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。

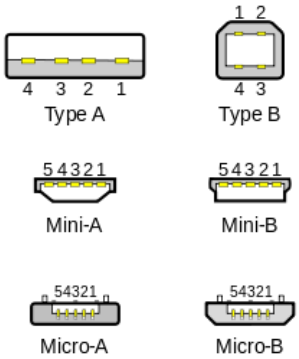


图 3-1 USB 接口

- 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。

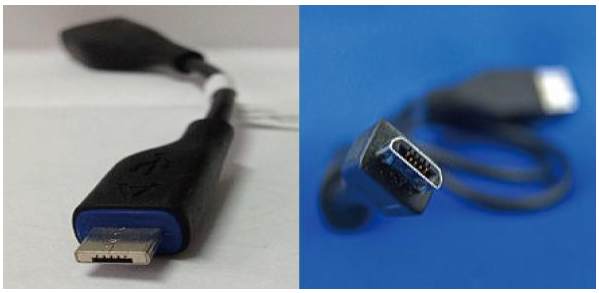


图 3-4 Micro USB 接口（左：A 型，右：B 型）

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。

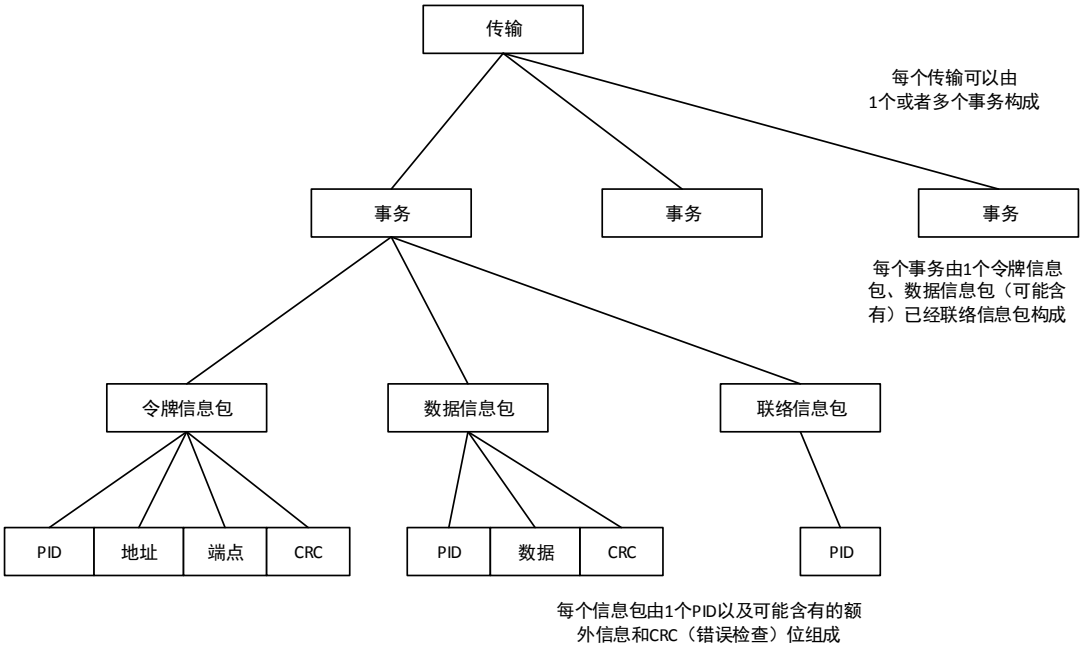


图 3-5 由事务组成的 USB2.0 传输

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。

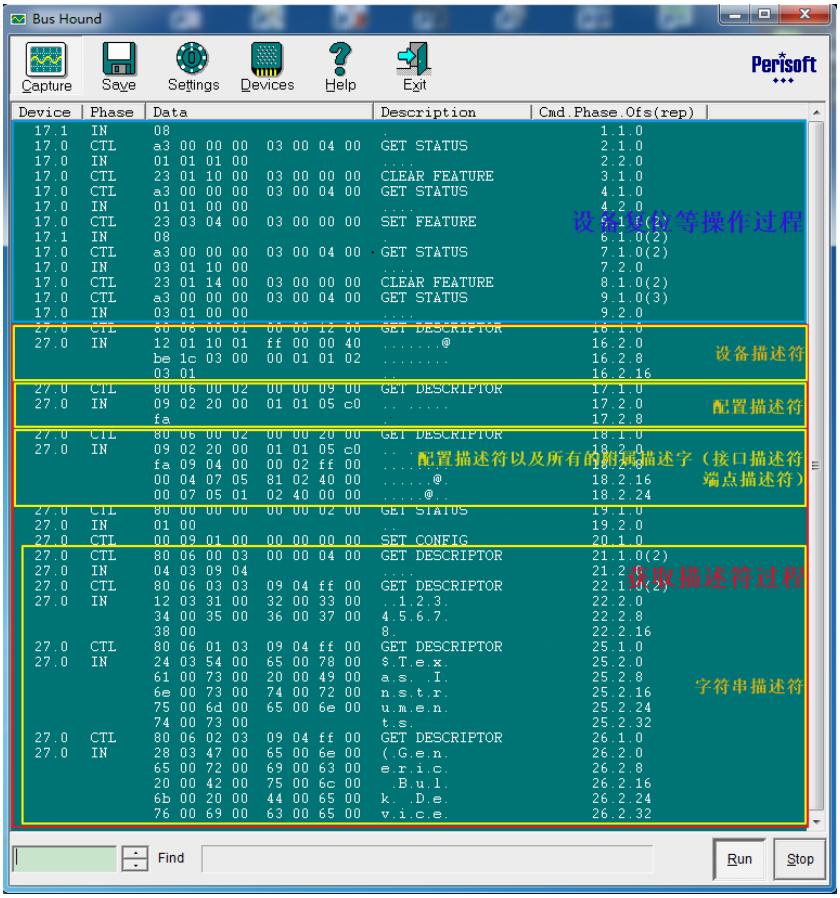


图 3-6 USB 枚举过程

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。

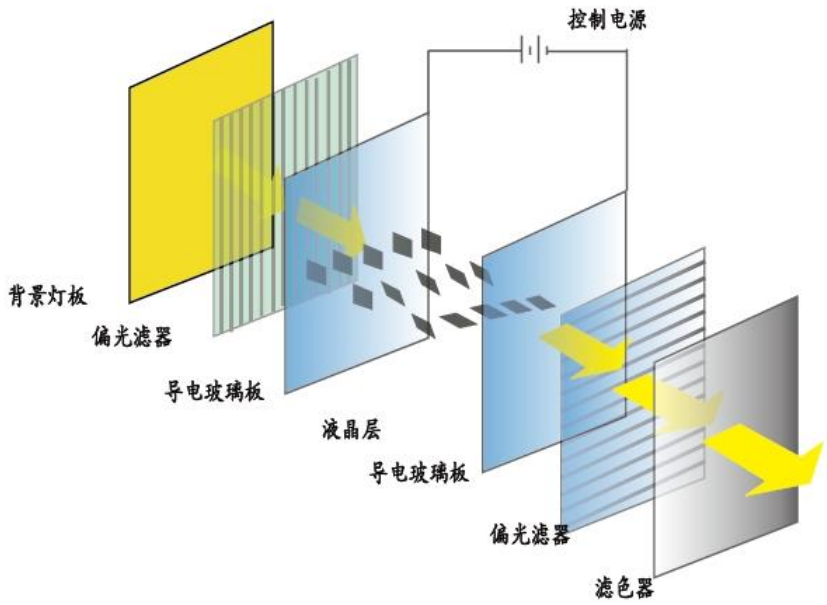


图 3-7 液晶结构

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：

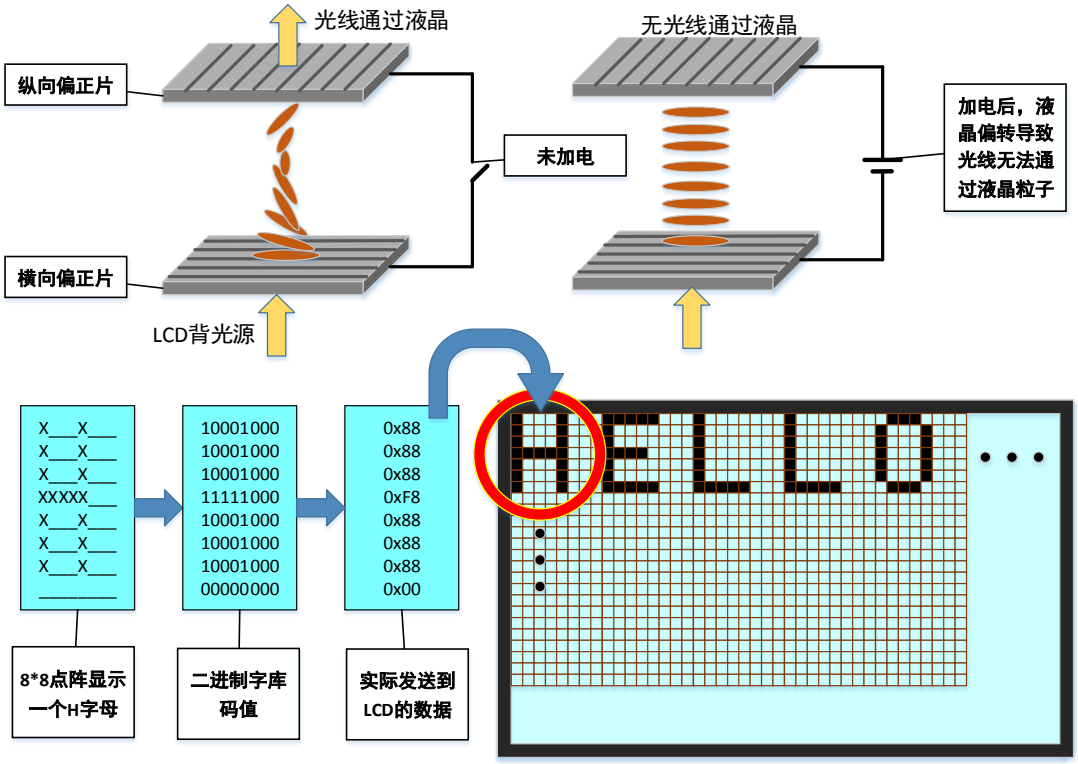


图 3-8 显示和数据的关系

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。

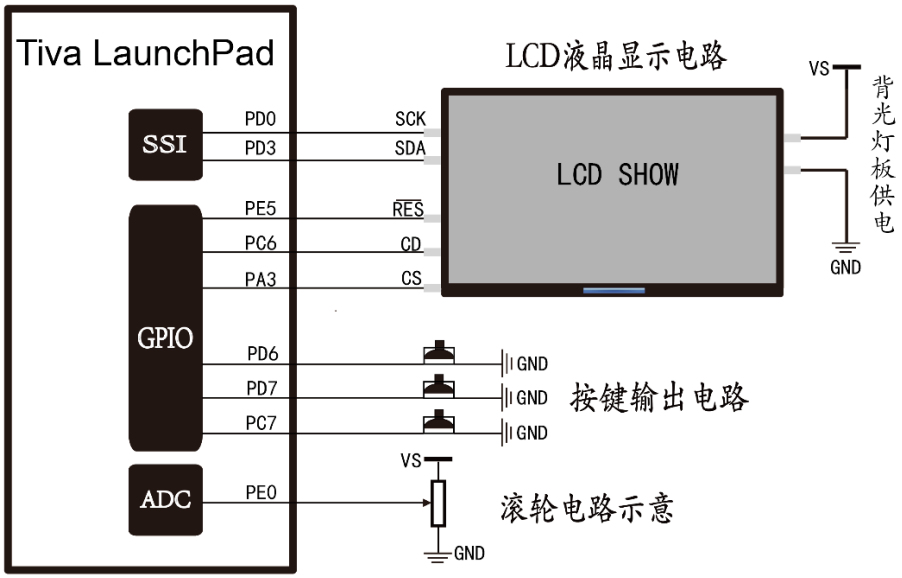


图 3-9 硬件结构图

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。

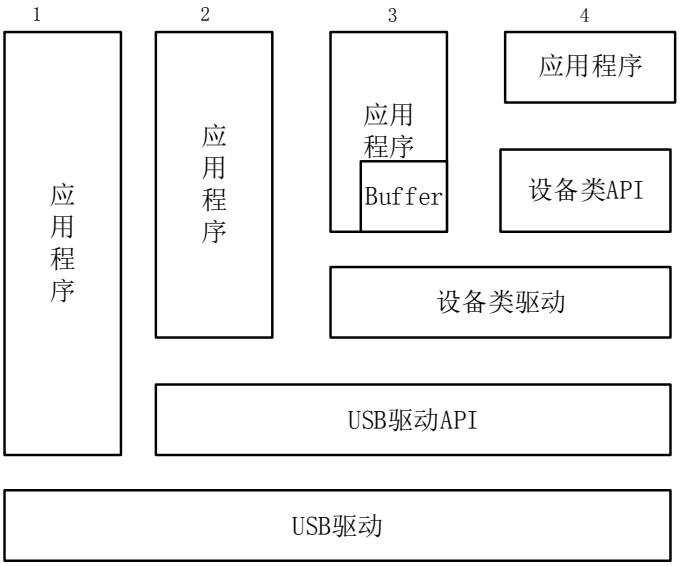


图 3-10 USB 开发框架

(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。

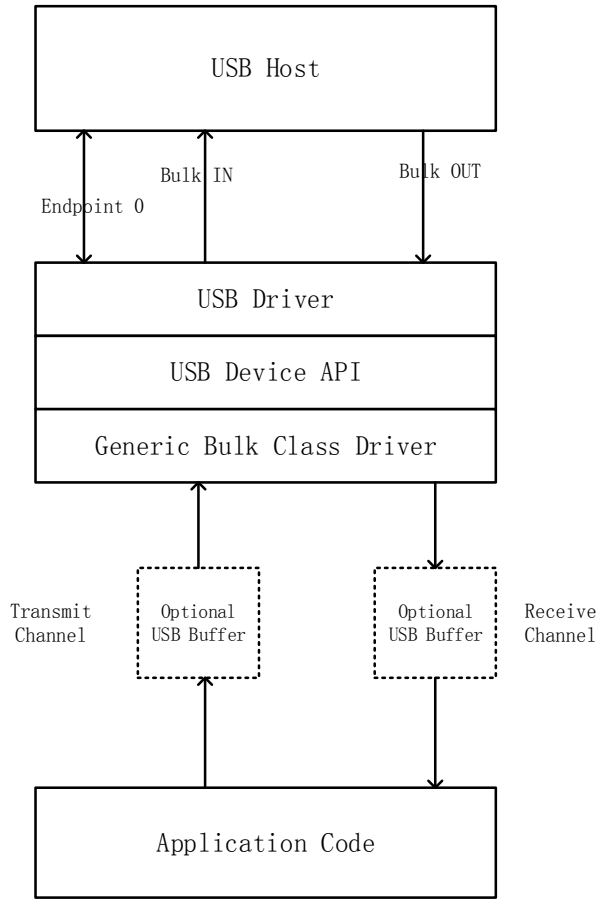


图 3-11 Bulk 模式开发

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

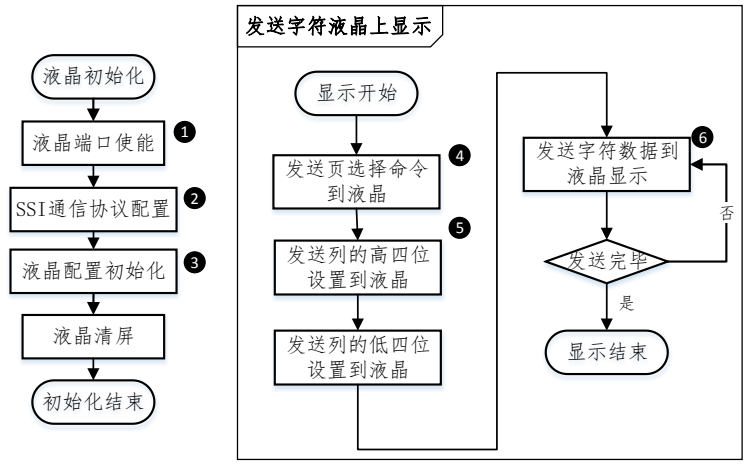
LCD 的函数库由 LCD_Matrix.h、LCDDriver.c、LCDDriver.h 构成，其中：

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

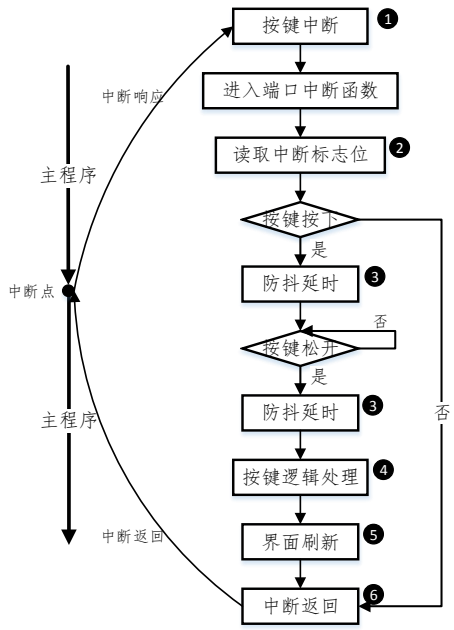
本模块和 USB 部分能完成的实验项目：（1）实验一：按键和滚轮控制液晶显示内容；主要目的是帮助体会和学习应用按键和滚轮的这两种不同输入控制方式，它们分别应用了 MCU 的不同外设，一个是采用了 GPIO，另一个则利用了 AD。 此外也让用户学习使用点阵式液晶的应用，同时学习 I^2C 串行通信的软件编程。（2）实验二：通过 USB 数据线与 PC 通信；通过实验学习 USB Bulk 模式通信，学习如何利用多种软件工具调试，学习 TivaWare 中 USB 函数库的应用。（3）实验三、SD 卡数据读写，通过实验学习 SPI 串行通信的软件编程，同时理解存储设备上的文件存储格式 FAT。

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
```

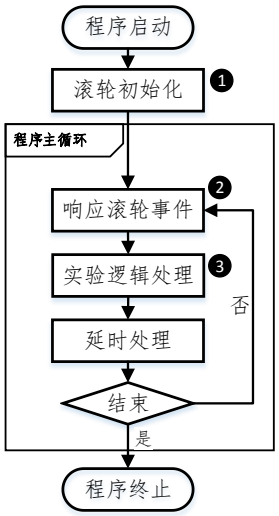
```
    *start address other than 0.
    ***************************************************
***********/
    #pragma DATA_SECTION(g_pfnVectors, ".intvecs")
    void (* const g_pfnVectors[])(void) =
    {
        (void (*)(void))((unsigned long)&__STACK_TOP),
                                        // 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。

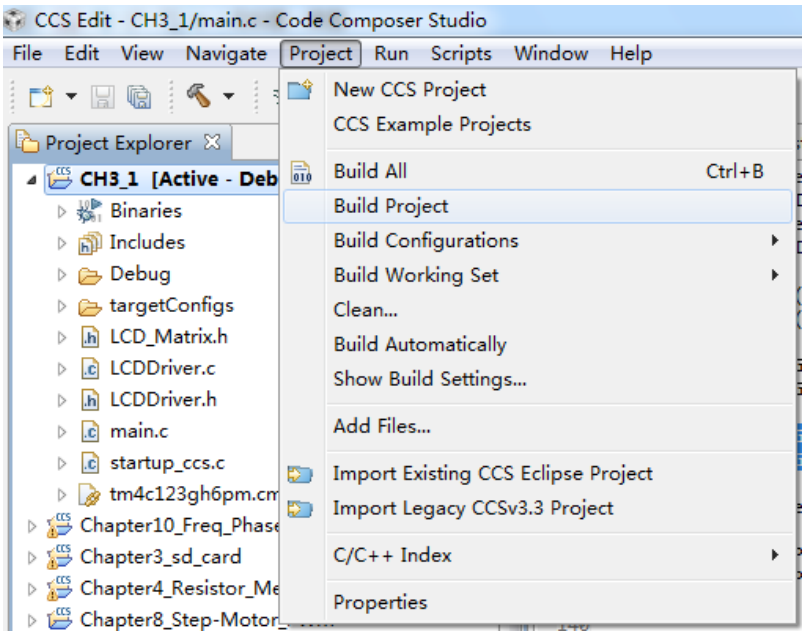


图 3-15 编译工程

(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）。

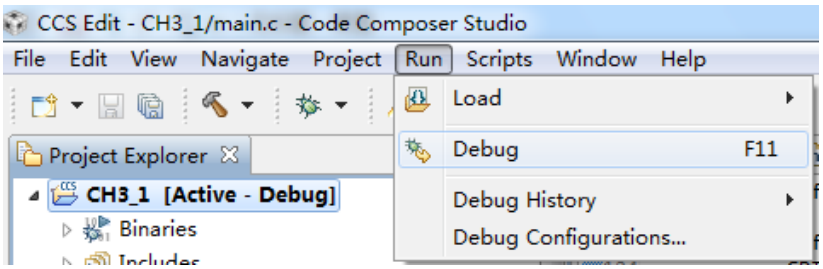


图 3-16 烧写工程到 Tiva LaunchPad

(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
};
```

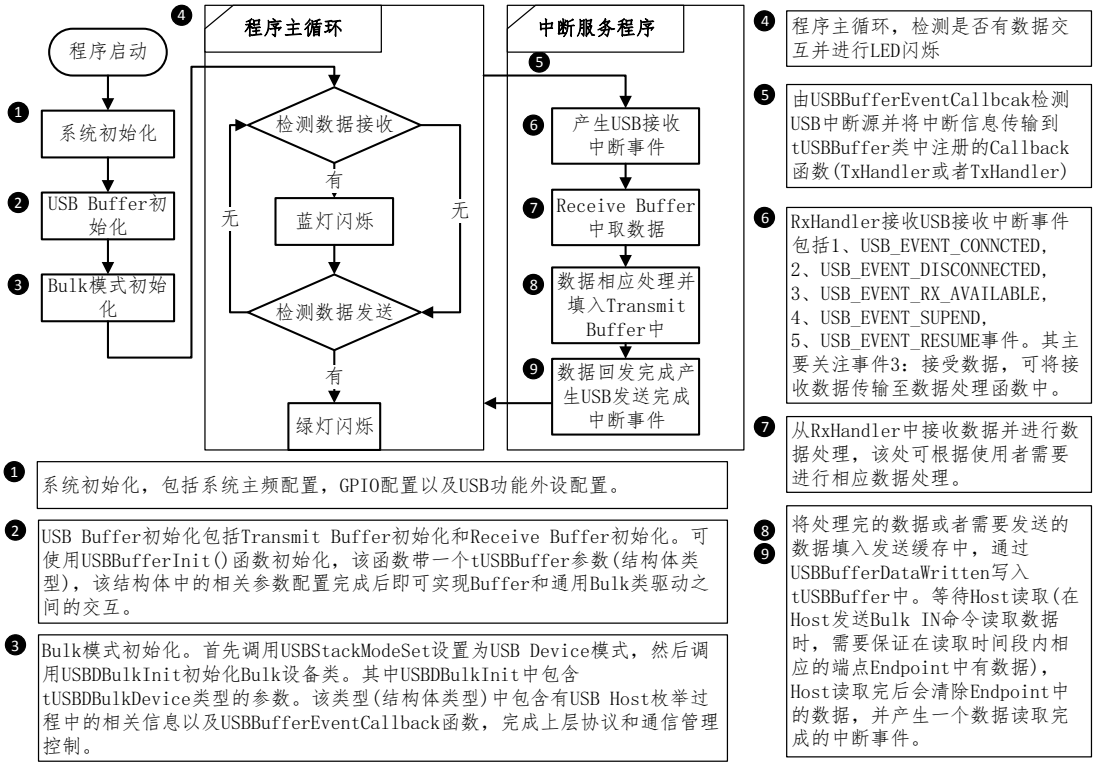


图 3-17 Bulk 模式软件流程图

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。

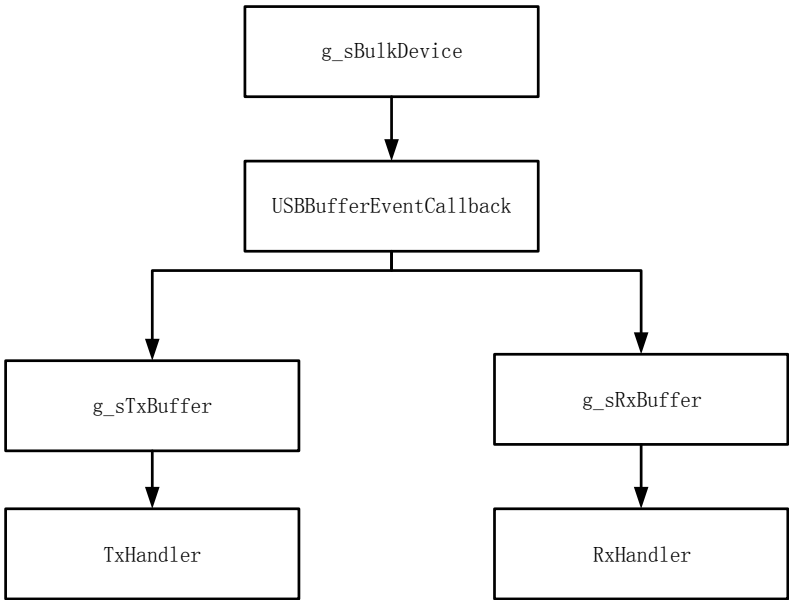


图 3-18 结构体关系图

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。

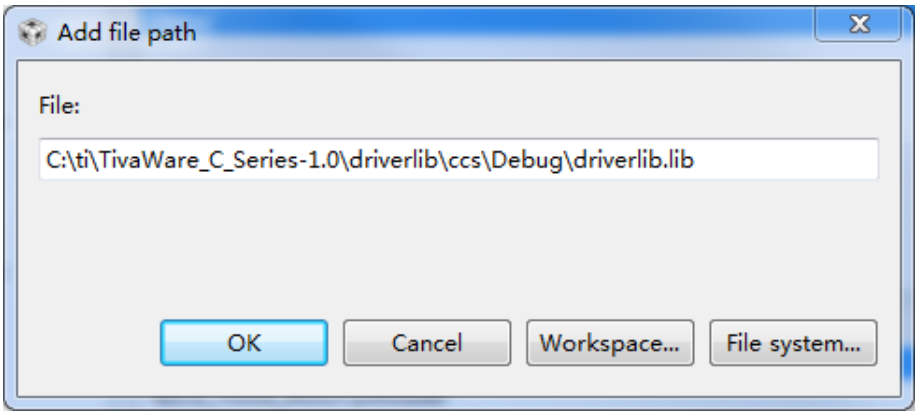


图 3-19 TivaWare 外设驱动库的引用

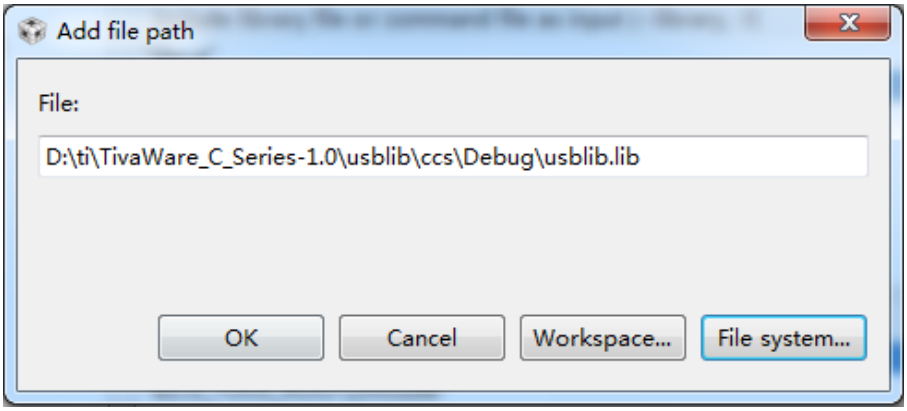


图 3-20 TivaWare USB 库的引用

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。

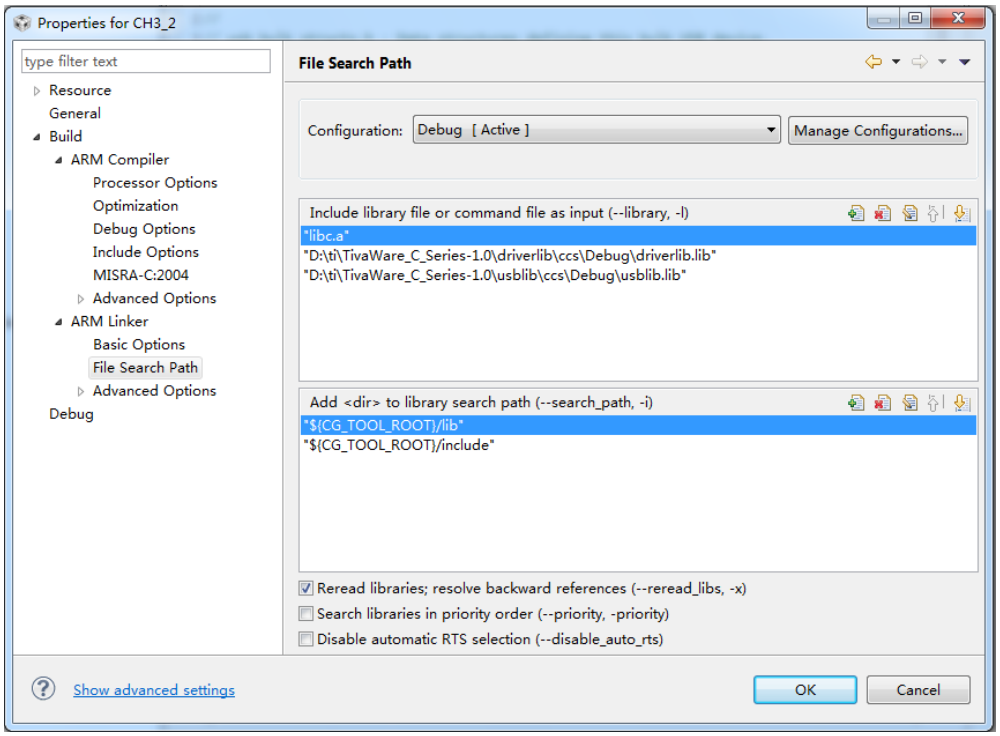


图 3-21 引用 TivaWare 库的链接属性窗口

(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。

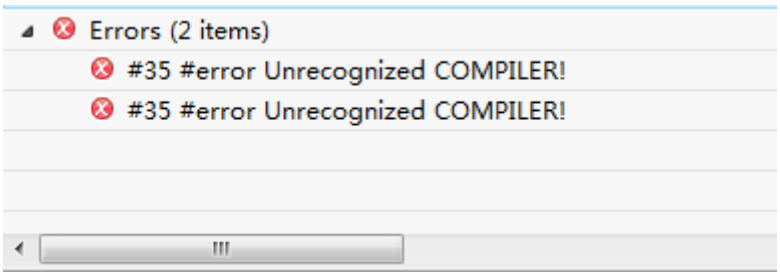


图 3-22 编译错误

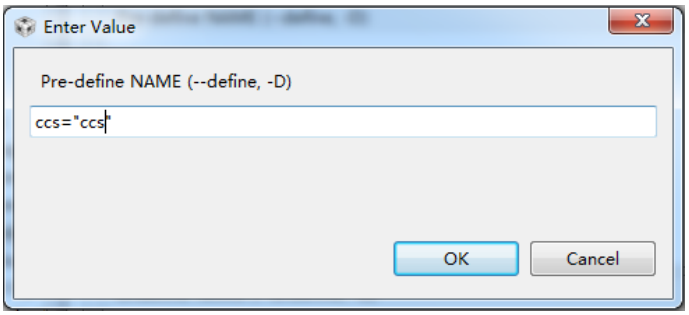


图 3-23 添加 ccs=“ccs”

(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。

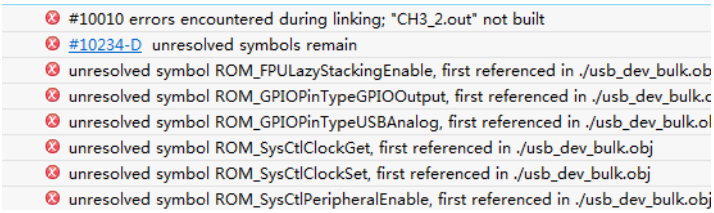


图 3-24 编译错误

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

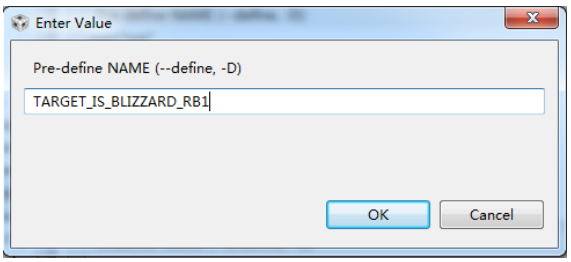


图 3-26 添加 TARGET_IS_BLIZZARD_RB1

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

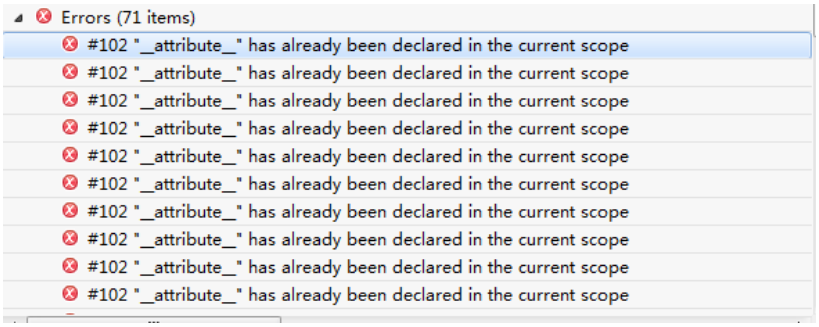


图 3-27 编译错误

(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。

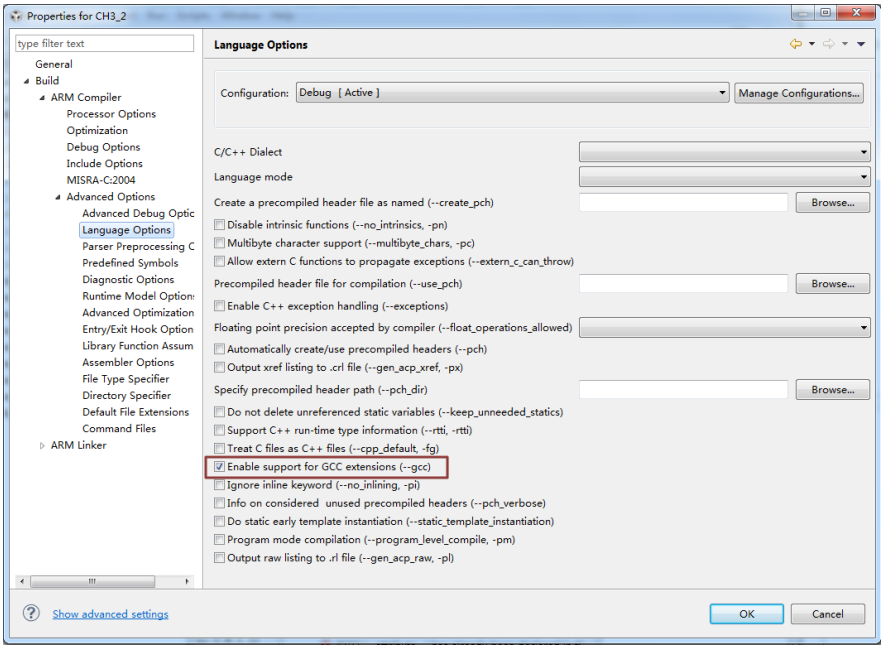


图 3-28 勾选 GCC

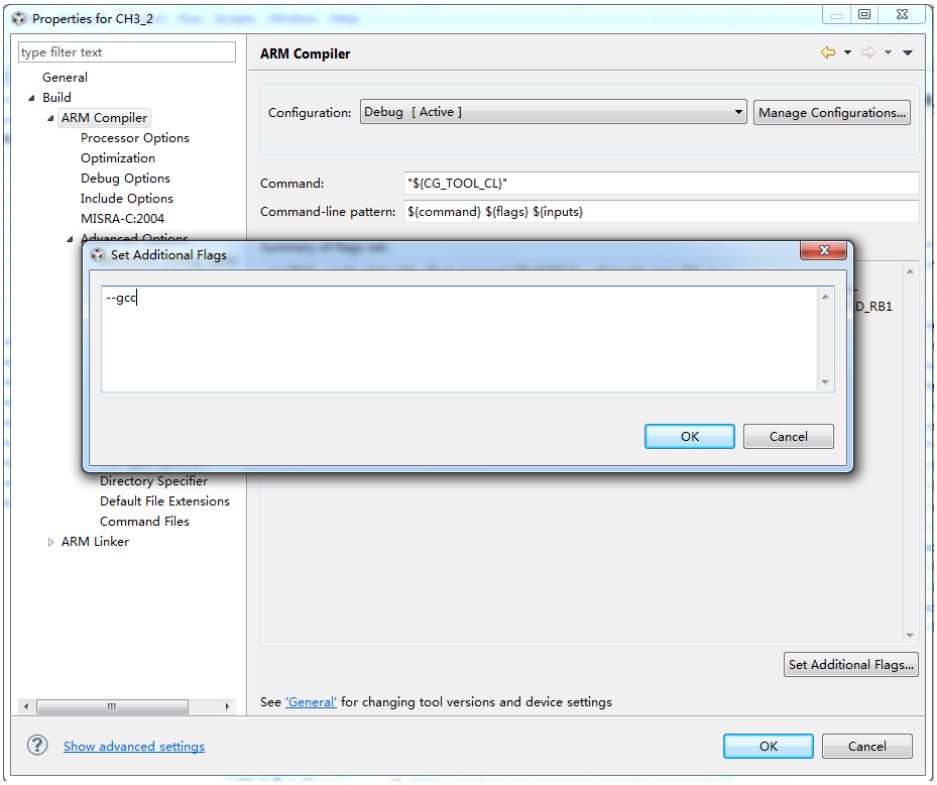


图 3-29 添加—gcc

(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。

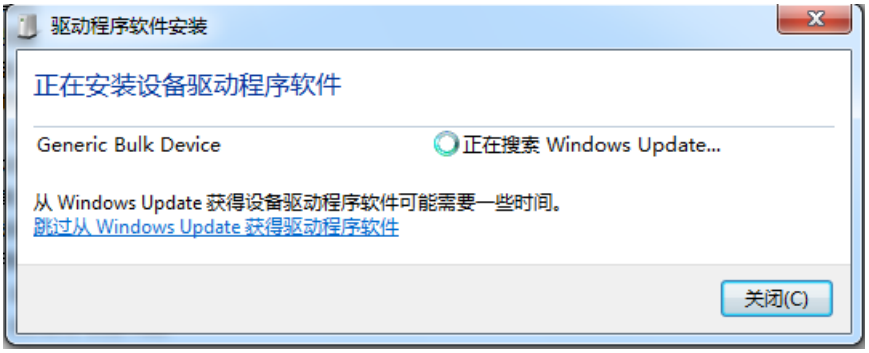


图 3-30 查找驱动程序

(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。

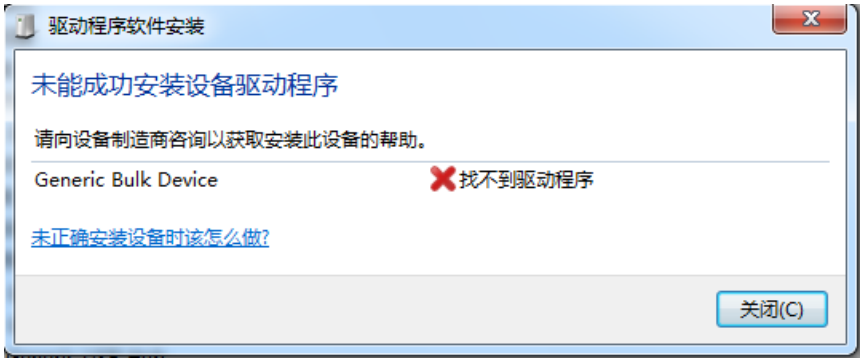


图 3-31 驱动程序安装失败

(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。

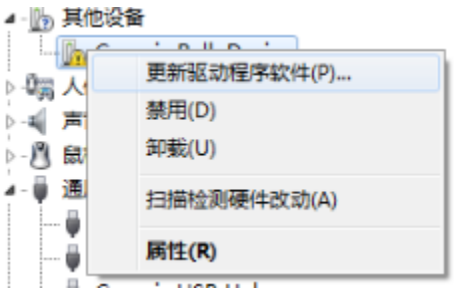


图 3-32 更新驱动程序

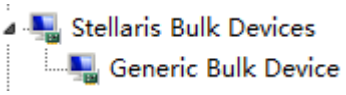


图 3-33 驱动安装完成

(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。

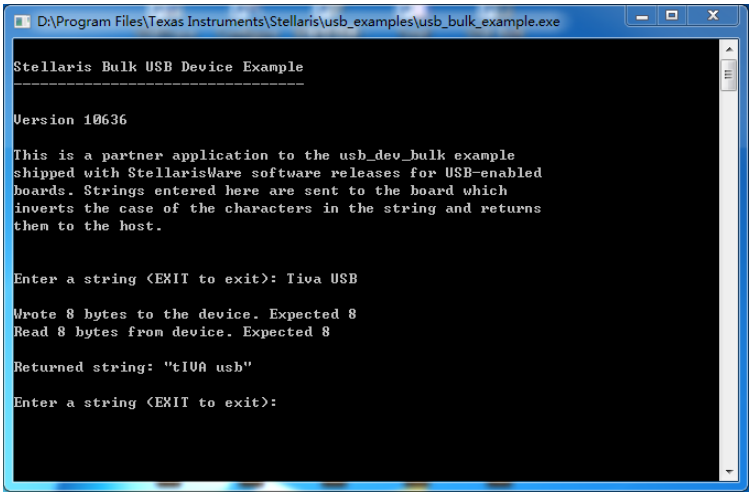


图 3-34 实验结果

(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。

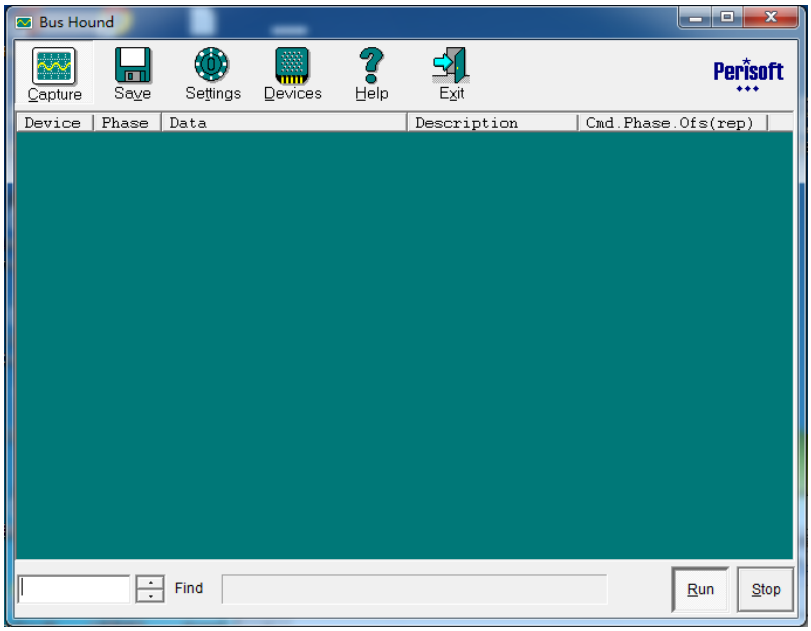


图 3-35 Bus Bound 软件界面

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。

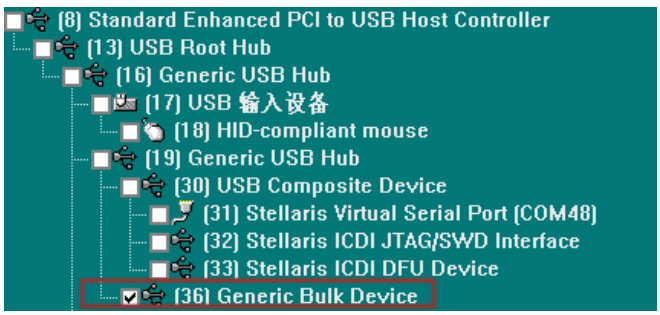


图 3-36 Devices 设置

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

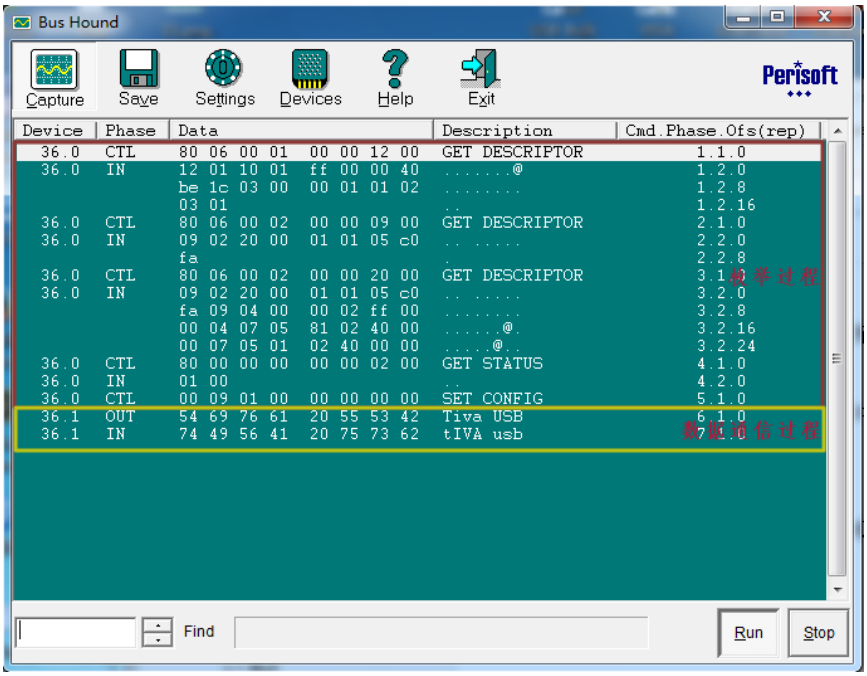


图 3-37 抓包过程

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：

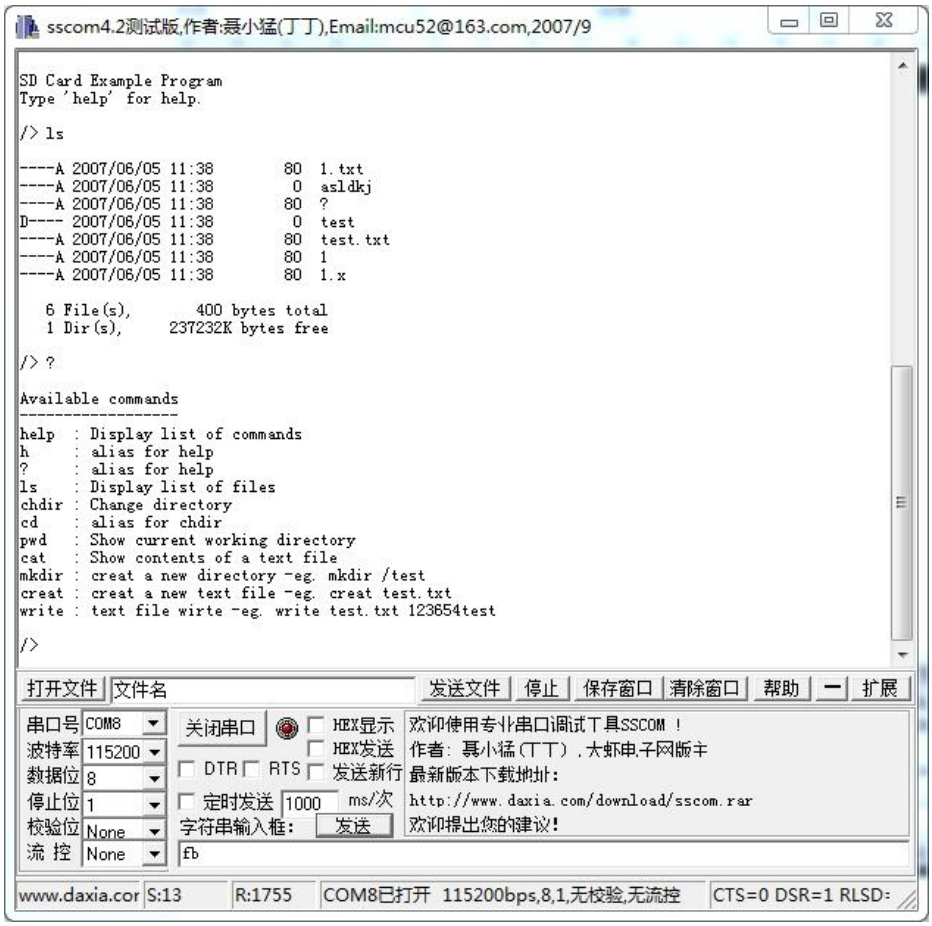
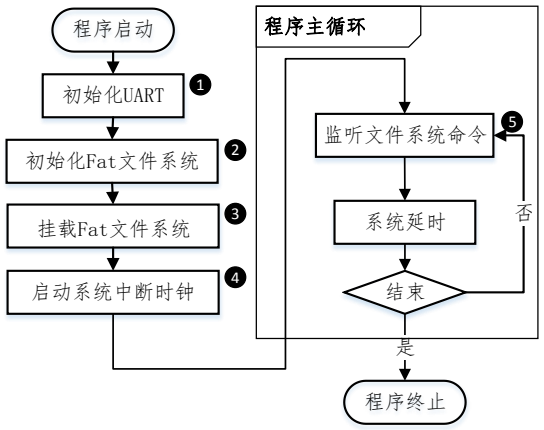


图 3-38 Tiva 与上位机串口通讯控制台示意图

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：

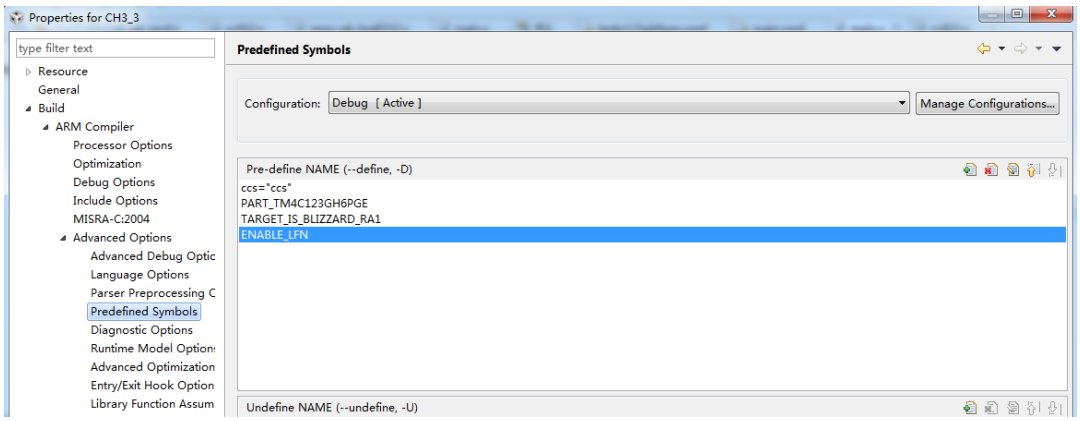


图 3-40 预定义符添加

(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：

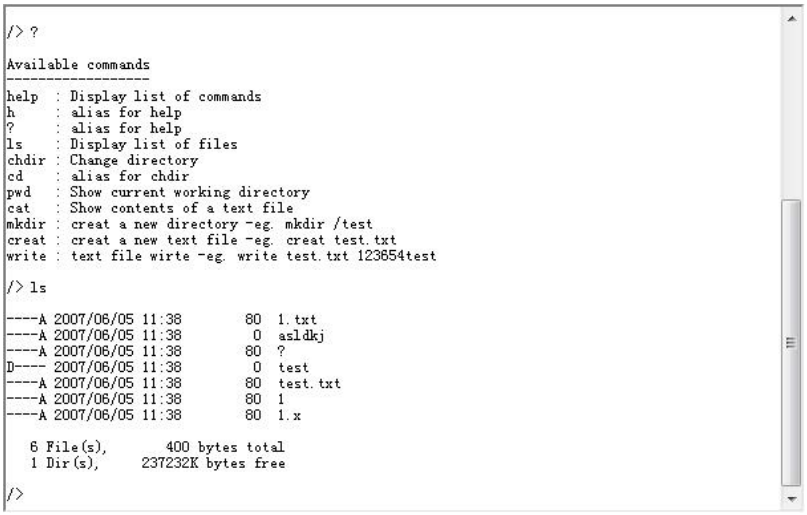


图 3-43 虚拟串口命令号操作

(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。