第5章　人机接口设计

人机接口提供了人与嵌入式系统进行信息交互的手段，通过人机接口，人可以给嵌入式系统发送操作指令，嵌入式系统的运行结果也可以通过显示等方式提交给人。人机接口的方式很多，在嵌入式系统中常用的人机接口设备有键盘、LED显示器、OLED显示器等。由于Zynq芯片内部没有集成键盘及显示器等的控制部件，因此，可以利用Zynq芯片内部PS部分的GPIO来扩展键盘或显示器的接口，也可利用FPGA逻辑来扩展键盘或显示器的接口。在利用FPGA来扩展功能时，可以采用成熟的IP核，这样可以提高嵌入式系统的开发效率。若没有成熟的IP核，用户也可以自己设计IP核，为以后开发相同功能的嵌入式系统时使用。本章首先对IP核的相关概念进行介绍，然后介绍键盘、LED显示器等几种人机设备的接口设计。

5.1　IP核的概述

IP核(Intellectual Property Core，知识产权核)是指一种基于FPGA逻辑上预先设计好的、并被验证是正确的、能完成某种特定功能的硬件功能模块。IP核实际上就是一种可以重用的硬件功能单元，类似于硬件功能芯片(但不是具体的芯片)。采用IP核可以缩短嵌入式系统的开发周期，提高开发的有效性和安全性。

5.1.1　IP核的分类

最早的IP核概念是Arm公司提出的，Arm公司通过出售Arm架构的微处理器核(即一种实现CPU功能的IP核)，使得Arm架构的微处理器迅速在市场上得到推广应用。同时，Xilinx公司或其他第三方IP核公司还提供了许多其他功能的IP核，如实现UART接口功能的IP核、实现AXI协议的IP核、实现FIR滤波器功能的IP核等。

1. 按提供给用户的体现形式分类

IP核按提供给用户的体现形式，可以分成软核、固核、硬核等3种形式。它们实际上也是IP核授权给他人的3种不同的级别，也对应着EDA功能设计的3种不同级别，即行为设计、结构设计、物理设计。

软核(Soft IP Core)是指硬件功能模块的寄存器传输级(RTL)的设计模型，是行为级的设计。即是用硬件描述语言(如Verilog)所设计的硬件逻辑电路，包括对电路的逻辑描述，并可以综合生成网表文件，但不涉及具体的实现芯片。换句话说，软核提供给用户的是电路逻辑设计的源代码文件，其功能经过行为级的仿真验证，但没有完成综合及布线(包括管脚约束)。因此，软核的最大优点是灵活性强、可移植性好、为使用者提供了较大的后续设计空间。其缺点是知识产权的保护不够安全，需要用户重视知识产权保护。

硬核(Hard IP Core)是指硬件功能模块的最终级的设计模型，它不仅完成了RTL级的

设计和验证，也完成了综合及布局布线，是以完整验证过的设计版图形式提供给用户。某个硬件功能的 IP 硬核，其布局和工艺是固定的，用户只能使用 IP 硬核，而不能对其进行修改，因此，硬核的灵活性差。但由于其不需要提供 RTL 级源文件，用户不可能知道该硬件功能的具体实现方法，而只能使用其硬件功能，因而更容易实现知识产权的保护。

固核(Firm IP Core)是介于软核和硬核之间的一种 IP 核的体现形式，是指带有布局规划的软核。即其提供给用户的 IP 核形式通常是 RTL 级原程序，以及对应具体网表的混合形式。在用户设计时，采用固核比采用软核的设计灵活性稍差，但其知识产权的保护比软核要强。

2. 按实现的功能分类

按实现的功能分类，IP 核可以分成 CPU IP 核、外设端口 IP 核、网络通信 IP 核、算法实现 IP 核及其他专用功能的 IP 核等。

CPU IP 核实现的是微处理器功能，目前主要有四大系列的 CPU IP 核，即 Arm 系列的微处理器核、MIPS 系列的微处理器核、x86 系列的微处理器核和 PowerPC 微处理器核。这些微处理器核在不同的应用领域均有不同级别的授权应用，典型的如 Zynq 芯片内部即集成有 Arm 的 Cortex-A9 微处理器核，它是一种硬核的形式。

外设端口 IP 核实现的是一些外部设计接口的功能，如 SPI 接口的 IP 核、CAN 总线接口 IP 核、UART 接口的 IP 核，利用这些外设接口 IP 核，用户在设计嵌入式系统时，可以方便地扩展外设接口。

算法实现 IP 核是需要进行大数据量运算的算法 IP 核，如 DES 加密算法 IP 核、FIR 滤波算法 IP 核、DCT 变换 IP 核等。

其他专用功能的 IP 核是完成某个专用功能的 IP 核，如 ADC(模数转换)IP 核、VGA 接口控制 IP 核、LCD 显示驱动 IP 核等。

5.1.2　IP 核的标准

在 20 世纪 90 年代，由于像 Arm 公司这样提供可复用硬件电路模块(IP 核)公司的出现，使得 SoC(System on Chip，片上系统)设计技术得到了推广和普及，也使得用户的嵌入式系统功能可以设计得更加复杂，并且用户在采用第三方成熟的 IP 核后，可以缩短其嵌入式系统的开发周期。

随着 IP 核的使用越来越广泛，由第三方公司设计的、作为商品的 IP 核也越来越多，因此，用户在设计基于 SoC 的嵌入式系统时，会遇到以下问题。

(1) 如何选择 IP 核。前面提到，IP 核的功能种类很多，即使是同样功能的 IP 核，也会有多个 IP 核供应商来提供。因此，IP 核的供应商需要提供标准的功能说明文件，以便用户评价 IP 核的功能是否能满足要求，并且需要一个对性能、质量进行评价的统一体系。

(2) 如何连接各种 IP 核。由于构建 SoC 的嵌入式系统时，所选择的 IP 核可能来自不同的供应商，如何使它们能相互集成在一起，就需要解决 IP 核接口的标准问题。

(3) 如何对选择的 IP 核(主要是软核)进行部分修改，使其适应用户需求。用户对所选择的 IP 核内部结构不一定非常清楚，因此，这也需要 IP 核的供应商提供标准的功能说明文件。

(4) 如何测试验证 IP 核。对 IP 核的功能、性能等进行测试，需要一种标准化的测试标

准,才能公平地对不同供应商提供的 IP 核进行测试及评价。

针对上述 IP 核使用中的问题,就需要制定 IP 核的标准,来促进 IP 核技术的发展及使用,这最终导致了 IP 核标准的产生及相关国际组织的出现。

1996 年,在国际上建立了最早的 IP 核标准化组织 VSIA,它是全方位制定 IP 核标准的国际联盟,成员包括系统设计公司、半导体生产商、EDA 设计公司、IP 核设计公司等。其目标是制定 SoC 工业的技术规范和标准,使得不同厂商的 IP 核能够集成在一起,完成一个统一的整体功能。在 VSIA 规范中,IP 核又称为虚拟元件 VC(Virtual Component)。2003 年 VSIA 组织制定了 4 类 IP 核规范及标准,即 IP 核交付使用文档规范、IP 核复用设计标准、IP 核质量评估标准、IP 核知识产权保护文件。

随着 IP 核产业的发展,出现了专业性更强的标准化组织,即 OCP-IP 和 SPIRIT。OCP-IP 标准化组织于 2001 年 12 月成立,它的目标是建立一套通用的 IP 核接口标准。SPIRIT 组织于 2003 年 6 月成立,主要在 IP 元数据描述标准和 EDA 工具的 API 标准方面进行工作,主要成员是 IP 核设计公司和 EDA 工具设计公司。例如,IP 核设计公司有 Arm 公司、Philips 公司等; EDA 设计公司有 Cadence 公司、Mentor 公司等。

总地来说,IP 核标准主要注重于 IP 核复用交付文档、IP 核接口标准的实用性、IP 核性能的评估手段、IP 核的技术保护手段等方面。解决的方案主要有两种:一种是采用标准总线结构方式,如采用 AMBA 总线;另一种是采用标准的接口信号,IP 核之间的互联不限制必须采用总线,可以采用点对点的连接。

2002 年我国成立了“信息产业部集成电路 IP 核标准工作组”,简称 IPCG。该组织负责制定我国的 IP 核技术标准。由于国内的集成电路发展水平有限,因此,目前还是在学习和借鉴国外的成熟 IP 核标准体系,并逐步建立适应我国集成电路设计水平、能与国际标准兼容的我国 IP 核标准体系。

本书是以 Zynq 芯片为背景来介绍 IP 核的设计技术,因此,涉及的 IP 核标准主要是采用 AMBA 总线接口标准。

5.2 键盘接口

键盘是最常用的人机输入设备,与通用个人计算机(即 PC)的键盘不一样,嵌入式系统中的键盘,其所需的按键个数及功能通常是根据具体应用来确定的,不同的应用其键盘中按键个数及功能均可能不一致。因此,在嵌入式系统的键盘接口设计时,通常需要根据应用的具体要求,来设计键盘接口的硬件电路,同时还需要完成识别按键动作、生成按键码和按键具体功能的程序设计。

5.2.1 按键的识别方法

嵌入式系统所用键盘中的按键通常是由机械开关组成的,通过机械开关中的簧片是否接触来断开或者接通电路,以便区别键是否处在按下或释放状态。键盘的接口电路有多种形式,可以用专用的芯片来连接机械按键,由专用芯片来识别按键动作并生成按键的键值,然后把键值传输给微处理器;也可以直接由微处理器芯片的 GPIO 引脚来连接机械按键,

由微处理器本身来识别按键动作，并生成键码。下面主要介绍由微处理器 GPIO 引脚连接键盘时的按键识别方法，掌握了这种识别方法，对于其他形式的键盘接口方法也就比较容易理解了。

即使采用 GPIO 引脚直接连接机械按键，通常也会根据应用的要求，其接口电路有所不同。若嵌入式系统所需的键盘中按键个数较少（一般不多于 4 个），那么通常会将每一个按键分别连接到一个输入引脚上，如图 5-1 所示。微处理器根据对应输入引脚上电平是“0”还是“1”来判断按键是否按下，并完成相应按键的功能。

若键盘中机械按键的个数较多，这时通常会把按键排成阵列形式，每一行和每一列的交叉点上放置一个机械按键。如图 5-2 所示，是一个含有 16 个机械按键的键盘，排列成了 4×4 的阵列形式。对于由原始机械开关组成的阵列式键盘，其接口程序必须处理 3 个问题，即去抖动、防串键和产生键码。

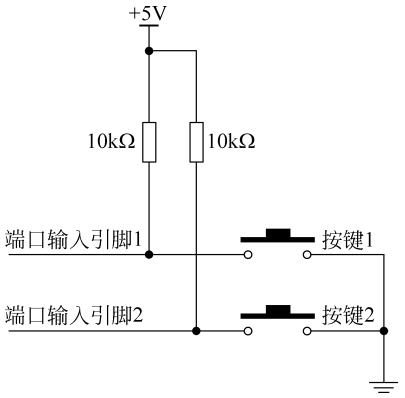


图 5-1　每个按键连接一根输入引脚

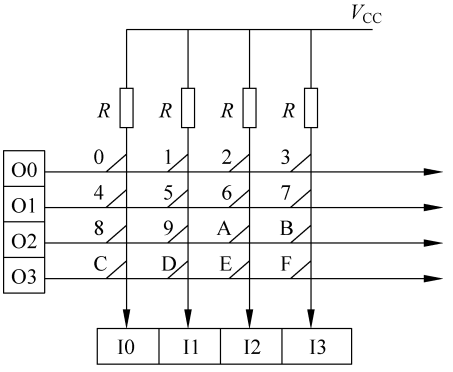


图 5-2　用作十六进制字符输入的键盘

① 抖动是机械开关本身的一个最普遍问题。它是指当键按下时，机械开关在外力的作用下，开关簧片的闭合有一个从断开到不稳定接触，最后到可靠接触的过程。即开关在达到稳定闭合前，会反复闭合、断开几次。同样的现象在按键释放时也存在。开关这种抖动的影响若不设法消除，会使系统误认为键盘按下若干次。键的抖动时间一般为 10～20ms，去抖动的方法主要采用软件延时或硬件延时电路。

② 串键是指多个键同时按下时产生的问题。解决的方法也是有软件方法和硬件方法两种。软件方法是用软件进行扫描键盘，生成键码是在只有一个键按下时进行。若有多键按下时，采用等待或出错处理。硬件方法则是采用硬件电路确保第一个按下的键或者最后一个释放的键被响应，其他的键即使按下也不会产生键码而被响应。

③ 产生键码是指键盘接口必须把按下的键翻译成有限位二进制代码，以便微处理器识别。在嵌入式系统中，由于对键盘的要求不同，产生键码的方法也有所不同。例如，可以直接把行信号值和列信号值合并在一起来生成键码，也可以采用一些特殊的算法来生成键码。但不管何种方法，产生的键码必须与键盘上的键一一对应。

下面以一个 4×4 阵列的键盘为例来说明键盘接口的处理方法及其流程。键盘的作用是进行十六进制字符的输入。如图 5-2 所示，该键盘排列成 4×4 阵列，需要两组信号线，一组作为输出信号线（称为行），另一组作为输入信号线（称为列），列信号线一般通过电阻与电

源正极相连。键盘上每个键的命名由设计者确定。

在图 5-2 所示的键盘接口中，键盘的行信号线和列信号线均由微处理器通过 GPIO 引脚加以控制，微处理器通过输出引脚向行信号线上输出全 0 信号，然后通过输入引脚读取列信号，若键盘阵列中无任何键按下，则读到的列信号必然是全 1 信号；否则就是非全 1 信号。若是非全 1 信号时，微处理器再在行信号线上输出“步进的 0”，即逐行输出 0 信号，来判断被按下的键具体在哪一行上，然后产生对应的键码。这种键盘处理方法称为“行扫描法”，具体流程如图 5-3 所示。

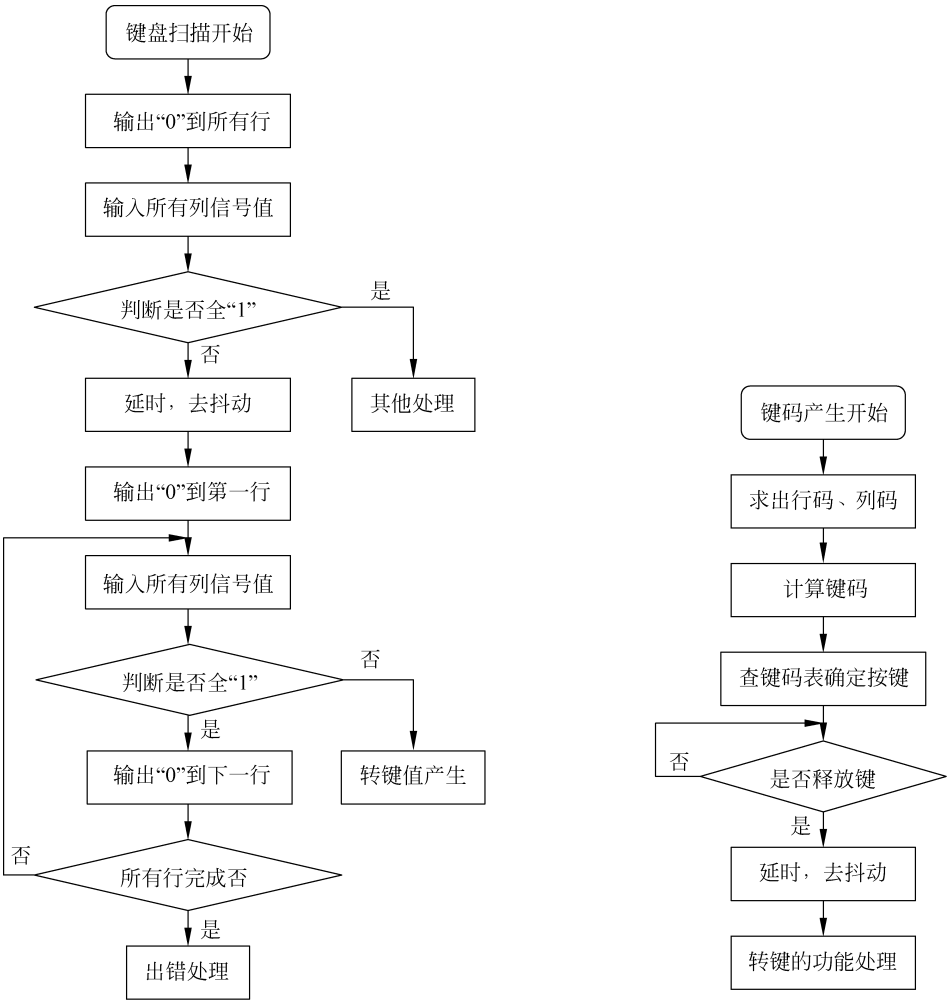


图 5-3 “行扫描法”键盘处理流程

键码的产生方法是多种多样的，但不论哪种方法都必须保证键码与键一一对应。通常情况下，常采用把行信号值和列信号值合并在一起生成键码。例如，若把行信号值和列信号值合并成一字节来做键码(注：行信号值在字节的高 4 位，列信号值在字节的低 4 位)，那么，图 5-2 中的按键 1，其键码为 0xED；按键 A，其键码为 0xBB。若要把行信号值和列信号值合并成一个 16 位的键码(注：即双字节，行信号值在高字节，列信号值在低字节，并且字节的高 4 位均置 1)，那么，图 5-2 中的按键“1”，其键码为 0xFEFD；按键 A，其键码为 0xFBFB。

除了采用行信号值和列信号值合并在一起生成键码的方法外，还经常采用一些键码产生的算法。下面再给出一种键码的产生算法，它比较适用于 16～64 键的键盘接口，并且键码采用 8 位二进制数表示。键码产生的算法步骤如下。

① 根据键盘扫描中所得到的行信号计算出被按下键所在行的行数，以数据最低位对应的键盘行为第一行，以此类推。

② 求行数的补(模为 256)，并求出其对应的二进制编码。

③ 将行数的补对应的二进制码左移 4 位，然后与列码相加，所得到的码即为键码。

例如，在图 5-2 所示的键盘接口电路中，键 9 的键码计算如下。

(1) 键 9 所在的行是第三行(对应的数据位是 O2)，因此其行数是 3。

(2) 3 的补(模为 256)是 253，其对应的二进制码为 0xFD。

(3) 0xFD 左移 4 位后得 0xD0，键 9 的列码是“00001101B”，即 0x0D，把 0xD0 和 0x0D 相加后的 0xDD 即为键 9 的键码。

采用相同的方法可以求出键盘中其他键的键码，键盘接口程序按照键码产生的算法求出键码后，即可知是哪个键按下，并根据键码转向键对应的功能处理程序。

5.2.2　基于 PS GPIO 的键盘接口

前面已经提到，Zynq 芯片内部没有键盘接口控制器，扩展键盘接口可以有两种途径，一种是利用 PS 部分的 GPIO 引脚来扩展键盘接口，另一种是利用 AXI GPIO IP 核先在 PL 部分扩展出 GPIO 引脚，然后再利用这些 GPIO 引脚来扩展键盘接口。本小节先介绍基于 PS GPIO 的键盘接口设计，下一小节再介绍基于 IP 核扩展的键盘接口设计。

例 5-1　假设要求设计一个阵列为 4×4 的键盘，通过 Zynq 芯片内部 PS 部分的 GPIO 引脚，来设计键盘行信号和列信号的接口电路，并完成键盘扫描识别及生成键码的程序设计。

具体设计时，若选用 Zynq 芯片 GPIO 端口 1 中的引脚 MIO[50]～MIO[53]作为输入，用于连接“键盘列”，引脚 MIO[46]～MIO[49]作为输出，用于连接“键盘行”，键码采用 8 位，是行信号值和列信号值合并而成，且列信号在高 4 位。那么，具体的键盘扫描及生成键码程序可设计如下：

```
/* 注：采用 C 语言编写，未用到的寄存器位应保持其原有值不变 */
//*********************************************************************
//** 函数名：Scankey()，无参数
//** 返回值：键扫描码(高 4 位是列信号值，低 4 位是行信号值，键码是两者合并)
//** 功 能：调用一次此函数，可以实现对键盘一次全扫描
//*********************************************************************
//** keyoutput 是键盘扫描时的输出地址，keyinput 是键盘读入时的地址
//** 此处的键盘输出地址和读入地址均对应 GPIO 端口 1 的数据寄存器地址
#define KEYOUTPUT        (*(volatile U8 *)0xE000A044)
#define KEYINPUT         (*(volatile U8 *)0xE000A044)
//** 定义 GPIO 引脚的功能设置寄存器
#define rMIO_PIN_46      (*(volatile unsigned long *)0xF80007B8)
#define rMIO_PIN_47      (*(volatile unsigned long *)0xF80007BC)
#define rMIO_PIN_48      (*(volatile unsigned long *)0xF80007C0)
#define rMIO_PIN_49      (*(volatile unsigned long *)0xF80007C4)
#define rMIO_PIN_50      (*(volatile unsigned long *)0xF80007C8)
#define rMIO_PIN_51      (*(volatile unsigned long *)0xF80007CC)
```

```
#define rMIO_PIN_52          (*(volatile unsigned long *)0xF80007D0)
#define rMIO_PIN_53          (*(volatile unsigned long *)0xF80007D4)
//** 定义 GPIO 引脚的方向设置寄存器和输出使能寄存器
#define rDIRM_1              (*(volatile unsigned long *)0xE000A244)
#define rOEN_1               (*(volatile unsigned long *)0xE000A248)
U8 ScanKey()
{
    U8 key=0xFF;
    U32 i;
    U32 temp=0xFFFFFFFF,output;
    //** 初始化 GPIO 引脚的功能
    rMIO_PIN_46 = 0x00000600;          //设置 MIO[46]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_47 = 0x00000600;          //设置 MIO[47]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_48 = 0x00000600;          //设置 MIO[48]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_49 = 0x00000600;          //设置 MIO[49]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_50 = 0x00000600;          //设置 MIO[50]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_51 = 0x00000600;          //设置 MIO[51]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_52 = 0x00000600;          //设置 MIO[52]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_53 = 0x00000600;          //设置 MIO[53]引脚功能为 GPIO,引脚电压 3.3V

    //设置 GPIO 引脚的方向,MIO[46]~MIO[49]为输出,MIO[50]~MIO[53]为输入
    rDIRM_1 = (rDIRM_1 & 0xFFC3FFFF) | 0x0003C000;
    rOEN_1 = rOEN_1 | 0x0003C000;      //输出使能
    //** 循环往键盘(4×4)输出线送低电平,因为输出为 4 根,所以循环 4 次 **//
    for (i=0x4000;(( i<=0x20000) && (i>0)); i>>=1) {
    //** 将第 i 根输出引脚置低,其余输出引脚为高,即对键盘按行进行扫描 **//
        output |= 0xFFFFFFFF;
        output &= (~i);
        KEYOUTPUT = output;
    //** 读入此时的键盘输入值 **//
        temp = KEYINPUT;
    //** 判断 4 根输入线上是否有低电平出现,若有则说明有键输入,否则无 **//
        if ((temp&0x0003C000)!=0x0003C000) {
    //** 有键按下,将此时的 temp 右移 8 位,并和读入的值合并为 16 位键码 **//
        temp >>= 14;
        key 1=temp;
        return(key);
        }
    }
    //* 如果无键按 F
    return 0xFF;
}
```

上面 Scankey()函数仅完成了键盘扫描及键码的生成,但没有考虑键盘的消抖动问题。下面在 Scankey()函数的基础上,再封装一层函数,在该函数中进行了延时消抖动处理,从而可以获得稳定的键码。

```
//*********************************************************************
//** 函数名: getkey(),无参数
//** 返回值: 读取到的稳定键码
//** 功 能: 调用 scankey()识别按键,然后消抖动,得到可靠键码
//*********************************************************************/
U8 getkey(void)
{
```

```
U8    key,tempkey;
U8    oldkey=0xFF;
U8    keystatus=0;
U8    keycnt=0;
//** 等到有合法的、可靠的键码输入才返回;否则无穷等待 **//
while(1) {
        //** key 设置为 0xFF,初始状态为无键码输入 **//
        key = 0xFF;
        //** 等待键盘输入,若有输入则退出此循环进行处理;否则等待 **//
        while(1) {
        //** 扫描一次键盘,将读到的键值送入 key **//
        key = ScanKey() ;
        //** 判断是否有键输入,如果有则退到外循环进行消抖动处理 **//
        tempkey = key;
        if ((tempkey&0xFF) != 0xFF)     break;
//** 若没有键按下,则延迟一段时间后,继续扫描键盘,同时设 oldkey=0xFF **//
        mydelay(20,50);     //延时函数(延时函数读者可以自行编写)
        oldkey=0xFF;
    }
//** 在判断有键按下后,延迟一段时间,再扫描一次键盘,消抖动 **//
    mydelay(50,5000);       //延时约十几毫秒(延时函数读者可以自行编写)
    if (key != ScanKey())
        continue;
//** 如果连续两次读的键码一样,并不等于 oldkey,则可判断有新的键码输入 **//
    if (oldkey != key)  keystatus=0;
//** 设定 Oldkey 为新的键码,并退出循环,返回键码 **//
    oldkey = key;
    break;
}
return key;
```

获得稳定的键码值后,即可以根据键码值来判断是哪个按键被按下,然后将程序转移到对应按键的处理程序处执行。下面的程序段仅给出了按键处理程序的框架,具体按键功能的程序需要根据具体应用来编写。

```
//*********************************************************************
//** 函数名: main(),无参数,无返回值
//** 功 能: 主程序,完成读键码,并根据键码调用具体的按键功能程序
//*********************************************************************/
  void main(void)
  {
    U8 key=0;
    while(1) {
        mydelay(10,1000);           //延时
        //** 读取键码 **//
        key = getkey();
    //下面根据键码完成具体的按键功能程序
    //假设 MIO[46]~MIO[49]分别对应第 1~4 行
    //假设 MIO[50]~MIO[53]分别对应第 1~4 列
    switch(key) {
            case 0xee:              //0xee 是一个键码,对应第 1 行第 1 列的按键
            //根据该键码完成对应按键的具体功能
```

```
                    case 0xed:                    //0xed 是一个键码,对应第 2 行第 1 列的按键
                                                  //根据该键码完成对应按键的具体功能
                    ……
                    break;
                    }
            }
    }
```

5.2.3 基于 IP 核扩展的键盘接口

利用 AXI GPIO IP 核,可以在 Zynq 芯片的 PL 部分扩展出 GPIO 端口。所扩展出的 GPIO 端口,通过芯片内部的 AXI 互联总线,连接到 PS 部分的 APU(应用处理单元),这样就与 PS 部分的嵌入式系统紧密连接在一起,可以由 PS 部分的 CPU 编程控制 GPIO 端口的输入或输出。

一个扩展了 AXI GPIO IP 核的系统功能框图如图 5-4 所示。图中显示利用 AXI GPIO IP 核扩展的 GPIO 端口,是一个 AXI_GP 总线的从端口,它可以被 APU 通过 AXI 总线进行访问。

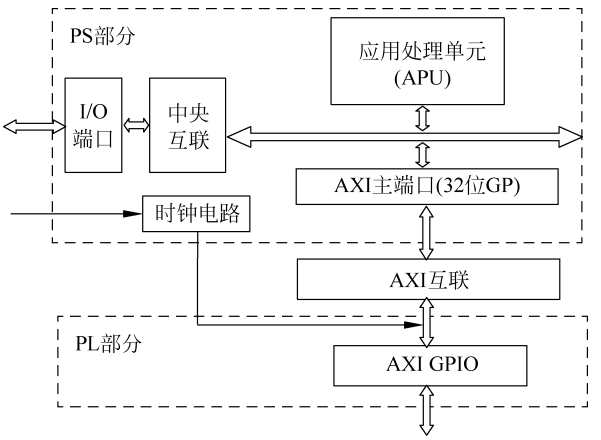


图 5-4 扩展了 AXI GPIO 端口的系统功能框图

在 Zynq 芯片的开发工具 Vivado 中,通过添加 IP Catalog 中的 AXI GPIO 实例来完成扩展 GPIO 端口,其设计步骤如下。

(1) 首先创建一个具有 AXI 互联的嵌入式系统新工程。在创建新工程时,需要配置 AXI 总线的参数,并配置 PS 部分的外设接口。

(2) 选择 IP Catalog 条目,并选中需要的 IP 核。本处需要选择 AXI General Purpose IO 的 IP 核,并配置其相关参数。例如,根据实际需求,配置其数据宽度、输入输出方向、默认的初始输出值、默认的初始输入状态值等。

(3) 完成扩展的 GPIO 端口与微处理器系统的连接,连接时将自动分配总线地址给 GPIO 端口,并使端口连接到外部。

(4) 编写新的约束文件或添加新的约束条件到约束文件中。

当完成了扩展的 GPIO 端口设计后,根据约束文件中的设定,选择作为输出功能的 PL 引脚连接键盘行信号,选择作为输入功能的 PL 引脚连接键盘列信号,从而设计出键盘阵列

的接口。然后根据分配的端口地址，按照键盘识别及键码生成方法来完成键盘的驱动程序设计。

5.3　LED 显示接口

LED 显示器是嵌入式系统中常用的输出设备，特别是 7 段(或 8 段)LED 显示器作为一种简单、经济的显示形式，在显示信息量不大的应用场合得到广泛应用。随着 LED 显示技术的发展，彩色点阵式 LED 显示技术越来越成熟，也已经得到许多应用，特别是在户外广告屏等领域中被广泛使用。

5.3.1　LED 显示控制原理

在嵌入式系统中，LED 显示器的形式主要有 3 种，即单个 LED 显示器、7 段(或 8 段)LED 显示器、点阵式 LED 显示器。

1. 单个 LED 显示控制原理

单个 LED 显示器实际上就是一个发光二极管，它的亮与灭代表着一个二进制数，因此，凡是能用一位二进制数代表的物理含义，如信号的有无、电源的通断、信号幅值是否超过其阈值等，均可以用单个 LED 显示器的亮与灭来表示。微处理器通过 GPIO 接口引脚中的某一位引脚来控制 LED 的亮与灭，如图 5-5 所示。图中引脚 D0 通过反相驱动器(也可以采用同相驱动器)控制一个单个 LED 显示器，D0 为“1”(高电平)时，单个 LED 显示器亮，代表一种状态；D0 为“0”(低电平)时，单个 LED 显示器灭，代表另一种状态。

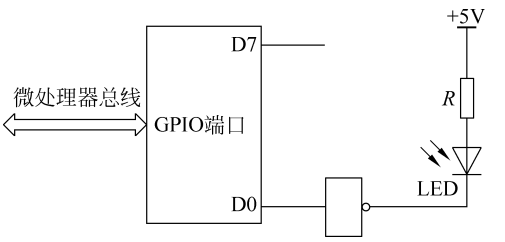


图 5-5　单个 LED 显示器控制原理

2. 7 段(或 8 段)LED 显示控制原理

7 段(或 8 段)LED 显示器是由 7 个(或 8 个)发光二极管按一定的位置排列成“日”字形(对于 8 段 LED 显示器来说还有一个小数点段)，为了适应不同的驱动电路，采用了共阴极和共阳极两种结构，如图 5-6 所示。

用 7 段(或 8 段)LED 显示器可以显示 0～9 的数字和多种字符(并可带小数点)，为了使 7 段(或 8 段)LED 显示器显示数字或字符，就必须点亮相应的段。例如，要显示数字 0，则要使 b、c、d、e、f、g 等 6 段亮。显示器的每个段分别由 GPIO 引脚进行控制，通常引脚的 D0～D7(即数据位的低位～高位)顺序控制 a～dp 段，所需的控制信号称为段码。由于数字与段码之间没有规律性，因此必须进行数字与段码之间的转换才能驱动要显示的段，以便显示数字的字形。常用的转换方法是将所要显示字形的段码列成一个表，称为段码表。显示时，根据字符查段码表，取出其对应的段码送到 GPIO 引脚上来控制显示。

值得注意的是，若采用的显示器其驱动结构不同，那么即使显示相同的字符，其段码也是不一样的。例如，若采用共阴极 7 段 LED 显示器，段信号采用同相驱动，则 0 的段码是 0x7E；若采用共阴极 7 段 LED 显示器，段信号采用反相驱动，则 0 的段码是 0x81。

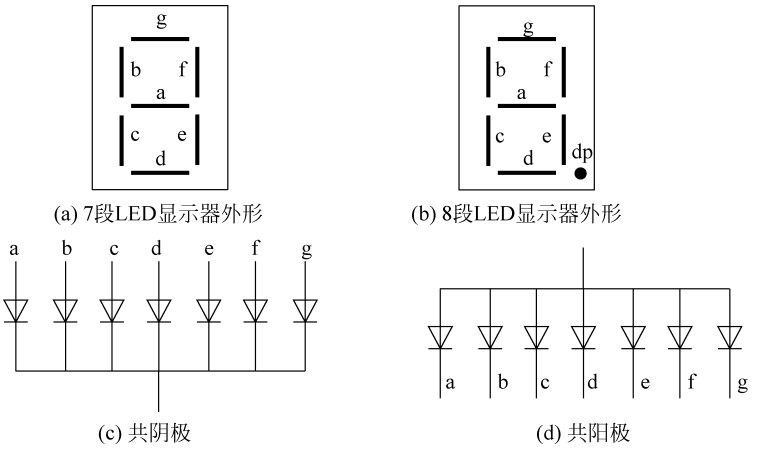


图 5-6　7 段(或 8 段)LED 显示器外形原理

在实际应用中,一般需要多位数据同时显示,这样就需要用多个 7 段(或 8 段)LED 来组成一个完整的显示器。图 5-7 是一个由 6 位 8 段 LED 组成的显示器接口电路,电路中采用了同相驱动、扫描显示方式。

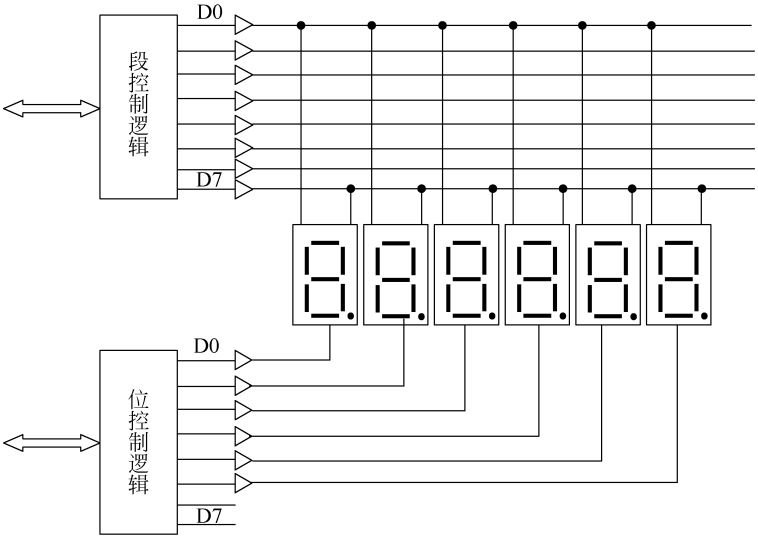


图 5-7　6 位 8 段 LED 显示器

扫描显示方式是根据人眼的视觉惰性,在多位 7 段(或 8 段)LED 组成的显示器中,所有位的段信号均连接在一起,由段控制逻辑控制,而该位能不能显示则由位控制逻辑中对应的位信号控制。位控制逻辑实际上是一扫描电路,它依次使 N 位 7 段(或 8 段)LED 显示器中的一位显示,其余位处于不显示状态。只要扫描的速度适当,人眼看到的是 N 位 LED 同时显示的状况。

例如,若要在图 5-7 所示的显示器中显示“12.08.18”字样,其典型的做法如下。

① 在数据存储区中选择 6 个存储单元作为显示缓冲区,存储单元地址从低到高依次对应显示器中从左至右的 8 段 LED,所需显示字符的段码存入对应的单元中。即显示缓冲区

中地址从低到高分别存有 0x30、0x6D、0x7E、0x7F、0x30、0x7F。

② 显示时，从低地址到高地址依次把显示缓冲区的内容通过段控制逻辑输出，同时位控制逻辑输出的位控制信号依次是 0x3E、0x3D、0x3B、0x37、0x2F、0x1F。

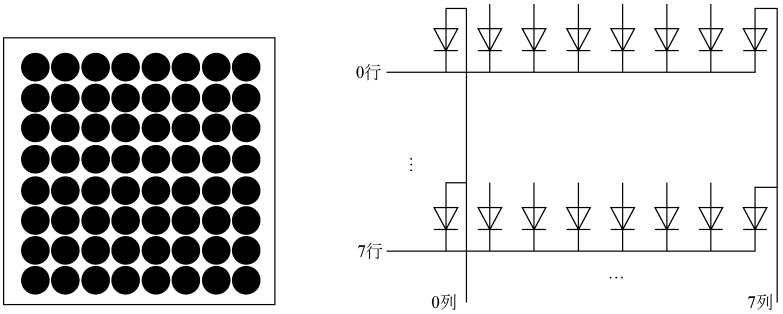
③ 循环进行上述动作，只要循环间隔适当，人眼在显示器上看到的是稳定的显示，而不会有跳动的感觉。

④ 若要改变显示器上显示的内容，只需改变存储在显示缓冲区中的段码即可。

在设计 7 段(或 8 段)LED 显示器接口电路时，还可以用专用的 8 段显示器控制芯片(如 ZLG7289AS 芯片)，这些芯片可以完成数字到段码的转换，并能控制扫描，因而不需要微处理器执行扫描程序控制显示，从而提高了微处理器的效率。

3. 点阵式 LED 显示控制原理

点阵式 LED 显示器的基本显示单元一般是 8 行×8 列的 LED 模块，如图 5-8 所示。通过若干块 8 行×8 列的 LED 模块，可以拼成更大的 LED 显示阵列。例如，若要设计一个分辨率为 800×600 的点阵式 LED 显示器，那么需要在行上选用 100 个 8 行×8 列的 LED 模块，在列上选用 75 个 8 行×8 列的 LED 模块。即需要选用 100×75 个 8 行×8 列的 LED 模块，从而组成 800×600 分辨率的 LED 显示阵列。



(a) 点阵式LED外形　　(b) 点阵式LED内部连接

图 5-8　点阵式 LED 外形及内部连接

点阵式 LED 显示器能显示各种字符、汉字及图形、图像，并具有色彩。点阵式 LED 显示器中，每个 LED 表示一个像素，通过每个 LED 的亮与灭来构造出所需的图形，各种字符及汉字也是通过图形方式来显示的。对于单色点阵式 LED 来说，每个像素需要一位二进制数表示，“1”表示亮，“0”表示灭。对于彩色点阵式 LED，则每个像素需要更多的二进制位表示，通常用 1 字节或 3 字节。

点阵式 LED 显示器的显示控制也采用扫描方式。在数据存储器中需开辟若干存储单元作为显示缓冲区，缓冲区中存有所需显示图形的控制信息。显示时依次通过列信号驱动器输出一行所有列的信号，然后再驱动对应的行信号，控制该行显示。只要扫描速度适当，显示的图形就不会出现闪烁。

5.3.2　基于 PS GPIO 的 LED 接口

在开发基于 Zynq 芯片的嵌入式系统时，可以利用 PS 部分的 GPIO 引脚来扩展 LED 接口。下面以 8 段的 LED 显示器接口设计为例，来说明基于 Zynq 芯片的嵌入式系统 LED

显示器接口设计方法。

例 5-2 假设需要设计一个由 6 个 8 段的 LED 组成的显示器，采用了 ZLG7289AS 芯片控制，并用 Zynq 芯片的 GPIO 端口来与其连接。ZLG7289AS 芯片是一个具有串行输入、8 位段信号并行输出，可同时驱动 8 个共阴 LED 的专用显示器控制芯片。该芯片能支持译码显示模式和非译码显示模式。译码显示模式指的是微处理器输出给 ZLG7289AS 芯片显示字符的对应值，由芯片 ZLG7289AS 译码产生显示需要的段信号；而非译码显示模式指的是微处理器直接输出给 ZLG7289AS 芯片显示字符对应的段码信号。因此，采用非译码显示模式时，设计者需要自行求出显示字符对应的段码。有关 ZLG7289AS 芯片的详细命令可参考其技术手册。

下面的程序代码是基于 ZLG7289AS 芯片来控制 LED 显示器程序的。显示器是共阴极 LED，其段的排列顺序如图 5-6(b)所示，采用了非译码显示模式控制。

```
/* 注:采用 C 语言编写,未用到的寄存器位应保持其原有值不变 */
//** 定义 GPIO 引脚的功能设置寄存器
#define rMIO_PIN_48        (*(volatile unsigned long *)0xF80007C0)
#define rMIO_PIN_49        (*(volatile unsigned long *)0xF80007C4)
#define rMIO_PIN_50        (*(volatile unsigned long *)0xF80007C8)
//** 定义 GPIO 引脚的方向设置寄存器和输出使能寄存器
#define rDIRM_1            (*(volatile unsigned long *)0xE000A244)
#define rOEN_1             (*(volatile unsigned long *)0xE000A248)
//** 定义 GPIO 端口 1 的数据寄存器(输出)
#define rDATA_1            (*(volatile U8 *)0xE000A044)
//** 定义了一些宏,包括 cs_disable、cs_enable、setdata_1、setdata_0、setclock_1、setclock_0
//** 假设选用 MIO[48]~MIO[50]来控制 ZLG7289AS 芯片
//** 它们对应了 ZLG7289AS 芯片的片选信号以及数据、时钟信号等,具体参见其技术手册
#define cs_enable {rDATA_1 = rDATA_1 | 0x10000;}          //MIO[48]置 1,即片选置 1
#define cs_disable {rDATA_1 = rDATA_1 & 0xFEFFFF;}        //MIO[48]置 0,即片选置 0
#define setdata_1 {rDATA_1 = rDATA_1 | 0x20000;}          //MIO[49]置 1
#define setdata_0 {rDATA_1 = rDATA_1 & 0xFDFFFF;}         //MIO[49]置 0
#define setclock_1 {rDATA_1 = rDATA_1 | 0x40000;}         //MIO[50]置 1
#define setclock_0 {rDATA_1 = rDATA_1 & 0xFBFFFF;}        //MIO[50]置 0
//** 下面数组用来映射 LED 模块非译码时,显示字符 0~9 和其段码的对应关系
char mapda[10]={0x7e,0x30,0x6d,0x79,0x33,0x5b,0x5f,0x70,0x7f,0x7b};
//*********************************************************************
//** 函数名: main(),功能是: 用 ZLG7289AS 控制的 LED 显示,无参数,无返回值
//*********************************************************************/
    void main(void)
    {
    int i,lednum=6;
    U8 boardtype;
    ledinit();                          //** 调函数,初始化 GPIO 端口和 ZLG7289AS
    //** 发送命令字,清除所有显示
    sendledcmd(0xa4);
    mydelay(10,1000);                   //延时函数
    i=0;
    //** 发送测试命令字,使所有段和点闪烁
    sendledcmd(0xbf);
    mydelay(10,1000);
```

```
    sendledcmd(0xa4);
    for(;;)  {
            //在相应位置的 LED 上显示 0～9 字符
            sendleddata(i%lednum , mapda[i%10]);
            mydelay(1000,1000);
            i++;
            if(i==40) i=0;
            sendledcmd(0xa4);
        }
}
```

在主函数 main()中，调用了函数 ledinit()来初始化 8 段 LED 接口。即需要把 MIO[48]～MIO[50]初始化为 GPIO 输出功能，并对 ZLG7289AS 芯片的控制引脚信号进行初始化。

```
//*************************************************************************
//** 函数名: ledinit(),无参数,无返回值
//** 功 能: 初始化 GPIO 端口以及 ZLG7289AS 芯片
//*************************************************************************/
void ledinit(void){
    //** 将 MIO[48]～MIO[50]设置为 GPIO 输出工作方式,同时设定不需要上拉电阻
    rMIO_PIN_48 = 0x00000600;          //设置 MIO[48]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_49 = 0x00000600;          //设置 MIO[49]引脚功能为 GPIO,引脚电压 3.3V
    rMIO_PIN_50 = 0x00000600;          //设置 MIO[50]引脚功能为 GPIO,引脚电压 3.3V
    //设置 GPIO 引脚的方向,MIO[48]～MIO[50]为输出
    rDIRM_1 = rDIRM_1 | 0x00070000;
    rOEN_1 = rOEN_1 | 0x00070000;     //输出使能
    //** 使片选信号不使能(即失效),且设定数据和时钟线均为高,初始化 LED
    cs_disable;
    setdata_1;
    setclock_1;
    mydelay(10,1000);                  //调延时函数,延时
}
```

ZLG7289AS 芯片是可以独立控制 LED 显示的，但需要微处理器给其发送相关的命令，其命令格式有单字节和双字节之分。下面两个函数分别是发送单字节的命令和发送双字节的命令。

```
//*************************************************************************
//** 函数名: sendledcmd(),功能是: 传送单字节命令到 ZLG7289AS,无返回值
//** 参 数: ZLG7289AS 命令字,参考 ZLG7289AS 资料
//*************************************************************************/
void sendledcmd(char context) {
     int count;
     //** 将时钟线和数据线均设置为低
     setclock_0;
     mydelay(20,10);                   //延时
     setdata_0;
     mydelay(20,10);                   //延时
     //** 使能片选
     cs_enable;
     mydelay(35,10);                   //延时
```

```
    //** 传送 8 比特,由高位到低位
    for (count=0x80;count>0;count>>=1) {
        //** 如果此位为 1,则数据线送 1;否则送 0
        if(context &count) {
              setdata_1;
        }
        else {
              setdata_0;
        }
        mydelay(3,10);                    //延时
    //** 时钟翻转为高,等待 ZLG7289AS 取数据
        setclock_1;
        mydelay(3,10);                    //延时
    //** 时钟翻转为低,ZLG7289AS 取数据结束
        setclock_0;
        mydelay(3,10);                    //延时
    }
    //** 一字节传送完毕后,使片选不使能(即失效)
    cs_disable;
    mydelay(3,10);                        //延时
  }
//*************************************************************************
//** 函数名: sendleddata(),功能是传送双字节命令到 ZLG7289AS,无返回值
//** 参 数: LED 显示位置序号,ZLG7289AS 命令字。参考 ZLG7289AS 资料
//*************************************************************************/
  void sendleddata(char i,char context)
  {
    char a[2];
    int count,k;
    //** 将时钟线和数据线均设置为低
    setclock_0;
    mydelay(3,100);                       //延时
    setdata_0;
    a[0]=0x90+i;
    a[1]=context;
    mydelay(3,10);                        //延时
    cs_enable;
    mydelay(3,10);                        //延时
    for (k=0;k<2;k++){
        for (count=0x80;count>0;count>>=1) {
              //发送一字节
              if(a[k] &count) {
                    setdata_1;
               }
               else {
                    setdata_0;
               }
               mydelay(3,10);             //延时
               setclock_1;                //设置时钟信号为高
               mydelay(3,10);             //延时
               setclock_0;                //设置时钟信号为低
```

```
                mydelay(4,10);          //延时
            }
        mydelay(3,10);                  //延时
    }
    cs_disable ;                        //使片选信号不使能
    mydelay(3,10);                      //延时
}
```

除了利用 PS 部分的 GPIO 来扩展 LED 显示接口外，也可以利用 AXI GPIO IP 核，在 Zynq 芯片的 PL 部分扩展出 GPIO 端口。然后利用扩展的 GPIO 端口来设计 LED 接口，其设计方法与 5.2.3 小节介绍的类似，在此就不再介绍。

5.4　OLED 显示接口

用 OLED(Organic Light-Emitting Diode，有机发光二极管)材料做成的显示器，具有自发光(即不需要背光源)、视角广、对比度高、低功耗等特点，已经广泛地应用于高端嵌入式系统中作为其显示设备，如应用于手机、数码相机、PDA 等产品中。

5.4.1　OLED 工作原理简介

OLED 的发光原理是通过把一个有机发光材料组成的发光层，嵌入两个电极之间，然后通过在两个电极端加上电源，使电流通过有机发光材料而使其发光。通常在构建 OLED 时，还会在电极和发光层之间各加上一层传输层，以便提高发光效率。一个典型的 OLED 结构如图 5-9 所示。

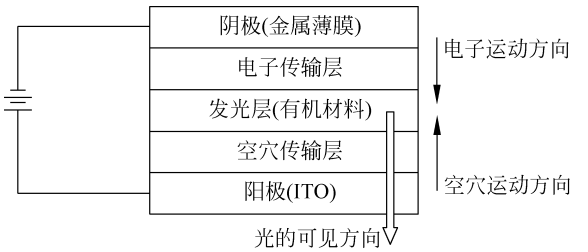


图 5-9　一个典型的 OLED 结构

图 5-9 所示的 OLED 结构中，阴极是一种金属薄膜，阳极是一种导电玻璃(其材料是铟/锡/锡氧化物，简称 ITO)。在阴极和有机发光材料层之间有一层电子传输层，而在阳极和有机发光材料层之间有一层空穴传输层。

1. OLED 显示原理

如图 5-9 所示，当直流电源(电压值为 2～10V)接通时，在阴极(金属薄膜)上将产生电子，而在阳极(ITO)上产生空穴。在电场的作用下，电子通过电子传输层而到达发光层(由有机发光材料组成)，同时空穴通过空穴传输层也到达发光层，由于电子带负电，空穴带正电，因此，它们在发光层相互吸引，从而激发有机材料发光。由于阳极(ITO)是透明的，因此可以在阳极端看见光。

OLED的发光亮度与发光层通过的电流有关，电流越大，亮度越大；电流越小，亮度越小。因此，通过控制电流的大小，就可以控制OLED的发光亮度。而光的颜色与发光层的发光材料有关，通过材料的不同配比，可以产生红(R)、绿(G)、蓝(B)3种基本颜色。利用R、G、B像素独立发光或混合发光，可以构造彩色的OLED显示器。在生产彩色OLED显示器的方法上，还有一种生产工艺，即可以采用白光OLED和彩色滤光片相结合的方式，通过在发白光的OLED上覆盖红、绿、蓝三基色的滤光片，从而实现红、绿、蓝及其他混合色彩的显示。

要实现 $n \times m$ 分辨率的彩色OLED显示器，就需要有 n 列 m 行的OLED像素阵列来组成，即一行上有 n 个像素、一列上有 m 个像素，每个像素中又包括R、G、B三基色，其组成示意图如图5-10所示。

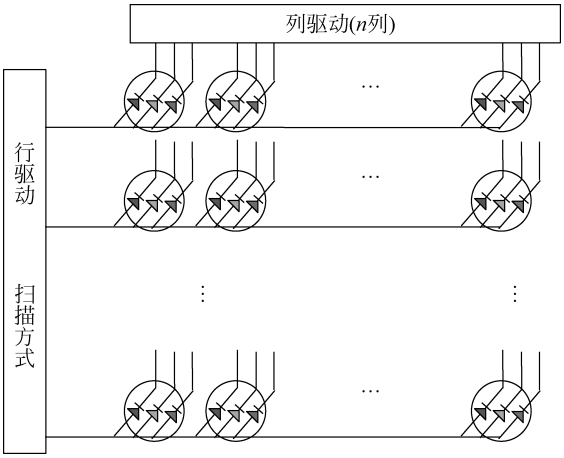
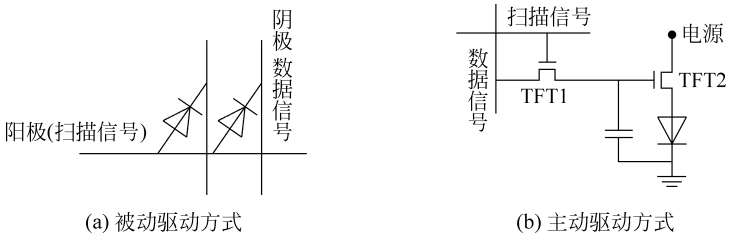


图5-10 $n \times m$ 分辨率的彩色OLED像素阵列

当需要在OLED显示器上显示文字、图像等信息时，就需要驱动相关的OLED被点亮。与普通点阵式LED显示驱动类似，也需要列驱动和行驱动电路，并且行驱动是扫描方式，即一次只点亮一行上的相关OLED。

2. OLED驱动控制

OLED的驱动方式分为被动驱动(Passive Matrix OLED，PMOLED)和主动驱动(Active Matrix OLED，AMOLED)两种方式，被动驱动方式又可称为无源驱动方式，主动驱动又可称为有源驱动方式。这两种OLED驱动方式的内部结构如图5-11所示。



(a) 被动驱动方式　　(b) 主动驱动方式

图5-11 OLED两种驱动方式内部结构

被动驱动方式如图5-11(a)所示，是在阳极和阴极之间设置一个OLED作为一个像素。若要让这个像素点亮，在阴极上接低电平、阳极上接高电平即可。在一个 $n \times m$ 分辨率的

OLED 显示器中，即有 $n \times m$ 个这样排列的 OLED 像素。要在这种驱动方式下显示一幅 $n \times m$ 分辨率的图像，通常采用在 m 个阳极上加扫描信号，即在 m 个阳极上逐次加正脉冲信号，在 n 个阴极上加图像数据信号(需要点亮的像素数据为 0，不亮的像素数据为 1)，只要阳极上的扫描信号足够快，人眼就能看到一幅完整的图像。

被动驱动方式由于是无源的，因此，其结构简单，制造成本低；但其驱动电流高，反应速度相对较低。被动驱动方式只适合于单色或多色且尺寸相对较小的 OLED 显示器。

主动驱动方式如图 5-11(b)所示，是一种有源的驱动方式，它采用至少两个薄膜晶体管(Thin Film Transistor，TFT)来控制一个 OLED 像素的驱动，并在 OLED 像素的控制端接有一个电容。也就是 OLED 的像素是否被点亮，是采用了具有开/关功能的 TFT 来控制，每个像素可以独立地连续发光，即图像数据信号(需要点亮的像素数据为 1，不亮的像素数据为 0)在扫描信号的高电平期间，可以驱动 TFT2 的开或关，从而使得 OLED 像素点亮或者关灭；在扫描信号的低电平期间，还可以由电容来保持 TFT2 所需要的控制电压，因此也能保持像素的状态。此处的扫描信号是一个固定周期的脉冲信号。

主动驱动方式由于在 OLED 像素的控制端接有电容，在两次扫描信号脉冲之间可以保持充电状态，因而可以更快速、更精确地控制 OLED 像素发光。并且其驱动电压低，组件寿命更长，适合做大尺寸的 OLED 显示器。但其制作工艺相对复杂，因而成本较高。

目前在市场上，OLED 显示屏有单色、多色、全彩色等几种。单色和多色显示屏多采用被动驱动方式，全彩色显示屏多采用主动驱动方式。在嵌入式系统设计中，通常都选用 OLED 显示模组，即由 OLED 显示屏和驱动芯片组合在一起的模块。市场上有许多可供选择的 OLED 显示模组产品，如单色或多色的有 UG-2832HSWEG04 OLED 模组(分辨率为 128×32，驱动芯片为 SSD1306)、DYS864 OLED 模组(分辨率为 128×64，驱动芯片为 SH1106)等。

OLED 显示模组与微处理器的接口主要是驱动芯片与微处理器的连接，它们之间的命令及数据传输通常可采用 SPI、I^2C 等，因此，在嵌入式系统设计时，OLED 显示模组的接口电路并不复杂，而复杂的是结合驱动芯片的特性，来完成相关的 OLED 驱动程序编写。

5.4.2　基于 PS GPIO 的 OLED 接口

在开发基于 Zynq 芯片的嵌入式系统中，可以利用 PS 部分的 GPIO 引脚来扩展 OLED 显示接口；也可以自己设计一个 OLED 的驱动控制 IP 核，利用 PL 部分的引脚来扩展 OLED 显示接口。下面以 PS GPIO 的 OLED 显示接口设计为例来说明其接口设计的方法。

例 5-3　假设需要设计一个 OLED 显示器接口，选用了 UG-2832HSWEG04 OLED 模组，该模组的驱动控制芯片为 SSD1306，它与微处理器的连接为 SPI 接口，主要的信号线有以下几个。

① SCLK 信号线，是 SPI 的时钟信号。

② SDIN 信号线，是 SPI 的串行数据线，即 MOSI 信号线。

③ DC 信号线，它是 UG-2832HSWEG04 OLED 模组的命令/数据指示信号线，该信号线为高电平时，指示读写的是数据；该信号线为低电平时，指示读写的是命令。

除了信号线外，UG-2832HSWEG04 OLED 模组还需要一个复位信号(RST)和逻辑电源 U_{DD}，以及升压电源 U_{BAT}(DC/DC 转换)。

在硬件电路设计时，若选用 Zynq 芯片 GPIO 的相关引脚，来与 UG-2832HSWEG04 OLED 模组相关引脚连接，如图 5-12 所示。其中，用 MIO[4]和 MIO[5]引脚分别与 SDIN 和

SCLK 信号相连，并且选用 MIO[7]引脚来控制 DC 信号，MIO[6]引脚来控制 RST 信号。

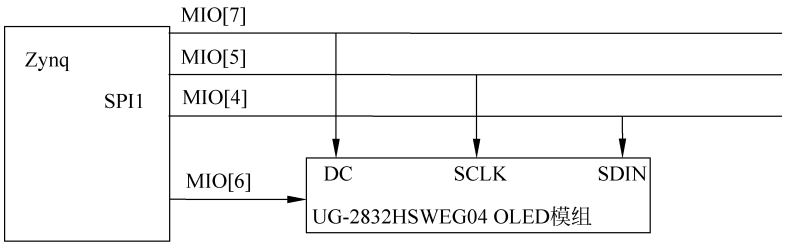


图 5-12 Zynq 芯片与 UG-2832HSWEG04 OLED 模组的连接

设计好 OLED 显示接口电路后，就需要编写相关的驱动程序。下面给出的 OLED 驱动程序示例，主要包括初始化 Zynq 芯片引脚功能函数、初始化 OLED 模组、写 OLED 模组命令函数、写 OLED 模组数据函数以及一个主函数的框架。

```
/* 注：采用 C 语言编写，未用到的寄存器位应保持其原有值不变 */
/*************************************************************************
功能：Zynq 芯片引脚功能初始化函数，完成 MIO 引脚功能设置，初始化其工作模式
*************************************************************************/
//** 定义 GPIO 引脚的功能设置寄存器
#define rMIO_PIN_7   (*(volatile unsigned long *)0xF800071C)
#define rMIO_PIN_6   (*(volatile unsigned long *)0xF8000718)
#define rMIO_PIN_4   (*(volatile unsigned long *)0xF8000710)
#define rMIO_PIN_5   (*(volatile unsigned long *)0xF8000714)
//** 定义 GPIO 引脚的方向设置寄存器和输出使能寄存器
#define rDIRM_0      (*(volatile unsigned long *)0xE000A204)
#define rOEN_0       (*(volatile unsigned long *)0xE000A208)
//** 定义 GPIO 端口 0 的数据寄存器(输出)
#define rDATA_0      (*(volatile unsigned long *)0xE000A040)
//** 定义了一些宏，包括 setDC_1、setDC_0、setRST_1、setRST_0
//** 以及 setSDIN_1、setSDIN_0、setSCLK_1、setSCLK_0
#define setDC_1     {rDATA_0 = rDATA_0 | 0x00000080;}     //MIO[7]置 1，即 DC 信号置 1
#define setDC_0     {rDATA_0 = rDATA_0 & 0xFFFFFF7F;}  //MIO[7]置 0，即 DC 信号置 0
#define setRST_1    {rDATA_0 = rDATA_0 | 0x00000040;}     //MIO[6]置 1，即 RST 信号置 1
#define setRST_0    {rDATA_0 = rDATA_0 & 0xFFFFFFBF;}  //MIO[6]置 0，即 RST 信号置 0
#define setSDIN_1   {rDATA_0 = rDATA_0 | 0x00000010;}     //MIO[4]置 1，即 DC 信号置 1
#define setSDIN_0   {rDATA_0 = rDATA_0 & 0xFFFFFFEF;}  //MIO[4]置 0，即 DC 信号置 0
#define setSCLK_1   {rDATA_0 = rDATA_0 | 0x00000020;}     //MIO[5]置 1，即 RST 信号置 1
#define setSCLK_0   {rDATA_0 = rDATA_0 & 0xFFFFFFDF;}  //MIO[5]置 0，即 RST 信号置 0

void Zynq_Init(void)
{
    rMIO_PIN_7 = 0x00000600;          //设置 MIO[7]引脚功能 GPIO，引脚电压 3.3V
    rMIO_PIN_6 = 0x00000600;          //设置 MIO[6]引脚功能 GPIO，引脚电压 3.3V
    rMIO_PIN_4 = 0x00000600;          //设置 MIO[7]引脚功能 GPIO，引脚电压 3.3V
    rMIO_PIN_5 = 0x00000600;          //设置 MIO[6]引脚功能 GPIO，引脚电压 3.3V
    //设置 GPIO 引脚的方向，MIO[4]～MIO[7]为输出
    rDIRM_0 = rDIRM_0 | 0x000000F0;
    rOEN_0 = rOEN_0 | 0x000000F0;     //输出使能
}
/* 注：采用 C 语言编写 */
/*************************************************************************
```

```
功能：初始化 OLED 模组函数，模组芯片为 SSD1306，初始化其工作模式
*************************************************************************/
void SSD1306_Init(void)
{
    //** 先对 SSD1306 芯片进行复位设置
    setRST_0;                          //RST 引脚置 0
    //** 此处需插入一段时间的延时(约 10ms)，延时函数可自行编写
    setRST_1;                          //RST 引脚置 1

    //** 下面按要求写入相关命令，以便初始化 OLED 模组，命令含义参见 SSD1306 手册
    OLED_cmd_send (0xA8);
    OLED_cmd_send (0x3F);
    OLED_cmd_send (0xD3);
    OLED_cmd_send (0x00);
    OLED_cmd_send (0x40);
    OLED_cmd_send (0xA0);
    OLED_cmd_send (0xC8);
    OLED_cmd_send (0xDA);
    OLED_cmd_send (0x02);
    OLED_cmd_send (0x81);
    OLED_cmd_send (0x7F);
    OLED_cmd_send (0xA4);
    OLED_cmd_send (0xA6);
    OLED_cmd_send (0xD5);
    OLED_cmd_send (0x80);
    OLED_cmd_send (0x8D);
    OLED_cmd_send (0x14);
    OLED_cmd_send (0xAF);
}
/* 注：采用 C 语言编写 */
/*************************************************************************
功能：写 OLED 模组命令函数，模组芯片为 SSD1306
*************************************************************************/
void OLED_cmd_send(U8 OLED_CMD)
{
    U8 i;                              //定义变量 i
    setDC_0;                           //** 先对 SSD1306 芯片的 DC 信号置 0：写命令
    //下面进行循环，把一字节的命令发送到 OLED 模组
    for ( i=0; i<8; i++)
    {
        setSCLK_0;                     //设置 SCLK 信号为 0
        if (OLED_CMD & 0x80 ==1)       //判断命令字节的相应位是否为 1
            setSDIN_1;                 //SDIN 信号置 1
        else
            setSDIN_0;                 //SDIN 信号置 0
        //** 若此处需要，可插入延时函数，延时时间根据实际确定，大约几微秒
        setSCLK_1;                     //设置 SCLK 信号为 1
        OLED_CMD <<= 1;                //左移一位
    }
}
/* 注：采用 C 语言编写 */
/*************************************************************************
功能：写 OLED 模组数据函数，模组芯片为 SSD1306
```

```
*****************************************************************************/
void OLED_data_send(U8 OLED_DATA)
{
    U8 i;                                   //定义变量 i
    setDC_1;                                //** 先对 SSD1306 芯片的 DC 信号置 0: 写数据
    //下面进行循环,把一字节的数据发送到 OLED 模组
    for ( i=0; i<8; i++)
    {
        setSCLK_0;                          //设置 SCLK 信号为 0
        if (OLED_DATA & 0x80 ==1)           //判断数据字节的相应位是否为 1
            setSDIN_1;                      //SDIN 信号置 1
        else
            setSDIN_0;                      //SDIN 信号置 0
        //** 若此处需要,可插入延时函数,延时时间根据实际确定,大约几微秒
        setSCLK_1;                          //设置 SCLK 信号为 1
        OLED_DATA <<= 1;                    //左移一位
    }
}
/* 注: 采用 C 语言编写 */
/*****************************************************************************
功能: 主函数 main 的一个框架.假设显示图像数据已存入显示缓冲区
*****************************************************************************/
void main(void)
{
    U8OLED_ARR[128] [4];                    //显示缓冲数组
    U8i, j;
    Zynq_Init();                            //** 初始化 Zynq 芯片的 GPIO 引脚功能
    SSD1306_Init();                         //** 初始化 OLED 模组
    //** 下面应根据需要显示的内容,来控制 OLED 模组的显示
    //** 具体实现时,可根据 OLED 的分辨率来定义一个显示缓冲数组,如此处 128×32
    //** 然后,根据需要显示的图像,更新显示缓冲数据,再刷新 OLED 显示
    …… 
    for( i=0; i<4; i++){
        OLED_cmd_send(0xb0+i);              //设置页地址
        OLED_cmd_send(0x00);                //设置显示位置
        for (j=0; j<128; j++)OLED_data_send(OLED_ARR[j] [i]);          //发送显示数据
    }
}
```

本章小结

人机接口是人与嵌入式系统进行信息交互的通道。常用的人机接口设备有键盘、LED显示器、LCD显示器以及触摸屏、鼠标、打印机等。本章主要介绍了键盘、LED显示器、OLED显示器等人机接口设备的工作原理及其接口驱动程序。

键盘是最常用的人机输入设备。嵌入式系统中,键盘通常都采用非编码式键盘,设计者除了要完成键盘接口硬件电路的设计外,还需要完成键盘的扫描、键码的生成等接口软件设计。键盘接口程序中的键码生成方法有许多,但无论采用哪种方法,均必须保证键码与键的一一对应关系。

显示器是最常用的人机输出设备。嵌入式系统中常用的显示器主要有 LED 显示器和 OLED 显示器。它们各有优、缺点，适用于不同的场合。本章详细介绍了它们的接口电路及接口软件。

习　题　5

1. 选择题

(1) 下面语句中错误的是(　　)。

A. 人机接口是人与嵌入式系统进行信息交互的接口，该接口仅指键盘和显示器的接口

B. Zynq 芯片内部没有集成键盘及显示器接口的控制部件

C. 可以利用 Zynq 芯片内部 PS 部分的 GPIO 来扩展键盘或显示器的接口

D. 可以利用 FPGA 逻辑及成熟的 IP 核来扩展键盘或显示器的接口

(2) 下面有关 IP 核的描述语句中，错误的是(　　)。

A. IP 核是指一种基于 FPGA 逻辑上预先正确设计好的，能完成特定功能的硬件模块

B. 采用 IP 核的优点是可以缩短硬件的开发周期，提高开发的有效性和安全性

C. IP 核必须由 Xilinx 公司或者其他第三方公司提供

D. IP 核按提供给用户的体现形式，可分成软核、固核、硬核等 3 种形式

(3) Zynq 芯片内部即集成许多 IP 核，如 Cortex-A9 微处理器核、UART 接口的 IP 核、CAN 总线接口 IP 核等。其中，Cortex-A9 微处理器核是一种(　　)的形式。

A. 软核　　B. 硬核　　C. 固核　　D. 算法 IP 核

(4) 若一个键盘接口的键码由 8 位二进制数组成，且其键码生成的算法采用行数的补左移 4 位加列码，那么第 3 行第 2 列上的按键键码是(　　)。

A. 0x32　　B. 0x23　　C. 0xDD　　D. 0xDB

(5) 若一个 4×4 键盘的接口电路中，采用 Zynq 芯片 GPIO 端口 1 中引脚 MIO[46]～MIO[49]来控制键盘阵列的行信号，MIO[50]～MIO[53]控制键盘阵列的列信号，那么在键盘驱动程序中需要初始化 GPIO 端口 1 的相关寄存器。下面是初始化引脚 MIO[46]～MIO[53]方向的语句，正确的是(　　)。

A. rDIRM_1 = (rDIRM_1 & 0xFFC3FFFF) | 0x0003C000

B. rDIRM_1 = (rDIRM_1 & 0x0003C000) | 0xFFC3FFFF

C. rDIRM_1 = (rDIRM_1 & 0xFF3CFFFF) | 0x000C3000

D. rDIRM_1 = (rDIRM_1 & 0x000C3000) | 0xFF3CFFFF

(6) 若一个 6×6 键盘的接口电路中，采用键盘逐行扫描，然后读取列信号判断是否为全“1”的方法来识别是否有键按下。键盘接口程序中，应该用语句(　　)来正确地控制逐行扫描的次数。(注：变量 i 的值求反后作为行扫描信号输出)

A. for (i=1; ((i<=16)&&(i>0)); i<<=1{…}

B. for (i=1; ((i<=32)&&(i>0)); i<<=1{…}

C. for (i=1; ((i<=64)&&(i>0)); i<<=1{…}

D. for (i=1; ((i<=128)&&(i>0)); i<<=1{…}

(7) 若某嵌入式系统中采用 7 段共阴极 LED 来构成其显示器,微处理器通过同相驱动电路输出段信号来驱动 LED,下面的段码中(　　)不可能是数字字符的显示段码。

A. 0x01　　B. 0x30　　C. 0x7E　　D. 0x70

(8) 点阵式 LED 显示器接口电路的设计中,下面有关其设计要点的说明语句中不正确的是(　　)。

A. 一个大型 LED 显示屏通常是由若干 8×8 点阵的 LED 模块拼接而成

B. 点阵式 LED 显示屏分成行和列分别进行控制

C. 点阵式 LED 显示屏的列信号通常需要锁存

D. 点阵式 LED 显示屏中的所有行必须保证同时被选中控制

(9) 下面有关 OLED 显示器的描述语句中,错误的是(　　)。

A. OLED 是有机发光二极管组成的显示器,其字符或图形的显示需要背光源

B. OLED 的发光亮度,与发光层通过的电流有关,电流越大,亮度越大

C. 实现 $n\times m$ 分辨率的 OLED 显示器,需要有 n 列 m 行的 OLED 像素阵列来组成

D. OLED 的驱动方式分为被动驱动和主动驱动两种方式

(10) 若选用了 UG-2832HSWEG04 OLED 模组作为以 Zynq 芯片为核心的嵌入式系统 OLED 显示器,在硬件设计时,下面的做法错误的是(　　)。

A. 选用 Zynq 芯片 PS 部分的 MIO 引脚并设置成 GPIO 功能来控制该模组

B. 选用 Zynq 芯片 PL 部分的引脚并设计驱动控制 IP 核来控制该模组

C. 选用 Zynq 芯片 PS 部分的 MIO 引脚并设置成 SPI 功能来控制该模组

D. 选用 Zynq 芯片 PS 部分的 MIO 引脚并设置成 UART 功能来控制该模组

2. 填空题

(1) 键盘接口中,产生键码的算法有许多种,但无论采用何种算法来生成键码,都必须保证一一对应。

(2) 若键码采用 16 位二进制数组成,其键码生成算法为:行扫描信号左移 8 位再加上读入的列信号。那么,第 3 行第 2 列上的按键,其键码是_____。

(3) 用多个 8 段的 LED 组成的显示器接口电路中,若每个 LED 段信号线均连接在一起,微处理器分时输出每位 LED 段信号来控制显示,但人们看到的是所有 LED 同时在显示,这是依据了_____原理。

(4) 点阵式 LED 显示器中,每个 LED 表示_____,通过每个 LED 的亮与灭来构造出所需的图形,各种字符及汉字也是通过图形方式来显示的。

(5) 图 5-13 是一个共阴极的 7 段 LED 外形,其内部 LED 管的排列顺序如图所示,通常硬件设计时用数据线 D0～D6 顺序连接 a～g 段,若采用同相驱动时,数字 2 的显示段码应该是_____。

(6) OLED 的驱动方式分为被动驱动和主动驱动两种方式,被动驱动方式又可称为_____方式,主动驱动(AMOLED)又可称为_____方式。

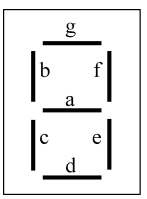


图 5-13　LED 外形排列