百问网
www.100ask.net

100ASK_AM335X

ARM Cortex-A8

Embedded Development Platform

Application Manual

Rev. 1.3

2019/07/10

0755-86200561

support@100ask.net

Shenzhen, Guangdong, China

Shenzhen 100ask Technology Co.

注意事项与售后维修

1. 注意事项

- 使用产品之前，请仔细阅读本手册，并妥善保管，以备将来参考；
- 请注意和遵循标注在产品上的所有警示和指引信息；
- 请使用配套电源适配器，以保证电压、电流的稳定；
- 请在凉爽、干燥、清洁的地方使用本产品；
- 请勿在冷热交替环境中使用本产品，避免结露损坏元器件；
- 请勿在湿气过重、温度过高或过低环境中使用本产品，使用时注意产品的通风；
- 请勿将任何液体泼溅在本产品上，禁止使用有机溶剂或腐蚀性液体清洗本产品；
- 请勿在多尘、脏乱的环境中使用本产品，如果长期不使用，请包装好本产品；
- 请勿在振动过大的环境中使用，任何跌落、敲打或剧烈晃动都可能损坏线路及元器件；
- 请勿在通电情况下，插拔核心板及外围模块(特别是串口模块)；
- 请勿自行维修、拆解本产品，如产品出现故障应及时联系本公司进行维修；
- 请勿自行修改或使用未经授权的配件，由此造成的损坏将不予保修；

2. 售后维修

1) 保修期限

- 底板、核心板：三个月（非人为损坏）
- 显示屏：七天（非人为损坏）

2) 保修说明

- 7 天内：产品（底板、核心板、屏幕）非人为损坏，本公司免费更换/维修，并承担来回运费；
- 7 天至 3 个月内：底板、核心板非人为损坏，本公司免费维修，并承担来回运费（屏幕不提供维修）；
- 3 个月至 1 年：底板、核心板非人为损坏或人为轻微损坏，只收更换元器件费用，免费维修，买家承担来回运费；
- 起始时间以快递签收日为准；

3) 联系方式

官方网站：www.100ask.net
淘宝网站：100ask.taobao.com

地　址：广东省深圳市龙岗区布吉南湾街道平吉大道建昇大厦 B1505
联系人：售后维修部
电　话：0755-86200561
邮　编：518114

邮寄须知：保修期限内，寄回本产品请预先垫付邮费，公司不接收任何到付快递。

技术支持与开发定制

1. 技术支持范围

1) 本公司提供的各类开发软件的安装，入门使用，环境搭建；
2) 本公司提供的所有裸机代码的烧写验证；
3) 本公司发布的操作系统的编译、烧写；
4) 本公司发布产品的工控板、模块的硬件原理；
5) 本公司发布的各种外设模块驱动及源码；
6) 本公司发布的配套手册在使用过程中遇到的问题；
7) 本公司产品的故障诊断及售后维修服务；

2. 技术讨论范围

由于嵌入式系统知识范围广泛，涉猎种类繁多，我们无法保证对各种问题都能一一解答，以下内容无法供技术支持，只能提供建议。

1) 本公司发布的教程之外的知识；
2) 非本公司发布的 U-Boot、Linux 内核的编译和移植；
3) 非本公司发布的工控板的各类驱动支持；
4) 非本公司发布的外设模块的硬件原理和驱动设计；

3. 技术支持方式

1) 官方论坛发帖提问(推荐)：bbs.100ask.net
2) 官方淘宝通过阿里旺旺咨询：100ask.taobao.com
3) QQ 群咨询（QQ 群号咨询淘宝客服，需提供淘宝购买订单号验证加入）；
4) 技术支持邮箱：support@100ask.net
5) 联系电话：0755-86200561

4. 技术支持时间

星期一到星期五;上午 9:00—12:00;下午 14:00—17:30;

公司按照国家法定节假日安排休息，在此期间无法提供技术支持，请将问题发送至技术支持邮箱或在论坛对应板块发帖，我们将在工作日尽快给您回复。

5. 投诉和建议

如果您对我们有不满意或者建议，可发送邮件到 support@100ask.net 进行反馈，也可拨打 0755-86200561 取得联系，我们将不断改进。

6. 定制开发服务

本公司提供嵌入式操作系统底层驱动、硬件板卡的有偿定制开发服务，以缩短您的产品开发周期。请将需求发送邮件到 support@100ask.net。

资料获取与后续更新

1. 资料的获取

1) 百度网盘下载

百度网盘里面有本产品的所有配套资料，包括原理图、发布的 U-Boot、内核镜像和源码、所需的开发软件、工具等等。

进入 www.100ask.net，在导航栏选择“资料下载”，点击“到百度网盘下载”，跳转到百度网盘后，找到对应的文件夹即可。

2) 视频配套教程

后续会为该工控板录制一套裸机、Linux 驱动、应用的配套付费教学视频，有需要的客户可以进入官方淘宝 100ask.taobao.com 选购。

3) 维基百科教程

维基百科里面会有视频配套的笔记，进入 wiki.100ask.net，选择对应的板块查看。

2. 后续更新

后续文档、视频等资料的更新，为了确保您的资料是最新状态，请密切关注我们的动态，我们将会通过微信公众号和 QQ 群公告推送，购买了本产品的客户请添加 QQ 群（QQ 群号咨询淘宝客服，需提供淘宝购买订单号验证加入）或关注微信公众号。

版权声明

更新记录

类别	100ASK_AM335X 系列文档
文档名	100ASK_AM335X Application Manual
当前版本	1.3
适用型号	C3358R512FN512C
编辑	百问科技文档编辑团队
审核	韦东山

修改日志		
版本	修改时间	更改说明
1.0	2019.03.18	初始版本，黄成
1.1	2019.06.26	修改，韦东山
1.2	2019.07.10	修改，黄成
1.3	2019.07.24	发布版，黄成

目录

前言

在过去十年的教育培训中，遇到很多程序员、计算机软硬件爱好者。他们对硬件、驱动并不了解，希望通过简单的学习就能控制硬件，实现自己创意想法，而不是看硬件芯片手册、编写驱动、应用程序。为此，我们在产品上编写了一套基于 Arduino 的应用程序库，让用户可以通过几行简单的代码，实现对各种硬件模块的控制。

本手册首先介绍了板子的配置和接口，然后以接口分类，介绍了 LED 灯、按键、红外遥控、温湿度传感器、人体红外检测、超声波测距、步进电机控制、OLED 显示、三轴加速度计、实时时钟、数模/模数转换、GPS 模块等的基本使用，最后再示例了两个实战项目供读者参考。

本手册适合嵌入式开发初学者、计算机软硬件爱好者阅读，帮助读者零基础快速操作外设硬件模块。

第一章 配置、接口介绍

1.1 配置介绍

100ASK_AM335X 使用“核心板+底板”的设计方式。在核心板上包含 CPU、RAM、ROM、PMU(电源管理)等最基本也是最重要的元器件。底板上包含一些常用的模块和对外接口。

100ASK_AM335X 使用美国 TI(德州仪器)公司基于 ARM Cortex-A8 内核设计的高性能、低功耗微处理器。该微处理器主频高达 1GHz，运算能力为 1600DMIPS，同时提供 3D 图形加速和关键外设的集成。板载 512MB 的 RAM(DDR)、512MB 的 ROM(Nand Flash)，支持板载 ROM 启动或 SD 卡启动。

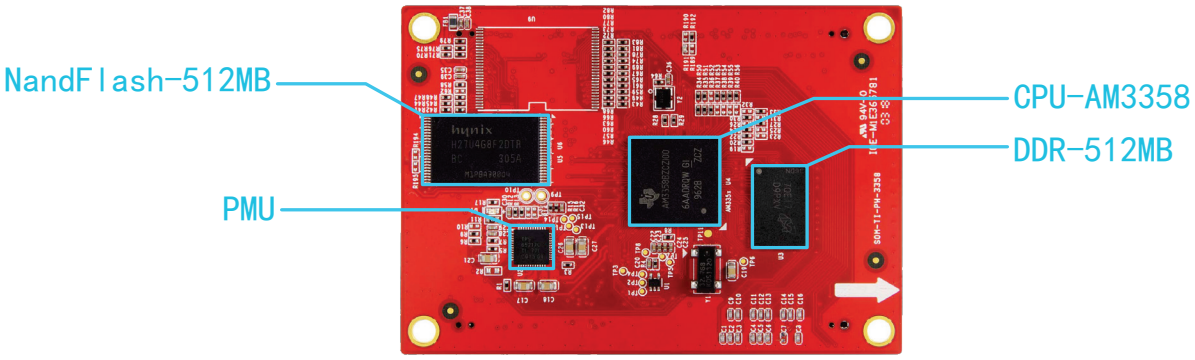

图 1.1.1 100ASK_AM335X 核心板重要器件实物图

1.2 接口介绍

板载的模块包含 LED 灯、按键、RTC 时钟。剩下的都是对外的接口，比如生活中常见的 USB 接口、有线网卡接口、SD 卡接口、耳机接口，还有一些生活中不常见的 RS232 电平串口接口、TTL 电平串口接口、GPIO 接口、SPI 接口、I2C 接口等。

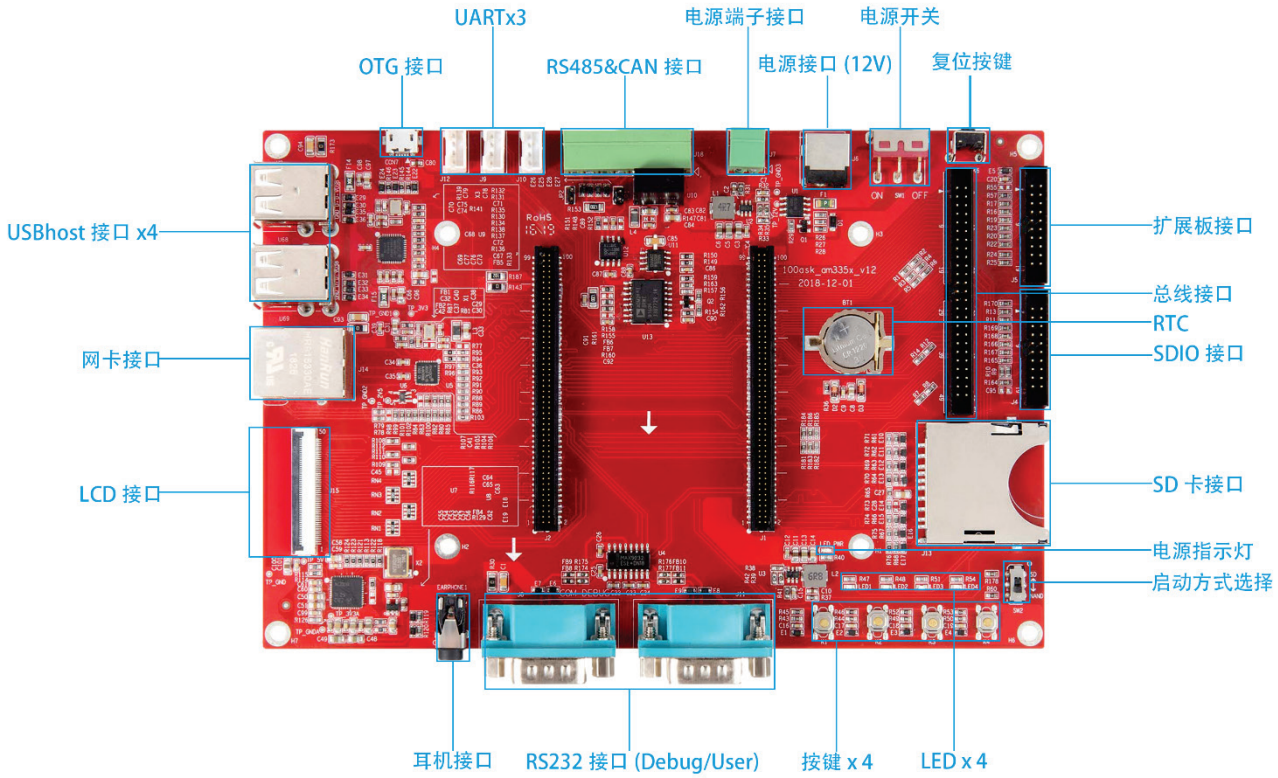

图 1.2.1 100ASK_AM335X 底板接口实物图

接口列表		
接口名称	**数量**	**介绍**
USB接口	4	用于外接USB接口设备：USB摄像头、U盘、键盘等
网口	1	用于外接网线，连接网络
LCD接口	1	用于外接RGB接口屏幕
耳机接口	1	用于外接耳机，播放声音和录音
R232电平串口	2	用于调试程序打印信息
SD卡接口	1	用于插入SD卡，扩充存储或作为SD卡启动
扩展接口(SPI)	2	用于外接SPI接口设备：ADXL345、OLED
扩展接口(I2C)	2	用于外接I2C接口设备：EEPROM、RTC
扩展接口(GPIO)	5	用于外接GPIO接口设备：温度传感器、红外遥控、超声波测距
TTL电平串口	3	用于外接TTL接口设备：蓝牙模块、GPS模块
OTG接口	1	用于连接电脑，文件传输
板载模块列表		
LED灯	4	用于状态显示、用户自定义显示
按键	4	用于按键输入
RTC实时时钟	1	用于提供系统时钟

以上配置、接口介绍为精简内容，更多详细的硬件参数，参考 100ASK_AM335X 硬件手册。

1.3 扩展板和模块介绍

百问网制作了配套的扩展板，可以在扩展板上接上各种模块。扩展板示意图如图 1.3.1 所示。

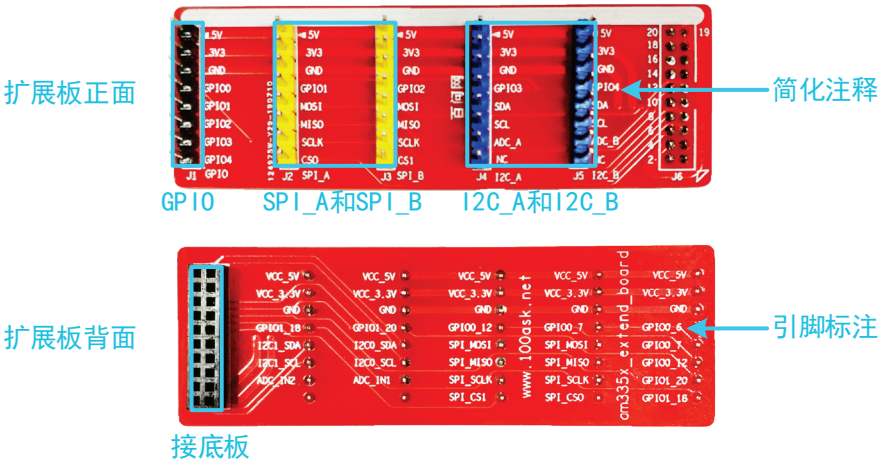

图 1.3.1 100ASK_AM335X 扩展板实物图

扩展板正面从左到右依次是一列 GPIO 接口、两列 SPI 接口(SPI_A 和 SPI_B)、两列 I2C 接口(I2C_A 和 I2C_B)，其中 I2C 接口所在列还包含 ADC 接口(ADC_A 和 ADC_B)。每列接口都提供 5V、3V3 电源，以便支持工作电压不同的模块。

扩展板正面文字注释为**简化注释**，后面文档、代码如果没有特殊说明，都是使用该简化注释。

扩展板背面文字注释为核心板或底板原理图上的**引脚标注**，供想深入学习的用户参考。

比如扩展板上的简化注释 GPIO0,含义是指扩展板中的第 0 个 GPIO 引脚。背面对应引脚注释为 GPIO0_6，表示使用的是 AM335X 芯片中第 0 组 GPIO 中的第 6 个 GPIO。初学者不需要深入理解这些内部知识，只要使用简化注释 GPIO0 即可。

对于 GPIO，一共有 5 个，分别为 GOIO0、GOIO1、GOIO2、GOIO3、GOIO4。

GPIO 的使用场景非常广泛，例如：

① 有些模块只需要一个 GPIO 就能实现功能，比如 IrDA、DS18B20；

② 有些模块可能需要多个 GPIO 引脚，比如步进电机驱动板同时需要四个 GPIO；

③ 有时可能想同时使用 2 个模块，比如同时使用 IrDA 和 DS18B20，就需要两组 GPIO 和配套电源；

④ 此外，一些其它接口的模块，需要 GPIO 协助，比如 OLED，还需要一个 GPIO 控制数据/命令的切换；

基于以上情况，使用扩展板的 GPIO 时要留意：

① 对于只需要一个 GPIO 的模块,可以插在五个排针座中的任意一个的上半部分,如图 1.3.2 所示的 GPIO0、GPIO1、GPIO2、GPIO3、GPIO4；

② 对于步进电机驱动板这种需要两个及以上 GPIO 的模块，只能插在最左边的黑色排针 GPIO0 上；

③ 同一个 GPIO，不能同时用于多个模块。比如不能同时使用电机驱动板和 OLED 模块。因为电机驱动板占用了 GPIO0～3；而 OLED 模块本来可以接在 SPI_A 或 SPI_B，但所需的 GPIO1～2 被电机驱动板占用了。

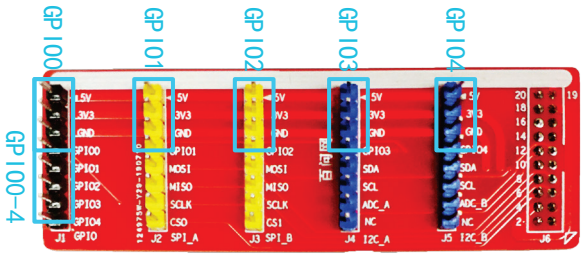

图 1.3.2 100ASK_AM335X 扩展板 GPIO 介绍

模块板数量众多，后续可能会继续生产更多模块，这些模块的安装、使用方法请看后续章节。现有模块如下图所示。

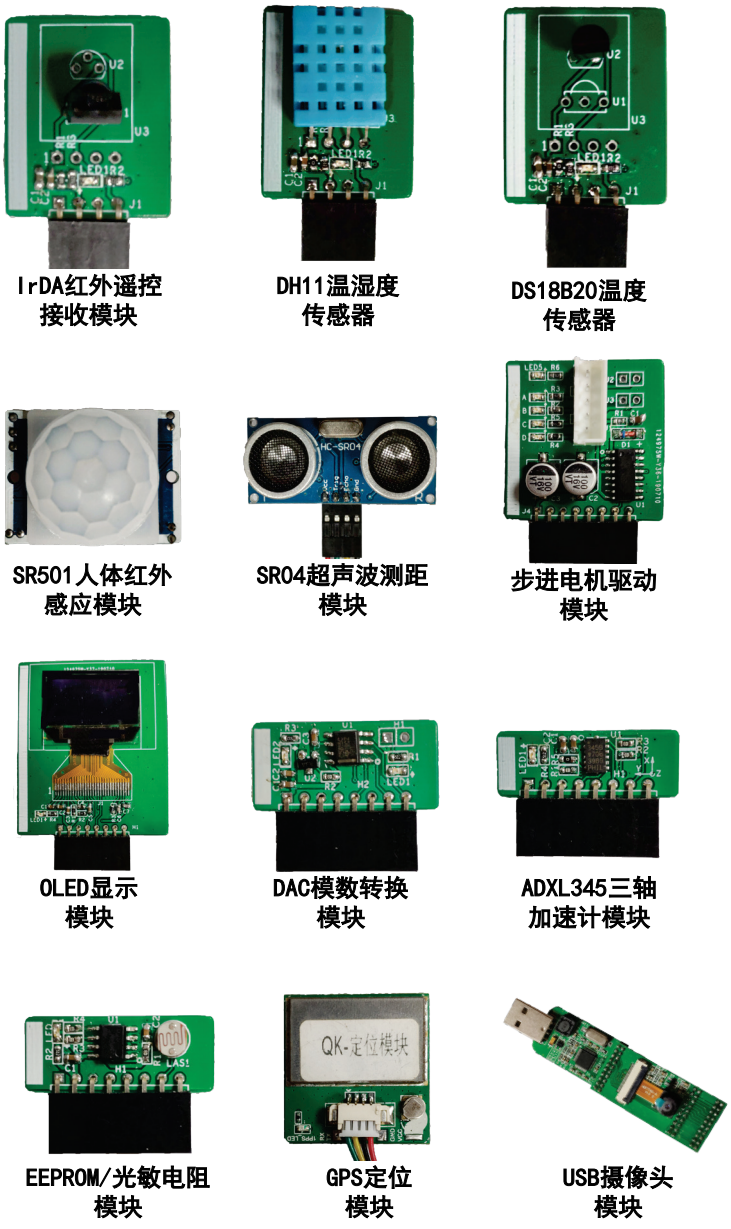

图 1.3.3 配套模块实物图

注意：模块插到转接板上去时，为防止插错，模块和转接板都画有一条长白线，接插时长白线对齐。

1.4 软件安装

1.4.1 串口工具软件

在后面的操作里，都是通过串口与板子进行“交流”。串口是串行接口的简称，是指数据一位一位地顺序传送，其特点是通信线路简单。

在电脑上先安装一个串口工具软件 MobaXterm，接上 USB 串口模块，并跟开发板连好线。

在 MobaXterm 里敲打键盘，就会通过 USB 串口模块，将数据经过 RS232 延长线传给板子，板子就能接收到我们在电脑上发送的数据。

反过来，板子发送的数据首先经过 RS232 延长线到 USB 串口模块，MobaXterm 读取数据后显示出来。

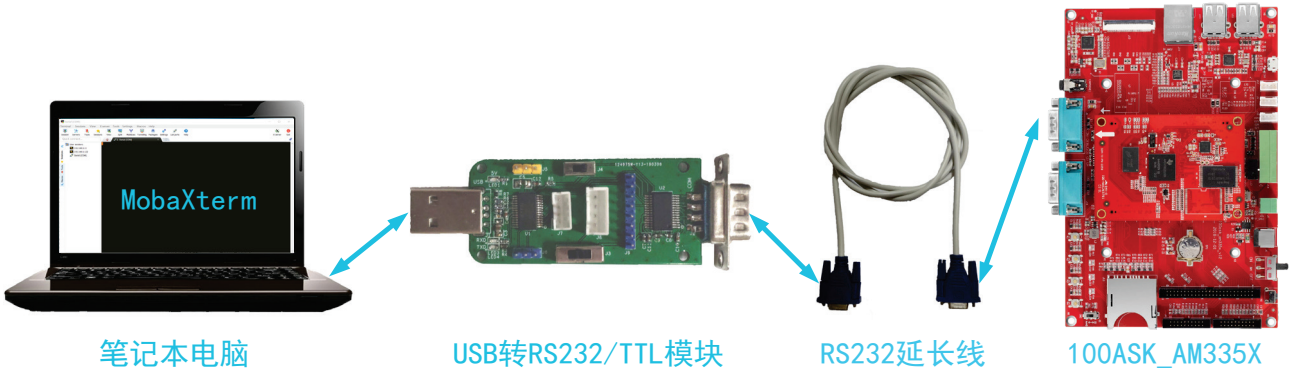

图 1.4.1 串口连接示意图

1) 下载软件安装包

可以从 MobaXterm 官网(https://mobaxterm.mobatek.net/download.html)下载家庭免费版，也可以直接使用我们百度云盘提供资料里的软件安装包。

MobaXterm 有家庭免费版和专业收费版，家庭免费版目前就够用了，因此如图 1.4.2 选择家庭免费版，点击“Download now”。

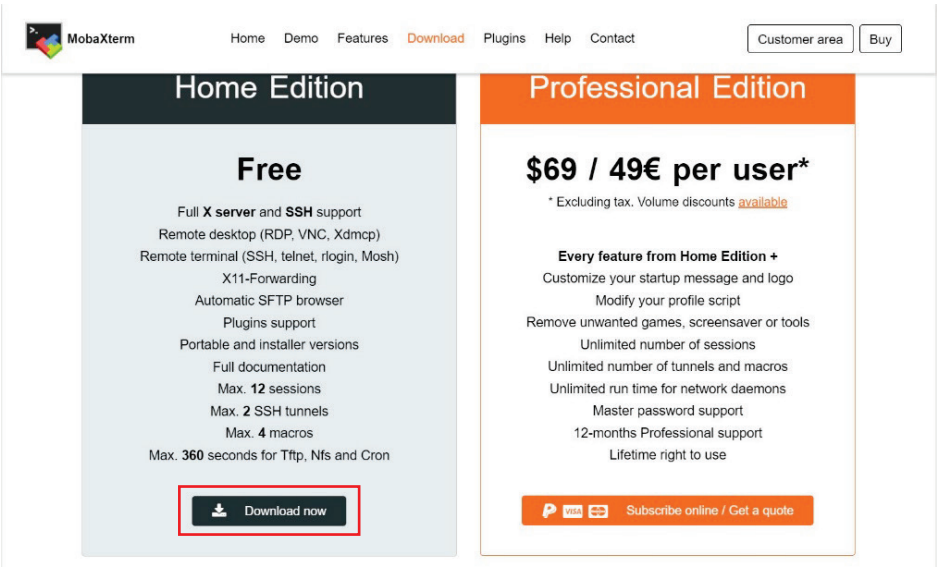

图 1.4.2 MobaXterm 版本选择

接着弹出安装版本的选择界面，左边的是免安装便携版，右边是安装版。为了方便，如图 1.4.3 选择免安装版即可。

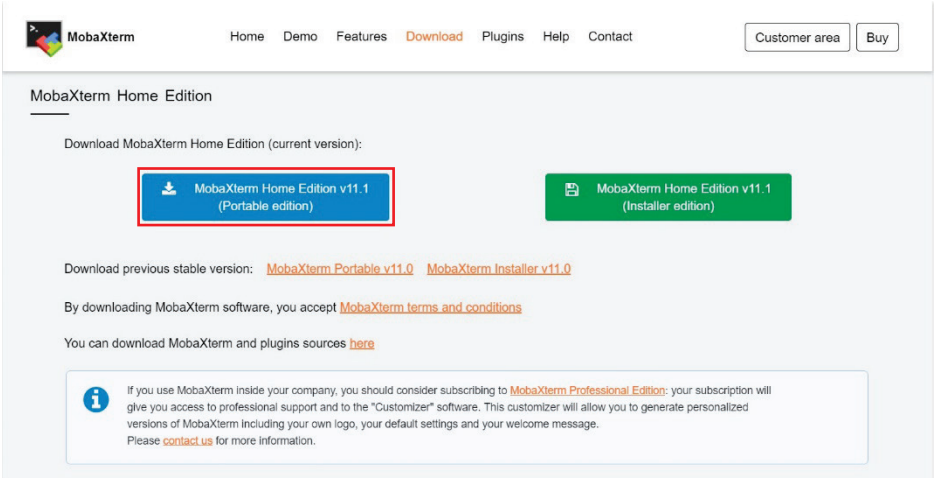

图 1.4.3 MobaXterm 安装版本选择

下载得到“MobaXterm_Portable_v11.1.zip”，解压后直接运行“MobaXterm_Personal_11.1.exe”即可。

2) 安装驱动

将 USB 转 RS232/TTL 串口模块插在电脑 USB 上，此时 Windows 会自动安装驱动(安装可能比较慢，等一分钟左右)。打开电脑的“设备管理器”，在“端口 (COM 和 LPT)”项下，可以看到如图 1.4.4 中的“Silicon Labs CP210x USB to UART Bridge (COM12)”。这里的“COM12”可能与你电脑上的不一样，记住你电脑上设备管理器中显示的数字。

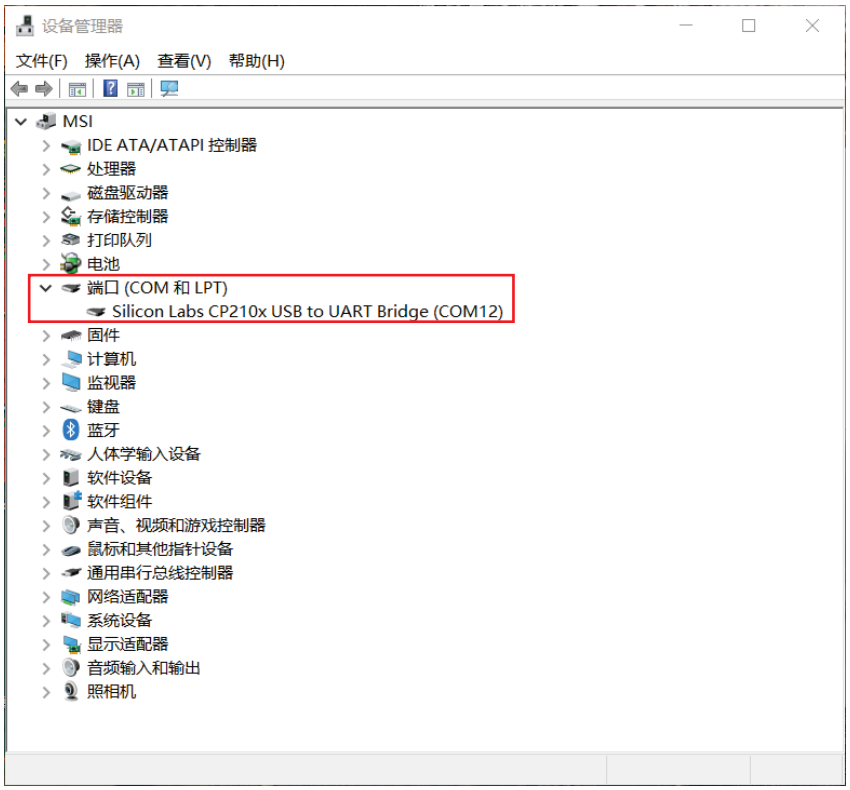

图 1.4.4 设备管理器查看串口端口号

如果电脑没有显示出端口号，就需要手动安装驱动，从驱动精灵官网（www.drivergenius.com）下载一个驱动精灵，安装、运行、检测，会自动安装上串口驱动。

3） 连线、配置

首先如图 1.4.5 所示将串口模块与电脑 USB 口、板子 RS232 口连接。

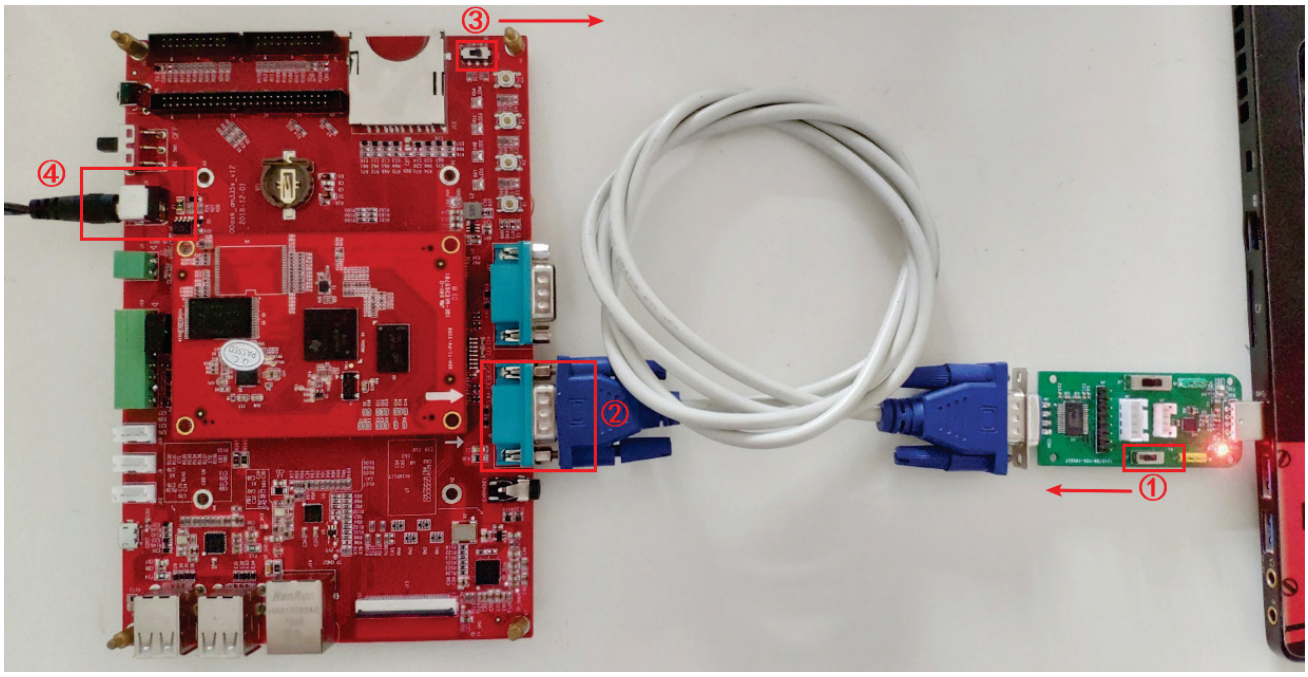

图 1.4.5 串口模块连接实物图

其中特别需要注意的几点：

① 串口模块的电平选择开关拨到如图所示的左边，表示切换到 RS232 电平；

② RS232 延长线连接板子如图所示下方的 RS232 接口，下方的才是默认的 debug 接口；

③ 板子的启动选择拨到如图所示的右边，以选择从 Nand Flash 启动；

④ 板子如图所示插上配套的电源，电源开关暂时不用打开；

打开 MobaXterm，点击左上角的“Session”，在弹出的界面选中“Serial”，如图 1.4.6 所示设置端口号（前面设备管理器显示的端口号）、波特率（Speed:115200）、流控（Flow control:None），最后点击“OK”即可。

注意：流控（Flow Control）一定要选择 none，否则你将无法在 MobaXterm 中向串口输入数据。

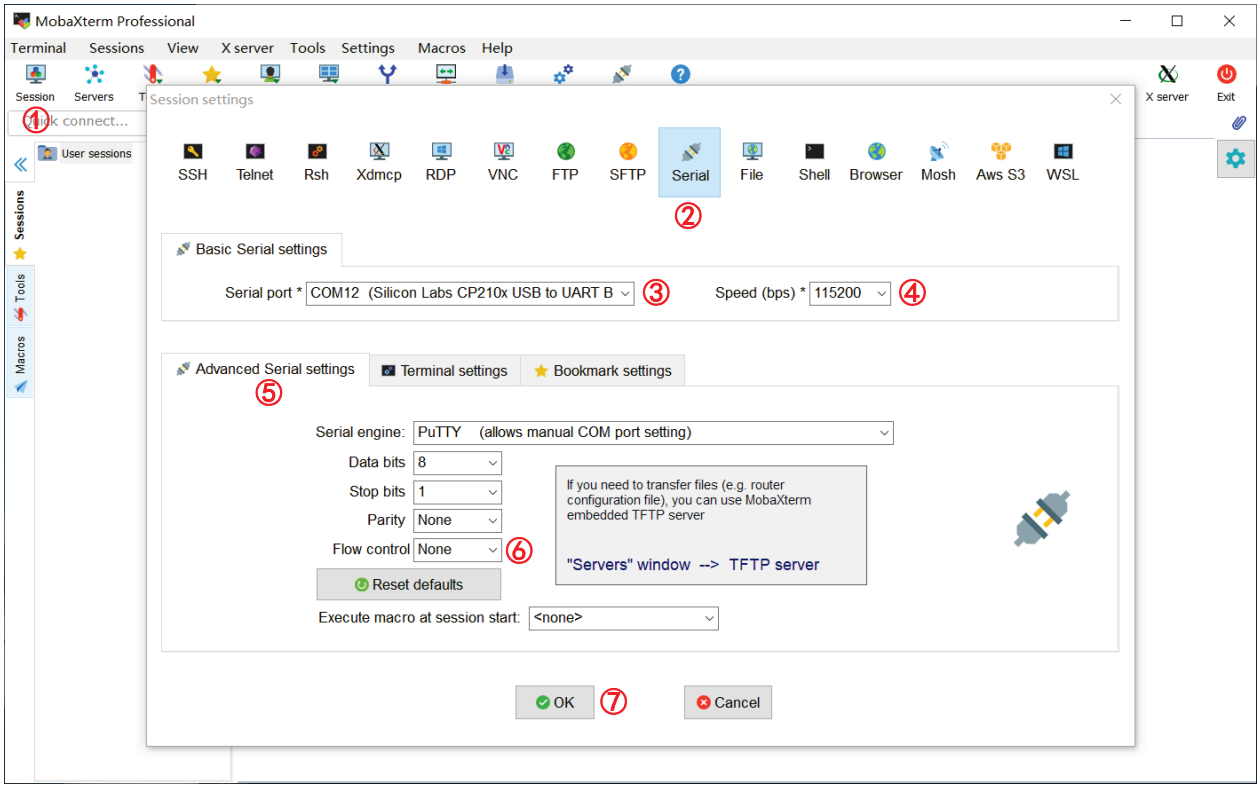

图 1.4.6 MobaXterm 串口对话设置

随后显示一个黑色的窗口， 此时打开板子的电源开关，将收到板子串口发过来的数据，如图 1.4.7 所示。

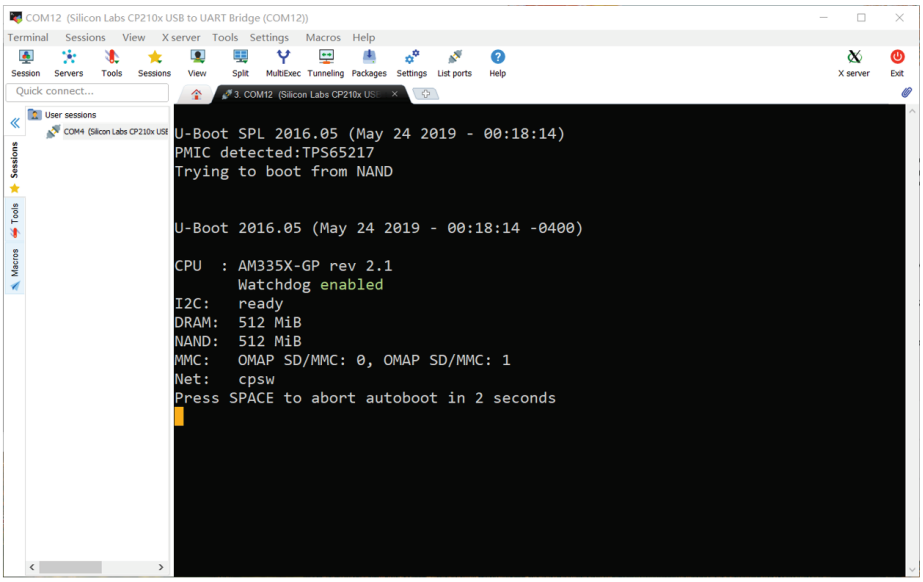

图 1.4.7 接收板子串口发送的数据

1.4.2 Arduino IDE

Arduino IDE 是由 Arduino 官方提供的一款支持 C/C++的集成开发环境。Arduino 简单易用，适合开发简单的程序，适合初学者。

通过简单的配置，就可以 Arduino 里添加对 100ASK_AM335X 开发板的支持。

1） 下载软件安装包

可以从 Arduino 官网(https://www.arduino.cc/en/Main/Software?setlang=cn)免费下载安装包，也可以直接使用我们百度云盘提供资料里的软件安装包。

Arduino 支持 Windows、MAC、Linux 等平台，这里以 Windows 平台为例进行介绍。Windows 下的安装包有两种，一种是需要安装的可执行程序，一种是免安装的 ZIP 压缩包，这里使用免安装的 ZIP 压缩包，如图 1.4.8 所示点击“Windows 免安装 ZIP 包”下载。

图 1.4.8 下载 Arduino 软件包

下载完成后，解压“arduino-1.8.9-windows.zip”，进入解压目录，双击“arduino.exe”启动 Arduino。

2） 添加 100ASK 库

启动 Arduino 后，点击“文件”->“首选项”，如下图 1.4.9 所示。

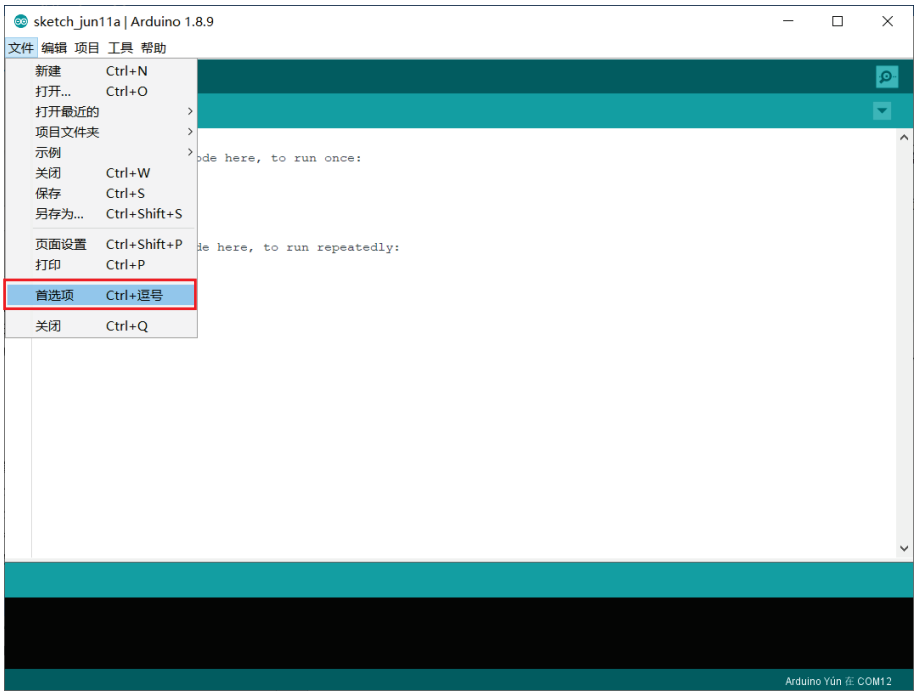

图 1.4.9 打开 Adrduino 首选项

在弹出的界面里，找到“显示详细输出：”，勾选上“编译”、“上传”，接着在“附加开发板管理器网址：”里添加以下内容：

```
https://raw.githubusercontent.com/100askTeam/Arduino-IDE/master/package_100ask_am335x_boards_index.json
```

设置完成后，点击“好”确定，如下图 1.4.10 所示。

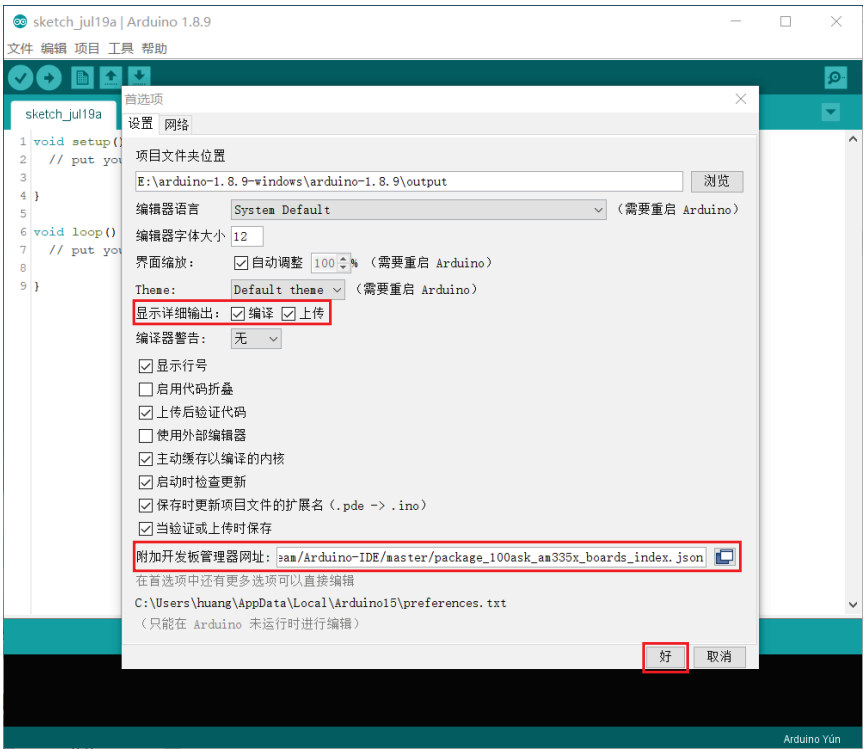

图 1.4.10 设置 Adrduino 首选项

接着，点击“工具”->“开发板：xx”->“开发板管理器”，如下图 1.4.11 所示。

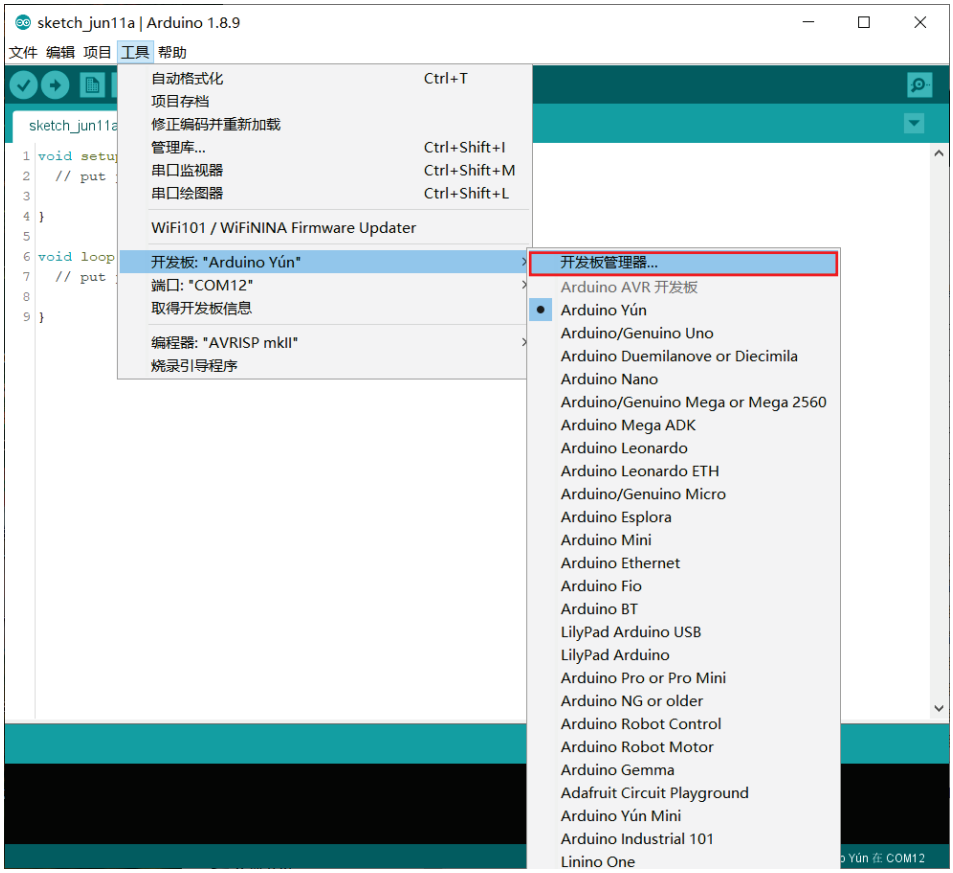

图 1.4.11 打开 Adrduino 开发板管理器

在弹出的“开发板管理器”界面搜索框里输入“100ask”，随即在下面出现“100ask modules Boards by 100ask”，点击后面的“安装”，如图 1.4.12 所示。

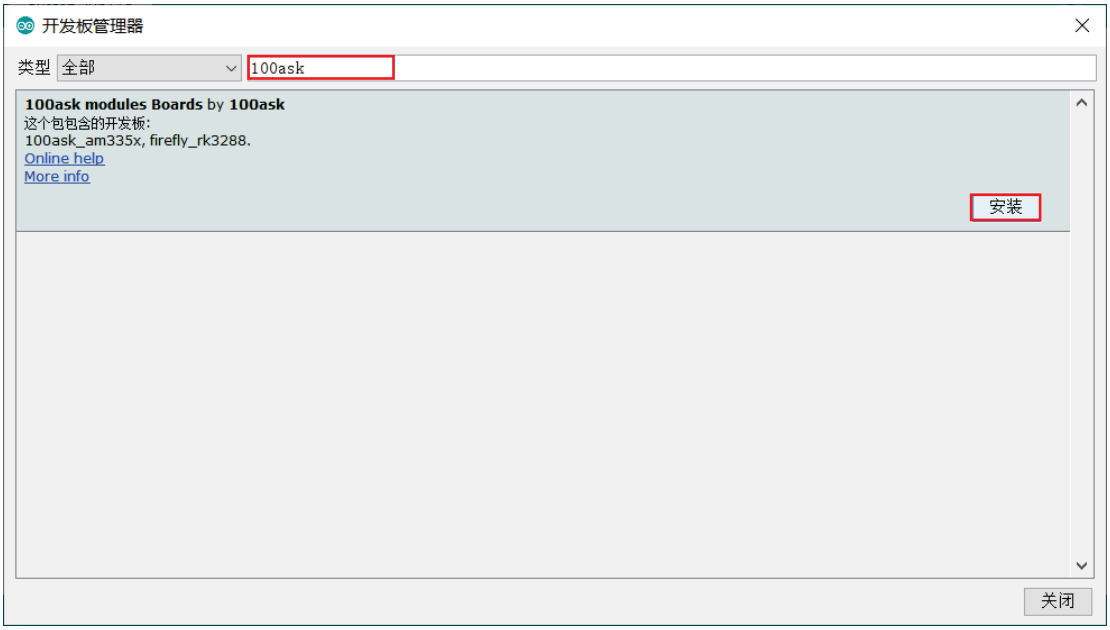

图 1.4.12 安装 100ask_am335x 库

安装过程需要先从网络下载，可能花费的时间比较长，耐心等待。待安装完成后，关闭“开发板管理器”界面，再次点击“工具”->“开发板：xxxx”，可以看到最下面多出了一个“100ask_am335x”，选中它，如图 1.4.13 所示。

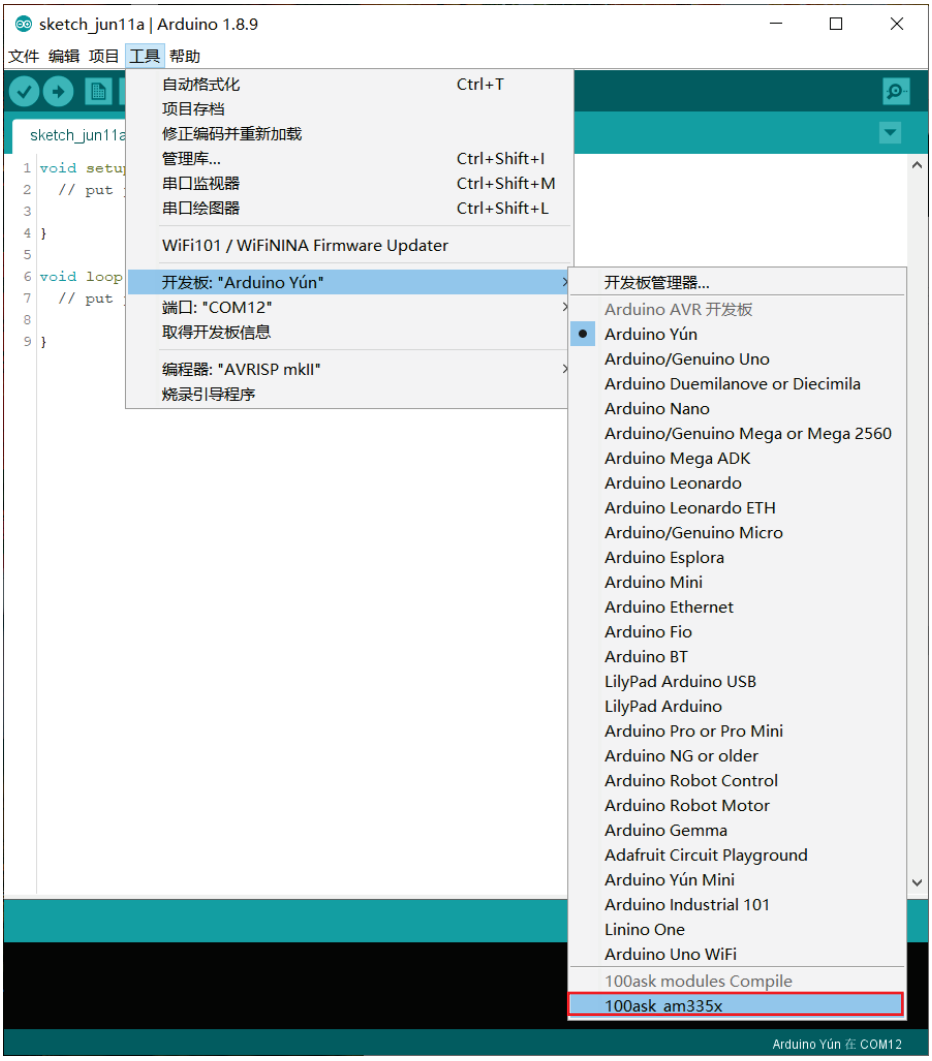

图 1.4.13 选择 100ask_am335x 库

至此，Arduino IDE 就安装、配置完成，对 Arduino 的使用在后面再讲解。

第二章 GPIO 接口

首先，简单介绍下 GPIO（General-purpose input/output），通用输入/输出端口。CPU 上有很多引脚，其中大部分都是 GPIO 引脚，它们经过电路板被引了出来。这些引脚默认都是 GPIO 功能，可以通过程序控制让它输出高/低电平，也可以通过程序读取它的状态(是高电平还是低电平)。

GPIO 最简单的应用就是控制 LED 的亮/灭，或是读取输入电平实现按键功能；复杂的应用就是通过 GPIO 输出或输入一系列复杂的脉冲信号，传输复杂的信息。

2.1 点亮 LED 灯

2.1.1 板载LED介绍

板子上有四个 LED 灯，在电路图中如下：

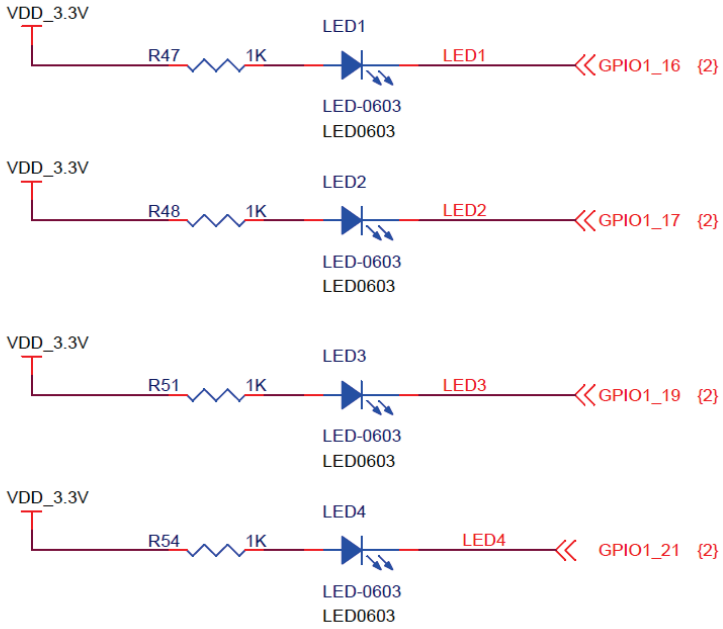

图 2.1.1 LED 原理图

从原理图我们可以得到两个重要信息：

1) LED 使用哪个引脚：
   LED1 接在 GPIO1_16 上，LED2 接在 GPIO1_17 上，LED3 接在 GPIO1_19 上，LED4 接在 GPIO1_21 上。
2) 怎么控制引脚才能点亮/熄灭 LED：
   LED 的左边是电源 VDD_3.3V，当右边引脚为低电平时才会产生电势差，灯才会亮。因此控制 GPIO 输出高电平时，灯熄灭；GPIO 输出低电平时，灯点亮。

2.1.2 使用Linux命令控制LED

在使用Arduino编程前，先感受下如何使用Linux命令控制LED灯。

首先如上一章的图1.4.5所示，将板子、RS232延长线、串口模块、电脑连接起来，再打开MobaXterm，并建立串口对话，接通板子电源，观察MobaXterm内容，等待板子进入如下登陆界面。

```
Welcome to 100ask-am335x system !

* Documentation:    http://wiki.100ask.net/100ask_am335x
* SourceCode:       https://dev.tencent.com/u/weidongshan/
* Support:          https://support@100ask.net
* Shop:             https://100ask.taobao.com/
Ti335x login:
```

此时在MobaXterm的串口对话框内输入以下内容(用户名)，按下回车，进入系统：

```
root
```

文档中有“[root@100ask-am335x:~]#”为输入提示符，后面内容为用户输入。没有该提示符的行，则是系统打印的信息。

进入系统后，依次输入以下命令，每行命令以回车结束。

```
[root@100ask-am335x:~]# echo 53 > /sys/class/gpio/export
[root@100ask-am335x:~]# echo out > /sys/class/gpio/gpio53/direction
[root@100ask-am335x:~]# echo 0 > /sys/class/gpio/gpio53/value
```

执行完上述命令后，板子上的LED4亮了起来，再输入以下命令可以熄灭LED4：

```
[root@100ask-am335x:~ ]# echo 1 > /sys/class/gpio/gpio53/value
```

简单解释下第1条命令的含义。

① “echo”是字符串输出命令，“echo 53”就会显示“53”这个字符串；

② “>”是重定向标记符，表示把结果写入某个文件；

③ “echo 53 > /sys/class/gpio/export”这条命令的含义就是把“53”这个字符串写入文件“/sys/class/gpio/export”；

因此，第一个命令就是向文件“export”写入“53”，内核中的驱动程序会因此生成文件夹“/sys/class/gpio/gpio53”及该文件夹下的文件。

上述命令中的“53”是指LED4所接的GPIO1_21引脚的引脚号，其算法是“1x32+21=53”。如果是GPIO0_3就对应的引脚号就是“0x32+3=3”。注意这里只支持LED4，其它的LED1-LED3分别用作了SD卡状态提示灯、CPU状态提示灯、NandFlash状态提示灯，用户无法再使用命令来控制它。

第二个命令就是向前面生成的“gpio53”目录下的文件“direction”写入“out”，作用是设置该GPIO为输出功能。

GPIO有输出、输入功能。用来控制LED时需要使用输出功能；后面我们将GPIO用作按键时就要设置为输入功能，这时要往“direction”写入“in”。

第三个命令就是向前面生成的“gpio53”目录下的文件“value”写入“0”，作用是设置该GPIO输出低电平。前面我们分析电路图要GPIO为低电平灯才会亮，反之写入“1”就熄灭LED灯。

可以发现，我们要在Linux控制硬件其实很简单，但这里还有几个问题：

① 假设我们每次要点亮四个LED灯，是不是要重复操作四次以上命令？

② 假设我们想要LED灯每隔1s交替亮灭，是不是需要不停的输入命令？

③ 假设我们不是操作LED，而是操作其它复杂接口，是不是要每次操作更多复杂命令？

解决上述几个问题也很简单，我们可以用C/C++语言，或者Shell脚本，甚至Python，编写代码，将上述几个命令写到代码里，让代码自动帮我们执行。百问网提供的100ask_am335x库，就是用C++编写的库，屏蔽了操作的细节，下面就先来体验一下。

2.1.3 使用Arduino编程控制LED

接好串口线之后，还要使用 micro USB 线将板子的 OTG 接口和电脑的 USB 口相连，如图 2.1.2 所示。

串口线用于输入命令控制开发板，USB 线用于将电脑上生成的应用程序拷贝到板子里。

接好线后，电脑的“文件资源管理器”会自动弹出一个盘符，里面有个“arduino-100ask”文件夹，表示连接 OTG 成功。

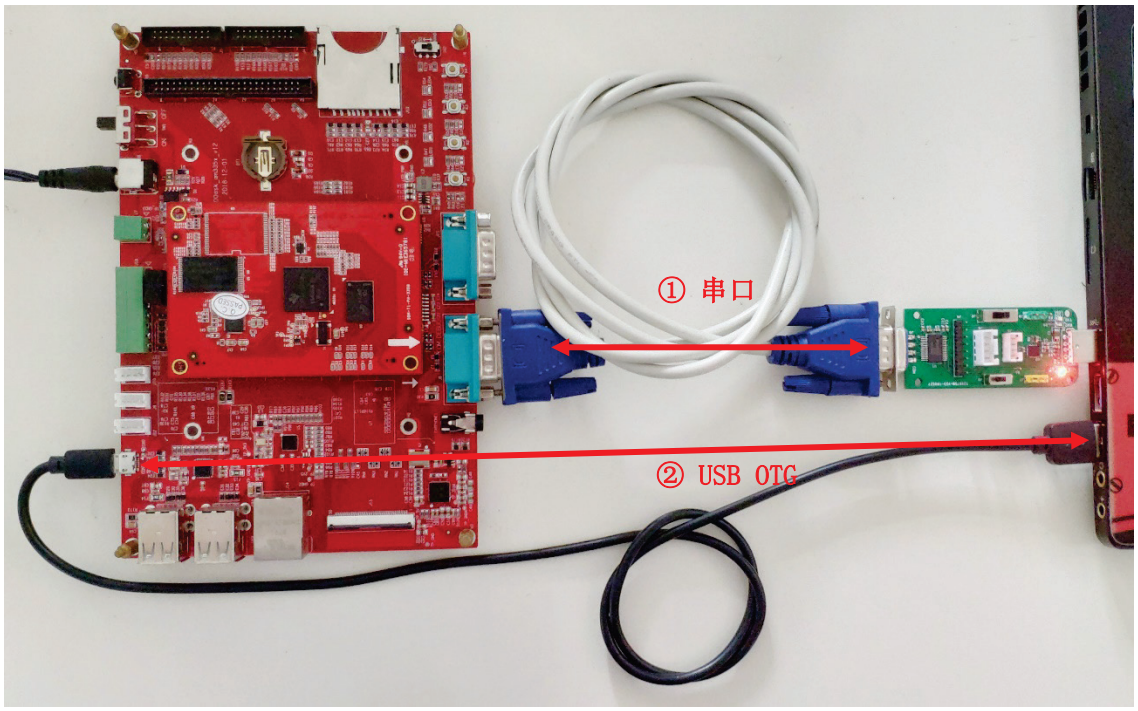

图 2.1.2 OTG 连接实物图

接着，打开 Arduino，点击“文件”->“示例”，找到“100ask_am335x 的例子”下的“GPIO.01.LED”，选中“Blink”，如下图 2.1.3 所示。

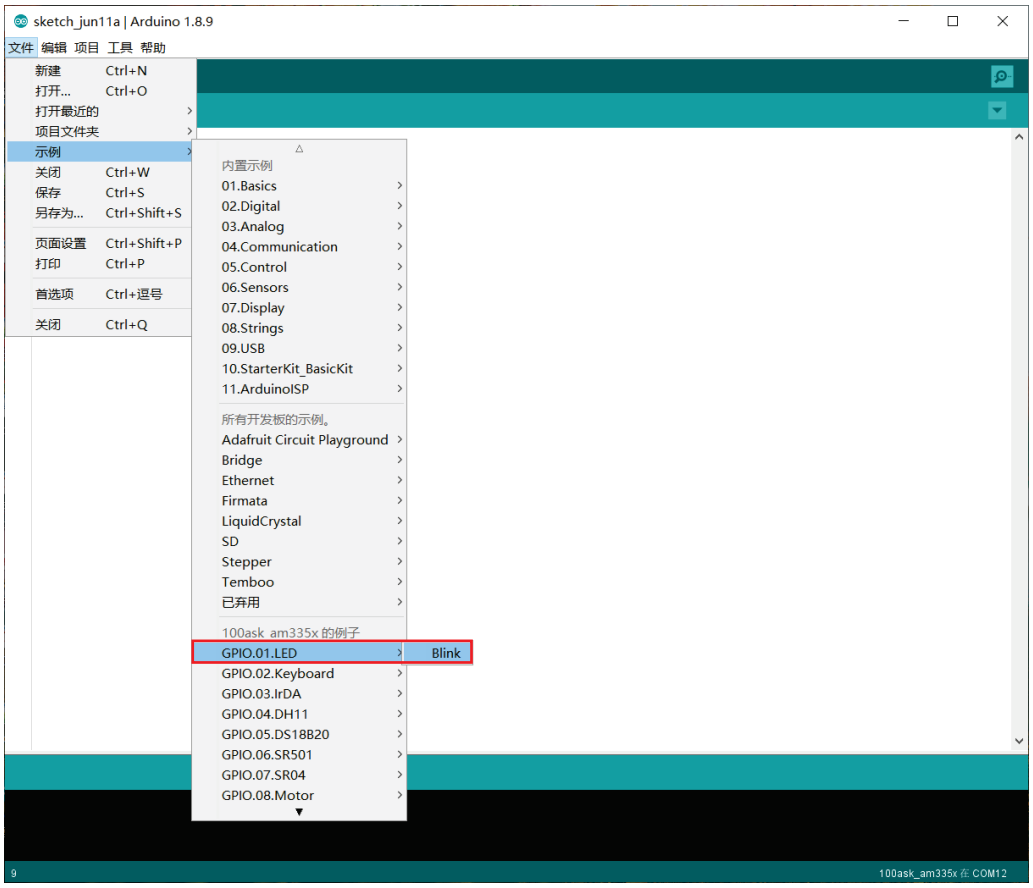

图 2.1.3 选择 LED 示例程序

此时会新弹出一个 Arduino 项目界面，这个 Arduino 项目就是 LED 的示例程序，点击该项目左上角的“  ”上传按钮。Arduino 会自动编译、上传该示例程序。待编译、上传完成后，下方会有“copy ok!”的提示，如下图 2.1.4 所示。

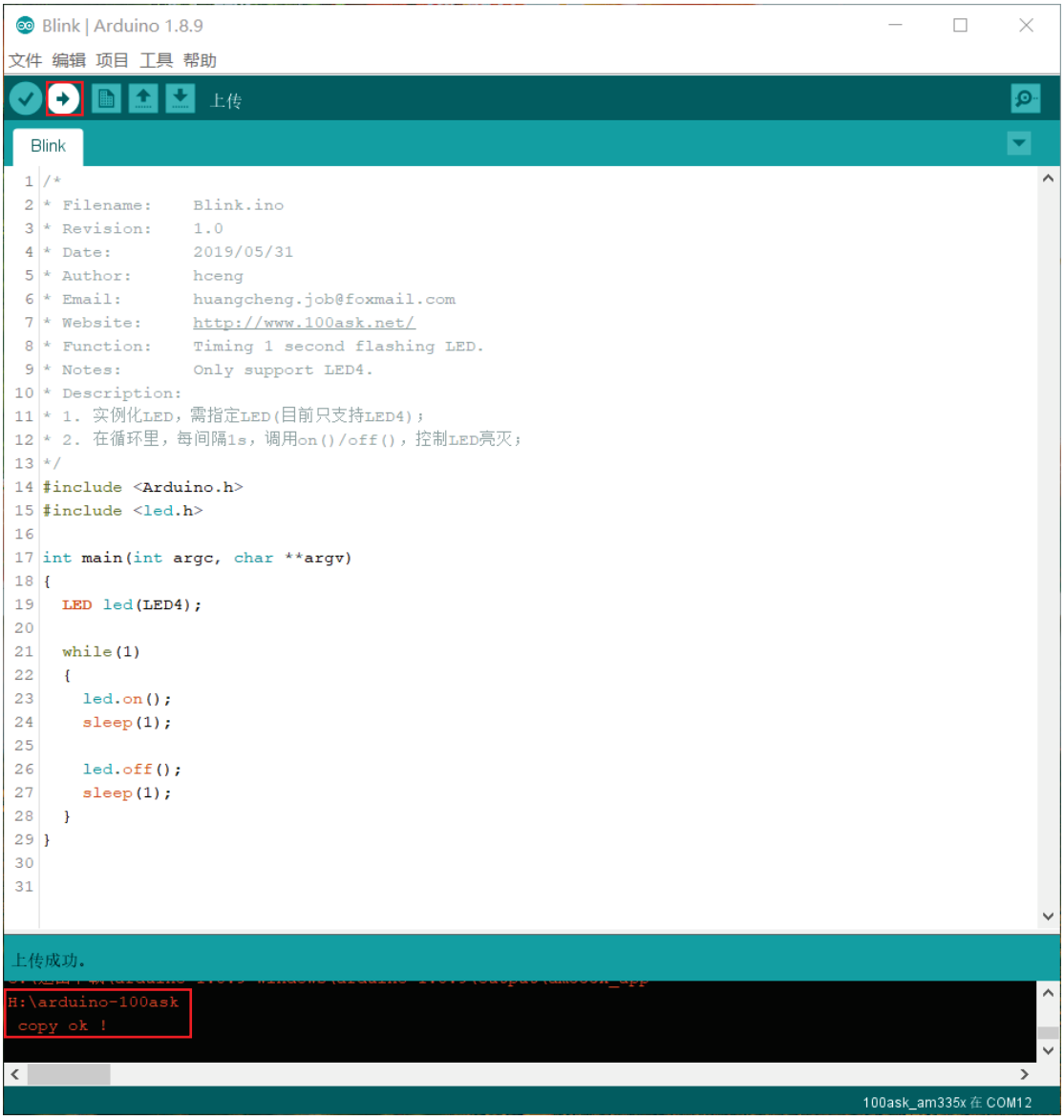

图 2.1.4 上传程序

同时，在 MobaXterm 中也会出现“Application copy succeeded !”的提示内容，表示已经收到了应用程序。

执行“ls”命令查看当前路径文件，可以看到多出来一个“am335x_app”文件：

```
[root@100ask-am335x:~]# ls
am335x_app
```

这个“am335x_app”文件就是 Arduino 里代码生成的可执行文件，输入如下命令执行该文件，即可看到 LED4 每隔 1 秒交替闪烁：

```
[root@100ask-am335x:~]# ./am335x_app
```

想终止该程序，按下组合键“Ctrl”+“C”即可。

2.1.4 百问网Arduino库介绍

目前为 Arduino 提供了 10 余个模块的示例代码。按接口类型，分成了四类：

① GPIO 接口

② SPI 接口
③ I2C 接口
④ 其它接口（UART 接口、ADC 等）

每类接口里面有几个模块。比如 GPIO 接口下，有 LED 灯、Keyboard 按键、IrDA 红外、DHT11 温湿度传感器等。

每个模块里面，至少有一个示例程序。比如 LED 灯模块下有一个 Bink 示例程序实现 LED 灯交替闪烁；Keyboard 按键模块下有两个示例程序：readKeyboard 程序实现读取按键，turnOffLED 程序实使用按键关闭 LED 灯。

后续会持续更新代码修复漏洞或增添模块，只需在“库管理器”里更新库即可，目前支持的模块信息参考附录一。

在 Arduino 中选择“文件”->“示例”->“100ask_am335x 的例子”下的任一示例程序后，都会弹出一个新的 Arduino 项目界面，可以直接点击“上传”图标即可编译、上传，然后需要你在串口工具里手动运行程序，观察效果。

也可以对示例代码稍作修改，以满足自己的需求，再上传、运行、观察效果。

值得一提的是，为了避免用户修改了原始例代码，修改后的代码在保存时，需要用户保存到其它位置。

注意：

① 每次通过“示例”菜单打开代码时，打开的都是我们提供的原始示例代码。
② 如果需要打开修改后的代码，需要通过菜单项“文件”->“打开”去之前的保存位置打开对应的代码。

以示例“GPIO.01.LED”里的“Blink”为例，对示例代码结构进行简单介绍。代码可以分成三部分：

① 代码注释：代码注释包含代码基本信息，该代码的功能，注意事项，以及代码的详细介绍。
② 头文件：要使用到什么模块，就要包含该模块的头文件。
③ 主函数：主函数是代码运行的入口，代码将从主函数开始执行。

在示例 Bink 里，首先实例化一个 C++对象 led，并指定使用哪个 LED，这里使用 LED4。

接着在一个“while(1)”死循环里，使用“led.on()”打开点亮 LED4，使用“led.off()”关闭熄灭 LED4，使用“sleep(1)”实现一秒间隔。

代码结构参考图 2.1.5 所示。


```
/*
* Filename:     Blink.ino
* Revision:     1.0
* Date:         2019/05/31
* Author:       hceng
* Email:        huangcheng.job@foxmail.com
* Website:      http://www.100ask.net/
* Function:     Timing 1 second flashing LED.
* Notes:        Only support LED4.
* Description:
* 1. 实例化LED，需指定LED(目前只支持LED4)；
* 2. 在循环里，每间隔1s，调用on()/off()，控制LED亮灭；
*/
#include <Arduino.h>
#include <led.h>

int main(int argc, char **argv)
{
    LED led(LED4);

    while(1)
    {
        led.on();
        sleep(1);

        led.off();
        sleep(1);
    }

    return 0;
}
```

图 2.1.5 代码结构

假如现在想间隔 2 秒点亮/熄灭 LED4，应该知道怎么做了吧。将主函数修改成如下内容，重新点击按钮上传，运行看看效果。

```
#include <Arduino.h>
#include <led.h>

int main(int argc, char **argv)
{
    LED led(LED4);

    while(1)
    {
        led.on();
        sleep(2);
```

```
        led.off();
        sleep(2);
    }
    return 0;
}
```

可以看到 LED4 亮灭的间隔变为了 2 秒，成功通过修改示例代码实现了自定义的效果。

如果希望下次也使用自定义修改后的代码，记得点击“文件”->“保存”，会弹出文件为只读的提示，点击“确定”，选择要保存的位置，点击“保存”即可。下次便可以通过“文件”->“打开”，找到之前保存的文件目录，使用自定义修改后的代码。

从代码可以感受到，操作变简单了，可自定义的更多了，也不再需要看电路图确定是高电平还是低电平点亮 LED。在实例化 LED 时，需要传入 LED 编号，当使用其它 GPIO 外接 LED 模块时，也可以传入 GPIO 编号。板载的所有 LED、扩展板的 GPIO 引脚编号，可以直接通过附录二查到。

2.2 Keyboard 按键

按键也可以像 LED 一样，通过设置 GPIO 为输出，再读取引脚值。但 Linux 提供了更好的输入子系统作为按键输入，可以同时对板子上的四个按键进行监测。

在 Arduino 打开菜单“示例”->“GPIO.02.Keyboard”->“readKeyboard”，代码内容如下：

```
#include <Arduino.h>
#include <keyboard.h>

int main(int argc, char **argv)
{
    int ret;

    KEYBOARD keyboard;

    while(1)
    {
        ret = keyboard.readKey();

        if (ret == 0)
            cout << "Code:" << keyboard.getCode() << " Value:" << keyboard.getValue() << endl;
    }

    return 0;
}
```

将示例代码上传，在板子里运行，然后按下板子上的按键 K1，再松开，可以看到串口打印如下：

```
[root@100ask-am335x:~] ./am335x_app
Code:105 Value:1
Code:105 Value:0
```

可以看到打印了两次，每次打印了两个值：“code”和“value”。

“code”用来表示哪一个按键被按下，“value”表示当前按键的状态(1 表示按下，0 表示松开，2 表示长按)。

K1 在 Linux 里被设定为了 KEY_LEFT，即左键的功能，对应的 code 值就是 105。四个按键对应的 code 值参考附录三按键映射表，我们只需要记住，K1 就是 105，当有按键发生，就可以根据 code 值来判断是哪一个被按下。

再来分析下代码，这里将板子上的所有按键作为了一个整体，同时进行监测。因此首先只需要实例化一个 keyboard；然后在一个“while(1)” 死循环里，使用 “readKey()”读取按键，再使用“cout”来打印输出到串口。打印的内容包含 keyboard 的 code 和 value，最后“endl”表示每次打印结束都换行。

扩展阅读

cout 语句用于向终端打印输出，其一般格式为：

```
cout<<表达式1<<表达式2<<……<<表达式n<<endl;
```

其语义是：cout 是一个对象，把要输出的字符串<<(插入)cout 中，这样就会显示在 out stream，也就是显示设备上(在这里是串口)。后面可以接多个表达式，如果用双引号“”标注的内容，就直接原样输出，没有引号标注的就输出该表达式(变量)对应的值。最后一般使用“endl”结尾，实现打印结束换行。

除“readKeyboard”示例外，还提供了一个“turnOffLED”的示例，它实现通过 KEY4 关闭 LED4 的功能，用户可以自行上传程序体验。

注意：上传新程序前，需要在 MobaXterm 里按下组合键“Ctrl”+“C”终止当前正在运行的程序。

2.3 IrDA 红外遥控

红外遥控被广泛应用于家用电器、工业控制和智能仪器系统中，像我们熟知的有电视机盒子遥控器、空调遥控器。红外遥控器系统分为发送端和接收端，如图 2.3.1 所示。

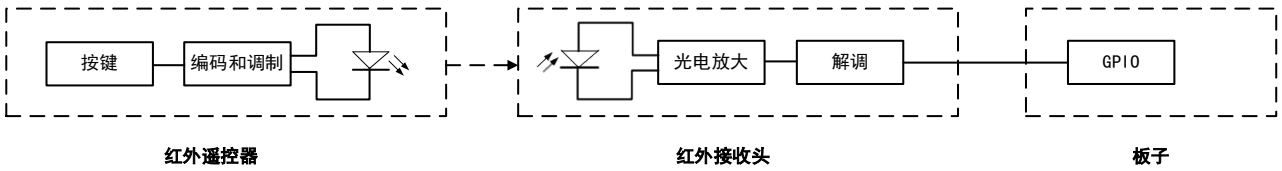

图 2.3.1 红外遥控系统原理示意图

发送端就是红外遥控器，上面有许多按键，当我们按下遥控器按键时，遥控器内部电路会进行编码和调制，再通过红外发射头，将信号以肉眼不可见的红外线发射出去。红外线线虽然肉眼不可见，但可以通过手机摄像头看到，常用该方法检查遥控器是否正常工作。

接收端是一个红外接收头，收到红外信号后，内部电路会进行信号放大和解调，再将数据传给板子上的 GPIO，板子收到数据后再解码才能确定是哪个按键被按下。

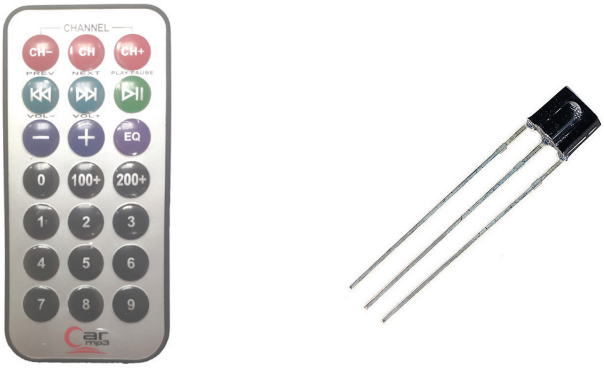

图 2.3.2 红外遥控器和红外接收头实物图

除了 LED 灯、Keyboard 按键、RTC 实时时钟这三个模块之外，其它的模块都需要通过扩展板与板子连接。首先将扩展板与板子连接，注意不要插错位，如图 2.3.3 所示。

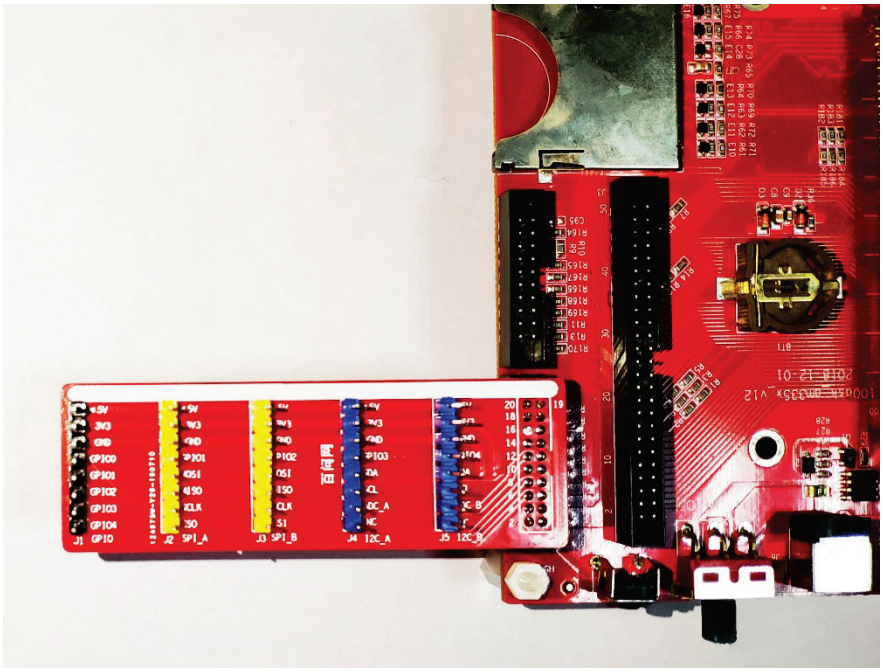

图 2.3.3 扩展板、底板实物连接图

为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。将红外接收模块接在扩展板的任意 GPIO 接口，假设接在 GPIO 接口 0，如图 2.3.4 所示。

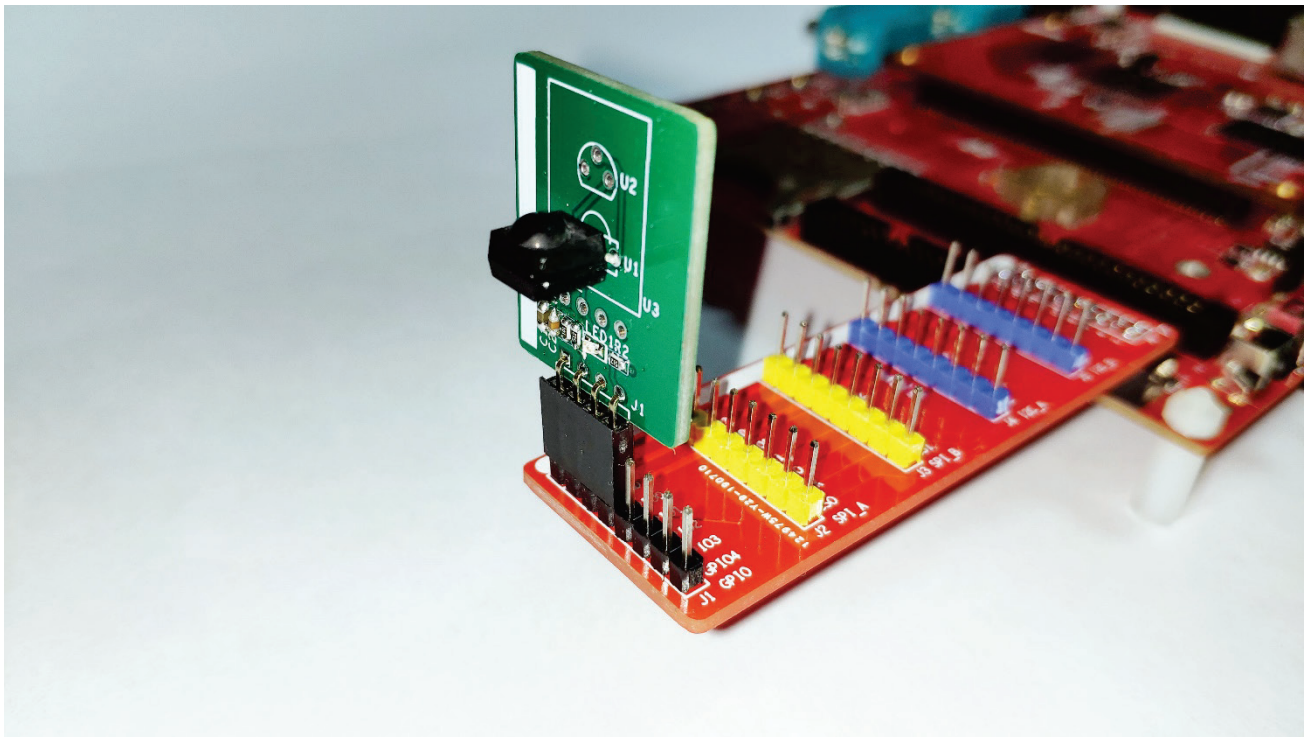

图 2.3.4 红外接收模块、扩展板、底板实物连接图

正常连接好后，红外接收模块上的电源指示灯会点亮。然后准备好配套的红外遥控器，如果是第一次使用红外遥控器，要先取出电池上的隔离薄膜。

在 Arduino 打开菜单“示例”->“GPIO.03.IrDA”->“readIRDA”，代码内容如下：

```
#include <Arduino.h>
#include <irda.h>

int main(int argc, char **argv)
{
    int ret;
    IRDA irda(GPIO0);

    while(1)
    {
        ret = irda.readKey();

        if(ret == 0)
            cout << "Code:" << irda.getCode() << " Value:" << irda.getValue() << endl;
    }

    return 0;
}
```

将示例代码上传，在板子里运行，然后按下红外遥控器上的 1，再松开，可以看到串口打印如下：

```
[root@100ask-am335x:~] ./am335x_app
```

```
Code:2 Value:1
Code:2 Value:1
Code:2 Value:0
Code:2 Value:0
```

可以看到打印的“code”和“value”。“code”用来表示哪一个按键被按下，“value”表示当前按键的状态，和前面的 Keyboard 含义一样。红外遥控器所有按键对应的 code 值参考附录三按键映射表。

简单分析下代码，首先实例化红外遥控 IRDA，需要指定 GPIO 引脚。接着在一个“while(1)”死循环里，使用 irda 的“readKey()”函数，通过返回值判断是否有红外遥控器按键被按下。如果有按键被按下，使用“getCode()”获取被按下的按键编号，通过“getValue()”获取当前按键的状态，是按下、松开还是长按。

除了“readIRDA” 示例，还提供了一个“turnOnLED”示例，实现按下 KEY_1 或者长按 KEY_1 点亮 LED4。

2.4 DHT11 温湿度传感器

下面介绍一个与温度、湿度相关的传感器——DHT11 温湿度传感器。它除了能测量温度以外，还能测量湿度，比如市面上一些空气加湿器，会测量空气中湿度，再根据测量结果决定是否继续加湿。

DHT11 测量温度的精度为±2℃，检测范围为-20℃-60℃。湿度的精度为±5%RH，检测范围为 5%RH-95%RH。

DHT11 的硬件连接和 IrDA 类似，选择任一 GPIO 接口插上即可，假设选择的 GPIO 接口 0，如图 2.4.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

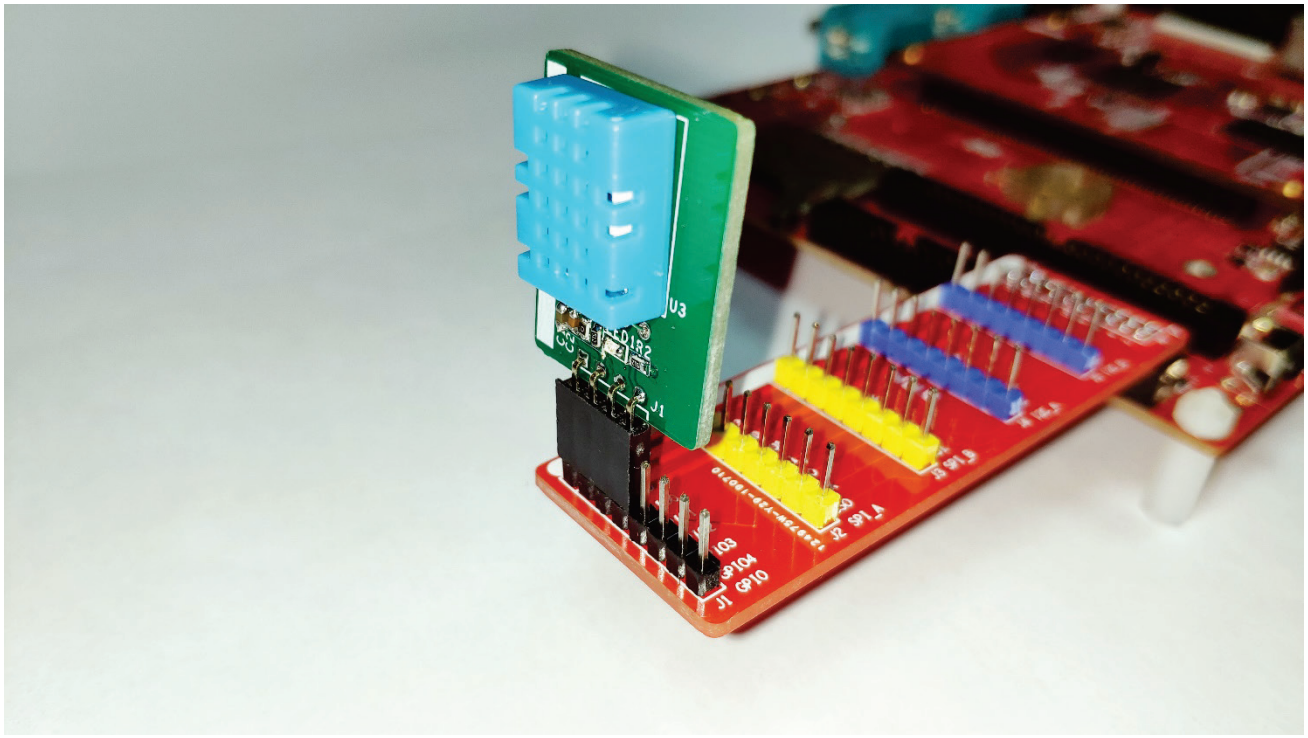

图 2.4.1 DHT11 温湿度传感器、扩展板、底板实物连接图

下面通过示例程序来获取到当前环境温度和湿度。在 Arduino 打开菜单“示例”->“GPIO.04.DHT11”->“readDHT11”，代码内容如下：

```
#include <Arduino.h>
#include <dht11.h>
```

```
int main(int argc, char **argv)
{
    DHT11 dht11(GPIO0);

    while(1)
    {
        cout << "Temperature: " << dht11.readTemperature() << "'C" << endl;
        cout << "Humdity: " << dht11.readHumdity() << "%" << endl;

        sleep(1);
    }

    return 0;
}
```

代码内容也与 IrDA 类似，首先实例化一个 DHT11，结合实物连接，需要指定它是接在哪一个引脚上。

接着在一个死循环里，使用 dht11 读取温度的函数“readTemperature()”读取温度，加上单位后打印输出，使用 dht11 读取湿度的函数“readHumdity()”读取湿度，加上单位后打印输出。

将示例代码上传，在板子里运行，即可看到当前温度、湿度打印信息如下：

```
[root@100ask-am335x:~] ./am335x_app
Temperature: 30'C
Humdity: 76%
```

2.5 DS18B20 温度传感器

前面的 DHT11 温湿度传感器的温度精度比较低，测量范围也比较小，因此再介绍另一款用途更广泛的温度传感器 DS18B20，默认精度是 0.0625℃，测量范围为-55℃~ 125℃。

DS18B20 依旧只使用一个 GPIO，和前面接法类似，选择任一 GPIO 接口插上即可，假设选择的 GPIO 接口 0，如图 2.5.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

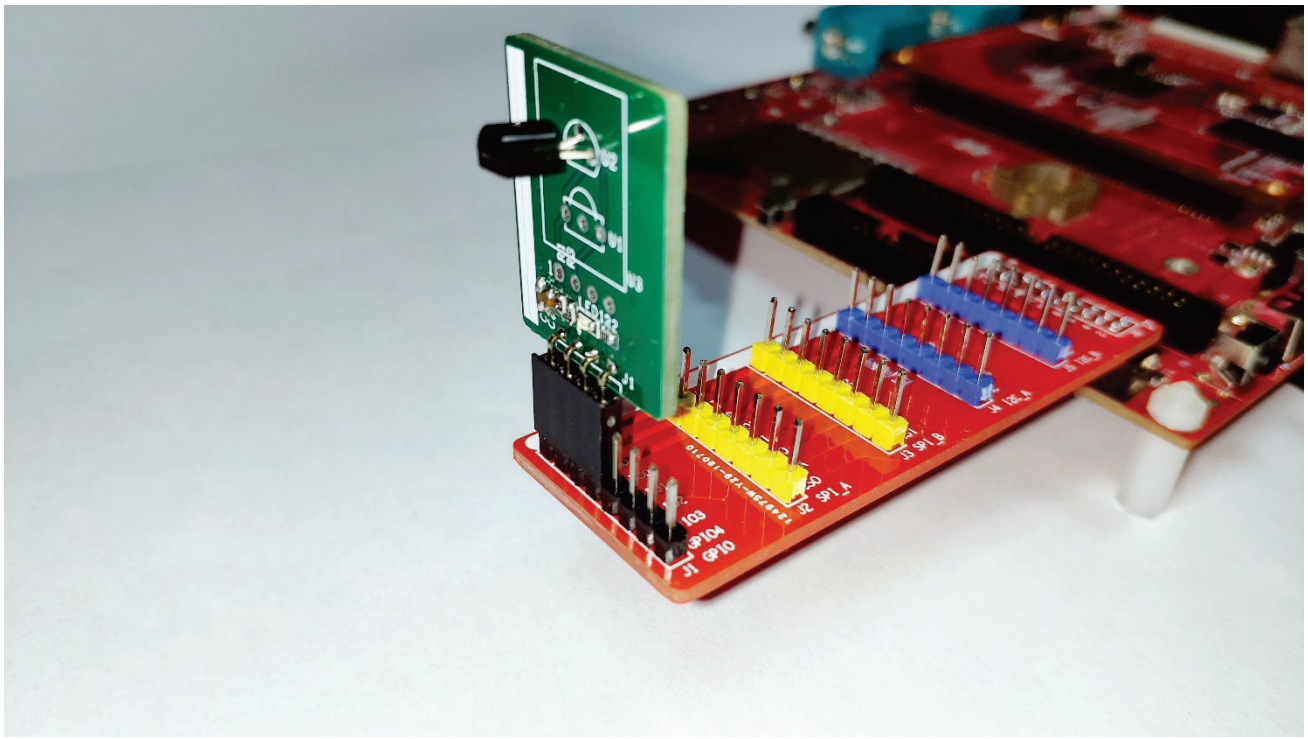

图 2.5.1 DS18B20 温度传感器、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“GPIO.05.DS18B20”->“readDS18B20”，代码内容如下：

```
#include <Arduino.h>
#include <ds18b20.h>

int main(int argc, char **argv)
{
	int i;
	unsigned char SerialNum[6];

	DS18B20 ds18b20(GPIO0);

	ds18b20.readSerialNum(SerialNum);
	cout << "SerialNum: ";
	for (i=0; i<6; i++)
		cout << (int)SerialNum[i];
	cout << endl;

	cout << "Temperature: " << ds18b20.readTemperature() << endl;
	cout << "FamilyCode: " <<    ds18b20.readFamilyCode() << endl;
	cout << "CRC: " <<    ds18b20.readCRC() << endl;

	return 0;
}
```

首先还是实例化 DS18B20，提供了四个函数，分别获取 DS18B20 唯一序列号“SerialNum”、温度值“Temperature”、产品系列号“FamilyCode”，以及用于 CRC 校验的“CRC”。只需要关心温度值“Temperature”即可。

将示例代码上传，在板子里运行，可以看到当前温度：

```
[root@100ask-am335x:~] ./am335x_app
SerialNum: 2171720318250
Temperature: 27.5625
FamilyCode: 40
CRC: 3
```

2.6 SR501 人体红外感应

热释红外传感器是一种能够检测人或动物发射的红外线而输出电信号的传感器。广泛应用于各种自动化控制装置中。比如常见的楼道自动开关、防盗报警等。

人体红外传感器如下图所示，上面有两个可调电阻，一个用于控制感应距离，一个用于控制延时时间。感应距离好理解，即感应人体的距离，能够最大达到 7 米，延时时间是指感应到红外信号后，输出高电平的时间，可调为 0.3 秒至 10 分钟。这两个可调电阻保持默认即可，需要的时候才去调节。

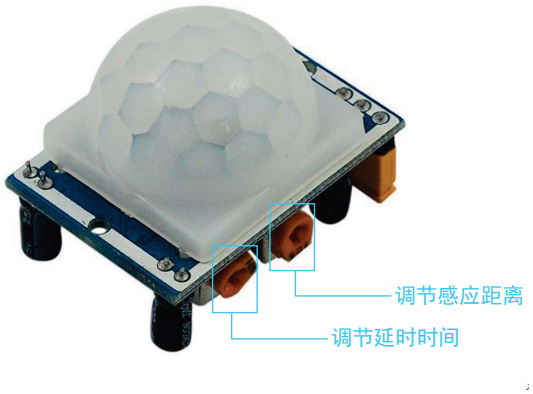

图 2.6.1 红外遥控器和红外接收头实物图

首先将人体红外模块接在扩展板的任意 GPIO 接口，假设接在 GPIO 接口 0。使用附赠的定做杜邦线可以直接连接 SR501 模块和转接板，注意电源(红色线)的方向，如图 2.6.2 所示。

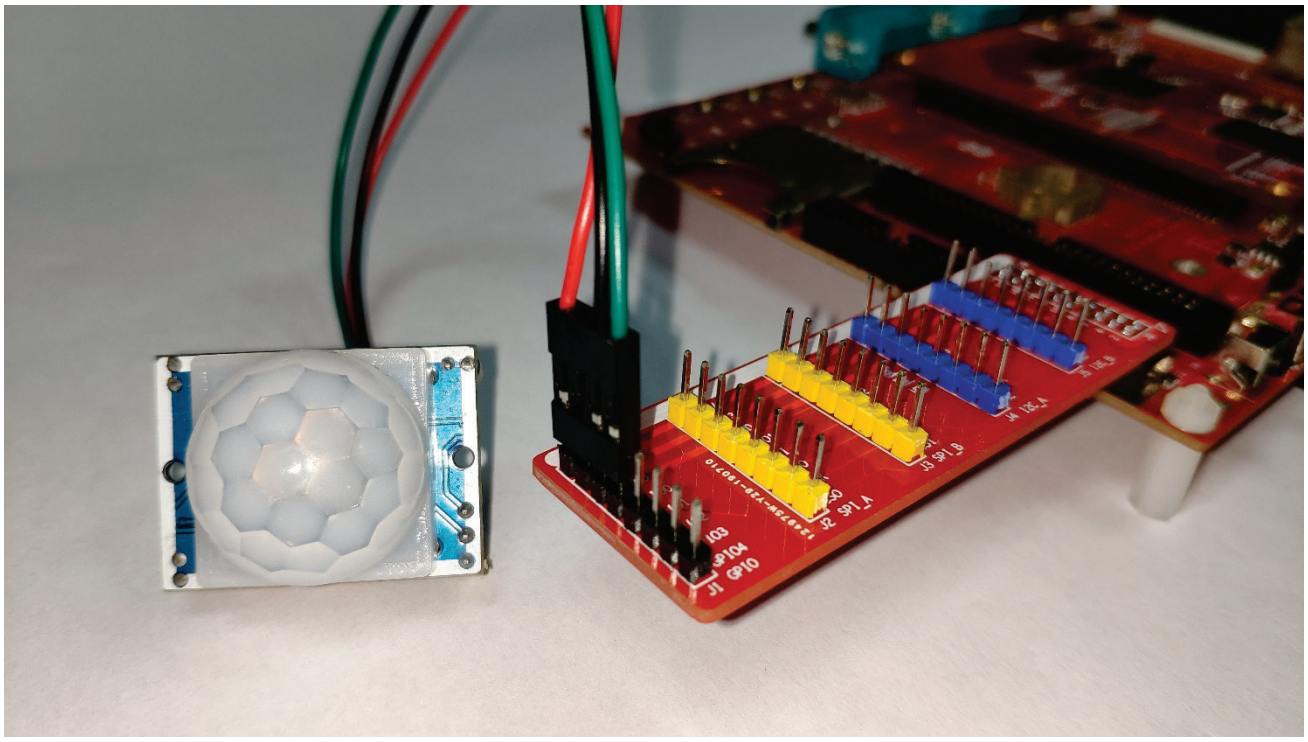

图 2.6.2 SR501 人体红外模块、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“GPIO.06.SR501”->“getSignal”，代码内容如下：

```
#include <Arduino.h>
#include <sr501.h>

int main(int argc, char **argv)
{
    SR501 sr501(GPIO0);

    while(1)
    {
        if(sr501.getSignal() == 1)
            cout << "Detected someone!" <<endl;
        else
            cout << "No one detected!" <<endl;

        msleep(200);
    }

    return 0;
}
```

代码里，首先实例化人体红外传感器 SR501，在 while(1)循环里，使用 sr501 读取信号函数“getSignal()”获取是否有人体红外信号。当有人时，“getSignal()”将返回高电平 1，串口打印“Detected someone!”，直到信号结束，否则打印“No one detected!”。

将示例代码上传，在板子里运行，再将手靠近 SR501 模块，可以看到类似如下打印：

```
[root@100ask-am335x:~] ./am335x_app
No one detected!
No one detected!
No one detected!
No one detected!
No one detected!
No one detected!
Detected someone!
Detected someone!
Detected someone!
```

注意：模块在每次检测后，会持续该状态一段时间，因此每次用手靠近触发后，再等一阵子再去执行“getSignal()”函数进行下一次判断。

2.7 SR04 超声波测距

超声波测距模块是利用超声波来测距。模块先发送超声波，然后接收反射回来的超声波，由反射经历的时间和声音的传播速度 340m/s，计算得出距离。

由于超声波模块需要外接两个 GPIO 引脚，因此只能接在 GPIO0 所在接口上，使用配套杜邦线可直接相连，注意电源(红线)方向，实物连接如下图 2.7.1 所示。

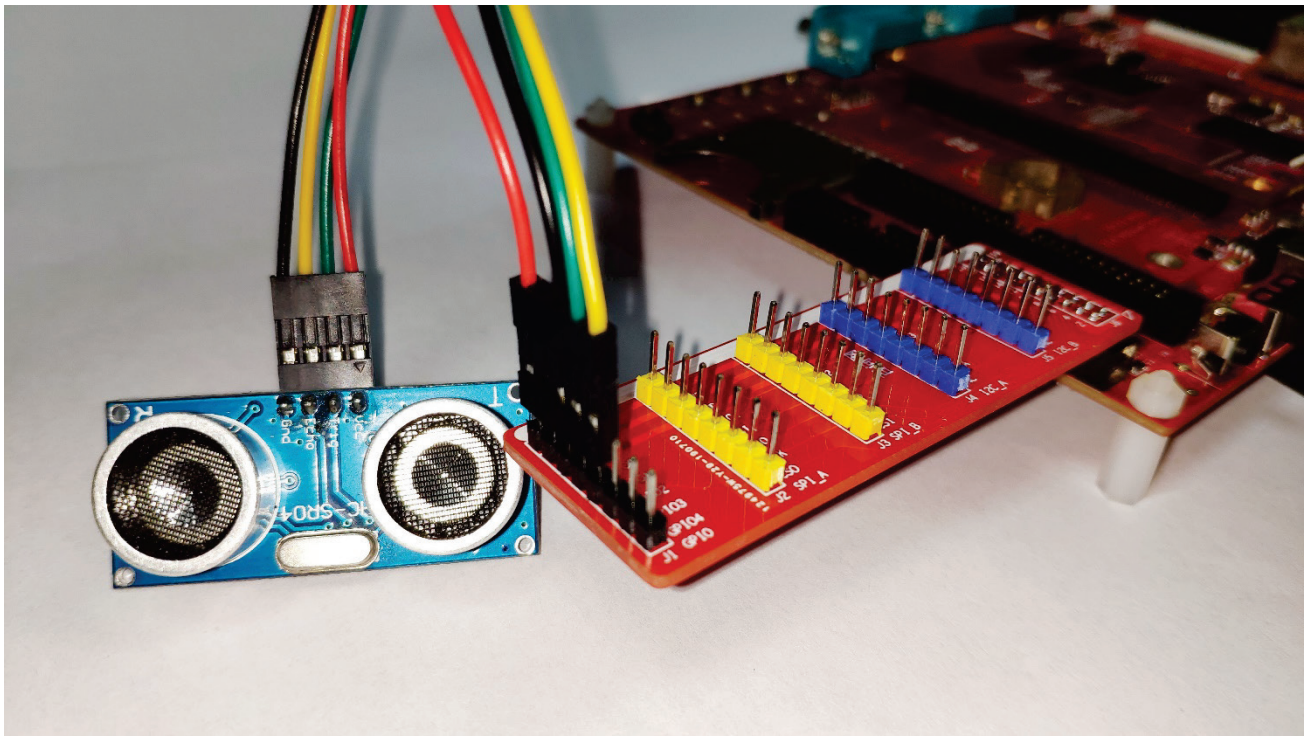

图 2.7.1 SR04 超声波测距模块、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“GPIO.07.SR04”->“getDistance”，代码内容如下：

```
#include <Arduino.h>
#include <sr04.h>
```

```
int main(int argc, char **argv)
{
	SR04 sr04;

	while(1)
	{
		cout << "distance: " << sr04.getDistance() << "cm" <<endl;
		msleep(200);
	}

	return 0;
}
```

代码内容就是先实例化一个超声波测距模块 SR04，然后在一个 while(1)循环里，使用超声波测距模块 sr04 提供的“getDistance()”函数获得距离，再用 cout 打印出来。为了防止发射信号对回响信号的影响，建议每次测量距离要间隔 60ms 以上，这里延时 200ms。

将示例代码上传，在板子里运行，再将手由远到近靠近模块，可以看到如下类似打印：

```
[root@100ask-am335x:~] ./am335x_app
distance: 28cm
distance: 24cm
distance: 21cm
distance: 21cm
distance: 17cm
distance: 14cm
distance: 10cm
distance: 9cm
distance: 10cm
distance: 7cm
distance: 7cm
distance: 3cm
distance: 3cm
```

注意：模块的盲区距离为 2cm，太靠近模块是测量不出来的。

2.8 Motor 步进电机

步进电机是一种将电脉冲转化为角位移的执行机构。通俗的讲，就是当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度。因此可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的，同时可以通过控制脉冲频率来控制电机转动的速度，从而达到调速的目的。

将步进电机与步进电机驱动模块相连，控制步进电机需要四个 GPIO，因此只能直接 GPIO0 上，连接实物图如 2.8.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

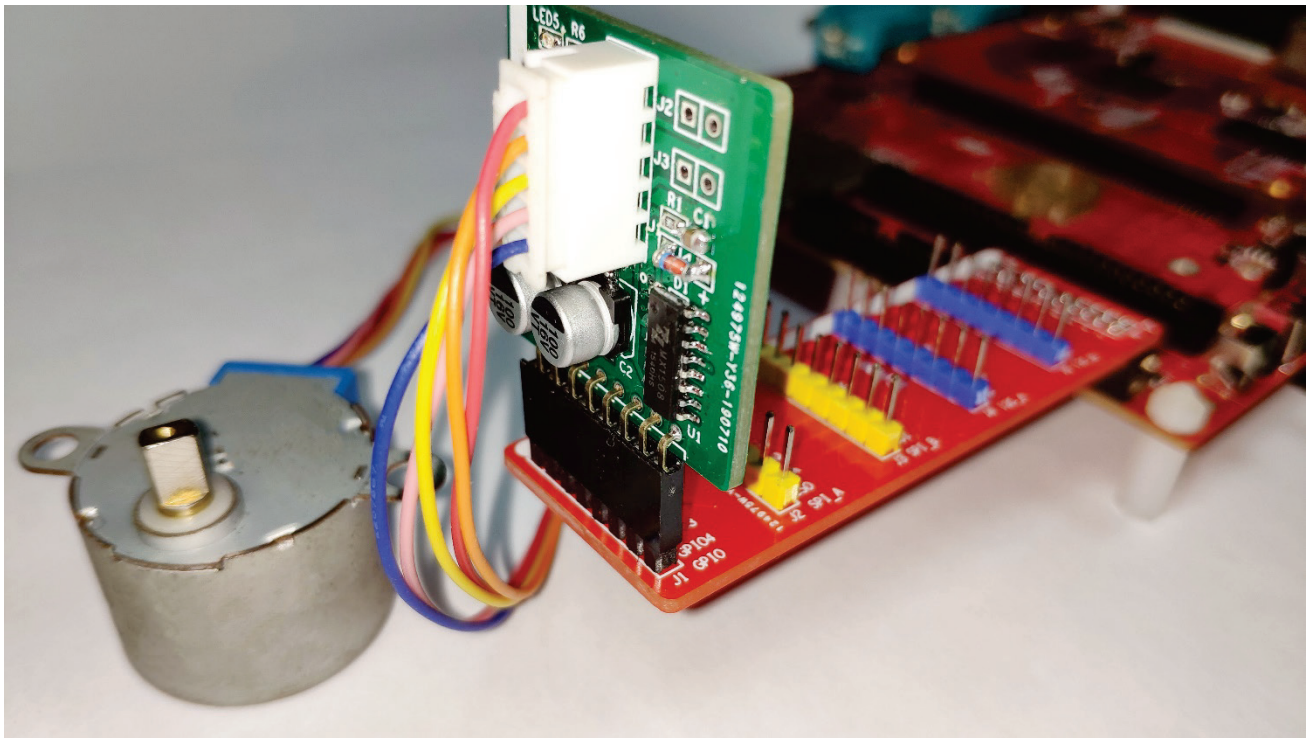

图 2.8.1 步进电机、步进电机驱动板、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“GPIO.08.Motor”->“stepperMotor”，代码内容如下：

```
#include <Arduino.h>
#include <motor.h>

int main(int argc, char **argv)
{
    MOTOR motor;

    motor.setSpeed(2); //1~10

    motor.motorForward(512); //正转一圈
    motor.motorStop(); //停止

    motor.motorReverse(512); //反转一圈
    motor.motorStop(); //停止

    return 0;
}
```

先分析下代码，首先实例化一个步进电机 MOTOR，然后调用它的“setSpeed()”函数设置步进电机转动速度，这里可设置的范围为 1 至 10，共 10 个等级，其中 1 为转动最快，10 为转动最慢。

接着使用“motorForward()”函数使步进电机正向转动，需要传入转动的角度，这里传入 512，刚好就是转动一圈。转动一圈后，使用“motorStop()”让步进电机停止转动。

在使用“motorReverse()”函数使步进电机反向转动，也同样传入 512，使其转动一圈，最后调用“motorStop()”让步进电机停止转动。

将示例代码上传，在板子里运行，即可看到步进电机按前面描述的进行转动转动。

注意：程序发生意外退出时，可能 GPIO 最后仍有输出，导致电机慢慢发热烧毁。因此在不使用电机时，应该取下步进电机或取下步进电机驱动板。

第三章 SPI 接口

前面介绍的模块从功能、接口角度看，都比较简单，大部分都只用一个 GPIO 引脚就能实现功能。实际上，前面的模块只是冰山一角，很多稍微复杂一点的模块，使用多个引脚与板子进行连接。比如后面的 DAC 模块，它使用 SPI 接口与板子连接，需要 4 个引脚。当然，我们不用关心这些细节，只需要把 SPI 接口的模块插到扩展板的 SPI 接口即可。

3.1 OLED 显示

板子适配一块 4.3 寸的 LCD，它的接口、操作都比较复杂。这里介绍 SPI 接口的 OLDE，它的显示空间虽然比较小，但成本低、功耗低，可以用来显示一些重要信息，或者作为副屏显示。

OLED 使用 SPI 接口，将其插在任意一个 SPI 接口，假设这里插在 SPI_A 上，如图 3.1.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

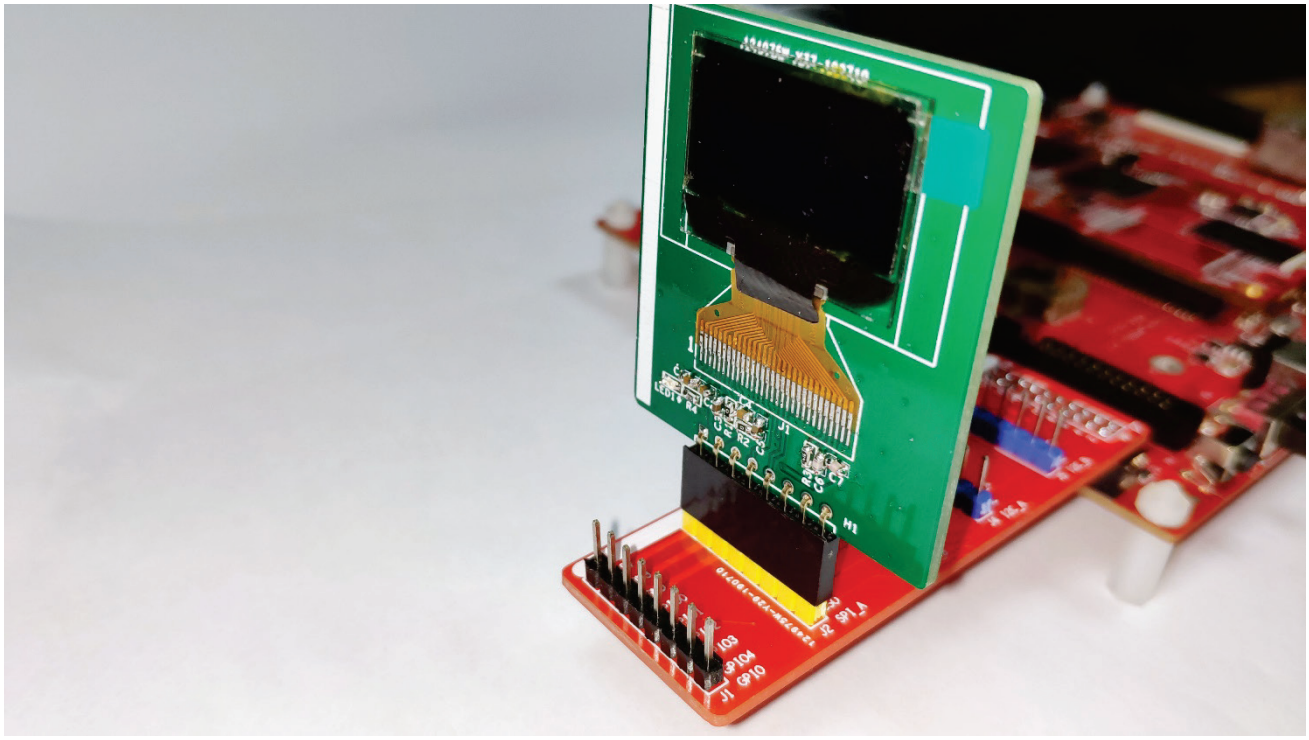

图 3.1.1 OLED 显示模块、扩展板、底板实物连接图

在写代码前，先介绍下字符和屏幕大小关系。

可以简单地认为，OLED 屏幕长为 16，宽为 8；而每一个字母或数字，长为 1，宽为 2。

所以，在 OLED 上可以显示最多 4 行文字，每行可以显示最多 16 个字符。

我们编写程序时，当某一行超过 16 个字符后，会自动换行继续显示；如果是在最后一行显示文字，将忽略多余文字。

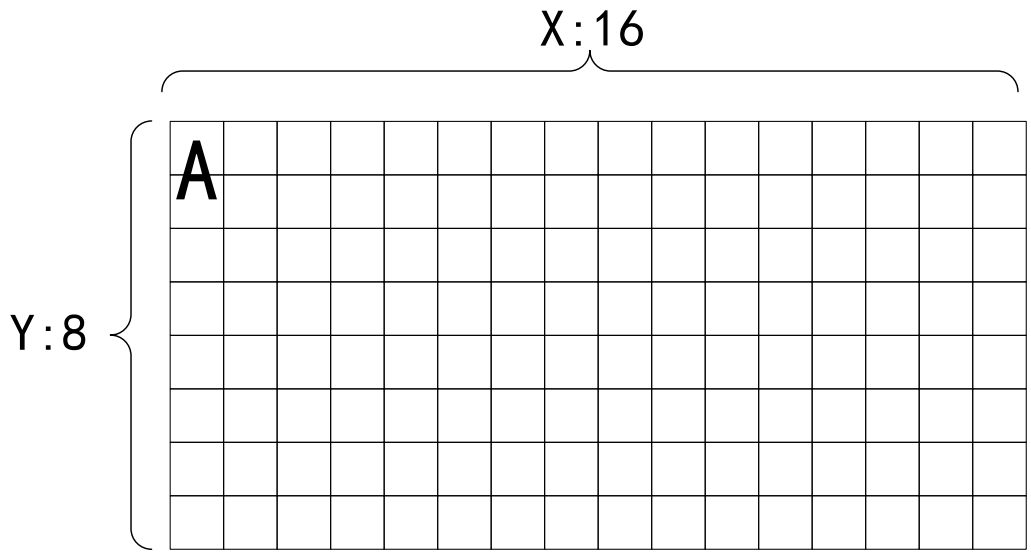

图 3.1.2 字符和屏幕大小关系示意图

在 Arduino 打开菜单“示例”->“SPI.01.OLED”->“oledDisplay”，代码内容如下：

```
#include <Arduino.h>
#include <oled.h>
int main(int argc, char **argv)
{
    OLED oled(SPI_A);
    int y = 0;

    while(1)
    {
        if(y < 8)
        {
            oled.OLEDPrint(0, y, "1234567890123456");
            y += 2;
            sleep(1);
        }
        else
        {
            y = 0;
            oled.OLEDClearAll();
            sleep(1);
        }
    }

    return 0;
}
```

首先实例化一个 OLED，需要指定接在哪一个 SPI 接口，这里插在 SPI_A 上，传入 SPI_A，如果插在 SPI_B 上，则传入 SPI_B。

接着在死循环里，使用“OLEDPrint()”函数显示想显示的内容，需要传入三个参数，第一个是横向坐标，第二个是纵向坐标，第三个是想显示的内容。这里为“ (0, y, "1234567890123456") ”，y 会逐次加 2，直到超过 8，又变为 0，一直循环。因此在一个完整的循环周期，相当于依次调用“OLEDPrint(0, 0, "1234567890123456") ”、“ OLEDPrint(0, 2, "1234567890123456") ”、“ OLEDPrint(0, 4, "1234567890123456")”、“OLEDPrint(0, 6, "1234567890123456")”，即间隔 1 秒，依次每行显示“1234567890123456”，共显示四行，最后调用“OLEDClearAll()”清除所有显示。

将示例代码上传，在板子里运行，即可看到 OLED 显示前面代码描述的效果。

3.2 DAC 数模转换

前面通过 GPIO 控制了 LED 灯的亮灭，但在生活中，有时不仅仅是控制 LED 灯的亮灭，还需要控制 LED 灯的亮度，比如常见的 LED 呼吸灯，灯光的亮度在亮和灭之间逐渐变化，感觉好像是人在呼吸。

控制 LED 灯亮度方式一般有两种，一种通过 PWM 控制脉冲的占空比，一种是输出模拟电压改变灯两端的电压，实现亮度控制。

这里我们使用 DAC 模块：程序通过 SPI 接口给 DAC 模块发送命令，DAC 模块根据命令改变输出电压，从 0V 依次到 4.092V 依次增加，实现 LED 缓缓变亮。

DAC 模块使用的 SPI 接口，将其插在任意一个 SPI 接口，假设这里插在 SPI 接口 A 上，连接实物图参考 3.2.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

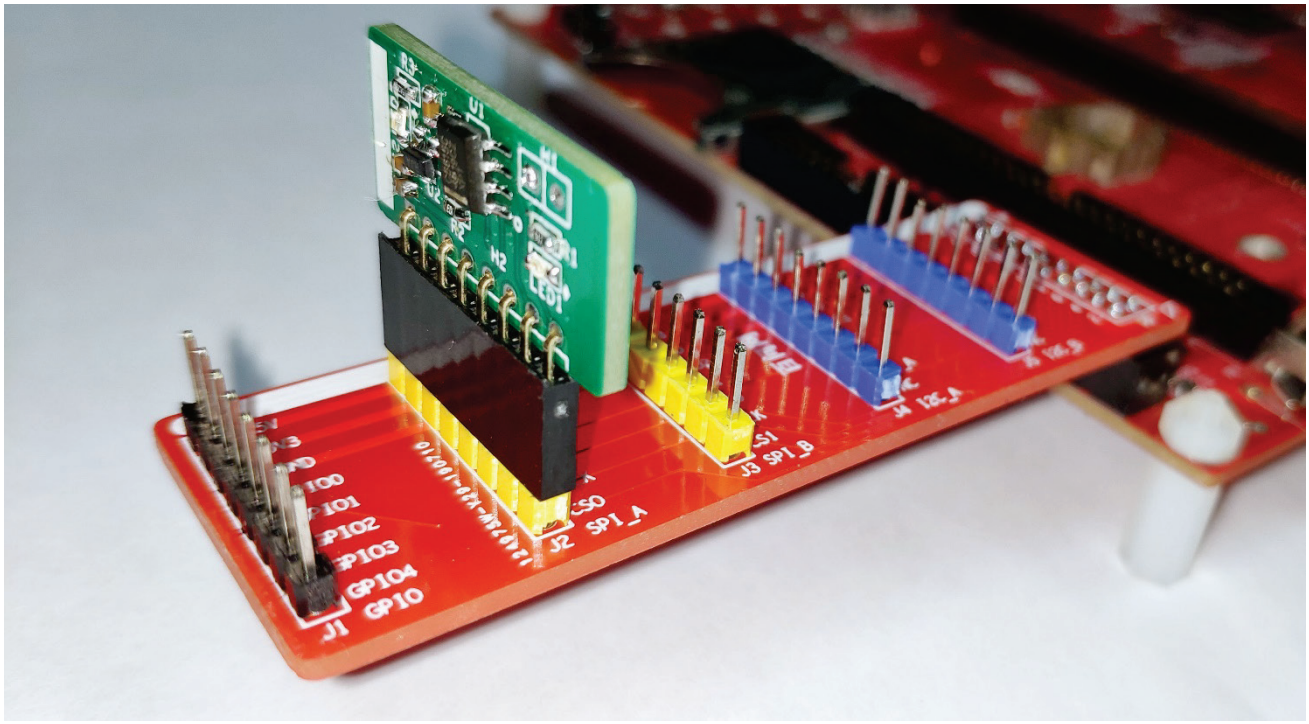

图 3.2.1 DAC 模块、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“SPI.02.DAC”->“breathingLight”，代码内容如下：

```
#include <Arduino.h>
```

```
#include <dac.h>

int main(int argc, char **argv)
{
    DAC dac(SPI_A);
    float vol = 1.7;//vol range: 0 ~ 4.092

    while(1)
    {
        while(vol < 3.6) //逐渐变亮
        {
            dac.setVoltage(vol);
            vol = vol + 0.01;
            msleep(5);
        }

        while(vol > 1.7) //逐渐变暗
        {
            dac.setVoltage(vol);
            vol = vol - 0.01;
            msleep(5);
        }
    }

    return 0;
}
```

首先实例化一个 DAC，需要指定接在哪一个 SPI 接口，这里插在 SPI_A 接口，因此传入 SPI_A。接着定义了一个 float 类型(即为小数)的变量 vol,表示要设置的电压,并设置初值为 1.7。之所以设置初值为 1.7V，是因为电压在 0～1.7V 之间，LED 都是熄灭状态，看不出渐变的效果。

然后在“while(1)”循环里，先是一个“while(vol < 3.6)”子循环，如果 vol 小于 3.6，就调用“setVoltage()”设置当前 vol 电压，然后 vol 每次加 0.1，直到等于 3.6。这样就实现了输出电压每次递增 0.1V，LED 逐渐变亮的效果。

接着是另一个“while(vol > 1.7)”子循环，vol 也依次减小 0.1，调用“setVoltage()”设置当前 vol 电压。这样就实现了输出电压每次递减 0.1V，LED 逐渐变暗的效果。

将示例代码上传，在板子里运行，即可看到 DAC 模块上的 LED 呼吸灯效果。

3.3 ADXL345 三轴加速度计

ADXL345 是一款小而薄的超低功耗 3 轴加速度计，可以用来测量加速度，或者检测倾斜、冲击、振动等运动状态，在工业、医疗、通信、消费电子和汽车等领域中有很多应用。这里提供了两个示例程序，一个用于检测冲击，另一个用于输出 XYZ 轴的绝对位置。

ADXL345 模块使用 SPI 接口，将其插在任意一个 SPI 接口，假设这里插在 SPI 接口 A 上，连接实物图参考 3.3.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

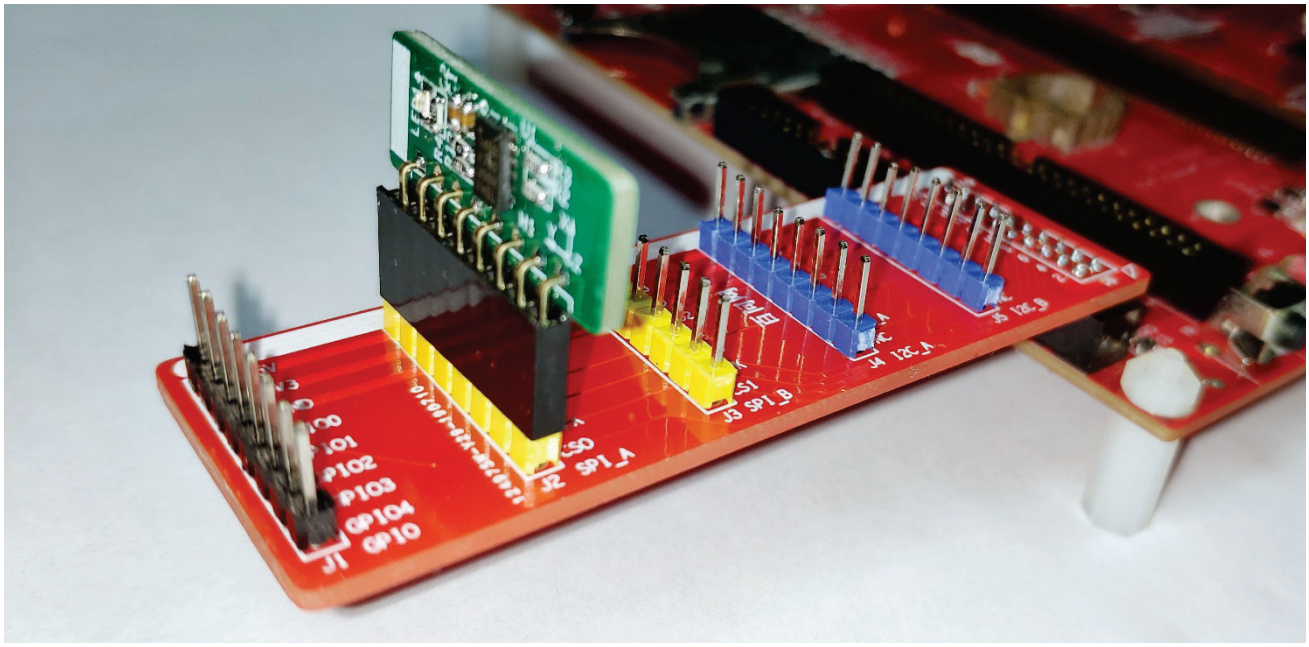

图 3.3.1 ADXL345 模块、扩展板、底板实物连接图

在 Arduino 打开菜单“示例”->“SPI.03.ADXL345”->“detectionTouch”，代码内容如下：

```
#include <Arduino.h>
#include <adxl345.h>

int main(int argc, char **argv)
{
    ADXL345 adxl345(SPI_A);

    while(1)
    {
        adxl345.readData();

        while(adxl345.getTouchValue() == 0)
        {
            adxl345.readData();
            if(adxl345.getTouchValue() == 1)
                cout << "You hit it" << endl;
        }
    }

    return 0;
}
```

首先实例化一个 ADXL345，需要指定接在哪一个 SPI 接口上，这里插在 SPI_A 上，传入 SPI_A。

接着在死循环里，使用“readData()”函数去获取三轴加速度计当前状态所有数据，根据“getTouchValue()”的值判断是否发生触摸，当返回值为 1 时，则表示发生触碰，打印“You hit it”。

将示例代码上传，在板子里运行，触碰一下 ADXL345 模块，会看到如下打印信息：

```
[root@100ask-am335x:~] ./am335x_app
You hit it
You hit it
```

在 Arduino 打开菜单“示例”->“SPI.03.ADXL345”->“readData”，代码内容如下：

```
#include <Arduino.h>
#include <adxl345.h>

int main(int argc, char **argv)
{
    ADXL345 adxl345(SPI_A);

    while(1)
    {
        adxl345.readData();

        cout << "X: " << setw(4) << adxl345.getXValue()
          <<   "   Y: " << setw(4) << adxl345.getYValue()
          <<   "   Z: " << setw(4) << adxl345.getZValue() << endl;

        msleep(200);
    }

    return 0;
}
```

示例“readXYZData”打印了 XYZ 轴绝对位置，其中“setw(4)”用于格式化输出，表示占据四个位置。

将示例代码上传，在板子里运行，当用手移动、翻转 ADXL345 模块时，会看到如下类似打印信息：

```
[root@100ask-am335x:~] ./am335x_app
X: -237   Y: -87   Z: -37
X: -237   Y: -88   Z: -37
X: -237   Y: -88   Z: -39
X: -237   Y: -88   Z: -39
X: -238   Y: -88   Z: -39
X: -238   Y: -88   Z: -39
X: -238   Y: -89   Z: -39
X: -238   Y: -89   Z: -39
X: -232   Y: -89   Z: -39
```

ADXL345 就暂时介绍到这里，可以参考示例程序的代码去扩展其它应用场景。

第四章 I2C 接口

与 SPI 接口相似，I2C 接口也是常见的接口，许多模块都使用该接口与 ARM 主控通信。下面来介绍几个使用 I2C 接口的模块。

4.1 EEPROM 存储模块

EEPROM 本质是 ROM 的一种，一般存储容量比较小，价格低廉。作为非易失性存储器，即使在掉电的状态下数据也不会丢失，常用于存储配置信息。比如将电视机中有关亮度、对比度、音量等用户个性化设置存储在 EEPROM 里，即使遇到突然停电，再打开电视时之前的设置数据也不会丢失，给用户带来了方便。

扩展板上有两个 I2C 接口，这里假设使用 I2C_A 接口，如图 4.1.1 所示连接。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

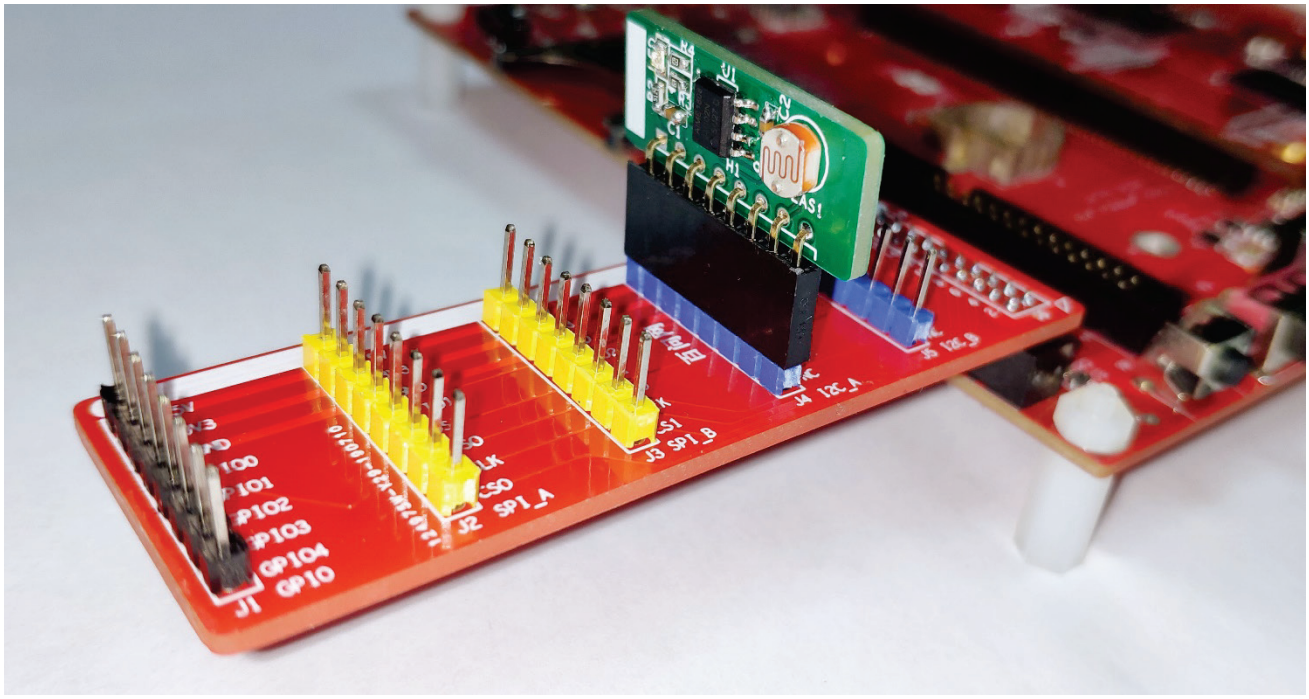

图 4.1.1 EEPROM 模块、扩展板、底板连接实物图

“readWriteArray”用来演示数组的保存和读取，先写入数组数据到 eeprom，再从 eeprom 读取数组数据。在 Arduino 打开菜单“示例”->“I2C.01.EEPROM”->“readWriteArray”，代码内容如下：

```
#include <Arduino.h>
#include <eeprom.h>

int main(int argc, char **argv)
{
    int i, ret;
    int txArray[5] = {10, 20, 30, 254, 255};
    int rxArray[5];
    int txLen = sizeof(txArray)/sizeof(txArray[0]);
    int rxLen = sizeof(rxArray)/sizeof(rxArray[0]);
```

```
    EEPROM eeprom(I2C_A);

    ret = eeprom.writeEEPROM(0, txLen, txArray);
    if (ret < 0)
    {
        cout << "writeEEPROM error" << endl;
        return -1;
    }

    ret = eeprom.readEEPROM(0, rxLen, rxArray);
    if (ret < 0)
    {
        cout << "readEEPROM error" << endl;
        return -1;
    }

    cout << "rxArray: ";
    for (i=0; i<rxLen; i++)
        cout << rxArray[i] << " " ;
    cout << endl;

    return 0;
}
```

首先定义待发送的数组 txArray 和设置待发送的数据，定义接收数组。

接着计算发送和接收长度。

实例化 EEPROM，需要传入所接的 I2C 接口编号，本实例是接在 I2C_A 接口，因此传入 I2C_A；如果是接在 I2C_B 接口，则传入 I2C_B。

然后使用“writeEEPROM()”函数写数据到 eeprom，并根据返回值判断是否写成功，函数需要三个参数，第一个参数为要写的 eeprom 地址，第二个为数组长度，第三个为含有待写数据的数组。

再接着调用“readEEPROM()”函数从 eeprom 读出数据，函数也需要三个参数，第一个参数为要读的 eeprom 地址，第二个为读取数组长度，第三个为用来保存读取数据的数组。

最后打印读出的数据，看写入 eeprom 的数组数据和读出来的数组数据是否一致。

将示例代码上传，在板子里运行，会看到如下打印内容：

```
[root@100ask-am335x:~] ./am335x_app
rxArray: 10 20 30 254 255
```

“readWriteString”是另一个示例，用来演示字符串的保存和读取；先写入字符串数据到 eeprom，再从 eeprom 读取字符串数据。在 Arduino 打开菜单“示例”->“I2C.01.EEPROM”->“readWriteString”，代码内容如下：

```
#include <Arduino.h>
#include <eeprom.h>

int main(int argc, char **argv)
```

```
{
    int ret;
    string txString = "www.100ask.net";
    string rxString;

    EEPROM eeprom(I2C_A);

    ret = eeprom.writeEEPROM(0, txString.size(), txString);
    if (ret < 0)
    {
        cout << "writeEEPROM error" << endl;
        return -1;
    }

   ret =   eeprom.readEEPROM(0, txString.size(), rxString);
    if (ret < 0)
    {
        cout << "readEEPROM error" << endl;
        return -1;
    }

    cout << "rxString: " << rxString << endl;

    return 0;
}
```

首先定义待发送的字符串 txString 和设置待发送的数据，定义用来接收字符串的 rxString。

然后实例化 EEPROM，并使用“writeEEPROM()”函数写数据到 eeprom。函数需要三个参数，第一个参数为要写的 eeprom 地址，第二个为字符串长度，第三个为要写的字符串。

接下来调用“readEEPROM()”函数从 eeprom 读出数据，函数也需要三个参数，第一个参数为要读的 eeprom 地址，第二个为字符串长度，第三个为数组（用来保存读出的字符串）。

最后打印读出的数据，看写入 eeprom 的字符串数据和读出来的字符串数据是否一致。

将示例代码上传，在板子里运行，会看到如下打印内容：

```
[root@100ask-am335x:~] ./am335x_app
rxString: www.100ask.net
```

4.2 RTC 实时时钟

RTC 是实时时钟(Real-Time Clock)的缩写。它可以像时钟一样实时的输出时间，包括年、月、日、星期、时、分、秒。配合纽扣电池，在模块断开电源的时候，仍能继续计时，在下次上电后继续提供准确的时间。

核心板上有一个 RTC 芯片，不再需要外接模块，它相当于接在了 I2C_A 上。

在 Arduino 打开菜单“示例”->“I2C.02.RTC”->“readTime”，代码内容如下：

```
#include <Arduino.h>
#include <rtc.h>

int main(int argc, char **argv)
{
    RTC rtc(I2C_A);

    while(1)
    {
        rtc.timePrintf();
        sleep(1);
    }

    return 0;
}
```

代码内容比较简单，先实例化一个 RTC（板载接在 I2C_A 上,只能传参数 I2C_A），然后在死循环里，间隔 1 秒调用“timePrintf()”函数打印一次时间。

将示例代码上传，在板子里运行，效果如下：

```
[root@100ask-am335x:~] ./am335x_app
a power voltage drop was detected, you may have to readjust the clock
Set the default time:2000-01-01 00:00:00, you may have to readjust the clock
Time: 2000-01-01 00:00:00 [Sat]
Time: 2000-01-01 00:00:01 [Sat]
Time: 2000-01-01 00:00:02 [Sat]
Time: 2000-01-01 00:00:03 [Sat]
```

该 RTC 模块首次运行时，会自动将时间设置为 2000-01-01 00:00:00。

下面尝试下修改时间，在 Arduino 打开菜单“示例”->“I2C.02.RTC”->“writeTime”，代码内容如下：

```
#include <Arduino.h>
#include <rtc.h>

int main(int argc, char **argv)
{
    int ret;
    RTC rtc(I2C_A);

    Time t;
    t.year = 2019;
    t.month = 6;
    t.day = 3;
    t.week = 1;

    t.hour = 23;
```

```
    t.minute = 59;
    t.second = 57;

    ret = rtc.setTime(t);
    if (ret < 0)
    {
        cout << "setTime error" << endl;
        return -1;
    }

    while(1)
    {
        rtc.timePrintf();
        sleep(1);
    }

    return 0;
}
```

先实例化 RTC，定义结构体 t，接着设置该结构体的成员 year 年、month 月、day 日、week 星期、hour 小时、minute 分、second 秒，最后调用“setTime()”将前面设置的数据写入 RTC，再调用“timePrintf()”读出时间验证。

将示例代码上传，在板子里运行，效果如下：

```
[root@100ask-am335x:~] ./am335x_app
Time: 2019-06-03 23:59:57 [Mon]
Time: 2019-06-03 23:59:58 [Mon]
Time: 2019-06-03 23:59:59 [Mon]
Time: 2019-06-04 00:00:00 [Tue]
Time: 2019-06-04 00:00:01 [Tue]
Time: 2019-06-04 00:00:02 [Tue]
```

第五章 其它接口

这一章继续介绍一些其它接口，包含 UART 和 DAC。

5.1 GPS 模块

在第一章的“1.4.1 串口工具软件”里，已经接触过了串口(也称为 UART)，它作为“桥梁”让我们可以发送命令给板子，也能接收从板子发过来的调试信息。在板子上不止一个串口，因此我们可以使用其它串口连接其它模块，收发数据。

GPS 模块使用 UART 接口上报 GPS 数据。该模块操作简单，每隔 1S 自动发送 NMEA-0183 标准的数据。100ASK_AM335X 的 UART 接口在底板通过防呆插座引出，使用配套的 1 米长连接线将 GPS 模块与底板连接。

因为 GPS 模块需要放在室外无遮挡的环境才能更好的工作，因此配套长达 1 米的连接线以方便使用。连接效果如图 5.1.1 所示。

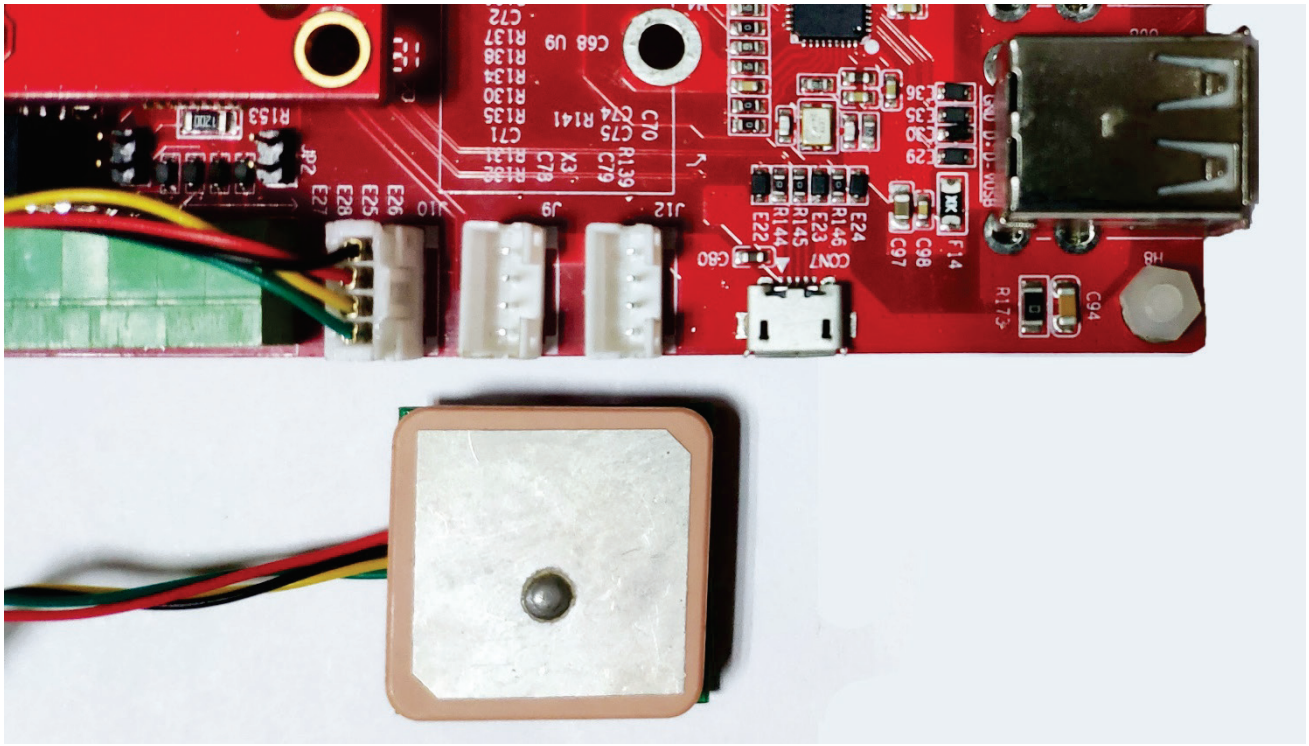

图 5.1.1 GPS 模块、底板连接实物图

注意这里只能连接图示所使用的 UART_A(对应着实际的串口 2，图中最左边的白色接口)，旁边两个分别用于调试、复用 GPIO，为了避免冲突，这里使用 UART_A。

在 Arduino 打开菜单“示例”->“Others.01.GPS”->“printfAnalyzeData”，代码内容如下：

```
#include <Arduino.h>
#include <gps.h>

int main(int argc, char **argv)
{
    GPS gps(UART_A);
```

```
    while(1)
    {
        gps.printfAnalyzeData();
    }

    return 0;
}
```

代码里首先示例化了一个 GPS 对象，只能传入参数 URAT_A。

然后在一个“while(1)”循环里，调用“printfAnalyzeData()”打印经过分析转换后的经纬度数据，如果想 NMEA-0183 格式的原始数据，可以参考示例“printfRawData”。

将示例代码上传，在板子里运行；并将 GPS 模块移至空旷开阔的地方，因为第一次是冷启动，需要重新搜索卫星定位，可能花费 3-5 分钟，效果如下：

```
[root@100ask-am335x:~] ./am335x_app
Invalid data, Please move the module to the open area and wait 3-5 minutes.
Invalid data, Please move the module to the open area and wait 3-5 minutes.
Invalid data, Please move the module to the open area and wait 3-5 minutes.
……
22.65146 , 114.11948
22.65146 , 114.11948
22.65146 , 114.11948
22.65146 , 114.11948
```

打开 GPS 位置查询网址，比如 GPSSPG 经纬度（http://www.gpsspg.com/maps.htm），输入前面经过换算的 GPS 坐标，验证是否正确，如图 5.1.2 所示。

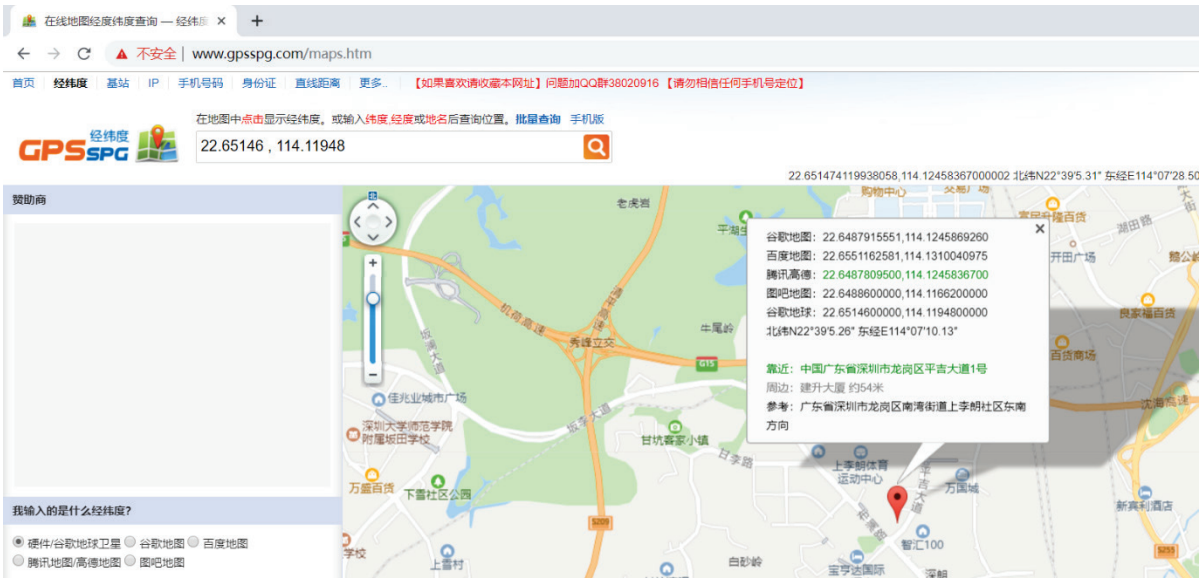

图 5.1.2 验证 GPS 模块数据

5.2 ADC 模数转换

前面第三章 3.2 介绍了 DAC，这里介绍一个与它效果相反的 ADC(Analog-to-Digital Converter)模数转换器。自然界的信号几乎都是模拟信号，比如光亮、温度、压力、声音；而为了方便存储、处理，计算机里面都是数字的 0/1 信号，将模拟信号转换成数字信号的装置就叫模数转换器(ADC)。

举个例子，光敏电阻的电路图如图 5.2.1 所示：

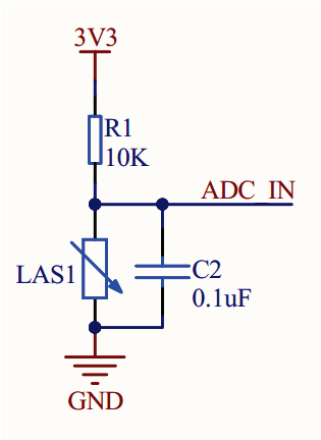

图 5.2.1 光敏电阻模块接线图

电路图中最上面是 3.3V 电源，往下是电阻 R1 和光敏电阻 LAS1，两个电阻之间，与 am335x 的 ADC 引脚相连，最下面是接地，一旁的电容 C2 可以暂时忽略(它起稳压作用，防止电压突然变化)。

当光敏电阻 LAS1 受到光照时，阻值会变小，此时它两端的电压会变小；当光照比较弱时，阻值会变大，导致它两端的电压也变大。假设上图中的 ADC_IN 接到某个 GPIO 引脚，通过读 GPIO 引脚只能得到 0 或 1 两个值之一，只能判断光照在某个阀值上下。如果使用 ADC 去读取光敏电阻两端电压，不仅可以知道是否有光照，还能知道光照的强度。

几乎所有的主控芯片都有 ADC，am335x 也不例外，并且在扩展板接口上引了出来。扩展板上有两个 ADC 接口，分别在 I2C_A 所在列和 I2C_B 所在列，如图 5.2.2 所示。

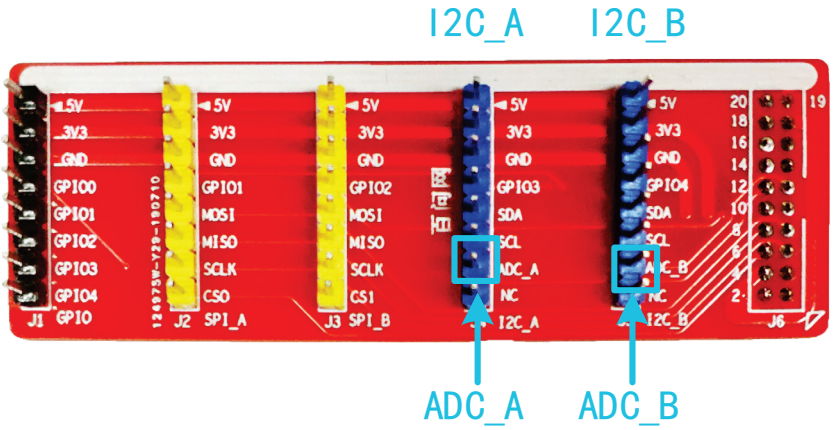

图 5.2.2 扩展板 ADC 引脚位置

光敏电阻与前面的 EEPROM 制作在了一起，直接将 EEPROM 模块接在 I2C 接口即可，如图 5.2.3 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

图 5.2.3 EEPROM 模块各元件位置

这里假设将光敏电阻模块插在 I2C_A 接口上，实物连接图如图 5.2.4 所示。

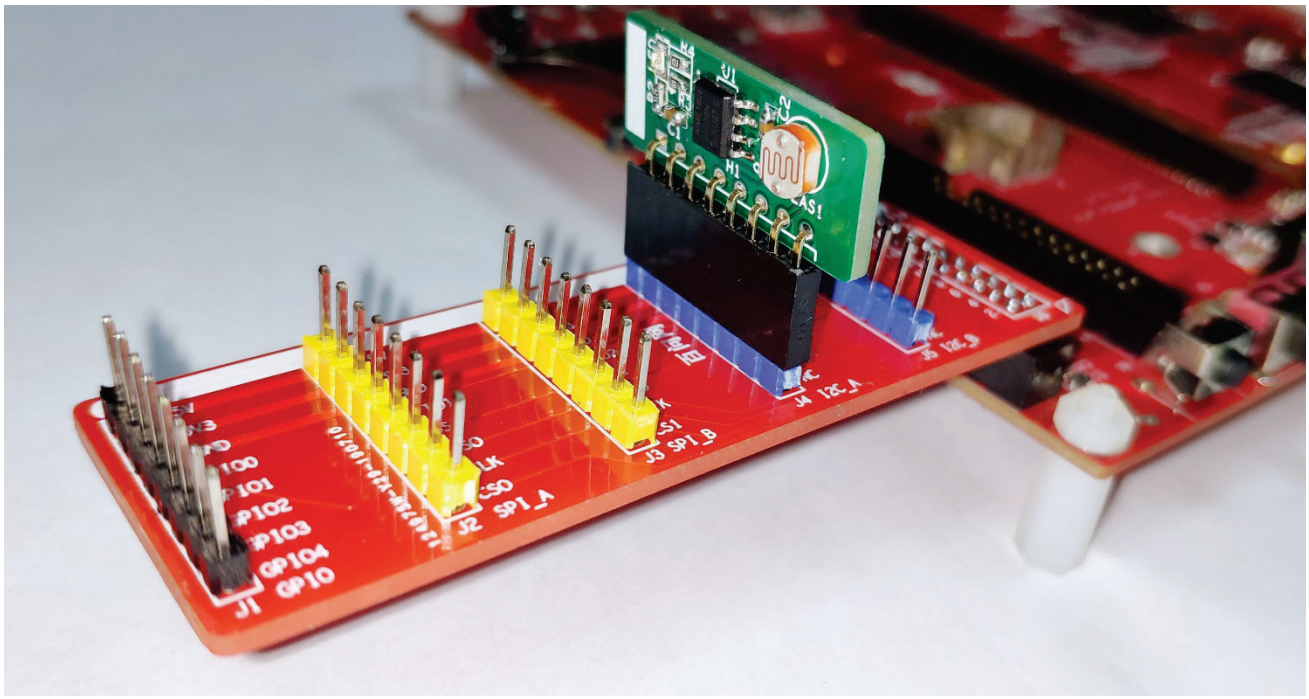

图 5.2.4 EEPROM 模块、扩展板、底板连接实物图

在 Arduino 打开菜单“示例”->“Others.02.ADC”->“readBringhtness”，代码内容如下：

```
#include <Arduino.h>
#include <adc.h>

int main(int argc, char **argv)
{
    ADC adc(ADC_A);

    while(1)
    {
        cout << "voltage: " << adc.getValue()    << "mv" << endl;
        sleep(1);
```

```
    }

    return 0;
}
```

先实例化一个 ADC 对象，传入参数为对应的 ADC 接口编号。然后调用“getValue()”函数得到电压值，再通过 cout 打印出来。

将代码重新编译、上传、运行，便可以看到打印的电压值，此时用手捂住光敏电阻模块，挡住光照，可以看到电压值明显上升，效果如下：

```
[root@100ask-am335x:~] ./am335x_app
voltage: 959mv
voltage: 960mv
voltage: 962mv
voltage: 961mv
voltage: 1100mv
voltage: 1469mv
voltage: 1456mv
voltage: 1458mv
voltage: 1481mv
```

第六章 项目示例

这一章介绍两个实战项目，希望可以抛砖引玉，给读者一些启发。

6.1 OLED 硬件监控

前面 2.5 DS18B20 温度传感器实验里，是通过串口来显示温度结果。使用串口观察结果，一般用于调试程序。实际应用中通过电脑串口查看并不方便，因此需要换个显示途径。前面 OLED 实验里提到了，OLED 可以作为副屏显示一些简单的数据，这里刚好可以用 OLED 显示。OLED 除了可以显示外接的 DS18B20 模块产生的温度信息，也可以显示 Linux 自身信息，比如 CPU 利用率、RAM 利用率等。

第一个项目的内容就是通过 OLED 显示板子上运行的 Linux 系统的 CPU、RAM 利用率，以及 DS18B20 温度传感器模块测出的的温度值。

6.1.1 硬件连接

硬件连接比较简单，只需要将 OLED 模块和 DS18B20 模块接到扩展板上，扩展板接到底板的 GPIO 扩展接口上。假设 DS18B20 模块接在了 GPIO0 上，OLED 模块接在了 SPI_A 上，如图 6.1.1 所示。为了防止用户接错方向，模块和扩展板都有一条长白线，连接时长白线在同一侧。

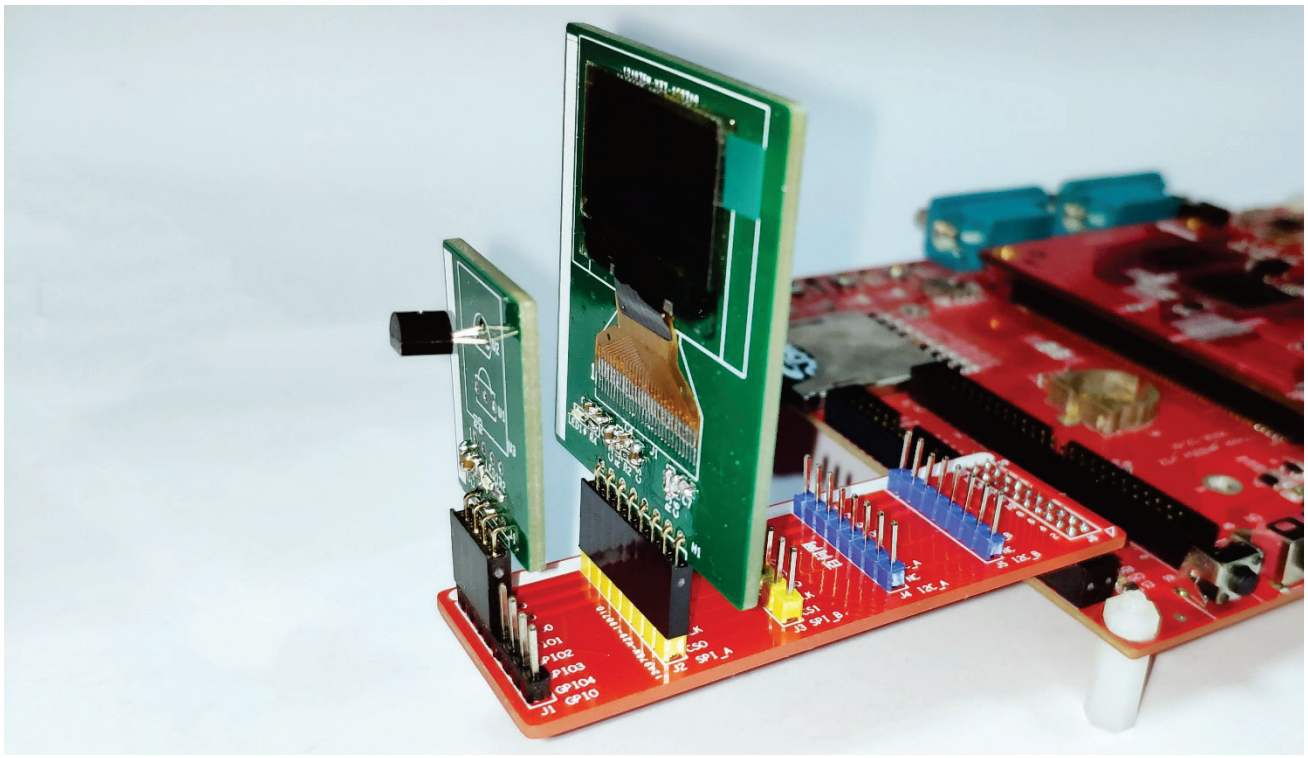

图 6.1.1 OLED 硬件监控硬件连接实物图

6.1.2 代码分析

在分析代码前，先介绍下原理。在 Linux 中，有“一切皆文件”的哲学核心思想，比如想查询当前 CPU 运行信息，输入命令“cat /proc/stat”即可，该命令就是查看“/proc/stat”这个文件。

因此，想知道 Linux 的任何状态信息，都可以查询对应的文件得到结果。

对于 CPU 利用率，解析“/proc/stat”文件即可。假设“/proc/stat”内容如下：

```
[root@100ask-am335x:~] cat /proc/stat
cpu   310 0 577 17814 153 0 6 0 0 0
cpu0 310 0 577 17814 153 0 6 0 0 0
intr 27255 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10405 0 0 0 1099 0 2 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 470 0 0 0
258 0 3447 0 0 0 0 0 0 0 0 0 0 0 0 28 16 0 0 11357 0 0 0 0 0 0 1 0 0 162 0
ctxt 50292
btime 0
processes 431
procs_running 1
procs_blocked 0
softirq 18636 5 9936 38 44 0 0 4860 0 0 3753
```

第一行的数值表示的是 CPU 总的使用情况，第二行是各 CPU 的情况，AM335X 是单核，所以“cpu”和“cpu0”的结果是一样的。这些数字指明了 CPU 执行不同的任务所消耗的时间（从系统启动开始累计到当前时刻），时间单位是 USER_HZ 或 jiffies（通常是百分之一秒）。以空格为分割，第一行的详细含义见下表：

/proc/stat 含义		
编号	内容	含义
1	cpu	name：指明该行表示CPU总的使用情况
2	310	user：正常的进程在用户态下执行时间累积值
3	0	nice： NICED的进程在用户态下执行时间列
4	577	system：进程在内核态的执行时间累积
5	17814	idle：空闲时间累积
6	153	iowait ：等待I/O完成的时间累积
7	0	irq ：硬中断时间
8	6	softirq：软中断时间
9	0	——
10	0	——
11	0	——

因为“/proc/stat”中的数值都是从系统启动开始累计到当前时刻的积累值，所以需要在不同时间点 t1 和 t2 取值进行比较计算，计算出的结果准确说是 t1 到 t2 时间段的结果，但如果 t1 到 t2 的时间段相对较小，可以把结果看作是当前的 CPU 利用率。

```
在t1到t2时间段，CPU总的使用时间 =
(user2+nice2+system2+idle2+iowait2+irq2+softirq2) -
(user1+nice1+system1+idle1+iowait1+irq1+softirq1)

在t1到t2时间段，CPU空闲时间 =
(idle2 - idle1)
```

```
CPU在t1到t2时间段即时利用率 =
(1 - CPU空闲时间) / CPU总的使用时间
```

获取 RAM 利用率的方法如下，在 Linux 里执行命令“cat /proc/meminfo”可以看到如下结果：

```
[root@100ask-am335x:~] cat /proc/meminfo
MemTotal:         507084 kB
MemFree:          447712 kB
MemAvailable:     478372 kB
Buffers:            5320 kB
Cached:            32444 kB
```

第一行是内存总大小，第二行是内存空闲大小，内存利用率=(总内存-空闲内存)/总内存。

DS18B20 的温度结果，可以直接调用“readTemperature()”函数得到。

整理清楚了思路，就可以写代码编程了，这需要 C/C++、Linux 环境编程等基础，不在本手册的介绍范畴，读者自行查阅相关资料。最后提供的示例代码有详细的注释，读者可以参考，下面假设读者了 C/C++、Linux 环境编程基础，对代码进行分析。

1）先是获得 CPU 利用率：

先定义一个结构体 CPU_OCCUPY，用来保存 CPU 各部分时间：

```
/*
* Description: 定义结构体CPU_OCCUPY, 保存CPU各部分时间
* Note: none
*/
typedef struct CPU_PACKED
{
    char name[20];
    unsigned int user;     //从系统启动累计到当前时刻，用户态的CPU时间（单位：jiffies），不包含
NICED的进程。1 jiffies=10ms
    unsigned int nice;     //从系统启动累计到当前时刻，NICED的进程在用户态下执行时间（单位：
jiffies）
    unsigned int system;   //从系统启动累计到当前时刻，进程在内核态的执行时间累积（单位：jiffies）
    unsigned int idle;     //从系统启动累计到当前时刻，空闲时间累积（单位：jiffies）
    unsigned int iowait;   //从系统启动累计到当前时刻，硬盘IO等待时间累积（单位：jiffies）
    unsigned int irq;      //从系统启动累计到当前时刻，硬中断时间（单位：jiffies）
    unsigned int softirq; //从系统启动累计到当前时刻，软中断时间（单位：jiffies）
}CPU_OCCUPY;
```

接着编写函数“getCPUOccupy()”获取一次节点“/proc/stat”内容，保存在 cpu_occupy 里：

```
/*
* FunctionName:getCPUOccupy
* Purpose: 获取节点/proc/stat内容，保存在cpu_occupy
* Parameters:cpu_occupy
* Return: return 0 if sucess,else return -1
* Note: none
```

```
*/
int getCPUOccupy(CPU_OCCUPY *cpu_occupy)
{
	FILE *fd;
	char buff[256];

	fd = fopen("/proc/stat", "r"); //打开节点/proc/stat
	if(fd == NULL) //如果打开失败
	{
		printf("getCPUOccupy:Can not open file\r\n");
		return -1;
	}

	fgets(buff, sizeof(buff), fd); //获取节点/proc/stat数据保存在buff
	sscanf(buff, "%s %u %u %u %u %u %u %u", cpu_occupy->name, &cpu_occupy->user, &cpu_occupy->nice,
&cpu_occupy->system, \
		&cpu_occupy->idle, &cpu_occupy->iowait, &cpu_occupy->irq, &cpu_occupy->softirq); //把
buff的数据依次保存到cpu_occupy里

	fclose(fd); //关闭节点/proc/stat

	return 0;
}
```

最后编写“getCPUInfo()”，调用两次“getCPUOccupy()”并计算结果。

```
/*
* FunctionName:getCPUInfo
* Purpose: 获取CPU利用率
* Parameters: 无
* Return: cpu_use CPU利用率
* Note: none
*/
float getCPUInfo(void)
{
	float cpu_use;
	unsigned long od, nd;
	unsigned long sum, idle;
	CPU_OCCUPY cpu_stat1;
	CPU_OCCUPY cpu_stat2;

	getCPUOccupy(&cpu_stat1); //第一次获取CPU时间
	msleep(500); //间隔时间，需大于10ms
	getCPUOccupy(&cpu_stat2); //第二次获取CPU时间
```

```
    od = (unsigned long) (cpu_stat1.user + cpu_stat1.nice + cpu_stat1.system + cpu_stat1.idle
                          + cpu_stat1.iowait + cpu_stat1.irq + cpu_stat1.softirq); //第一次CPU所有时间
    nd = (unsigned long) (cpu_stat2.user + cpu_stat2.nice + cpu_stat2.system + cpu_stat2.idle
                          + cpu_stat2.iowait + cpu_stat2.irq + cpu_stat2.softirq); //第二次CPU所有时间

    sum = nd - od; //两次采集间隔总时间
    idle = cpu_stat2.idle - cpu_stat1.idle; //增加的空闲时间
    if (sum > 0) //防止除0
        cpu_use = (sum - idle) / sum; //利用率=(总时间-空闲时间)/总时间
    else
        cpu_use = 0;

    return cpu_use;
}
```

2）接下来获得 RAM 利用率：

先定义一个结构体 MEM_OCCUPY，保存 RAM 的总大小和空闲大小：

```
/*
* Description: 定义结构体MEM_PACKED，保存内存的总大小和空闲大小
* Note: none
*/
typedef struct MEM_PACKED
{
    float total;
    float free;
}MEM_OCCUPY;
```

接着读取节点“/proc/meminfo”，计算内存利用率：

```
/*
* FunctionName:getMemInfo
* Purpose: 获取内存利用率
* Parameters:无
* Return: info 内存利用率
* Note: none
*/
float getMemInfo(void)
{
    FILE *fd;
    char buf[64], name[32];
    float ram_use;

    MEM_OCCUPY meminfo;
    memset(&meminfo, 0x00, sizeof(MEM_OCCUPY));
```

```
    fd = fopen("/proc/meminfo", "r"); //打开节点/proc/meminfo
    if(fd == NULL) //如果打开失败
    {
        printf("getMemInfo:Can not open file\r\n");
        return 0;
    }

    memset(buf, 0x00, sizeof(buf)); //清空buf
    fgets(buf, sizeof(buf), fd); //获取节点/proc/meminfo内容
    sscanf(buf, "%s %f %s", name, &meminfo.total, name); //得到第一行总内存

    memset(buf, 0x00, sizeof(buf)); //清空buf
    fgets(buf, sizeof(buf), fd); //获取节点/proc/meminfo内容
    sscanf(buf, "%s %f %s", name, &meminfo.free, name); //得到第二行空闲内存

    ram_use = (meminfo.total - meminfo.free) / meminfo.total; //利用率=(总内存-空闲内存)/总内存

    fclose(fd); //关闭节点/proc/meminfo

    return ram_use;
}
```

注意：“getCPUInfo()”、“getMemInfo()”返回的结果都是 float 数据类型，而“OLEDPrint()”需要传入字符串，因此还需要将 float 转为 string:

```
/*
* FunctionName:numToString
* Purpose: 数字转字符串
* Parameters:i
* Return: string 字符串
* Note: none
*/
string numToString(float i)
{
    char str[10];

    sprintf(str, "%02.2f", i);

    return str;
}
```

3) 主函数：

主函数里先实例化 OLED、DS18B20，清空 OLED 显示，让第一行显示"System info"。

然后调用 getCPUInfo()获取 CPU 利用率、调用 numToString()将它转换为字符串，然后传给 OLEDPrint()显示。

接下来调用 getMemInfo()获取 RAM 利用率、调用 numToString()转换为字符串，也是传给 OLEDPrint()显示。

最后调用 readTemperature()获取 DS18B20 温度、调用 numToString()转换为字符串，也传给 OLEDPrint()显示。

```
int main(int argc, char **argv)
{
    OLED oled(SPI_A); //实例化OLED
    DS18B20 ds18b20(GPIO0); //实例化DS18B20

    oled.OLEDClearAll(); //清空OLED

    oled.OLEDPrint(0, 0, "System info"); //OLED第一行显示System info

    while(1)
    {
        oled.OLEDPrint(0, 2, "CPU:" + numToString(getCPUInfo()*100) + "%"); //第二行显示CPU信息
        oled.OLEDPrint(0, 4, "RAM:" + numToString(getMemInfo()*100) + "%"); //第三行显示RAM信息
        oled.OLEDPrint(0, 6, "TEP:" + numToString(ds18b20.readTemperature()) + "'C"); //第四行显示
DS18B20信息
        sleep(1);
    }

    return 0;
}
```

该项目示例位于 Arduino 的“示例”->“Project.01.OLEDMonitor”->“OLEDMonitor”。

6.1.3 实际效果

将示例代码上传，在板子里运行，即可在 OLED 看到 CPU、RAM、TEP 信息，如图 6.1.2 所示。

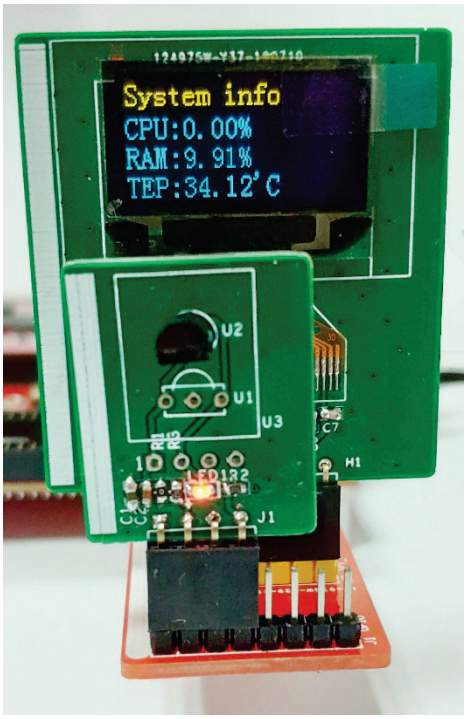

图 6.1.2 OLED 监控效果

这里看到的 CPU、RAM 利用率几乎没有变化，可以简单的模拟一下高 CPU、RAM 负载情况，观察数值是否变动。

- CPU 高负载测试：

```
[root@100ask-am335x:~]./am335x_app &
[1] 18455
[root@100ask-am335x:~]cat /dev/urandom | gzip -9 > /dev/null
```

先执行“./am335x_app &”，这里多了个“&”表示在后台执行“am335x_app”，这样才能在串口继续输入后面的命令。执行此命令后可以看到提示信息“[1] 18455”，它表示 “am335x_app” 这个程序运行时的进程 PID（读者返回的值可能不一样），以后可以通过此 PID 来结束此程序。

然后执行“cat /dev/urandom | gzip -9 > /dev/null”。这是一个不断获取随机数并进行压缩的操作，该过程会大量占用 CPU，使得 CPU 利用率升高。此时可以观察到 OLED 显示的 CPU 利用率变为 100%。

该测试完成后，先按下“Ctrl”+“c”，结束“cat /dev/urandom | gzip -9 > /dev/null”，再执行“kill 18455”结束后台程序“am335x_app”。

- RAM 高负载测试：

```
[root@100ask-am335x:~]./am335x_app &
[1] 21053
[root@100ask-am335x:~]mkdir ram_test
[root@100ask-am335x:~]mount -t ramfs /proc ram_test/
[root@100ask-am335x:~]dd if=/dev/zero of=ram_test/file bs=1M count=128
128+0 records in
128+0 records out
```

先执行“./am335x_app &”，让程序后台执行，系统返回该程序的 PID。

然后执行“mkdir ram_test”创建一个名为“ram_test”的目录，接着执行“mount -t ramfs /proc ram_test/”将该目录挂载到 RAM 里，最后执行“dd if=/dev/zero of=ram_test/file bs=1M count=128”向“ram_test”目录写 128M 文件，也就是向 RAM 写了 128M 文件。此时可以看到 OLED 上的 RAM 利用率明显上升，意味着 OLED 的显示反应了 RAM 的实际情况。

该测试完成后，先执行“umount ram_test/”取消挂载，再执行“rm ram_test/ -r”删除刚才新建的测试文件夹，最后执行“kill 21053”结束后台程序“am335x_app”。

- 温度测试：

```
[root@100ask-am335x:~]./am335x_app
```

直接执行“./am335x_app”，用手捏住 DS18B20 传感器，可以看到 OLED 里温度示数缓慢上升。

6.2 LCD 和 Web 摄像头监控

前面第一个项目通过 OLED 显示了一些简单的信息，但有些复杂的内容是无法在 OLED 显示的，比如摄像头的监控画面。像这类复杂的显示，一般有两种显示方式，一种是在本地 LCD 屏幕显示，另一种远程显示，比如在 Web 网页显示。

LCD 显示和 Web 网页显示，涉及 Linux 驱动、应用编程，如果读者对整个实现过程感兴趣，可以系统地学习百问网配套的 Linux 驱动、应用的相关教程。本手册面向嵌入式开发初学者、计算机软硬件爱好者，就不过多分析细节，只向读者展现项目效果。

6.2.1 硬件连接

读者可以网上购买任意 USB 接口支持 UVC 的免驱动摄像头，将 USB 摄像头插到 ROC-RK3399-PC 的任意一个 USB 接口即可。

如果要使用 Web 网页显示，还需要连接网线；还要保证开发板与电脑的 IP 位于同一个网段，并能互 ping 成功，连接如图 6.2.1 所示。

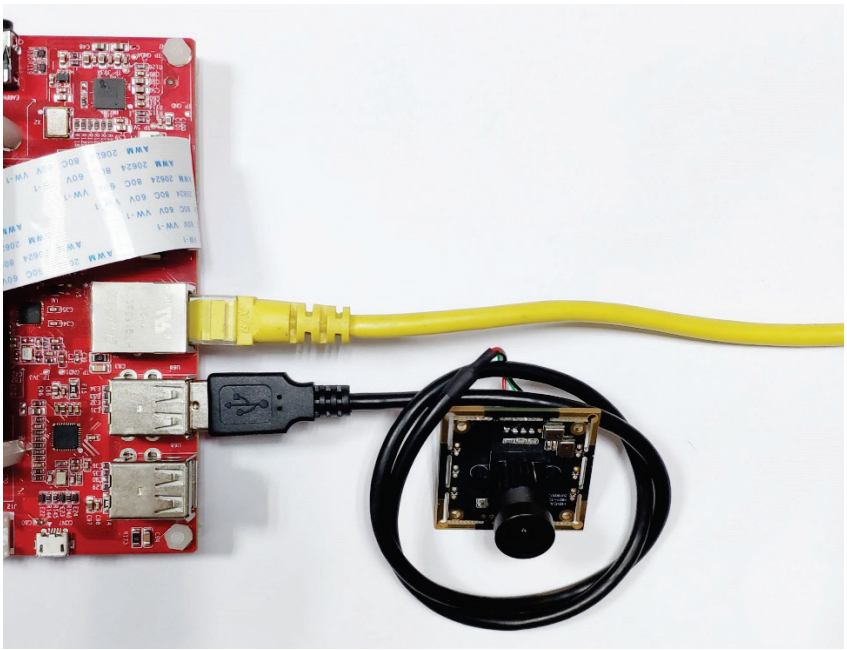

图 6.2.1 底板、USB 摄像头、网线连接实物图

6.2.2 实际效果

- LCD 摄像头监控

出厂的板子 LCD 默认有个显示示例程序，该程序开机自启动，会影响后面摄像头的显示，因此需要先取消该程序开机自启动。

在串口上依次输入以下命令：

```
[root@100ask-am335x:~]cp /etc/init.d/S93digitpic /etc/init.d/S93digitpic_bak
[root@100ask-am335x:~]cp /usr/camera/lcd/S93digitpic /etc/init.d/S93digitpic
[root@100ask-am335x:~]reboot
```

第一行命令是备份源文件，第二行是覆盖源文件，第三行是重启命令。

执行完这些命令后，开发板会重启，重启后 LCD 就不再显示示例程序。

实验完后，如果想恢复默认显示示例程序，将备份源文件改复制回去即可：

```
[root@100ask-am335x:~]cp /etc/init.d/S93digitpic_bak /etc/init.d/S93digitpic
```

重启完成，输入“root”进入系统，执行如下命令即可看到 LCD 上显示摄像头画面：

```
[root@100ask-am335x:~]/usr/camera/lcd/video2lcd -v /dev/video0 -d /dev/fb0
```

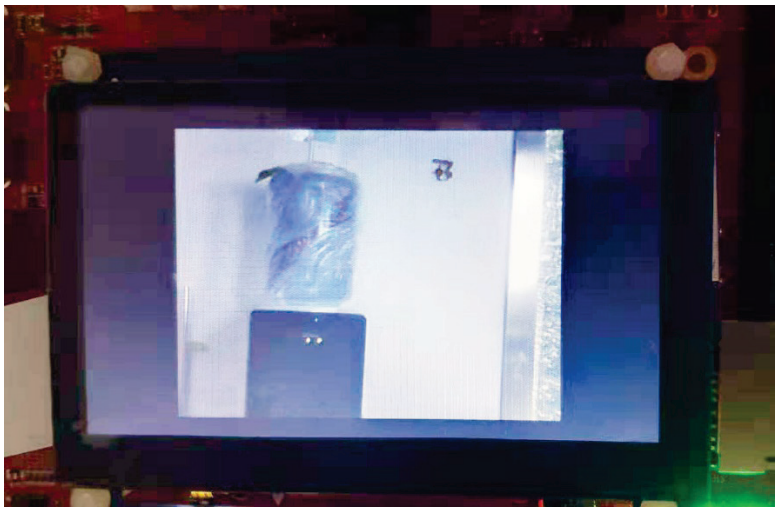

图 6.2.2 LCD 摄像头监控画面

实验测试完成后，按下“Ctrl”+“c”结束。

如果读者想实现开机自启动 LCD 监控，可以在文件“/etc/init.d/S93digitpic”末尾加上“/usr/camera/lcd/video2lcd -v /dev/video0 -d /dev/fb0 &”，修改涉及“vi”或“vim”编辑器的使用，读者自行搜索研究。

- Web 摄像头监控

先在串口输入“ifconfig”命令，查看板子的 IP 地址，可以看到如下类似内容：

```
[root@100ask-am335x:~]ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.76  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::9fa9:2ecc:127e:a256  prefixlen 64  scopeid 0x20<link>
        ether 98:5d:ad:42:a9:6b  txqueuelen 1000  (Ethernet)
        RX packets 18387  bytes 1956578 (1.8 MiB)
```

```
        RX errors 0   dropped 1   overruns 0   frame 0
        TX packets 25023   bytes 29959807 (28.5 MiB)
        TX errors 0   dropped 0 overruns 0   carrier 0   collisions 0
        device interrupt 177
```

可以看到板子局域网 IP 地址为“192.168.1.76”，这个 IP 是路由器分配的，读者的可能不一样，记下自己板子的 IP 地址。如果没有显示 IP 地址，需要用户根据路由器网段，自行设置 IP 地址。

接着输入以下命令启动程序：

```
[root@100ask-am335x:~]mjpg_streamer -i "/usr/lib/input_uvc.so -d /dev/video0 -r 640x480 -f 30 -q 90 -n"
-o "/usr/lib/output_http.so -w /usr/camera/web"
```

命令中的“640x480”，为读者所使用的摄像头支持的分辨率，读者根据所使用的摄像头硬件特性和需求，自行参考设置。

然后打开浏览器，输入“自己板子的 IP：8080”，比如我这里输入“192.168.1.176:8080”，随后就会显示如图 6.2.3 所示页面。

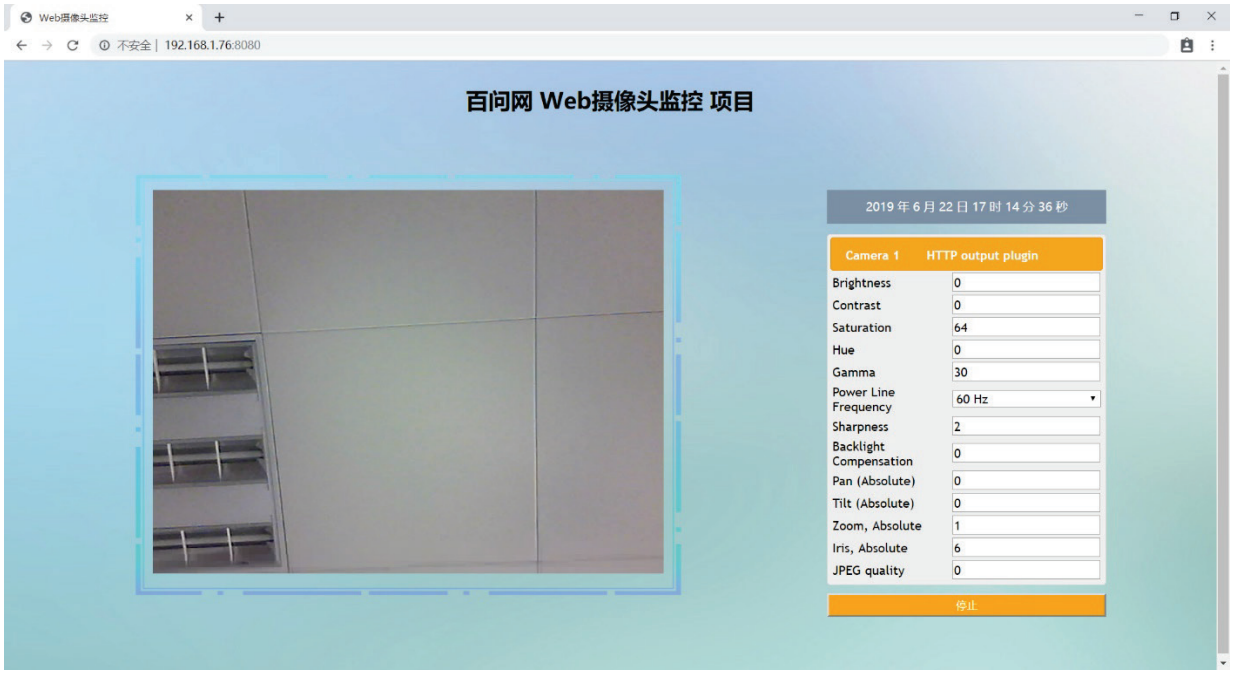

图 6.2.3 Web 摄像头监控画面

网页左边显示摄像头监控图像，右边为控制面板，可以控制摄像头画面的亮度、对比度、饱和度等，以及开始、停止监控。

附录一 模块列表

GPIO接口			
编号	**模块名**	**接口**	**备注**
1	LED模块	板载	用户只能使用LED4
2	Keyboard按键模块	板载	——
3	IrDA红外遥控	接扩展板	一个GPIO、配遥控器
4	DHT11温湿度模块	接扩展板	一个GPIO
5	DS18B20温度模块	接扩展板	一个GPIO
6	SR501人体红外模块	接扩展板	一个GPIO
7	SR04超声波测距模块	接扩展板	两个GPIO
8	Motor步进电机驱动板	接扩展板	四个GPIO、配步进电机
SPI接口			
编号	**模块名**	**接口**	**备注**
1	OLED显示屏	接扩展板	复用GPIO
2	DAC数模转换	接扩展板	——
3	ADXL345三轴传感器	接扩展板	复用GPIO
I2C接口			
编号	**模块名**	**接口**	**备注**
1	EEPROM存储模块	接扩展板	与光敏电阻在一个模块
2	RTC实时时钟模块	板载	——
其它接口			
编号	**模块名**	**接口**	**备注**
1	GPS定位模块	接TTL串口接口	用户只能使用UART_A(uart2)
2	ADC模数转换	板载	通过光敏电阻展示效果
项目			
编号	**模块名**	**接口**	**备注**
1	USB摄像头	USB接口	网上任购支持UVC的免驱USB摄像头

附录二 引脚编号对照表

LED GPIO			
名称	对应引脚	对应值	描述
LED1	GPIO1_16	48	板载LED1
LED2	GPIO1_17	49	板载LED2
LED3	GPIO1_19	51	板载LED3
LED4	GPIO1_21	53	板载LED4
扩展板 GPIO			
名称	对应引脚	对应值	描述
GPIO0	GPIO0_6	6	扩展板GPIO接口0
GPIO1	GPIO0_7	7	扩展板GPIO接口1
GPIO2	GPIO0_13	13	扩展板GPIO接口2
GPIO3	GPIO1_20	52	扩展板GPIO接口3
GPIO4	GPIO1_18	50	扩展板GPIO接口4

附录三 按键映射表

板载按键		
名称	对应值(code)	定义
K1	105	KEY_LEFT
K2	106	KEY_RIGHT
K3	28	KEY_ENTER
K4	1	KEY_ESC
红外遥控器按键		
名称	对应值(code)	定义
CH-	403	KEY_CHANNELDOWN
CH	553	KEY_CHANNEL
CH+	402	KEY_CHANNELUP
⏮	412	KEY_PREVIOUS
⏭	407	KEY_NEXT
⏯	164	KEY_PLAYPAUSE
-	114	KEY_VOLUMEDOWN
+	115	KEY_VOLUMEUP
EQ	13	KEY_EQUAL
0	11	KEY_0
100+	59	KEY_F1
200+	60	KEY_F2
1	2	KEY_1
2	3	KEY_2
3	4	KEY_3
4	5	KEY_4
5	6	KEY_5
6	7	KEY_6
7	8	KEY_7
8	9	KEY_8
9	10	KEY_9