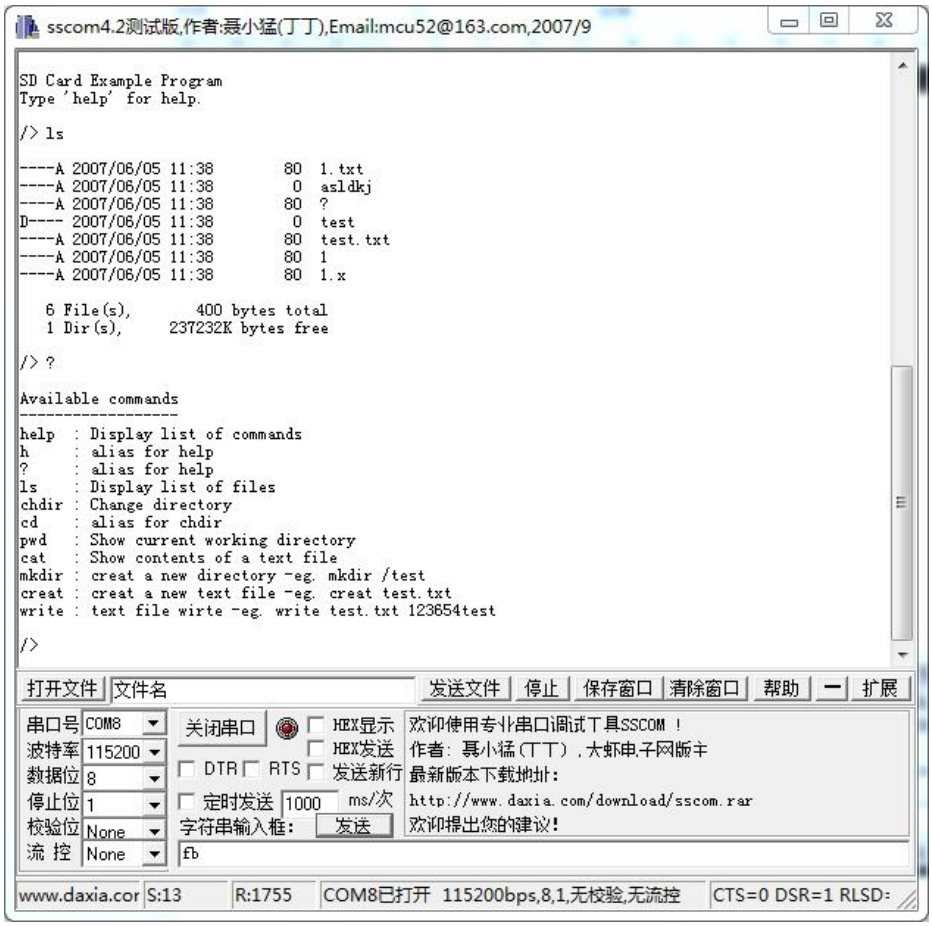
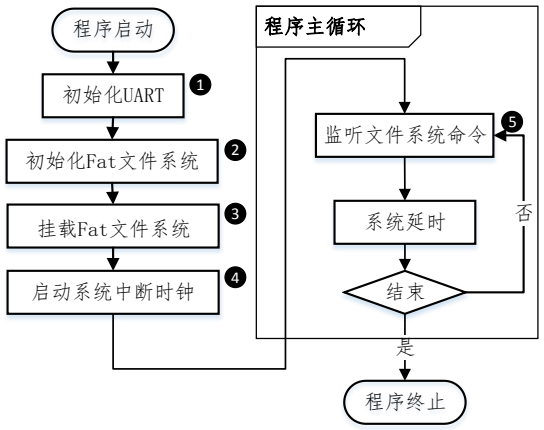


图 3-38 Tiva 与上位机串口通讯控制台示意图

软件流程图及代码解析



❶ Micro SD卡内容的读写显示操作需要更为高效的输入输出方式实现。因此启动通用异步串行数据总线（UART）实现上位机和TIVA LaunchPad的通信。这样既可以通过命令行输入方式下达文件读写指令，也可以输出显示指令执行结果并查看Micro SD卡的文件内容。

❷ Fat文件系统是一个为小型嵌入式系统设计的通用FAT(File Allocation Table)文件系统模块。Fat的编写遵循ANSI C，并且完全与磁盘I/O层分开。因此它独立(不依赖)于硬件架构。它可以被嵌入到低成本的微控制器中，而不需要做任何修改。TivaWare的third_party中也支持Fat文件系统。

❸ 挂载Fat文件系统由函数f_mount实现。f_mount 函数在FatFs模块上注册/注销一个工作区。在使用任何其他文件函数之前，必须使用该函数为每个卷注册一个工作区。要注销一个工作区，只要指定FileSystemObject为NULL即可，然后该工作区可以被丢弃。该函数只初始化给定的工作区，以及将该工作区的地址注册到内部表中，不访问磁盘 I/O 层。卷装入过程是在f_mount 函数后或存储介质改变后的第一次文件访问时完成的。

❹ Fat文件系统需要一个100Hz的中断来实现磁盘IO读写逻辑的接口。通过系统时钟创建一个10ms的定时中断即可。

❺ 监听串行通信窗口的Fat命令输入，如文件读取命令cat、显示文件列表命令ls、创建文件夹命令mkdir、修改文本内容命令wirte、改变文件路径命令chdir、显示当前文件路径命令pwd、创建文件命令creat等等。当Fat文件系统监听到这些命令后，会根据命令执行相应的文件操作并将处理结果通过串口通信在上位机窗口显示出来。

图 3-39　SD 卡读写流程图

(1) 代码解析

配置 UART

```
    /*********************************************************
***********
* Configure the UART and its pins.  This must be called before
 UARTprintf().
    *********************************************************
**********/
    void ConfigureUART(void)
    {
        // Enable the GPIO Peripheral used by the UART.
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

        // Enable UART0
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
```

```
        // Configure GPIO Pins for UART mode.
        ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
        ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
        ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 |
GPIO_PIN_1);

        // Use the internal 16MHz oscillator as the UART clock
source.
        UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

        // Initialize the UART for console I/O.
        UARTStdioConfig(0, 115200, 16000000);
    }
```

挂载 Fat 文件系统

```
    /*********************************************************
    * Mount the file system, using logical disk 0.
    *********************************************************/
        iFResult = f_mount(0, &g_sFatFs);
        if(iFResult != FR_OK)
        {
            UARTprintf("f_mount error: %s\n",
StringFromFResult(iFResult));
            return(1);
    }
```

监听串口命令行

```
/*********************************************************
*****
   Enter an infinite loop for reading and processing commands
from the user.
*********************************************************
****/
        while(1)
        {
            // Print a prompt to the console.  Show the CWD.
            UARTprintf("\n%s> ", g_pcCwdBuf);
```

```
            // Get a line of text from the user.
            UARTgets(g_pcCmdBuf, sizeof(g_pcCmdBuf));

           // Pass the line from the user to the command
processor.  It will be
            // parsed and valid commands executed.
            nStatus = CmdLineProcess(g_pcCmdBuf);

            // Handle the case of bad command.
            if(nStatus == CMDLINE_BAD_CMD)
            {
                UARTprintf("Bad command!\n");
            }

            // Handle the case of too many arguments.
            else if(nStatus == CMDLINE_TOO_MANY_ARGS)
            {
                UARTprintf("Too many arguments for command
processor!\n");
            }

            // Otherwise the command was executed.  Print the
error code if one was
            // returned.
            else if(nStatus != 0)
            {
                UARTprintf("Command returned error code %s\n",
                           StringFromFResult((FRESULT)nStatus));
            }
    }
```

枚举当前路径下的文件及文件夹

```
/*************************************************************
******* This function implements the "ls" command.  It opens
the current directory and enumerates through the contents, and
prints a line for each item it finds.  It shows details such
```

```
as file attributes, time and date, and the file size, along
with the name.  It shows a summary of file sizes at the end
along with free space.
*******************************************************************
******/
    int Cmd_ls(int argc, char *argv[])
    {
        uint32_t ui32TotalSize;
        uint32_t ui32FileCount;
        uint32_t ui32DirCount;
        FRESULT iFResult;
        FATFS *psFatFs;

        // Open the current directory for access.
        iFResult = f_opendir(&g_sDirObject, g_pcCwdBuf);

        // Check for error and return if there is a problem.
        if(iFResult != FR_OK)
        {
            return((int)iFResult);
        }

        ui32TotalSize = 0;
        ui32FileCount = 0;
        ui32DirCount = 0;

        // Give an extra blank line before the listing.
        UARTprintf("\n");

        // Enter loop to enumerate through all directory
entries.
        for(;;)
        {
            // Read an entry from the directory.
            iFResult = f_readdir(&g_sDirObject, &g_sFileInfo);

            // Check for error and return if there is a problem.
```

```
            if(iFResult != FR_OK)
            {
                return((int)iFResult);
            }

    // If the file name is blank, then this is the end of the
listing.
            if(!g_sFileInfo.fname[0])
            {
                break;
            }

    // If the attribue is directory, then increment the
directory count.
            if(g_sFileInfo.fattrib & AM_DIR)
            {
                ui32DirCount++;
            }

    // Otherwise, it is a file.  Increment the file count, and
add in the
    // file size to the total.
            else
            {
                ui32FileCount++;
                ui32TotalSize += g_sFileInfo.fsize;
            }

     // Print the entry information on a single line with
formatting to show
     // the attributes, date, time, size, and name.

UARTprintf("%c%c%c%c%c %u/%02u/%02u %02u:%02u %9u  %s\n",
                        (g_sFileInfo.fattrib & AM_DIR) ? 'D' : '-',
                        (g_sFileInfo.fattrib & AM_RDO) ? 'R' : '-',
                        (g_sFileInfo.fattrib & AM_HID) ? 'H' : '-',
                        (g_sFileInfo.fattrib & AM_SYS) ? 'S' : '-',
```

```
                    (g_sFileInfo.fattrib & AM_ARC) ? 'A' : '-',
                    (g_sFileInfo.fdate >> 9) + 1980,
                    (g_sFileInfo.fdate >> 5) & 15,
                    g_sFileInfo.fdate & 31,
                    (g_sFileInfo.ftime >> 11),
                    (g_sFileInfo.ftime >> 5) & 63,
                    g_sFileInfo.fsize,
                    g_sFileInfo.fname);
        }

        // Print summary lines showing the file, dir, and size
totals.
        UARTprintf("\n%4u File(s),%10u bytes total\n%4u
Dir(s)",
                   ui32FileCount, ui32TotalSize, ui32DirCount);

        // Get the free space.
        iFResult = f_getfree("/", (DWORD *)&ui32TotalSize,
&psFatFs);

        // Check for error and return if there is a problem.
        if(iFResult != FR_OK)
        {
            return((int)iFResult);
        }

        // Display the amount of free space that was
calculated.
        UARTprintf(", %10uK bytes free\n", (ui32TotalSize *
                                        psFatFs->sects_clust /
2));

        // Made it to here, return with no errors.
        return(0);
    }
```

实验步骤

(2) 和本章实验一步骤 1、2 一致，首先准备好模块和实现与 PC 端的链接。

(3) 将 MicroSD 卡插入 LCD 模块背后的插槽中。

(4) 按照前一章节中介绍的方式添加创建一个新的 CCS 工程 CH3_3。

(5) 参照第二章实验 CH2_1 中图 2-11 所述，添加工程包含路径：

（..\TI\TivaWare_C_Series-1.0）

（..\TI\TivaWare_C_Series-1.0\third_party）其中..\代表实验中本地电脑中 TI 的安装路径。

(6) 参照第二章实验 CH2_2_2 中图 2-18 所述，添加工程编译链接属性设置。将（..\TI\TivaWare_C_Series-1.0\driverlib\ccs\Debug\dirverlib.lib）链接库添加到工程配置中。

(7) 添加对应配套程序包中源代码到 CH3_3 工程目录下。其中包含文件夹 third_party、utils，文件 main.c、startup_ccs.c。

(8) 参照本章实验二步骤第七步中所述，在高级配置（Advanced Options）中的编译语言使能配置（Language Options）中添加使能外部 gcc 编译功能。

(9) 在 ARM 编译器（ARM Compiler）高级配置（Advanced Options）中的预定义符号里（Predefined Symbols）添加 ccs="ccs"、PART_TM4C123GH6PGE 、TARGET_IS_BLIZZARD_RA1、ENABLE_LFN 等几个变量符号。如下图所示：

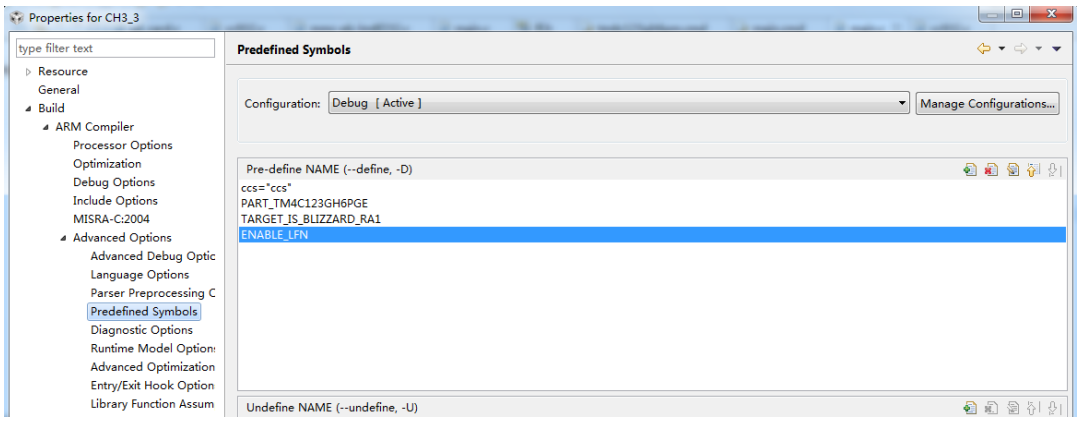


图 3-40 预定义符添加

(10) 编译通过并烧写到 Tiva LaunchPad 中。

(11) 打开 sscom4.2 串口监听程序。在设备管理器中查找到 Tiva 对应的虚拟串口 COM8（实际情况的串口端口号依赖与实际使用电脑设备的分配机制，不一定就是 COM8 口）。

图 3-41 虚拟串口信息

配置串口信息如下图所示：

图 3-42 虚拟串口配置

(12) 成功连接后，可以用 ls 命令行向 Tiva 发送查找 MicroCD 卡文件系统信息的命令。

Tiva 会响应查找并反馈 MicroSD 卡文件系统信息如下图所示：

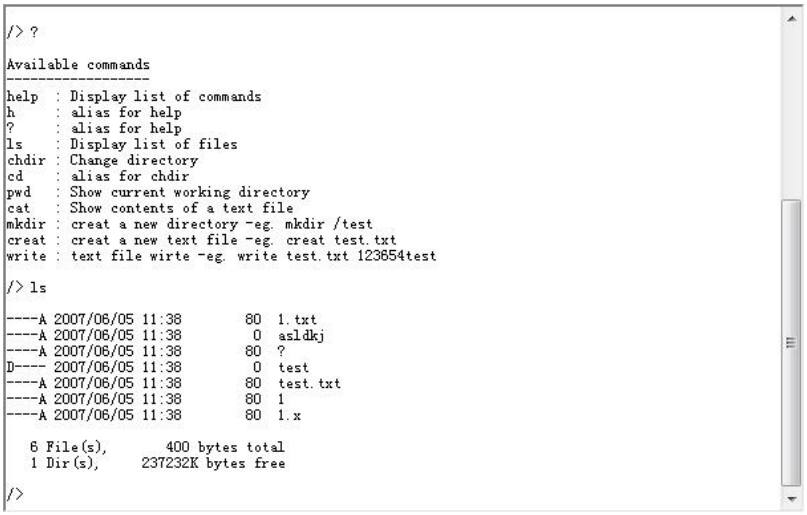


图 3-43 虚拟串口命令号操作

(13) 使用 mkdir、creat、write 等命令行操作尝试创建一个新的文本文件，并采用 cat 命令查看创建的文本文件的内容信息。