第四章多功能桌面式时钟设计

通过前面四章的铺垫，我想大多数同学对STM32这颗芯片应该都有个大致的 认识了，到此为止，不管大家对此颗芯片掌握了多少，都没关系。因为接下来的 实战设计教程，我们会就我们所遇到的知识点，针对性的来讲解。如果说前面几 张是战略性的部署我们的大局，那从这一章开始，我们就是针对局部的“战役” 来制定正确的开发套路。

1产品开发流程

收看我们视频的绝大多数同学，应该都没有设计过一个完整的产品，甚至有 些同学还是从没有接触过单片机或者电路的。尽管我们的视频可以非常细致地无 死角授课，但是针对最基本的一些知识，大家还需要自己去学习。打个比喻来说， 我们学习知识，就好像在打扫房子，尽管我每一步细致地教你怎么打扫，包括把 别人容易忽略的死角也详细说明要怎么打扫，但是如何是用扫帚，就不是我们教 授的范围了，这个还需大家开小灶自己去补习。收看我们视频的同学，有些基础 知识是必备的，如下：

* 电路基础；

>模拟电路基础；

>数字电路基础；

* 〔语言程序设计；

>基本元器件的知识（电阻，电容，晶体管等）。

在有了上面所述的知识之后，那理解我们的视频就变得非常简单了。那先让 我们来叙述一个产品设计的主要线索。

总的来说，一般一件产品的设计主要包括以下几个部分：确立产品定位-> 市场调研-> 确立产品任务书-> 研发验证阶段-> 工程验证阶段-> 小批量 试产-> 量产。

在上面整个固定的设计流程里面，“确立产品定位”和“市场调研”，主要

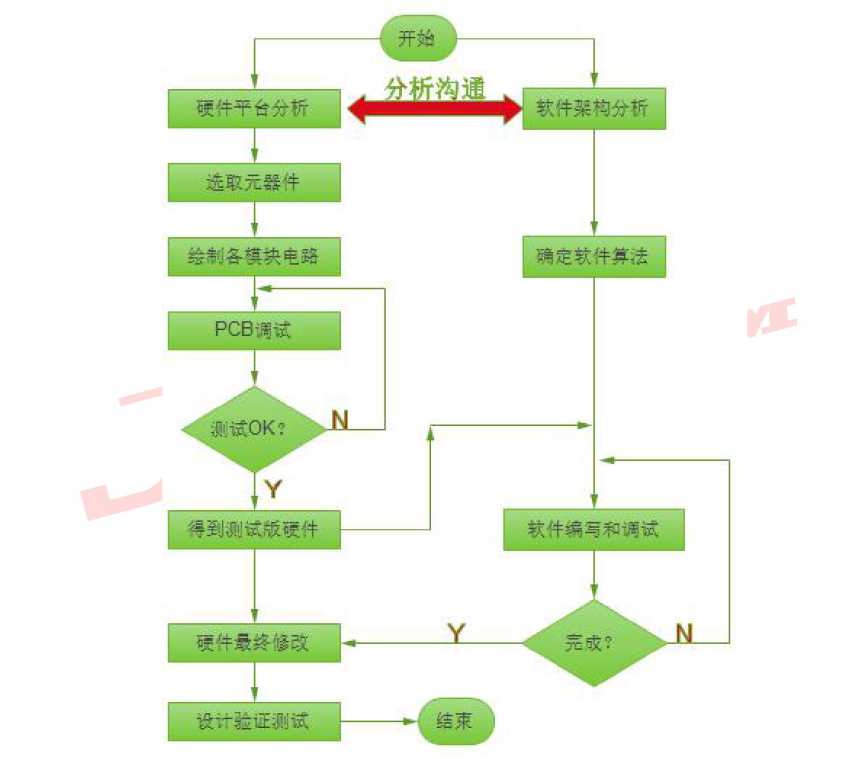
是属于产品策划阶段，属于市场分析；我们做技术的主要涉及的环境就是“研发

验证阶段”和“工程验证阶段”以及“小批量试产”这三个环节。一般来说，这

1

三个环境当中“研发验证阶段”就是给大家足够的时间，去实现“产品任务书” 里面的研发设计，以及产品可靠性测试和产品的安规认证；“工程验证阶段”是 基于工程反馈的问题点，进行讨论和修改；而“小批量试产”就是整个产品设计 阶段的一个最终验证，此验证是针对产线生产时候问题的反馈和修改。到此为止， 如果按照标准流程来说的话，产品设计已经完成，接下来就是量产了。

我们此份文档讲的内容，是基于“研发设计阶段”的。关于“研发设计阶段” 的标准流程如图1所示。



图i研发设计流程

有一点需要说明的是，我们“乐创DIY学院”已经帮大家做好了硬件设计的 绝大多数工作，然而考虑到有些有必要的朋友可能会需要这一部分的内容，因此， 我们将硬件设计的思路方法独立出来做一部分享给大家，如果对它们没有兴趣， 或是本身就具有硬件设计经验的朋友可以跳过那一部分。我们接下来的内容，主 要分成硬件和软件两大块来说，每一块里面按照独立的模块来讲，最后再用一大 块内容来完成我们的功能设计。大家可以适当的选取自己最需要的部分来收看和 练习。



2硬件部分

再设计硬件之前，我们先来当一次产品经理，来制定出我们多功能闹钟需要 有哪些要求。这边我们先来自己拟定几个：

>时间/日期设置和显示；

>温湿度显示；

>设置闹钟；

>设置整点报时；

>定时器以及定时报警功能；

>串口通讯功能。

我们暂时的功能就先设置这么几个，以后又需要再加，或者实现不了的可以 删掉，这个先不去管它。

按照上面所述的一些功能，我们大致可以把它分成如下几个硬件模块，如： 电源模块，控制模块，显示模块，输入模块，温湿度米集模块，串口通讯模块以 及蜂鸣器模块。以下的内容，我们就按照这么6大模块进行硬件电路的设计。

2.1电源模块

这是我们的第一个实例教程，因此我们关于电源部分，就采用最简单的USB 电压通过线性稳压器来稳压的设计，以降低设计的难度。

关于线性稳压器，可能很多同学在我的“51单片机视频教程”里面就已经 略有所闻了。我们在“51单片机视频教程”里面使用的是最常用的三端稳压集 成芯片7805。由于当时我们侧重的是一个认知性的使用，并没有多花时间在线 性稳压器上面花功夫，那借助这次机会，我们在这里重新来认知一下，何为线性 稳压器。

2.1.1线性稳压器

关于线性稳压器，百度百科上面的定义是这样的：线性[稳压器](http://baike.baidu.com/view/781492.htm)使用在其线性 区域内运行的晶体管或FET,从应用的输入电压中减去超额的电压，产生经过调 节的输出电压。可能说到这里，大多数同学都不能很好的理解它在说什么。接下 来，我们来细细分析。

首先，我们需要明白的一点就是，何为晶体管的线性区？以双极型晶体管为

例（BJT，三极管），我们知道一颗三极管具有截止状态，放大状态，饱和状态 这三种基本状态，我们可以改变偏置，来改变其工作区。其中截至状态和饱和状 态的三极管被用来作为开关使用。而放大状态的三极管，如果以NPN共射电路为 例的话，输入电压^和输出电压匕之间是呈现线性关系的，即匕越大匕越大。

有了上一段落的三极管线性区的基本状态作为铺垫，那我们可以继续来看线 性稳压电路的内部原理图，如图2所示。

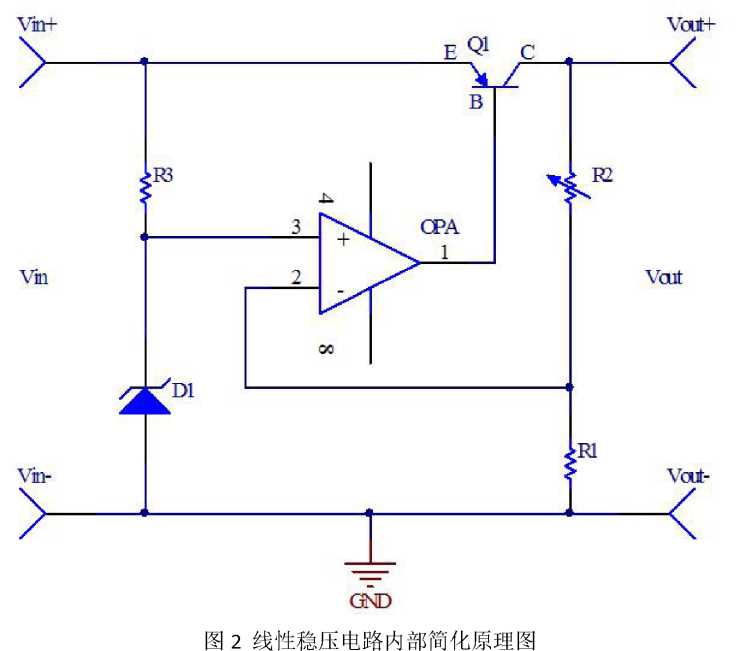


图2中，D1是一颗稳压二极管，产生一个参考电压。假设输入电压为5V， 那么此时放大器0PA的正相输入端稳定为一稳定电压。首先，假设此电路当前状 态是没有接入Vin的。在以上准备完了之后，我们给它Vin上面加上一个5V电 路，由于输出的滞后性，因此运放反相输入端上面的电压开始增长，同时使得运 放OPA的输出开始增长。进而晶体管开始进入线性区，然后使得电压Vce开始增 大，直到当运放的反相输入电压等于正向输入电压，那此时Vce就可以得到一个 固定的电压值，因此，我们可以得到Vo = Vin - Vce。以上就是线性稳压的基 本原理。

我们回顾一下之前的7805,我们明确提到，如果想要它正常工作，就必须 需要一个2V以上的压差，但是考虑到由于STM32是3.3V供电的，而输入5V时，

5

压差就只有5V - 3.3V = 1.7V，显然达不到三端线性稳压器的要求。因此，我

们在这里给大家介绍一个新概念 低压差线性稳压器（LD0，low dropout

regulator)。

2.1.2低压差线性稳压器LDO

我们在这里选用的LDO为AMS1117-3.3V。AMS1117系列稳压器有可调版与多 种固定电压版，设计用于提供1A输出电流且工作压差可低至IV。在最大输出电 流时，AMS1117器件的压差保证最大不超过1.3V，并随负载电流的减小而逐渐降 低。具体详见数据手册。SOT-223封装的AMS1117-3.3V和引脚定义如图3和图4 所示。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  | | |  | |
| AMS1117  SOT-223 | | | | | | |
|  |  | | 2 |  | |  |

图 3SOT-223 封装的 AMS1117-3.3V

|  |  |  |  |
| --- | --- | --- | --- |
| 管脚号 | 管脚名称 | I/O | 功能 |
| 1 | GND/ADJ | 一/0 | 祕DJ。 |
| 2 | VOUT | 0 | 输出电压- |
| 3 | VIN | I | 输入工作电压。 |

图4AMS1117引脚定义

\|> i>

n§/mv

因此我们可以很简单的得到我们的电源电路，如图5所示。

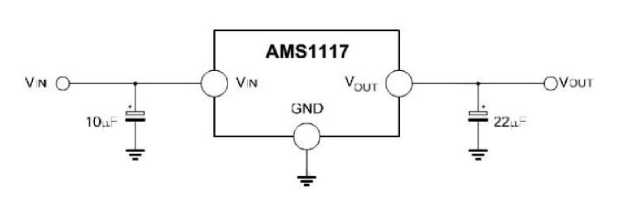


图5AMS1117应用电路

在这里，我们注意一下，关于输入输出电容。它的数据手册上面只标注了可 调电压版本，输出至少要接一个22uF的电容，而固定电压输出版本，可以选择 更小的电容。但是，由于我们需要过滤一些LD0输入电压里面的高频和低频电容， 因此我们选用大电容和小电容组合的方式。如图6所示。

U3 A]VKI 117-33 33WDD

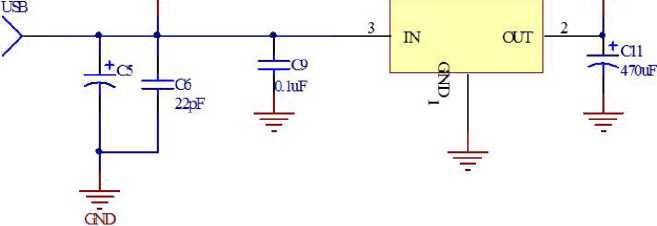


图6实际电源电路

可能细心的同学已经发现了图6中的USB接口，在这里，我们多提一点关于 USB的引脚定义。我们采用的是5脚的Micro USB接口，大多安卓手机都是用的 此接口。其顶视图如图7所示。

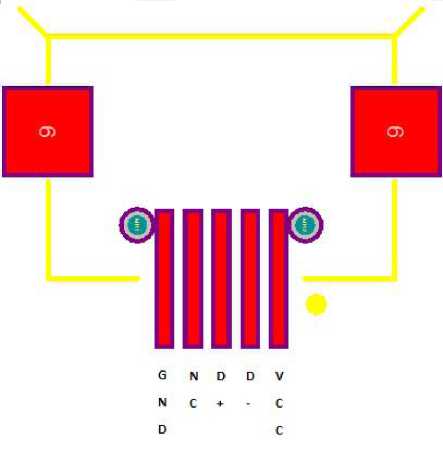


图7 USB接口顶视图

2.2控制模块

控制模块我们还是选择了 10元左右性价比最高的单片机STM32F103C8T6， 关于STM32F103C8T6的最小系统，我们在第三章里面已经详细分析过，这里就不 再重复了。为了便于大家方便开发，我们在这里将STM32F103C8T6的最小系统做 成了一个单独的模块，因此大家以后再又需要用到的时候，直接选购就行，模块

7

的Protel原理图和封装库我已经传上去（JC\_MODULE.PcbLib和 JC\_MODULE.SchLib)，有需要的同学可以直接下载使用。

STM32F103C8T6最小系统模块的原理图如图8所示。

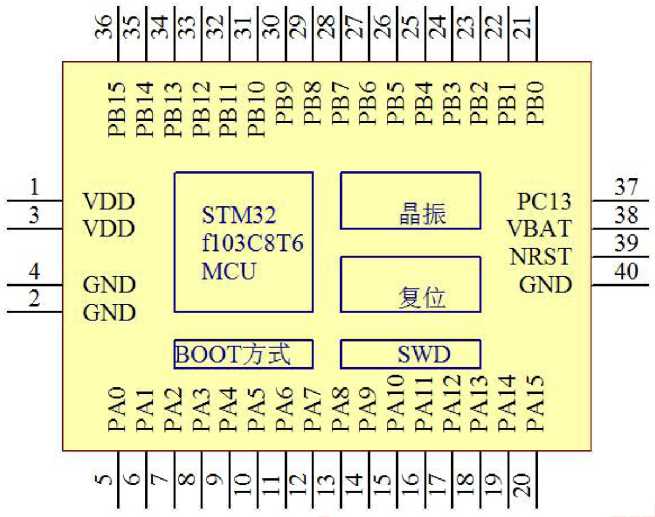


图8 STM32F103C8T6最小系统模块接口图

我们以后设计电路，只需要基于图8的最小系统模块就好，首先，需要把电 源和地供上，用2.1里面设计的电源直接加上就可以，其他端口引脚直接可以作 为驱动脚去连接外部其它器件。

这里还需要说明一下，因为我们以后会使用到STM32内部自带的实时时钟， 来用做我们的时钟基准，因此我们需要外加外部的不间断电源，使得RTC—直处 于不断电的状态。因此这里我们就需要设计VBAT引脚的供电部分。我们来参考 《STM32F10XXX硬件开发》手册，上面提到“VBAT引脚必须被连接到外部电池 (1.8V < VBAT < 3.6V)，如果没有外部电池，这个引脚必须和100nF的陶瓷电容 一起连接到VDD电源上”。因为我们没有外部电源，因此需要用一颗3V的纽扣 电池给系统供电。参考的供电设计电路，如图9所示。

图9中，需要注意D1和D2两个二极管。假设1N4148压降为0.7V，当外部 供电时，由于VDD>VBAT，D2截止，因此VBAT主要由VDD供电；当外部没有电 压时，D2导通，VBAT由电池供电。这里还需要补充一下，如果D1和D2不接时， 当外部电源去除后，电池的能量会倒灌给系统去供电，会导致无法关机。因此这 里需要注意。

8

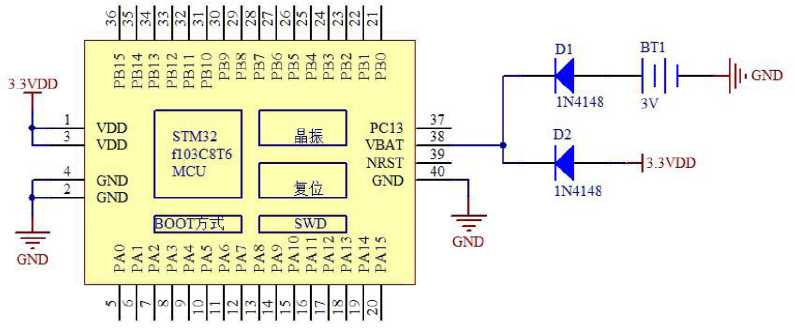


图9控制系统供电方案

1. 3显示和输入模块

为了有一个有好的用户操作界面，我们一般选择将显示功能和输入功能做成 一个独立的模块，而我们本次采用的布局方案如图10所示。

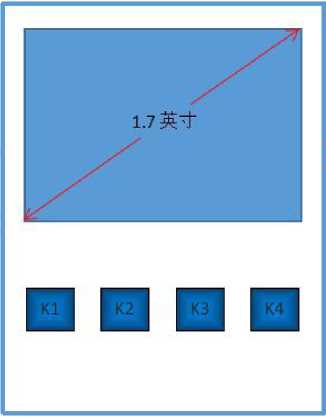


图10操作界面布局

之所以采用图10中的布局，是因为它比较简洁，并且4个按键在做液晶UI 的时候，已经具备了可以完成“确定”，“返回”，“上”，“下”这四种操作 功能的基本要求。而1.7英寸的显示器，是基于ST7567-G4D主控芯片的128\*64 像素的黑白FSTN液晶，配备了一组单向的SPI串口进行控制。

关于液晶显示器的内部原理和电性参数，可以参照相应的数据手册。图11 中，我们截取了此款液晶的外观图，图12中为它的引脚定义。



图11液晶显示器外观图

|  |  |  |
| --- | --- | --- |
| No. | Symbol | Description |
| 1 | LED-A | Backlight anode |
| 2 | LED-K | Backlight cathode |
| 3 | GND | Ground |
| 4 | CSB | Chip select input pin. Interface access is enabled when CSB is |
| 5 | RESET | Hardware reset input pin. When RSTB is internal initialization is executed and the internal registers will be initialized. |
| 6 | AO | Data or command selected signal ,A0='H'—data;A0=\*L' command |
| 7 | SCL | Serial dock input,latched data at rising edge |
| 8 | SDA | Serial data input. |
| 9 | VDD | Power supp!y^.8V(TYP) |
| 10 | VSS | Ground |
| 11 | VO | VO is the LCD driving voltage for common circuits at negative frame. Connect to a capacitor(0.1 Uf^luF) to XVO. |
| 12 | XVO | XVO is the LCD driving voltage for common circuits at positive frame. Connect to a capadtor(0.1Uf-1uF ) to VO. |
| 13 | VG | VG is the LCD driving voltage for segment circuits. Place a stabilizing capacitor(0.1Uf-1uF ) between Ground |
| 14 | GND | Ground |

图12液晶显示器引脚定义

根据引脚定义，我们可以得到如图13的模块原理图。在图13中，1脚接于

一个三极管开关，通过控制此三极管的基极即可以控制液晶背光板的亮灭，以避

免夜间光照，影响睡眠。其余各脚，即按照引脚定义的来连接。同时，我们也将

四个输入按键以及四个LED指示灯一并放在此模块上面。如图主控模块，我们也

将此模块封装传入JC\_MODULE.PcbLib和JC\_MODULE.SchLib里面，供大家方便开

2.4温湿度采集模块

发

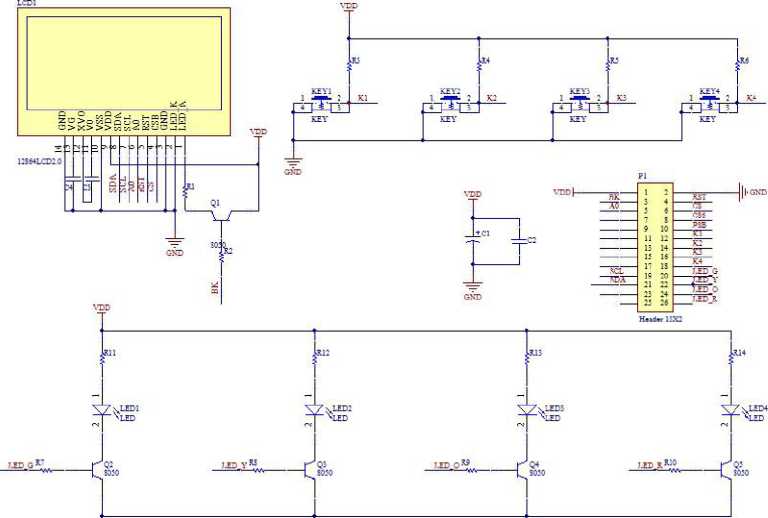
****

图13LCD显示模块原理图

就我们所知的对温度采集已经有了非常多的方法，就温度采集来说，最简单 的就是根据液体体积受温度的膨胀率来获取温度。除此之外，比如热电阻，热电 偶，以及光谱测温，都是非常有效的测温手段。然而对湿度测量来说，可能大家 接触的比较少，一般来说，我们常说的“湿度”指的是相对湿度，即空气中实际 水汽压与当时气温下的饱和水汽压之比的[百分数](http://baike.baidu.com/view/41104.htm)表示，取整数。而对湿度的测量 方法有湿敏电阻，库伦湿度计，光学形湿度计，气象[色谱法](http://baike.baidu.com/view/28457.htm)，化学物质电特性法， 离子晶体冷凝湿度计等。但就我们实际使用的过程中，湿敏电阻是最方便的方法。 而对于环境温度之类的不精确的条件下，测量温度的最方便方法就是热敏电阻。 因此在这里，我们引入一个传感器一一DHT11。

DHT11数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传 感器。它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高的 可靠性与卓越的长期稳定性。传感器包括一个电阻式感湿元件和一个NTC测温元 件，并与一个高性能8位单片机相连接。因此该产品具有品质卓越、超快响应、 抗干扰能力强、性价比极高等优点。每个DHT11传感器都在极为精确的湿度校验 室中进行校准。校准系数以程序的形式储存在0TP内存中，传感器内部在检测信

11

第四章多功能时钟

号的处理过程中要调用这些校准系数。单线制串行接口，使系统集成变得简易快 捷。超小的体积、极低的功耗，信号传输距离可达20米以上，使其成为各类应 用甚至最为苛刻的应用场合的最佳选则。产品为4针单排引脚封装。连接方便， 特殊封装形式可根据用户需求而提供。DHT11的外观和引脚图，如图14所示。

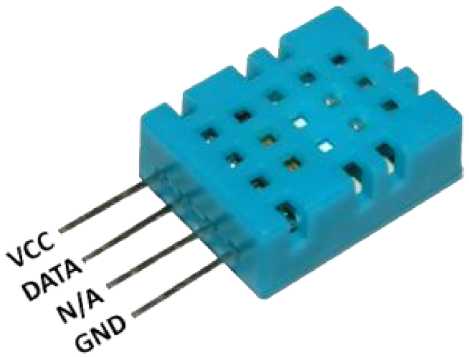


图14 DHT11的外观和引脚图

图14中，“VCC”脚是DHT11的供电引脚，DHT11的供电电压为3 — 5.5V。 传感器上电后，要等待1s以越过不稳定状态在此期间无需发送任何指令。“GND” 脚是DHT的参考地脚。电源引脚（VDD，GND)之间可增加一个100nF的电容，

用以去耦滤波。“DATA”脚是数据通讯脚用于微处理器与DHT11之间的通讯和 同步，采用单总线数据格式，一次通讯时间4ms左右，为了增加驱动能力，我们需 要把“DATA”上拉一个5K以上的电阻。关于时序操作部分，我们放到软件部分 来讲。因此我们可以得到DHT11的电路图，如图15所示。

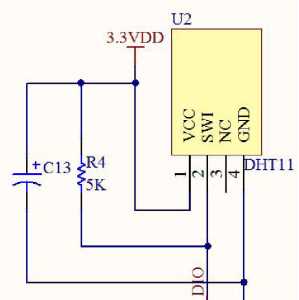




图15 DHT11电路

2.5串口通讯模块

STM32F103C8T6具有3个USART串口，USART是一个全双工通用同步/异步串 行收发模块，该接口是一个高度灵活的串行通信设备。因此我们设计外部串口的 时候，只需要建立在STM32自带的串口上面，有必要时，对其进行时序或者协议 转换，就可以达到我们的要求。

再说串口，目前较为常用的[串口](http://baike.baidu.com/view/69108.htm)有9针串口（DB9)和25针串口（DB25)， 典型的的九针串口引脚如图16所示，引脚功能如表1所示。



图16九针串口外观

表1DB9串口引脚定义

|  |  |  |
| --- | --- | --- |
| 针号 | 功能说明 | 缩写 |
| 1 | 数据载波检测 | DCD |
| 2 | 接收数据 | RXD |
| 3 | 发送数据 | TXD |
| 4 | 数据终端准备 | DTR |
| 5 | 信号地 | GND |
| 6 | 数据设备准备好 | DSR |
| 7 | 请求发送 | RTS |
| 8 | 清楚发送 | CTS |
| 9 | 振铃提示 | DELL |

以上的内容，我们可以理解为介绍了基本的串口引脚，然而关于通讯协议的 话，有很多种方式，如RS232RS485等等。然而我们目前在短距离的模块之间 通讯时，一般采用3线制的接口，即TXD，RXD和GND三根信号线。STM32的串 口包含了 USARTx\_CK，USARTx\_TX，USARTx\_RX，USARTx\_CTS，USARTx\_RTS，但是

13

常用的只有USARTx\_TX，USARTx\_RX和GND三根引脚。表2中是STM32F103C8T6 里面所有串口的引脚定义。

表2STM32F103C8T6串口引脚定义

|  |  |  |  |
| --- | --- | --- | --- |
| 引脚号 | 串口 | 功能 | 默认复用GPIO |
| 30 | USART1 | TX | PA9 |
| 31 | RX | PA10 |
| 12 | USART2 | TX | PA2 |
| 13 | RX | PA3 |
| 21 | USART3 | TX | PB10 |
| 22 | RX | PB11 |

我们可以在表2中任意选择一组串口作为我们的通讯串口，例如我们选择 USART1。这里再说明一点，其实STM32的串口不仅仅可以作为通讯口这么简单， 还能当ISP编程口来用，有兴趣的同学可以向我咨询，考虑到ISP无法调试，因 此我们本视频就只用SWD去下载。

再回到我们的设计上面，我们给出以下几个情况来为大家介绍串口：

1. 近距离单片机和单片机之间的通讯。近距离单片机和单片机之间的串 口通讯，只需将两个系统的地线连接到一起，然后RX和TX交换就行，如图17 所示。

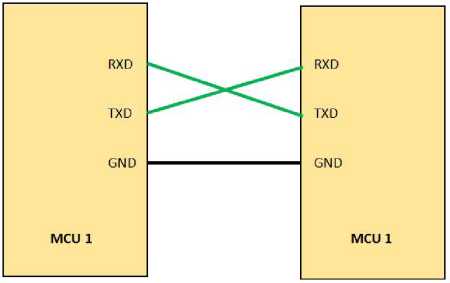


图17近距离单片机通讯串口连接图

1. 远距离串口传输。远距离的串口传输一般需要用到协议转换，如 RS232，RS485等等，如果距离足够长的时候，需要加上中继器来做信号的中继。 一般是传输端信号调制-> 传输（有必要时中继）-> 接收端信号解调。如图 18所示。

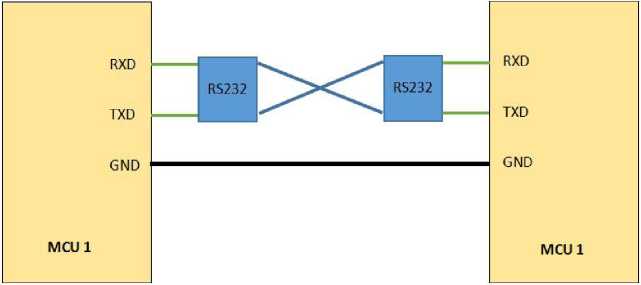


图18远距离串口传输

由于我们的路径比较短，因此采用图17里面的传输架构。考虑到目前主流 的计算机上面，早已没有了之前的9针串口接口，因此我们需要将串口数据用芯 片转换成USB信号给电脑读取。

具体在这里用到的芯片是CH340〇CH340是一个USB总线的转接芯片，实现 USB转串口、USB转IrDA红外或者USB转打印口。在串口方式下，CH340提供常 用的MODEM联络信号，用于为计算机扩展异步串口，或者将普通的串口设备直接 升级到USB总线。连接电路如图19所示。

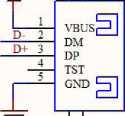
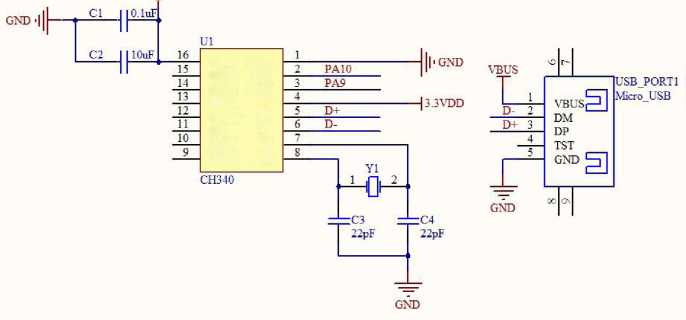
3.3VDD

图19串口转USB电路

ITXDIV3D+SXO

Milrii

2.6蜂鸣器模块

蜂鸣器一般作为报警器件来用，根据其加载信号的不同，可以分为有源蜂鸣 器和无源蜂鸣器两种

无源蜂鸣器是要有驱动的音频信号（20 - 20000 Hz)才能发声。人能听到 20到20kHZ的声音.无源蜂鸣器对相应的交流信号，可以产生不同频率的声音。

而有源蜂鸣器，由于内部有振荡电路，产生的声音频率固定.因此只要通电就 能发出单个频率的声音。

15

第四章多功能时钟

其对应的控制电路也是不同的。如图20所示。

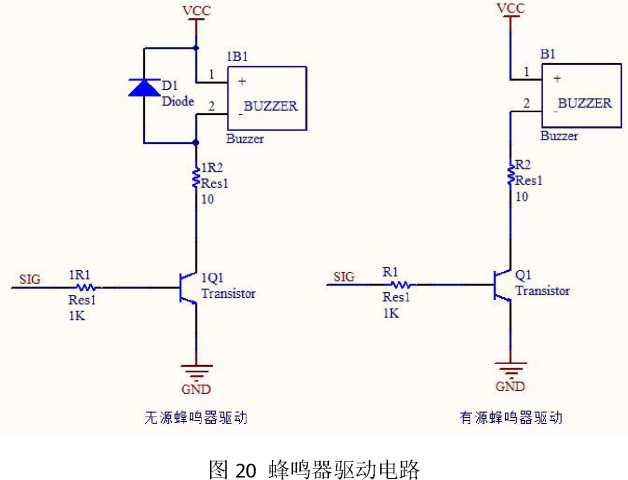


图20中，大多数电路器件还是相似的，唯一不同的是，无源蜂鸣器驱动时， 需要加一颗反向的二极管用作电流泄放。因为蜂鸣器本身的发声原理就是电磁感 应，而我们控制无源蜂鸣器时，直接控制的是其线圈，因此当采用交流信号的时 候，为了加快每次信号变化时线圈的反向压降，因此需要一颗二极管进行泄流。

考虑到我们的功能只是单一的报警，因此我们采用比较简单的有源蜂鸣器电 路。

2.7总结

基于以上内容，我们可以得到每个模块的线路，最后只要通过端口分配，就 得到了最终的系统电路，我们的电路分成以下几个部分：

“MCU Core.SchDoc”：核心板模块；

“LCD and Key.SchDoc”：显示按键模块；

“Main Board.SchDoc”：主板模块；

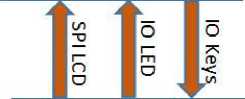
“Buzzer.SchDoc”：蜂鸣器模块。

以上电路可以参考，但是具体元件参数需要认真阅读上文的内容，才可以掌 握。有兴趣的朋友可以下载参考。

为了便于大家更好地理解我们的硬件设计，我在这里画出了我们各个模块的 组成，以及其之间的通讯，控制方式。如图21所示。



显示按键模块



主控模块

实时时钟

**A**

...

**DHT1 Buzzer**

SD DHTll

10 Buzsr

**USB**

主板模块

图21系统硬件框图

3软件部分

在本章的第二部分中，我们非常详细地介绍了硬件的设计图，有了以上的基 础之后，我们的软件开发就非常简单了，考虑到这是第一次大家用STM32芯片写 程序，因此我们会从最简单的知识点去讲解。我们软件部分主要分成两大部分来 讲解一一硬件驱动和功能实现。这里还需要再申明一下，如果你需要和我们一起 来动手开发这份软件，那你必须要有一份编译过的固件库。

3.1硬件驱动

3.1.1 GPIO输出驱动——点亮第一个LED

可以说所有的单片机教学视频都是以点亮第一个LED灯作为切入点来展开 的，就好像所有的软件教程都是以“Hello World”作为第一个程序。但关于为 什么要以这个为实例，很多人都说不上来。其实以这个为第一个实例是很有科学 依据的一件事情，首先第一点就是效果明显，第二点就是，我们以后的操作，可 能70°%以上的外设驱动都是通过I/O 口去实现的。

1. STM32 的 GPI0 口介绍

回顾之前我们学习的51单片机，我们对IO端口的操作非常简单，只需要使 对应的某一位为“1”，那这个端口就会变成高电平状态，如果使之位“0”，那 它就会变成低电平状态，这就是我们51单片机的输出方式，由于这种方式可以 寻址到某一端口的某一位，因此我们称之为位寻址。而在STM32单片机上面，我 们就不能独立地对特定的某一位进行寻址操作了，并且在操作此端口前需要对其 输出模式进行设定。以上就是我们在操作上面可以感受出STM32和51 GPIO最明 显的不同点。然而这种是表象而已，他们的GPIO之间基本上没有一处是一样的。

51单片机的IO 口仅仅只用了一个8位的寄存器来实现复用的输入输出模 式。而STM32每个GPI/O端口有两个32位配置寄存器(GPIOx\_CRL，GPIOx\_CRH)， 两个32位数据寄存器(GPIOx\_IDR和GPIOx\_ODR)，一个32位置位/复位寄存器 (GPIOx\_BSRR)，一个16位复位寄存器(GPIOx\_BRR)和一个32位锁定寄存器 (GPIOx\_LCKR)。通过设置这些寄存器，GPIO的每一位都可以被独立设置成以下 八种模式：

输入浮空

输入上拉 输入下拉 模拟输入 开漏输出 推挽式输出 推挽式复用功能 开漏复用功能

关于这八种模式的具体功能，请见附件“[**STM**32中**GPIO**的8种工作模式](http://blog.csdn.net/kevinhg/article/details/17490273)”。

1. STM32 GPI0配置流程

以上是关于STM32GPIO的简单介绍，我们在实际操作的过程中，无需去做 复杂的寄存器设置，只需调用固件库里面的函数，即可实现对GPI0的输出操作。 一般的操作流程如下：

\*开启GPIO时钟（在STM32中，所有的外设固件，都需要开启相应的时钟）； \*选择需要配置的引脚；

\*选择引脚工作速度；

\*配置GPI0模式（八种模式之一）；

\*初始化GPIO 口

1. STM32 GPIO配置相关函数和程序

正如我们前文中所说的，STM32固件库提供给我们了一系列强大的系统函 数，具体库函数的使用参照附件《STM32固件库使用手册》。因此我们只需要使 用这些系统函数，即可以完成对GPIO的配置，在这里，我们主要使用到的配置 函数有两个：

RCC\_APB2PeriphClockCmd(uint32」RCC\_APB2Periph, FunctionalState NewState); GPIO」nit(GPIO\_TypeDef\* GPlOx, GPIO\_lnitTypeDef\* GPIO\_lnitStruct);

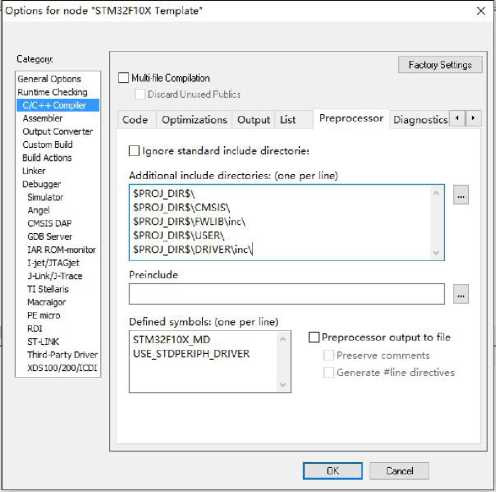
这两个函数各个参数的具体说明，请见《STM32固件库使用手册》。这里有两点小技巧：

(1)细心的同学可能发现，在stm32f10x\_rcc.h里面还定义了 RCC\_APB2PeriphClockCmd函数，很多初 学者可能在如何选择这两个函数的时候无从下手，这里教大家一个小技巧，只要看他的系统框图，如果对 应固件是连接在APB1总线上的，那就使用RCC\_APB1PeriphClockCmd，如果是连接在APB2上的，那就使用 RCC\_APB2PeriphClockCmd。

(2)很多同学刚刚使用这个固件库的时候，感觉每个参数都要参考手册有点麻烦，其实不需要，我们 只要去找这个函数的定义，就可以看到定义列表那边列出的每个参数列表，具体请看视频操作。

好了，有了以上两个基础知识点之后，我们就开始来写代码。打开第二章我 们建立好的固件库模板，首先我们要操作的就是在模板的跟文件夹里面建立一个 文件夹，我们命名为“DRIVER”，用于存放外设的配置代码。在“DRIVER”底下， 我们再建立两个文件夹“inc”（存放.h文件）和“src”（存放.c文件）。接 下来，由于我们需要操作gpio用于控制LED灯，因此我们分别在“inc”和“src” 里面加入“led.h”和“led.c”文件，并且按照之前教过大家的方法将“led.c” 文件加入IAR的代码管理文件夹“DRIVER”里面，如图22所示。同时，我们需 要在设置里面添加“led. h”的包含路径，如图23所示。

STM32F10X Template - IAR Embedded Workbench IDE  
File Edit View Project Simulator Tools Window Help

图22增加“led.c”

□运B g|晨|其\_魯|幻n || …豸V V々團險舞赍過鉍|呦邶土也

|  |  |
| --- | --- |
| Workspace \* | stm32fl0x.h | stm32fl0x\_gpio,c | stm32fl0x\_gpio,h |misc,c |main.c led.c |
| Debug |  |
| Files  □ g STM32F1 OX Template... ^  l-EDCMSIS |-pD DRIVER |
| Mu I?) |  |
| HuDFWUB HuC] USER ■-E D Output |  |

图23添加丨ed.h的文件包含路径

完成以上操作之后，我们就开始来写led.h的预处理代码。很多同学可能以 前受某些影响，喜欢把函数写在头文件里面，这是非常不可取的，从现在开始， 我们就来规范的把头文件和函数分离出来。一个空的头文件，为了防止多次重复 包含，因此我们需要程序片段1的格式代码，以“led.h”为例。

#ifndef\_\_LED\_H\_

#define LED H #endif

程序片段1

这样，我们就可以用#include〃led.h〃包含了。我们再来细致地分析程序片 段1里面的代码，其实就是#ifndef (标号）#define (标号）#endif的一个 预处理。每次遇到#include〃led.h〃的时候，编译器就开始来判断：如果（标号） 没有被定，就定义，万一定义了，就结束这个头文件的包含。

我们在“led.h”里面包"led.h〃即可。然后我们接着写“led.h” 里面的内容。第一点，所有与STM32固件库相关的操作，都需要包含 “stm32f10x.h”这个头文件；第二点，由于这里需要使用到GPIO,因此首先需 要包含STM32固件库里面GPIO相关代码的头文件“stm32f10x\_gpio.h”；第三 点，由于GPIO的操作和RCC相关，因此需要包含“stm32f10x\_rcc.h”。如代码 片段2所示。

#ifndef—LED\_H\_

#define —LED\_H\_

r包含相关的头文件\*/

#include "stm32f10x.h"

#include "stm32f10x\_gpio.h"

#include "stm32f10x\_rcc.h"

#endif

程序片段2

到此为止，我们对于“led.h”的编写就暂时告一段落，我们把目光转到 “led.c”上面。如我们上文所说的一样，“led.c”首先要做的一件事情，就是 把其头文件“led.h”包含进去。包含好之后，就可以按照我们的步骤来一步步 实现GPIO的配置函数，我们把GPIO的配置函数称为led\_gpio\_config(void)。

首先第一步，开启GPIO时钟。查看STM32的内部结构图，我们可以得知所 有的GPIO都是挂载在APB2总线上买的，因此，我们需要用到的函数就是 RCC\_APB2PeriphClockCmd(u32 RCC\_APB2Periph, FunctionalState NewState)。关于 此函数，有两个传递参数。第一个RCC\_APB2Periph表示需要操作的外设，我们查看 原理图“Main Board.SchDoc”可知：PB15 控制 LED\_G，PA11 控制 LED\_Y，PA12 控制 LED\_O, PB10控制LED\_R。我们现在先来以LED\_G为例，因此第一个传递的参数就是应该是GPIOB (具体参数列表，请看固件库手册或者视频）；第二个参数表示新状态，“ENABLE”代 表使能，“DISABLE”代表使能，由于我们要使能它，因此应该为“ENABLE”。因此这 个函数可以写为：“RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE);”。

第二步，选择需要配置的引脚。首先，我们需要定义一个GPIO\_InitTypeDef 的结构体GPIO\_InitStructure，这个结构体的具体描述，我们可以在 “stm32f10x\_gpio.h”里面找到，包含了所有与引脚相关的元素。由于我们需要 控制的引脚是PB15,因此我们选择的引脚是GPIO\_Pin\_15，即 “GPIO\_InitStructure.GPIO\_Pin = GPI0\_Pin\_15;”。

第三步，选择引脚速度（频率）。一般来说，我们定义成最快的速度。 “GPIO\_InitStructure.GPIO\_Speed = GPI0\_Speed\_50MHz;”。

第四步，配置GPIO模式。通过第二节里面的电路设计，我们知道LED是通 过一颗NPN型的三极管去做开关驱动的，因此，我们设置成端口推挽输出模式， 去推动三极管打开，“GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP”。

第五步，初始化GPIO〇STM32固件库里面提供了一个GPIO的初始化函数 GPIO\_Init，因此我们只需要使用此函数，就可以完成端口的初始化，这个函数 的传递参数有两个个，第一个是我们需要初始化的端口名，第二个即是我们上文 里面定义的GPIO\_InitTypeDef类型的结构体。

“GPIO\_Init(GPIOB, &GPIO\_InitStructure);”

具体程序如程序片段3所示。函数写好之后，记住需要在相应的.h文件里 面进行申明。

#include "led.h"

"LED\_G驱动GPIO初始化函数\*/ void led\_gpio\_config(void)

{

GPIO」nitTypeDefGPIO」nitStructure;

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE);

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_15;

GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO」nit(GPIOB,&GPIO\_InitStmcture);

}

程序片段3

至此为止，我们的驱动程序已经完成，接下来要做的只要在主函数里面调用 端口配置函数，并且使得这个引脚输出高电平，即可以点亮LED\_G。输出高电平 的程序很简单，只要使用固件库里面的函数“GPIO\_SetBits(GPIO\_TypeDef\* GPIOx, uint16\_tGPIO\_Pin);”即可完成相应位的输出。具体参照固件库使用手 册的函数，我们可以在主函数里面写出函数“GPIO\_SetBits(GPIOB, GPIO\_Pin\_15);'Main函数如程序片段4所示。接下来我们进行make，然后进 行下载即可，下载方式详见《JC-Debugger使用手册》。

#include "led.h"

|  |  |
| --- | --- |
| int main(void)  {  led\_gpio\_config(); |  |
| //配置GPIO |
| GPIO\_SetBits(GPIOB, GPIO\_Pin\_15); | //将PB15设置成高电平 |
| while(1); | //等待 |
| return 0;  } |  |

程序片段4

之后，我们就可以看到LED\_G被点亮了。如图24所示。



图24点亮第一个LED

3.1.2延迟函数相关（系统滴答）一一 LED闪烁灯

回顾我们的51单片机编程，当我们需要做系统延迟的时候，最常采用的一 种方式就是使用for循环的空语句等待来实现。当然，在STM32里面也可以这么 实现。但是在STM32的Cortex内核里面，有个比其更加精准的定时器专业用于 系统定时，我们称之为Cortex系统定时器（SysTick，系统滴答）〇Systick就 是一个定时器而已，只是它放在了 NVIC (中断事件）中，主要的目的是为了给 操作系统提供一个硬件上的中断（号称滴答中断）。这样，只要设置好其中断的 时间，就可以每隔一定时间跳入其处理程序，通过这种方式，我们可以做一些分 时的任务处理。然而，由于我们刚刚接触STM32,因此我们本课程内容，仅仅是 用其做一些延迟函数的处理。可能有些同学有疑问，微控制器的定时器资源一般 比较丰富，比如STM32存在8个定时器，为啥还要再提供一个SYSTICK?原因就 是所有基于ARM Cortex\_M3内核的控制器都带有SysTick定时器，这样就方便了 程序在不同的器件之间的移植。而使用RT0S的第一项工作往往就是将其移植到 开发人员的硬件平台上，由于SYSTICK的存在无疑降低了移植的难度。具体 Systick的概述，请参考《Cortex-M3权威指南》179页。

参考《STM32F10xxx Cortex-M3编程手册》我们可知，SysTick是一个24 位的系统时钟，它是一个从重载值递减到0会产生一个SysTick的事件，。整个 SysTick—共有四个相关的寄存器，如图25所示。

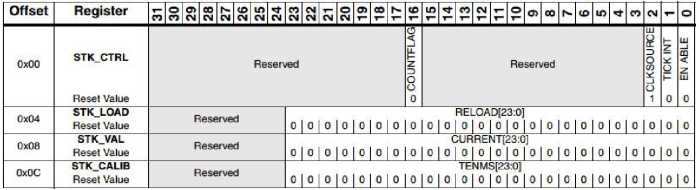


图25 SysTick相关寄存器

SysTick\_CTRL :控制和状态寄存器 SysTick—LOAD :重装载寄存器 SysTick\_VAL :当前值寄存器 SysTick—CALIB :校准值寄存器

同样的，对其硬件部分的了解，我们先止步于此，我们主要关心的是其编程 思路。

新建timer.c和timer.h，存放所有时钟相关的代码。关于SysTick的编程 流程如下：

\*配置系统时钟；

•配置 SysTick；

籲写SysTick中断处理函数；

籲编写delay延迟函数；

第一步，先让我们来设置系统时钟。关于系统时钟的配置，我们可以直接使 用默认的固件库函数“void SystemInit(void);”，这个函数在固件库手册上面 是没有的，一旦使用默认配置之后，整个STM32的系统时钟就会被配置成： SYSCLK(系统时钟）=72MHZ (系统最高允许时钟）；

AHB总线时钟 =72MHZ(AHB最高允许时钟）；

APB1总线时钟 =36MHZ(APB1最高允许时钟）；

APB2总线时钟 =72MHZ(APB2最高允许时钟）；

第二步，配置SysTick。我们在设置SysTick的时候，只用到“core\_cm3.h” 文件的函数 “—STATIC\_INLINE uint32\_t SysTick\_Config(uint32\_t ticks)”。 这个函数在固件库里面是没有介绍的，因为这个函数是在“core\_m3.h”里面定 义的，所以不属于STM32固件库的范畴。参考《STM32F10xxx参考手册》第80 页的STM32系统时钟框图，我们可以知道，系统时钟（AHB，此时为72MHz)经8 分频或者不分频之后产生的时钟给Systick作为时钟震荡源，因此此时的 Systick默认为频率为72MHz，如果需要使用8分频之后的频率，可以使用函数 “SysTick\_aKSourceConfig(SysTick\_aKSource\_HCLK\_Div8);，，，因此我们只 需要把 Systick 设置成 72000 时（计算方式：（1/72000000Hz)\*72000 次=1ms)， 就能产生1ms时间基准，说白了就是一个中断信号。

我们先来编写Systick的初始化函数。如程序片段5所示。

voidSystick」nit(void)

{

if (SysTick\_Config(72000))

{

/\* Capture error \*/ while(1);

}

}

程序片段5

第三步，编写Systick的中断处理函数。对于STM32所有的中断处理函数，

我们都可以在对应的“startup\_stm32f!0x\_xx.s”里面找到其入口。比如，在做

Systick中断处理的时候，我们选择的入口地址就是“SysTick\_Handler”。因

此，我们可以写如下的代码，如程序片段6所示。同时，需要把“stm32f10x\_it.c”

里面的“SysTick\_Handler”入口屏蔽，不然会报错。

\_\_IO uint32\_t TimingDelay; void TimingDelay\_Decrement(void)

{

if (TimingDelay != 0x00)

{

TimingDelay--;

}

}

void SysTick\_Handler(void)

{

TimingDelay\_Decrement();

}

程序片段6

第四步，写delay\_ms函数。如程序片段7所示。

void delay\_us(\_丨O uint32\_t nTime)//延迟函数，设置为 US {

TimingDelay = nTime;//时钟滴答数 while(TimingDelay != 0);

}

程序片段7

完成了上述步骤之后，我们接下来做的，只需要在main.c里面，每隔一定 的时间，将灯闪烁点亮就可以了。主函数如程序片段8所示。

|  |  |
| --- | --- |
| #include "led.h" #include"timer.h" int main(void)  {  SystemInit(); |  |
| //初始化系统时钟 |
| Systick一Init(); | //配置系统滴答 |
| led\_gpio\_config(); | //配置GPIO |
| //SysTick一CLKSourceConfig(SysTick一CLKSource一HCLK一Div8); | |
| while(1)  {  GPIO一SetBits(GPIOB, GPIO一Pin一15); |  |
| //将PB15设置成高电平 |
| delay一ms(1000);  GPIO一ResetBits(GPIOB,GPIO一Pin一15); | //将PB15设置成低电平 |
| delay一ms(1000); | "等待 |
| }  return 0;  } |  |

程序片段8

“timer.h”如程序片段9所示。

#ifndef\_\_TIMER\_H\_

#define\_\_TIMER\_H\_

#include "stm32f10x\_tim.h"

#include"stm32f10x\_rcc.h"

#include "stm32f10x\_it.h"

#include "misc.h"

extern —IO uint32\_t TimingDelay;

voidSystick」nit(void);

void TimingDelay\_Decrement(void);

void delay\_ms(\_IO uint32\_t nTime); //延迟函数，设置为 ms

#endif

程序片段9

“timer.c”程序如程序片段10所示。

#include"timer.h"

\_\_IO uint32\_t TimingDelay;

/\*SystemCoreClock / 1000000 1us\*/

/\*SystemCoreClock /100000 10us\*/

/\*SystemCoreClock /10000 100us\*/

/\*SystemCoreClock /1000 1ms\*/ voidSystick」nit(void)

{

if (SysTick\_Config(SystemCoreClock / 1000)) //可以直接写 72000

{

/\* Capture error \*/ while (1);

}

}

void TimingDelay\_Decrement(void)

{

if (TimingDelay != 0x00)

{

TimingDelay--;

}

}

void delay\_ms(—丨O uint32\_t nTime)//延迟函数，设置为 US

{

TimingDelay = nTime;//时钟滴答数 while(TimingDelay != 0);

}

void SysTick一Handler(void)

{

TimingDelay一Decrement。；

}

程序片段10

3.1.3 GPIO输入一一按键点灯程序

STM32每个GPI0都能独立地配置成输入模式，并且还可以与外部中断进行 复用。因此对于按键的操作，我们分成两个部分来讲，第一个部分即是通用的查

询式按键输入，第二个部分即是中断方式输入。我们先来看下原理图，发现K1, K2，K3，K4 分别连接到 PB11，PB12，PB13，PB14。

1. 通用查询式按键输入

同样的，对按键的操作，我们也将其放入两个新的文件里面，叫做key.c 以及key.h。按键输入的GPIO配置流程与LED驱动的配置流程一样，如下所示。

\*开启GPIO时钟（在STM32中，所有的外设固件，都需要开启相应的时钟）；

\*选择需要配置的引脚；

\*选择引脚工作速度；

\*配置GPI0模式（八种模式之一）；

\*初始化GPIO 口

第一步，开启 GPI0 时钟。RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE);

第二步，选择需要配置的引脚。首先，我们需要定义一个GPIO\_InitTypeDef的结 构体GPIO\_InitStructure。由于我们需要检测的引脚是PB11,因此我们选择的引脚是 GPIO\_Pin\_11，即“GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_11;，，。

第三步，选择引脚速度（频率）。一般来说，我们定义成最快的速度。

“GPIO\_InitStructure.GPIO\_Speed = GPI0\_Speed\_50Mfc;’，。

第四步，配置GPIO模式。由于我们检测按键的实质，就是检测GPIO的电平状态， 因此需要将端口设置成输入模式，而仔细观察我们的电路，每颗按键都用电阻上拉，因 此我们只需要使用浮空输入即可。“GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_m\_FLOATmG;”。

第五步，初始化GPIOATM32固件库里面提供了一个GPIO的初始化函数GPIO\_Init, 因此我们只需要使用此函数，就可以完成端口的初始化，这个函数的传递参数有两个个， 第一个是我们需要初始化的端口名，第二个即是我们上文里面定义的GPIO\_InitTypeDef 类型的结构体。“GPIO\_Init(GPIOB，&GPIO\_InitStructure);”

完成了以上的端口配置之后，我们在主函数里面只要实时读入此端口的状态，假设 检测到K1有输入，[已0\_0就被点亮，再按一下，LED\_G就被熄灭。GPIO配置代码如程序 片段11所示，主程序控制代码，如程序片段12所示。

#include"key.h"

void key\_gpio\_config (void)

{

GPIO」nitTypeDef GPIO\_lnitStructure;

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE);

GPIO」nitStructure.GPIO\_Pin = GPIO—Pin—ll- GPIOJnitStructure.GPIO—Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING; GPIO」nit(GPIOB,&GPIO」nitStructure);

}

程序片段11

#include "led.h" #include"timer.h" #include"key.h" int main(void)

{

u8 led\_state;

//定义变量觉得LED状态 //系统初始化 //初始化系统滴答 //配置 LED GPIO //配置按键GPIO //设置LED初始状态

SystemInit();

Systick一Init(); led\_gpio\_config(); key一g p i o一co nfi g();

GPIO一ResetBits(GPIOB,GPIO一Pin一15);

while⑴

{

if(GPIO一ReadInputDataBit(GPIOB, GPIO—Pin一11) == RESET)//如果按键按

下

{

delay一ms(10); //延迟 10ms 消抖

if(GPIO一ReadInputDataBit(GPIOB, GPIO一Pin一11) == RESET)

{

}

}

while(GPIO一ReadInputDataBit(GPIOB, GPIO\_Pin\_11) == RESET);//松键检

测

if((led一state % 2) == 1)

{

GPIO一ResetBits(GPIOB, GPIO一Pin一15); //关灯

}

else

{

GPIO\_SetBits(GPIOB, GPIO\_Pin\_15); //开灯

}

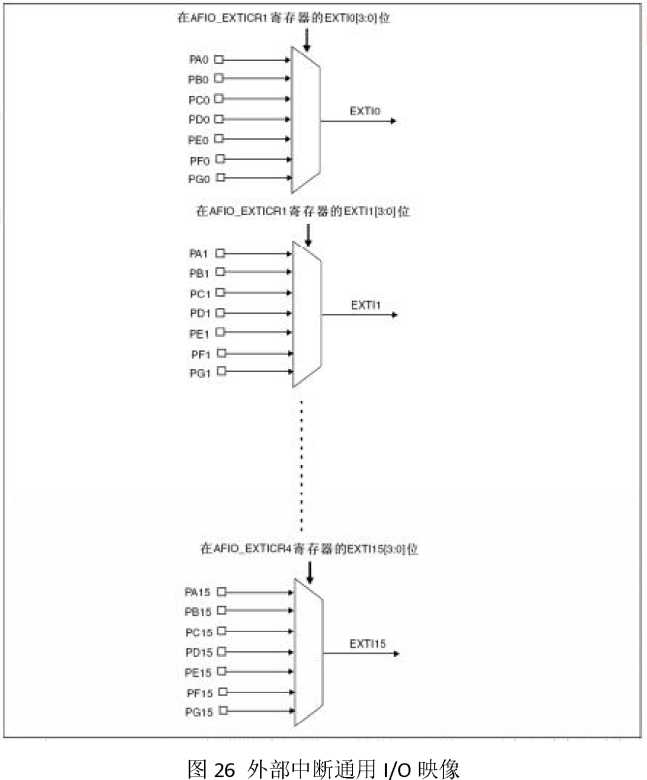
}

}

程序片段12

1. 通用中断式按键输入

STM32内部有19个能产生事件/中断请求的边沿检测器。每个输入线可以独 立地配置输入类型(脉冲或挂起)和对应的触发事件(上升沿或下降沿或者双边沿 都触发)。每个中断线都可以选择性的与外部端口相连接。具体可以参照图26。 除了以下16个中断和外部GPIO相连，还有EXTI线16连接到PVD输出，EXTI 线17连接到RTC闹钟事件，EXTI线18连接到USB唤醒事件，EXTI线19连接到 以太网唤醒事件(只适用于互联型产品)。



关于STM32所有的中断向量举例，可以参照文档《STM32中文参考手册》 第130 ~ 133页。在这里，我们只是简单地来看下STM32外部中断（EXIT)的工 作流程。要产生中断，必须先配置好并使能中断线。当外部中断线上发生了期待 的边沿时，将产生一个中断请求，随机跳到相应的中断入口进行服务程序的处理， 处理好之后，清除中断标志位以及返回入口处。

在STM32中，所有与中断有关的配置（不包含Systick)，我们都需要用到 一个寄存器 NVIC (嵌套向量中断控制器，Nested Vectored Interrupt Controller)。 那接下来，我们只需要查看STM32固件库手册与NVIC有关的函数，即可进行中 断的配置。因此，关于中断的操作，我们同样可以得到如下的思路。

\*配置时钟；

籲配置GPIO (—般设置成浮空输入）；

鲁将GPIO与外部中断线连接起来；

籲配置EXIT;

籲配置NVIC;

\*写中断服务程序。

接下来，我们一步一步来完成上面的步骤。

第一步，配置时钟。我们需要同时开启GPIO和AFIO的时钟，注：在这里由 于IO 口不是当作普通IO来用的，而是启用其复用功能，因此需要开启AFIO时 钟。“RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA | RCC\_APB2Periph\_AFIO, ENABLE);”。

第二步，配置GPIO,假设我们还是使用PB11:

GPIO」nitStructure.GPIO\_Pin = GPI0\_Pin\_11;

GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode」N\_FLOATING; GPI〇」nitStructure.GPI〇\_Speed = GPIO\_Speed\_50MHz; GPI〇」nit(GPI〇B,&GPI〇」nitStructure);

第三步，将GPI〇和外部中断线连接起来。即，将PB11与EXIT\_11连接起来。 “GPI〇\_EXT旧neConfig(GPI〇\_PortSourceGPI〇B, GPI〇\_PinSource11);”。

第四步，配置EXIT。

EXTI\_InitTypeDef EXTI\_InitStructure;

EXTI\_InitStructure.EXTI\_Line = EXTI\_Line11;

EXTIJnitStructure.EXTI一Mode = EXTI\_Mode」nterrupt;

EXTIJnitStructure.EXTI一Trigger = EXTI一Trigger\_Falling; EXTI」nitStructure.EXTI\_UneCmd = ENABLE;

EXTI」nit(&EXTI」nitStructure);

分析以上的程序代码，我们来具体研宄一下。首先，定义了一个关于外部中 断的初始化结构体。接着所做的工作即是对此结构体的每一个元素进行赋值。最 后再用初始化函数是能此结构体。关于结构体每个元素的赋值参数，参考《STM32 固件库中文手册》即可。

第五步，配置NVIC。

NVICJnitTypeDef NVICJnitStructure;

NVIC一PriorityG roupConfig(NVIC一PriorityGroup一0); //设置组优先级

NVIC」nitStructure.NVIC\_IRQChannel = EXTI15一10」RQn;//外部中断线 NVIC」nitStructure.NVIC」RQChannelPreemptionPriority = 0; //设置抢占优先

级

NVIC」nitStructure.NVIC」RQChannelSubPriority = 0; //设置子优先级

NVIC」nitStructure.NVIC」RQChannelCmd = ENABLE; "使能 IRQ 中断

NVIC」nit(&NVIC」nitStructure);

以上部分是对中断控制器来进行配置的，具体操作，请看程序注释，并且联 合《STM32固件库中文手册》进行理解和学习 第六步，写中断服务程序。 void EXTI15一10一IRQHandler(void)

{

if (EXTI一GetITStatus(EXTI一Linell) != RESET)

{

EXTI一ClearITPendingBit(EXTIJJnell);//清除标志

}

}

main.c具体程序，请参考代码片段13所示。

|  |  |
| --- | --- |
| #include "led.h" #include"timer.h" #include"key.h，丨  int main(void)  {  //u8 led\_state; |  |
| SystemInit(); | //初始化系统时钟 |
| Systick一Init(); | //初始化系统滴答 |
| led\_gpio\_config(); | //初始化LED GPIO |
| key一g p i o一co nfi g(); | //初始化按键GPIO |
| GPIO一ResetBits(GPIOB,GPIO一Pin一15); | //默认灯初始为关 |
| while(1);  } | //等待中断 |

程序片段13

按键GPIO中断初始化程序“void key\_gpio\_config (void)”如程序片段14所

示。

#include"key.h，丨 #include"timer.h" u8 led\_on\_off; void key\_gpio\_config (void)

{

GPIOJnitTypeDef GPIOJnitStructure; //定义 GPIO 结构体 NVIC\_InitTypeDef NVIC\_InitStructure; //定义外部中断结构体 EXTIJnitTypeDef EXTI\_InitStructure; //定义中断控制器结构体

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB | RCC\_APB2Periph\_AFIO, ENABLE);//开启GPIOB和AFIO (复用端口）时钟

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_11;

GPIO」nitStmcture.GPIO\_Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING; GPIO」nit(GPIOB,&GPIO」nitStructure); //初始化 GPIO

GPIO\_EXTILineConfig(GPIO\_PortSourceGPIOB, GPIO\_PinSource11); //将 PB11 和 EXIT\_11 连接

EXTI\_InitStructure.EXTI\_Line = EXTI\_Line11; //选择 EXTI\_Line11

EXTI」nitStructure.EXTI\_Mode = EXTI\_Mode」ntermpt; //外部中断模式 EXTI」nitStructure.EXTI\_Trigger= EXTI\_Trigger\_Falling; //下降沿出发 EXTI\_InitStructure.EXTI\_LineCmd = ENABLE;

EXTI\_Init(&EXTI」nitStructure);

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_0); //设置中断组优先级 0 NVIC\_InitStmcture.NVIC\_IRQChannel = EXTI15\_10」RQn; //PPP 外部中断线 NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 3; //设置抢占优

先级为3

NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0; //设置从优先级为 0

NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;

NVIC一Init(&NVIC一InitStructure);

}

程序片段14

中断处理程序的代码，如程序片段15所示。

void EXTI15一10一IRQHandler(void)

{

u16 i, j;

if(EXTI一GetITStatus(EXTI一Linell) != RESET) //如果是 EXIT\_11 出发的中

断，则处理

{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO一ReadInputDataBit(GPIOB, GPIO一Pin一11) == RESET) //消抖 {

led一on一off = led一on一off > 3 ? 0 : led一on一off + 1;

}

if((led一on一off % 2) == 1)

{

GPIO一SetBits(GPIOB, GPIO一Pin一15);

}

else

{

GPIO一ResetBits(GPIOB, GPIO一Pin一15);

}

EXTI一ClearITPendingBit(EXTI\_Line11);//清除标志

}

}

程序片段 15

3.1.4串口实验

对于STM32来说，串口是使用频率非常高的一个外设，因为现在实际使用 中，串口和电脑通讯可能使用的频率不是很高了，更多的情况下是用在系统和系 统之间通讯，或者单片机和传感器进行通讯。这里对于串口的传输协议，我们就 不再赘述，有兴趣的同学可以自行去理解。

STM32F103C8T6为用户提供了 3个USART，STM32的USART利用分数波特 率发生器提供宽范围的波特率选择。它支持同步单向通信和半双工单线通信，也 支持LIN(局部互连网)，智能卡协议和IrDA(红外数据组织)SIR ENDEC规范，以及 调制解调器(CTS/RTS)操作。它还允许多处理器通信。使用多缓冲器配置的DMA 方式，可以实现高速数据通信。

STM32的串口，不仅可以用来做通讯使用，还能用作ISP编程（In-System Programming，在线编程）。这就可以为一些没有SWD编程器的用户，提供方便。 我们这一讲的ISP，是基于USART1来讲的，目前STM32也只支持USART1来进

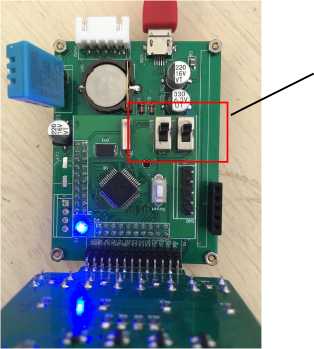
行ISP编程。

我们本节课的内容分成4部分来讲，第一部分，介绍ISP编程，第二部分介 绍简单的串口单字节发送程序，第三部分介绍简单的串口单字节接收程序，第四 部分介绍多字节串口协议通讯程序。

(1)使用USART1进行ISP编程。

使用ISP编程时，我们先要来设置STM32的启动方式。在我们的核心板上面， 有两个拨码开关，他们对应得就是BOOTO和BOOT1两个选项，对于ISP编程时， 我们需要将BOOTO和BOOT1接高电平，拨码开关的选择如图27所示。

两个都拨 向上方



图**27 ISP**时**BOOT**配置

然后插上USB与电脑连接，打开“FlyMCU”软件，点击工具栏的“搜索串 口 ”即可自动找到电脑上面的COM 口（注：如果不是开发板对应的USART,则

需要手动更改）。然后在“联机下载时的程序文件”里面，装入需要下载的“.HEX” 文件，点击“开始编程”，同时按下核心板上的复位按钮（可能需要多按几次， 编程时，最好插入显示板，因为显示板上面的大电容对CH340具有很好的稳压 作用）。这样就完成了 ISP下载，如图28所示。

0 FlyMcu V0.188--单片机汪线编程专家--[www.mcuisp.com](http://www.mcuisp.com)

系统(X)帮助(Y) Language 搜索串 口〇〇 PortC0M9 bps:115200 联机下载时的程序文件:

**iAww**.**mcuisp.com** ^**S**^**IS**(**W**) 0编程前重装文件

f 材读材 \STM32F 10X Template pebug 疋 xe\STM32F 10X Template, hex •

手持万用编程器 免费 STMIAP NXPISP EP968\_RS232

^校验

开始谝程CP)

|读器件信息演除芯片(Z) 迭顷字节区：

0编程到FLASH时写迭项宇节 设定迭项字节等

读FLASH

芯片己不包含此信息>

开始全片瘵涂《全片瘵狳时间会比较长.请酎心等

侯！）

全片瘵狳成功

开始连接...5,接收到：1F 1F

在串口C0M9连接成功@115200bps,耗时750毫秒

芯片内BootLoader版本号：2.2

芯片PID: 00000410 STM32F10xxx\_Hedium-

density

读出的选项宇节：

A5 5AFF0 0FF00FF00FF00FF00FF00FF00 96位的芯片唯一序列号： [57FF6E067583575233211481]

[066EFF57 52578375 81142133]

芯片FLASH容量为64KB

芯片SRAM容量为65535KB(此信息仅供参考，新版本 芯片己不包含此倌息>

第e59毫秒，己准备好

共写入2KB,进度100%,耗时1625毫秒

写入选攻宇节：A5 5A FF 00 FF 00 FF 00 FF 00

FF 00 FF 00 FF 00

成功写入选项宇节

写入的选项宇节：

A5 5AFF0 0FF00FF00FF00FF00FF00FF00 [www.mcuisp](http://www.mcuisp). com{全找机手持编程器EP968,全球首 创）向您报告.侖令执行完毕.一切正笨

不使用RTS和DTR

图28 ISP下载成功提示

下载成功后，将图27中的两个拨码开关，都拨到反方向，再按下核心板上 的复位按键，即可运行程序。

(2)简单的串口单字节发送程序

首先我们先来看我们的开发板，我们的开发板上面有一个USB 口，这边引 出的是USART1,即和PA9和PA10复用的串口。我们首先需要在“DRIVER”文件 夹的“inc”和“src”里面建立“usart.h”和“usart.c”，然后将其加入我们的工 程里面，这是一个出现频率非常高的操作，因此，我们就不做重复解释。建立好 了之后，我们就开始来编写串口的发送程序。关于串口的配置问题，我们还是先 给大家来罗列一下思路，然后再按照我们的思路来一步步实现。配置的一般步骤 如下：

开启相关的RCC时钟;

配置相应的GPIO 口；

配置相应的串口模式；

第一步，我们先使能对应外设的时钟。这里，我们不仅要开启串口的时钟， 还需要把用到的GPIO 口和AFIO 口的时钟也开启起来。查询《STM32中文手册》 我们可以知道，STM32的串口是挂载在APB2总线上面的，和GPIO，AFIO—样， 由于我们使用的是USART1,对应GPIO为PA9和PA10，因此，我们只需使用 “RCC\_APB2PeriphClockCmd( RCC\_APB2Periph\_USART1 |RCC\_APB2Periph\_GPIOA |RCC\_APB2Periph\_AFI0, ENABLE);” 即可开启。

第二步，配置相应的GPIO 口。由于我们使用的是USART1,查找数据手册我 们可知，USART1对应的复用引脚为PA9和PA10,因此我们只需要设置PA9和PA10 即可。因为PA9对应的是TXD，PA10对应得实RXD，因此我们需要将PA9设置 为复用推挽输出，将PA10设置为浮空复用输入即可，代码参照程序片段16所示。

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_9; //USART1TX

GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_AF\_PP; //复用推挽

输出

GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO」nit(GPIOA, &GPIO\_InitStructure);

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_10; //USART1 RX

GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING; //复用浮空

输入

GPIO」nit(GPIOA, &GPIO\_InitStructure);

程序片段16

第三步，配置相应的串口模式。我们这里主要设置的是串口的发送方式，通 过函数“USART\_Init()”来实现。查询《STM32固件库中文手册》，我们可以完成 如代码片段17的配置。

USART\_lnitTypeDefUSART」nitStmcture; //串 口初始化结构体声明

USART\_lnitStmcture.USART\_BaudRate = 115200;"设置波特率为 115200bps USART\_InitStructure.USART\_WordLength = USART\_WordLength\_8b; //数据位 8 位 USART\_InitStructure.USART\_StopBits = USART\_StopBits\_1; //停止位 1 位

USART\_InitStructure.USART\_Parity = USART\_Parity\_No; //无校验位

USART\_InitStructure.USART\_HardwareFlowControl =

USART\_HardwareFlowControl\_None; //无硬件流控

USART\_InitStructure.USART\_Mode = USART\_Mode\_Tx; //发送模式打开

USART\_Init(USART1, &USART\_InitStructure); //配置串口参数函数

USART\_ClearFlag(USART1,USART\_FLAG\_TC); //清除发送完成标志位

USART\_Cmd(USART1, ENABLE); "使能串口 1

程序片段17

对于逻辑函数，我们采用如下的形式，当按键将LED点亮之后，我们就发送 数据0x01，LED熄灭后，就发送0x00。串口初始化程序如程序片段18所示，主 函数如程序片段19所示。而我们需要改变的仅仅是中断接收程序。如程序片段 20所示。

void usart\_config(void)

{

USART\_InitTypeDef USART\_InitStructure; //串口初始化结构体

声明

USART\_InitStructure.USART\_BaudRate = 115200; //设置波特率为 115200bps

USART\_InitStructure.USART\_WordLength = USART\_WordLength\_8b; //数据位 8 位 USART\_InitStructure.USART\_StopBits = USART\_StopBits\_1; //停止位 1 位

USART\_InitStructure.USART\_Parity = USART\_Parity\_No; //无校验位

USART\_InitStructure.USART\_HardwareFlowControl =

USART\_HardwareFlowControl\_None; //无硬件流控

USART\_InitStructure.USART\_Mode = USART\_Mode\_Tx; //接受模式打开

USART\_Init(USART1, &USART\_InitStructure); //配置串口参数函数

USART\_ITConfig(USART1, USART\_IT\_RXNE, ENABLE); "使能接收中断

USART\_ClearFlag(USART1,USART\_FLAG\_TC); //清除发送完成标志位

USART\_Cmd(USART1, ENABLE); "使能串口 1

}

程序片段18

|  |  |
| --- | --- |
| #include "led.h" #include "timer.h" #include "key.h" #include "usart.h"  int main(void)  {  //u8 led\_state; SystemInit(); Systick\_Init(); led\_gpio一config(); | //初始化系统时钟 //初始化系统滴答 //初始化LED GPIO |

key\_gpio\_config(); //初始化按键 GPIO

GPIO\_ResetBits(GPIOB,GPIO\_Pin\_15);//默认灯初始为关 usart1\_init();

|  |  |
| --- | --- |
| while(1); | //等待中断 |
| } | 程序片段19 |

void EXTI15\_10\_IRQHandler(void)

{

u16 i, j;

if(EXTI\_GetITStatus(EXTI\_Line11) != RESET) //如果是 EXIT—11 出发的中断，则处理

{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO\_ReadInputDataBit(GPIOB, GPI0\_Pin\_11) == RESET) //消抖

{

led\_on\_off = led\_on\_off > 3 ? 0 : led\_on\_off + 1;

}

if((led\_on\_off % 2) == 1)

{

GPIO\_SetBits(GPIOB, GPIO\_Pin\_15);

USART\_SendData(USART1,0x01);//发送 0x01

while(USART\_GetFlagStatus(USART1, USART\_FLAG\_TXE) == RESET);//等待发

送完 }

else

{

GPIO\_ResetBits(GPIOB, GPIO\_Pin\_15); USART\_SendData(USART1,0x00); //发送 0x00 while(USART\_GetFlagStatus(USART1, USART\_FLAG\_TXE)：

RESET);//等待发

}

送完

EXTI\_ClearITPendingBit(EXTI\_Une11);//清除标志

}

}

程序片段20

1. 简单的串口单字节接收程序

串口程序的发送方式和接收方式其实有着本质的不同，一般来说，串口发送 时候，仅仅只是使用顺序结构的发送函数来实现的，发送函数的任务也只是简单 地将串口的发送字节送入发送BUFFER寄存器里面。而串口的接受程序却是一般 采用中断的方式去做。因而在设置串口时候，我们还需要设置NVIC，以及使能 串口中断。而函数实现起来，仅仅只需要将串口单字节发送程序做简单的修改。 修改的内容有三个，第一个设置NVIC，第二个使能串口中断，第三个编写串口 接收中断函数。

第一步，设置NVIC。NVIC的设置主要使用函数NVICJnit，设置的方式和我 们前面所说的外部中断一样，只不过是修改一些配置的参数，如程序片段21所

示。

voidNVIC一Configuration(void)

{

NVIC一InitTypeDef NVIC一InitStructure; //NVIC 初始化结构体声明

NVIC一PriorityGroupConfig(NVIC一PriorityGroup一1);

NVIC一InitStructure.NVIC一IRQChannel = USART1一IRQn; "设置串口 1 中断

NVIC一InitStructure.NVIC一IRQChannelPreemptionPriority = 0; //抢占优先级

0

NVIC一InitStructure.NVIC一IRQChannelSubPriority = 0; //子优先级为 0

NVIC一InitStructure.NVIC一IRQChannelCmd = ENABLE; //使會 ^

NVIC一Init(&NVIC一InitStructure);

}

程序片段21

第二步，使能串口中断。这里面只需要在上述的串口配置程序里面修改几个 参数即可。

USART一InitStructure.USART一Mode = USART一Mode一Tx | USART一Mode一Rx;

//接受和发送模式都打开

USART一ITConfig(USART1, USART一IT一RXNE, ENABLE); //使能接收中

断

第二步，编与串口接收中断程序。我们在这里主要完成如下功能，当我的串 口发送“0x01”时，就将LED打开；发送“0x00”时，就将LED关闭。因此我们 需要找到串口接收程序的入口 “voidUSART1JRQHandler(void)”。在中断程序一 开始，我们首先要判断的就是是否接受标志位有效，只要使用代码 “USART一GetITStatus(USART1, USARTJT一RXNE) != RESET” 做判断即可，因此我们 可以得到如程序片段22的代码。在代码片段22里面，每次进入后，判断是否是 串口接收中断，如果是，则用 “rev一data = USART一ReceiveData(USART1);” 将数据 读入，并且把接收到的数据发送出来，便于我们调试。最后再去做判断，如果是 “00”，则关闭灯，如果是“01”，则打开灯。这样就完成了程序的操作。

|  |  |
| --- | --- |
| void USART1一IRQHandler(void)  {  u8 rev一data;  if(USART一GetITStatus(USART1, USART一IT一RXNE) != RESET)  是否非空 {  rev一data = USART一ReceiveData(USARTl); | //判断读寄存器 |

USART一SendData(USART1,rev\_data);

while(USART\_GetFlagStatus(USART1, USART\_FLAG\_TXE) == RESET); if(rev一data == 0x00)

|  |  |
| --- | --- |
| {  GPIO一ResetBits(GPIOB, GPIO一Pin一15);  }  else if(rev一data == 0x01)  { | //点亮LED |
| GPIO一SetBits(GPIOB, GPIO一Pin一15);  }  else  { | //熄灭LED |
| /  }  USART一ClearITPendingBit(USART1, USART一IT一RXNE);  //清除接受中断标志 }  }  程序片段22 | //不做操作 |

1. 多字节串口协议通讯程序

对于单字节的串口收发程序，是个非常简单的程序。然而，对于我们日常使 用却基本没什么帮助，因为我们平时用的比较多的是基于单字节收发程序基础上 的一种协议性质的通讯。

假设我们需要完成这样几个功能：

鲁通过PC的串口软件，控制电亮对应的LED灯。

鲁每次点灯灭灯之后，把最新的LED状态发送个PC的串口软件。

对于上述两点功能，我们完全可以用单字节传输的方式来做，只要用每一位 来代表一种状态就可以，这个实现起来很简单，就留给读者自己去研宄深入。

而我们在这里讲的是一种基于多字节通讯协议的实现方式。首先，我们来看 下什么是协议。协议从字面上理解，就是大家所约定的一种规定。而在计算机上面的解释就是：通信计算机双方必须共同遵从的一组约定。如怎么样建立连接、 怎么样互相识别等。只有遵守这个约定，计算机之间才能相互通信交流。既然有 了以上的解释，我们完全可以给我们的开发板拟定一个远程通讯的协议。由于我 们需要控制四个灯，因此我们至少需要4个字节，再加上1个起始字节，1个结 束字节，还有1个数据方向流字节，因此一共需要7个字节。如图29所示。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 字节0 | 字节1 | 字节2 | 字节3 | 字节 4 | 字节5 | 字节6 |

图29串口通讯协议格式

字节0：协议起始字节，永远为“Oxaa”；

字节1：数据流向字节，“0x09”为单片机传入电脑;

‘0xf9”为电脑发送

0x00” 为 LED1 灭，

给单片机；

字节2： LED1的状态， 字节3： LED2的状态， 字节4： LED3的状态， 字节5： LED4的状态，

£0xff” 为 LED1 亮 £0xff” 为 LED1 亮 £0xff” 为 LED1 亮 £0xff” 为 LED1 亮

0x00” 为 LED1 灭，

0x00” 为 LED1 灭，

0x00” 为 LED1 灭，

字节6：协议结束字节，永远为“0xdd”。

其实，一个标准的串口通讯协议，校验字节是必备的，不管是奇偶校验，或 者和校验等，都是不可获缺的。然而这里考虑到，我们刚刚开始来拟定简单的通 讯字节，并且我们协议的字节比较少，因此我们先不去加校验字节，后续如果有 复杂的超长指令字节通讯，我们再来拟定。

我们这个程序的大致思路如下：

\*完成四路按键点灯程序；

\*完成四路按键点灯程序时，将协议从单片机发送给电脑；

\*完成多字节串口接收程序；

\*完成多通讯协议的解析程序，并且开关相应的LED灯。

第一步，我们先将剩余的按键点灯程序来写好，由于我们之前详细讲过中断 式的按键点灯程序，因此我们这里只附上程序，不做详细的讲解。

在这里，我们还需要补充一点，由于我们现在需要控制的LED比较多，因此 如果还用之前的端口操作程序去操作，无法避免会出现很多问题，所以我们这里

需要用宏定义来定义每个灯的开关操作。宏定义如代码片段23所示。GPIO初始

化程序，如代码片段24所示。

#define LED1\_ON() do{GPIO\_SetBits(GPIOB, GPIO\_Pin\_15);}while(0); #define LED1\_OFF〇 do{GPIO\_ResetBits(GPIOB, GPIO\_Pin\_15);}while(0);

#define LED2\_ON() do{GPIO\_SetBits(GPIOA, GPIO\_Pin\_11);}while(0); #define LED2\_OFF() do{GPIO\_ResetBits(GPIOA, GPIO\_Pin\_11);}while(0);

#define LED3\_ON() do{GPIO\_SetBits(GPIOA, GPIO\_Pin\_12);}while(0); #define LED3\_OFF() do{GPIO\_ResetBits(GPIOA, GPIO\_Pin\_12);}while(0);

#define LED4\_ON() do{GPIO\_SetBits(GPIOB, GPIO\_Pin\_10);}while(0);

#define LED4\_OFF() do{GPIO\_ResetBits(GPIOB, GPIO\_Pin\_10);}while(0);

程序片段23

void led\_gpio\_config(void)

{

GPIO」nitTypeDef GPIO\_InitStructure;

/\*开启GPIOA和GPIOB的时钟\*/ RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA |

RCC\_APB2Periph\_GPIOB, ENABLE);

/\*配置 PB15 和 PB10\*/

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_15 | GPIO\_Pin\_10; GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP; GPIO」nit(GPIOB,&GPIO\_InitStmcture);

/\*配置 PA11 和 PA10\*/

GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_11 | GPIO\_Pin\_12; GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP; GPIO」nit(GPIOA,&GPIO\_InitStmcture);

}

程序片段24

第二步，完成四路按键点灯程序时，将协议从单片机发送给电脑。这里面只 列出中断电灯的一些代码而已，具体实现方式请参照上文内容。

我们先来复制上位中中断点灯的程序，利用剩余的按键，把对应的LED点亮， 如程序片段25所示。

voidEXTI15\_10」RQHandler(void)

{

u16 i, j;

if(EXTI\_GetITStatus(EXTI\_Line11) != RESET) //如果是 EXIT\_11 出发的中

断，则处理  
{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO\_ReadInputDataBit(GPIOB, GPIO\_Pin\_11) == RESET) //消抖 {

led\_on\_off[0] = led\_on\_off[0] > 3 ? 0 : led\_on\_off[0] + 1;

}

if((led\_on\_off[0] % 2) == 1)

{

LED1\_ON();

}

else

{

LED1\_OFF();

}

EXTI\_ClearITPendingBit(EXTI\_Line11);//清除标志

}

if(EXTI\_GetITStatus(EXTI\_Line12) != RESET) //如果是 EXIT\_11 出发的中

断，则处理  
{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO\_ReadInputDataBit(GPIOB, GPIO\_Pin\_12) == RESET) //消抖 {

led\_on\_off[1] = led\_on\_off[1] > 3 ? 0 : led\_on\_off[1] + 1;

}

if((led\_on\_off[1] % 2) == 1)

{

LED2\_ON();

}

else

{

LED2一OFF();

}

EXTI\_ClearlTPendingBit(EXTI\_Une12);//清除标志

}

if(EXTI一GetITStatus(EXTI一Line13) != RESET) //如果是 EXIT\_11 出发的中

断，则处理  
{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO一ReadInputDataBit(GPIOB, GPIO一Pin一13) == RESET) //消抖 {

led一on一off[2] = led一on一off[2] > 3 ? 0 : led一on一off[2] + 1;

}

if((led一on一off[2] % 2) == 1)

{

LED3一ON();

}

else

{

LED3一OFF();

}

EXTIJClearITPendingBit(EXTIJJne13);//清除标志

}

if(EXTI一GetITStatus(EXTI一Line14) != RESET) //如果是 EXIT\_11 出发的中

断，则处理  
{

for(i = 100; i > 0; i--)

for(j = 720; j > 0; j--); //延迟一会

if(GPIO一ReadInputDataBit(GPIOB, GPIO一Pin一14) == RESET) //消抖 {

led一on一off[3] = led一on一off[3] > 3 ? 0 : led一on一off[3] + 1;

}

if((led一on一off[3] % 2) == 1)

{

LED4一ON();

}

else

{

LED4\_OFF();

}

EXTI\_ClearlTPendingBit(EXTI\_Une14);//清除标志

}

}

程序片段25

然后要做的一件事情就是，每次在LED的状态改变时，我们要将当前状态的 LED状态发送给电脑端。因此，我们首先需要做的就是定义一个协议栈类型的结 构体。

typedef struct {

u8 start\_byte; u8 direction; u8 led\_state[4]; u8end一byte;

} usart\_stack;

接下来关于按键中断处理函数，为：

if((led一on一off[1] % 2) == 1)

{

LED2一ON();

tx一buffer.led一state[1] = 0xff;

}

else

{

LED2一OFF();

tx一buffer.led一state[1] = 0x00;

}

然后再通过发送函数发送出去，每次进入按键中断，处理完之后，就将当前 的状态进行发送，如程序片段26所示。

void led\_state\_tx (void)

{

u8i;

USART一SendData(USART1,tx\_buffer.start\_byte); while(USART\_GetFlagStatus(USART1, USART\_FLAG\_TXE) == RESET);

USART一SendData(USART1,tx一buffer.direction);

while(USART一GetFlagStatus(USART1, USART一FLAG一TXE) == RESET);

for(i = 0;i< 4; i++)

{

U SART一SendData(USART1, tx一b uffer.led一state [i]);

while(USART一GetFlagStatus(USART1, USART一FLAG一TXE) == RESET);

}

USART一SendData(USART1, tx一buffer.end一byte);

while(USART一GetFlagStatus(USART1, USART一FLAG一TXE) == RESET);

}

程序片段25

完成了以上的操作之后，我们再来写串口的接收函数，也同样如此。首先定 义一个接收的缓冲区。

然后每次进入串口接收中断之后，就将相应的数据先进行判断，然后存入对 应的结构体元素中。由于我们选择的字符比较特殊，因此只要直接判断开始、流 向是否都满足我们的要求，就可以决定是否接收后续的字节。而判断一组数据是 否接收完成，只需要判断结尾是否被放入相应的字符即可。

串口接收中断处理函数如代码片段26所示的程序。中断接收到串口协议数 据之后，我们只需要定时的去查询rx一buffer.bufferjockjlag这个变量，就可以知 道，串口协议有没有接受完成，一旦接收完成了，就把这个接收缓冲区锁住。关 于接受缓冲区，我们定义成一个宏定义结构体。因而，我们的协议解析函数，只 要查询到rx一buffer.bufferjockjlagzzl 了，即可判断协议接受完成，进行解析。

typedef struct {

u8 start一byte ; u8 direction; u8 led一state[4]; u8 end一byte; u8 data一pt; u8 buffer一lock一flag;

} usart一receive一buffer;

void USART1\_IRQHandler(void) {

u8 rev\_data;

if(USART\_GetITStatus(USART1, USART\_IT\_RXNE) != RESET) //判断读寄存

器是否非空

{

rev\_data = USART\_ReceiveData(USART1);

if(rx\_buffer.buffer\_lock\_flag == 0) //如果串口没锁住，代表协议未满

{

if(rev\_data == 0xaa)

//如果

是头部，则放入头部字节里面

{

rx\_buffer.start\_byte = rev\_data

}

else if(rev\_data == 0xf9)

//如果是

流向，则放入流向里面

{

rx\_buffer.direction = rev\_data;

}

else if(rev\_data == 0xdd)

"如果

是结束，则放入结束里面

{

rx\_buffer.end\_byte = rev\_data;

}

else

{

rx\_buffer.led\_state[rx\_buffer.data\_pt] = rev\_data; rx\_buffer.data\_pt ++;

}

if((rx一buffer.data一pt >=3)&& (rx\_buffer.end一byte== 0xdd))//如果接受状

态满，且结束位也被接收到，则锁住，直到数据被处理之后解锁

{

rx一buffer.buffer一lock一flag = 1;

}

USART一ClearITPendingBit(USART1, USART一IT一RXNE);

//清除接受中断标志  
}

}

}

程序片段26

串口协议解析程序如代码片段27所示。

void usart\_deal(u8 \*data\_group)

{

u8 \*data\_base, data\_pt = 0;

data\_base = data\_group;

for(data\_pt = 0; data\_pt <= 3; data\_pt ++)

{

if(\*data\_base == 0x00)

{

if((led\_on\_off[data\_pt]) % 2 == 0) //原本就灭

{

led\_on\_off[data\_pt] = led\_on\_off[data\_pt]; //不做操作

}

else

{

led\_on\_off[data\_pt] = led\_on\_off[data\_pt] > 3 ? 0 : led\_on\_off[data\_pt] + 1;

}

}

else if(\*data\_base == 0xff)

{

if((led\_on\_off[data\_pt]) % 2 == 1) //原本就亮

{

led\_on\_off[data\_pt] = led\_on\_off[data\_pt]; //不做操作

}

else

{

led\_on\_off[data\_pt] = led\_on\_off[data\_pt] > 3 ? 0 : led\_on\_off[data\_pt] + 1;

}

}

data\_base ++;

}

}

程序片段27

我们在这里，写了一个具有传递参数的程序，调用时如程序片段28所示。 这里面我们定义了一个指针作为传递参数，每次调用时，将指针指向接收缓存里 面，数据部分的首地址，逐个判断，判断完成之后，将指针指向下一个地址单元， 重复判断动作。

}

**if(rx\_buffer.buffer\_lock\_flag== 1)**

**{**

**usart\_deal(rx\_buffer.led**一**state); rx\_buffer.buffer\_lock\_flag = 0;**

//如果数据接受满

//处理协议中LED等状态 //处理完成后，将接收缓存解锁

程序片段28

1. 利用IAR重定向printf函数

可能以前很多的资料上面都会教大家如何来重定向标准C库“stdio.h”里面 的“printf”函数，使得在STM32的程序里面，使用“printf”函数，即可向PC 的串口监视软件里面，打印调试信息。

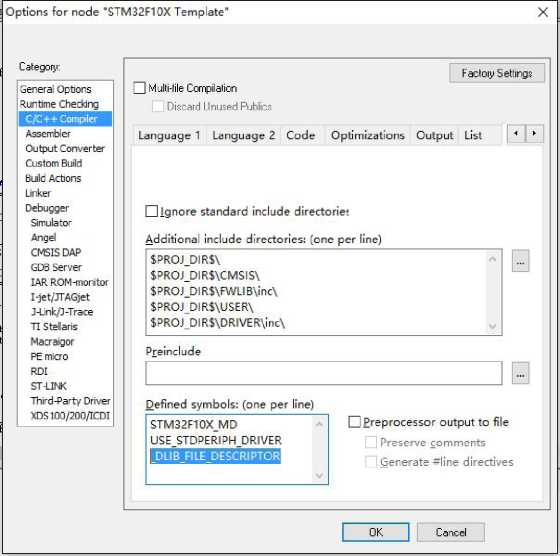
首先第一步，我们先要在“usart.h”里面包含“stdio.h”，#include"stdio.h" 第二步，需要在IAR里面包含一个宏“一DLIB一FILE一DESCRIPTOR”，如图30 所示。

图 30 定义“一DLIB一FILE一DESCRIPTOR” 第三步，改写“fputc”函数，代码如程序片段29所示。

int fputc(intch, FILE \*f)

{

while(USART\_GetFlagStatus(USART1,USART\_FLAG\_TC)==RESET)

{ }

USART\_SendData(USART1,(uint8\_t) ch); return ch;

}

程序片段29

C:\Users\pc-14\AppData\Local\Temp\FineReader12.00\media\image29.jpeg最后，只要调用“printf”函数，即可正常输出。如图32所示。

III IIIIIIII

§ §k§k§kliTIkliTIkJTIkIJTIklJTIk§

oaoaoaaoaoaoaoaoaoaoaoa

k a k a k kalkalkalk a k a k a kalk a k 000000000000 III IIIIIIII

kunkurlkIJrl^1k ^1k ^ oaoaoaaoaoaoaoaoaoaoaoa

kalkalk kalkalkalkalkalkalkalkalk oooooooooooo III IIIIIIII

IK ^1K ^1K ^llrlr-^llrlkllrlkllrlkllrlkllrlkllrlr-^llrlr-^llrl

oaoaoaaoaoaoaoaoaoaoaoa

ImIInlIil Ilntlililililililil

alkalkalk kalkalkalkalkalkalkalkalk oooooooooooo III IIIIIIII

VK ^Ifnr^illrlkllrlr^llrlkllrlkllrlkllrlr^llrlr^llrl oaoaoaaoaoaoaoaoaoaoaoa

ImIilllTII IlmIInlIInlIInlIilililil

alkalkalk kalkalkalkalkalkalkalkalk oooooooooooo III IIIIIIII

kJTIklfTIkIITIInl^tlfTIkInlkInlkInlkJTIkIJTIkllTIklln

oaoaoaaoaoaoaoao&oaoaoa

ImIInlIInlI IlmIlmIInlIInlIInlIInlIInlIInlI l

alkalkalk kalkalr-^alkalkalr-^alkalkalka  
oooooooooooo  
III IIIIIIIII

konkIInkIITI^tk ^lx ^lx ^Tx ^1x

oaoaoaaoaoaoaoao&oaoaoao

ImIlmIlmI IlmIIITIIInlIililInlIilillnl

alkalkalk kalkalkalr^alr^alritalkalkalkal oooooooooooo III IIIIIIIII klmklnlklnl

oaoaoaaoaoaoaoaoaoaoaoao  
InlIJTIIInlI IJTIIJTIIInlIInlIInlIInlIlnlIJTIIlnl

alkalkalk k 田r-^atkalkalkalkalkalr^a

oooooooooooo  
III IIIIIIIII

klmklnlklnlImkInlkInlkInlkInlklmkIITIkIITIkInlk

oaoaoaaoaoaoaoaoaoaoaoao

InlIInlIInlI IJTIIJTIIInlIInlIInlIInlIInlIInlIlnl  
al爻alkalk kalkalr^alkalkalkalkalr-^alkal  
oooooooooooo  
III IIIIIIIII  
olcgolcanlokanlanlo^anlolcanlokanlokanlokanlokalTIolcfolcanlolc

ImIJTIIJTII IJTIIJTIIInlIJTIIJTIIJTIIJTIIJTIIllTI

alkalkalk k ak ak ak a  
oooooooooooo  
III IIIIIIIII  
olcgolcalllokanll^lol(anlok§ok§ok§ok§ol(§ol(§c.lc

JTIIJIIIJTII IJnIJTIIInlIJTIIInlIJTIIInlIInlIJTI  
alkalkdlk k^Ik 恕kalkalkalkalkalkalkal

oooooooooooo  
III IIIIIIIII

klmkUTIkllTIImklmkllTIkInlkInlkllTIkIITIkIITIkllTIk

oaoaoaaoaoao&o&o^oaoaoao

JTIIJTIIJTII IJTIIJTIIJTIIJTIIJTIIJTIIJTIIJTIIJTI  
ajk a IKalk k4Jkdjk a k. a hr a IK a IKajk a IK a  
oooooooooooo  
III IIIIIIIII

kUTIkl/TIk § § k § k § klmkllTIkIJrlk § kUTIk § k  
oaoaoaaoaoao&o&o&oaoaoao

JTIIJTIIJTII IJTIIJTIIJTIIJTIIJTIIJTIIJTIIJTIIJTI

a k a IKajk kajk a IK a k a k a IK a k a k a IKaj  
000000000000  
III IIIIIIIII

kUTIkIJTIk §UTIkIJTIkIJTIkliTIkliTIkJTIkIJTIklJTIkIJTIk

oaoaoaaoaoaoaoao4oaoaoao  
JnIJnIJnI IJnIJnIJnIJnIJnIJnIJrlIJrlIJn

a k a IKajk kajk a IK a k a k a IK a k a k a IKaj  
000000000000  
III IIIIIIIII

kUTIkIJTIk §

oaoaoaaoaoao&oao4oaoaoao

号率位位位空 口特掮止验—

串波教傳校流

|  |  |
| --- | --- |
| C0H11 | 1 |
| 115200 | ▼ |
| 8 | ▼ |
| 1 ▼ | |
| None | ▼ |
| None | ▼ |

图**32 “printf**”打印输出信息

3.1.5 12864液晶实验

相信看过我51单片机的同学，很多都已经熟悉了怎么去使用内置字库的 1602液晶，驱动起来非常简单，只需要用到RS，RW和CS三根控制线，以及D7-D0 这八根数据线即可。然而，在我们大多数的情况下，还是希望使用可以显示任意 字体以及图片的显示器，而不是像1602液晶一样显示出非常不美观的界面。因 此在这里，我们选用了一块1.7英寸的并且基于ST7567的128\*64点阵液晶。而 此时再去做界面设计时，应该将将思路转变过来，重新去认识点阵式液晶。

关于STM32控制液晶的方式，目前主流的有这么几种方式：

采用并口+控制线的方式，比如1602液晶以及12864液晶，这种方式一般都 是做一些简单的字符型液晶控制；

采用SPI或者IIC+控制线的方式，这种控制方式一般用来控制小屏幕的点阵 液晶用的非常多，比如点阵型TFT，STN。

采用FMSC接口的方式，这种方式是目前STM32支持的最新方式，直接通 过内存来实现，因此速度非常快，控制一些大屏幕的设备也没有刷新过慢的现象。

由于控制LCD是每个做单片机的基本功，之前我们讲了第一种方式，这次机 会，我们将第二种方式，以后的TFT我们再来讲第二种和第三章方式。

我们下面的课程会分成四部分来讲。

1、 基于SPI单向发送的12864驱动编写；

2、 编写一些简单的字符型UI;

3、 编写一些简单的图片型UI;

4、 编写一些复杂的综合性UI。

(1)基于SPI单向发送的12864驱动编写

我们要对一块陌生芯片的编写，首先要做的第一步必然是阅读其数据手册。 我们使用的这个液晶的数据资料，已经上传。

这里我们选用了一款基于ST7567驱动芯片的128\*64像素FSTN (格式化超 扭曲向列型）液晶，当中不集成任何字库，集成了可独立控制的琥珀色背光板。 之所以选择这样的液晶，是因为其与目前可穿戴设备广泛使用的0LED (有机发 光二极管）具有相似的控制原理，并且价格还比0LED便宜很多。

我们在前面硬件设计部分，就已经对其引脚进行详细地描述了，因此这里不 再赘述。对于我们软件部分来说，唯一需要知道的就是控制引脚和MCU的引脚 对应，如下：

PB2：背光开关；

PB3：复位；

PB4： A0,数据或者命令选择，“1”：写数据，“0”：写命令；

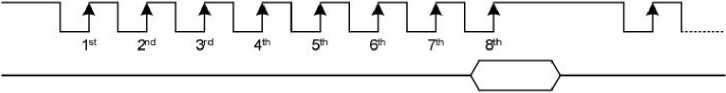
PA5: SCL (时钟端）；

PA7: SDA (数据端）；

在原理图上，还有很多引脚，比如“PSB”之类的，这些引脚是和我们这里 另一款液晶（未生产）相匹配的，因此这里就不需要去使用。使用“PSB”的液 晶，即可以使用并口和串口来传输数据，类似于1602液晶。而我们这里使用的 液晶只可以支持SPI 口通讯，SPI可以有效地减少数据端口的连接，便于布板和 节约端口资源，因此我们优先推荐。关于LCD的介绍，我们就像讲到这里，接下 来我们研宄一下LCD的驱动程序。

要研宄LCD的驱动程序，首先必须了解其控制芯片的时序。打开ST7567的 数据手册，我们来找一下其时序图。

C:\Users\pc-14\AppData\Local\Temp\FineReader12.00\media\image31.jpegCSB

SDA **X~57~X~5j~X~5i^X~5^~)( D3 pi X~d**〇**7**

**SCLK**

**AO**

图**33 ST7567**的时序

每次需要发出的时候，先要拉低CSB,然后SDA发出字节中最高位的bit, 接着SCLK进行一次上升沿的变化，然后按照上述内容依次发送其余各位。在最 后全部发送完成数据后，A0需要根据当前发送的是命令还是数据，进行变化。 发送完成之后，CS需要拉高，准备下一次字节的发送。

这里有同学可能会有疑惑“老师，为什么你的电平变化时候，不加上延迟， 进行电平保持？ ”这里我们需要再来看下SPI总线是什么。以下为引用百度百科。

**SPI**接口的全称是**"Serial** **Peripheral** **Interface**",意为串行外围接口，是**Motorola**首先在其**MC**68**HCXX**系

列处理器上定义的。SPI接口主要应用在EEPROM、FLASH、[实时时钟](http://baike.baidu.com/view/1031044.htm)、AD[转换器](http://baike.baidu.com/view/110440.htm)，还有[数字信号处理器](http://baike.baidu.com/view/529231.htm)和

数字信号解码器之间。

SPI接口是在CPU和外围低速器件之间进行同步串行数据传输，在主器件的移位脉冲下，数据按位传 输，高位在前，低位在后，为全双工通信，[数据传输速度](http://baike.baidu.com/view/2827566.htm)总体来说比[I2C总线](http://baike.baidu.com/view/751581.htm)要快，速度可达到几Mbps。

1. MOSI -主器件[数据输出](http://baike.baidu.com/view/324739.htm)，从器件数据输入
2. MISO -主器件数据输入，从器件数据输出
3. SCLK -[时钟信号](http://baike.baidu.com/view/188811.htm)，由主器件产生,最大为fPCLK/2,从模式频率最大为fCPU/2
4. NSS -从器件使能信号，由主器件控制，有的IC会标注为CS(Chipselect)

因此，我们在这里用72MHz的系统时钟去驱动它，完全不需要担心速度不 匹配的问题。

接着，我们开始来写程序。在此之前，我们在创建一个“Icd.c”和一个“Icd.h” 文件。然后，还是和往常一样，罗列出写驱动的步骤：

鲁初始化相应的GPIO;

籲写LCD的SPI写数据程序；

籲写LCD的SPI写命令程序；

籲初始化LCD。

第一步，初始化相应的GPIO。在这里，由于我们仅仅用了单向的通讯，即 控制信号永远从MCU到LCD，因此只需要配置上面5个控制引脚为推挽式输出 模式“GPIO\_Mode\_Out\_PP”。初始化代码如程序片段30所示。

void lcd\_gpio\_spi(void)

{

GPIO\_InitTypeDefGPIO」nitStructure;

RCC\_APB2Peripha〇ckCmd(RCC\_APB2Periph\_GPIOA, ENABLE); GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_5 | GPIO\_Pin\_7; GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP; GPIO\_Init(GPIOA,&GPIO」nitStructure); RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB, ENABLE); GPIO」nitStructure.GPIO\_Pin = GPIO\_Pin\_2 | GPIO\_Pin\_3 | GPIO\_Pin\_4 | GPIO\_Pin\_5; GPIO」nitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; GPIO」nitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;

GPIO\_Init(GPIOB, &GPIO」nitStructure);

程序片段30

在定义了以上的初始化程序之后，为了便于后续的操作方便，我们可以釆用 宏定义的方式来定义每个引脚的高低电平输入输出。如程序片段31所示。

GPIO\_SetBits(GPIOB, GPIO\_Pin\_2) GPI〇\_ResetBits(GPI〇B,GPI〇\_Pin一2) GPIO\_SetBits(GPIOB, GPIO\_Pin\_3) GPIO一ResetBits(GPI〇B,GPI〇一Pin一3) GPIO一SetBits(GPI〇B, GPIO一Pin一4) GPI〇一ResetBits(GPI〇B,GPI〇一Pin一4) GPI〇一SetBits(GPI〇B, GPI〇一Pin一5) GPI〇一ResetBits(GPI〇B,GPI〇一Pin一5) GPIO一SetBits(GPI〇A, GPIO一Pin一7) GPI〇一ResetBits(GPI〇A,GPI〇一Pin一7) GPI〇一SetBits(GPI〇A, GPI〇一Pin一5) GPI〇一ResetBits(GPI〇A, GPI〇一Pin一5)

//led background on //led back ground off //led reset pin high //led reset pin low //led A0 pin high //led A0 pin low //led CS pin high //led CS pin low //led M〇SI pin high //led M〇SI pin low //led SCK pin high //led SCK pin low

#defineLCD\_BK\_ON #define LCD\_BK\_OFF #define LCD\_RST\_H #define LCD\_RST\_L #define LCD\_A0\_H #define LCD\_A0\_L #define LCD\_CS\_H #define LCD\_CS\_L #define LCD\_MDO\_H #define LCD\_MDO\_L #define LCD\_SCK\_H #define LCD SCK L

**}**

程序片段31

第二步，编写LCD的写数据程序。在这里，我们先来看下时序图，如果要写 命令的话，首先需要将A0拉高，CS在写的过程中需要拉低，接着通过SCK和SDA 的配合，即可将数据完整的发送出去。发送的时候，以一个字节为最基本单元，

从最高位开始发送，直至最低位发送出去。如程序片段32所示。

void write一led一data一spi(u8 data)

{

u8 data一bit, trans一data; trans一data = data;

LCD一A0\_H;

LCD一CS一L;

for(data一bit = 0; data一bit < 8; data一bit ++)

{

LCD一SCK一L;

if((trans一data & 0x80) == 0x80)

LCD一MD〇一H;

else

LCD一MD〇一L;

LCD一SCK一H; trans一data <<= 1;

}

}

程序片段32

第三步，编写LCD的写命令程序。写命令程序和写数据程序类似，只需要将 A0设置成低电平即可，如程序片段33所示。

void write\_lcd\_cmd\_spi(u8 cmd)

{

u8 data\_bit, trans\_cmd; trans\_cmd = cmd;

LCD\_A0\_L;

LCD\_CS\_L;

for(data\_bit = 0; data\_bit < 8; data\_bit ++) {

LCD\_SCK\_L;

if((trans\_cmd & 0x80) == 0x80) LCD\_MDO\_H;

else

LCD\_MDO\_L;

LCD\_SCK\_H; trans\_cmd <<= 1;

}

}

程序片段33

第四步，初始化LCD。初始化LCD的时候，我们需要做两件事情，第一件事 设置LCD的工作方式；第二件是控制LCD的背光开关（打开或者关闭）。关于 此LCD的工作模式设置，请参照ST7566数据手册的第21页。这里我们直接给出

般的初始化程序，如程序片段34所示。

|  |  |
| --- | --- |
| void Init\_ST7567(void) {  lcd\_gpio\_spi();  lcd\_delay(10);  HDReset(); | "复位LCD |
| lcd\_delay(100);  write\_lcd\_cmd\_spi(0xe2); | //Soft rest |
| write\_lcd\_cmd\_spi(0xa2); | //SET LCD bias(A2-1/9bias;A3-1/7bias) |
| write\_lcd\_cmd\_spi(0xa0); | //SET ADC NORMAL(OB-POR seg0-00h) A0: NORMAL |
| write\_lcd\_cmd\_spi(0xc8); | //A1: REVERSE  //SET COM OUTPUT SCAN |

//DIRECTION(0XXXB-NORMAL)-POR COM63-->COM0

// SET DISPLAY NORMAL (OB-NORMAL)-POR //A4:NORMAL A5:ENTIRE DISPLAY ON //SET NORMAL DISPLAY MODE(OB-NORMAL) //A6:NORMAL A7:REVERSE //SET INTERNAL REGULATOR RESDASTOR //RATIO(100B)-POR

**write\_lcd\_cmd\_spi(0xa4);**

**write\_lcd\_cmd\_spi(0xa6);**

**write\_lcd\_cmd\_spi(0x25);**

write」cd\_cmd\_spi(0x81);

write」cd\_cmd\_spi(0x1a); // SET CONTRAST CONTROL

//REGISTER(00,0000H-11,miH)30对比度请修改此值，调浓增大此值，反之调淡。 write」cd\_cmd\_spi(0x2f); //SET POWER CONTROL REGISTER (ALL

//INTERNAL)

write\_lcd\_cmd\_spi(0x40); //end of initialzation

write」cd\_cmd\_spi(0xaf); //DisPlay On

}

程序片段33

打开背光程序非常简单，我们只需要用LCD\_BK\_ON即可，后续我们可以以PWM 的方式来调节背光亮度，在这里我们先不讲。

最后的一个程序，是Icd复位程序和转为Icd操作编写的延迟程序如程序片段34所

示。

void HDReset(void)

{

LCD\_RST\_L;

lcd\_delay(2);

LCD\_RST\_H;

lcd\_delay(4);

}

void lcd\_delay(unsigned long value) {

u16 i,j;

for (i=0;i<value;i++) for (j=0;j<500;j++);

}

程序片段33

(2)编写一些简单的字符型Ul

我们在上一节里面，阐述了如何来编写点阵液晶的写数据和写命令函数，单 单的写数据和写命令函数是无法驱动LCD的，因此我们这节课来讲如何驱动LCD， 使它可以进行一些字符的显示。这一小节主要分成以下几个部分来讲：

籲12864点阵液晶的扫描方式；

\*利用字符取模软件来编写字库；

籲显示字符。

首先，我们来讲下12864点阵液晶的扫描方式。目前互联网上面关于12864 点阵液晶的描述资料五花八门，我们在这里就不做任何参考了，因为大多数资料 阅读之后，大家还是会发现毫无头绪。这里我们给出12864液晶的模拟电子格点。 如图34所示。

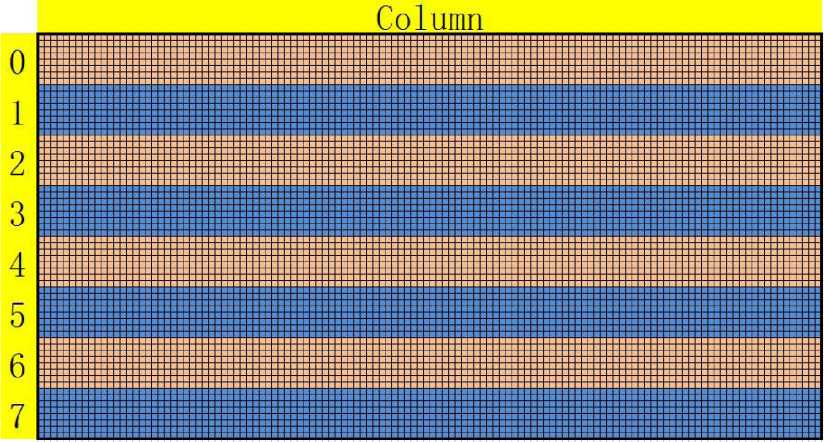


图34 12864模拟液晶晶格

因此我们很直观地可以看出，在这128\*64个晶格上面，纵向的64格被分成 了 8个部分，每个部分有8个晶格，我们把这里称为12864的页地址，从上到下 分别为第0页~第7页。接下来横向的被称为“Column”即为“列”这个含义， 而12864顾名思义即有128列，从左到右分别为第0列~第127列。

我们每次在写入LCD的时候，分成三步来做：

首先，寻找到需要写入的页地址，用写命令函数去找；

其次，寻找到固定页的列地址，用写命令函数去找；

最后，以8为的形式写入某一页。

写入时候，ST7567具有两个命令，如图35所示。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (3) Set Page Address | 0 | 0 | 1 | 0 | 1 | 1 | Y3 | Y2 | Y1 | Y0 | Set page address |
| (4)  Set Column Address | 0 | 0 | 0 | 0 | 0 | 1 | X7 | X6 | X5 | X4 | Set column address (MSB) |
| 0 | 0 | 0 | 0 | 0 | 0 | X3 | X2 | X1 | XO | Set column address (LSB) |

图35 ST7567的寻址命令

我们先脱离代码来写个例子，假设你对第0页的第8列写了一个数值0x22, 那么我们可以使用如下的条理：

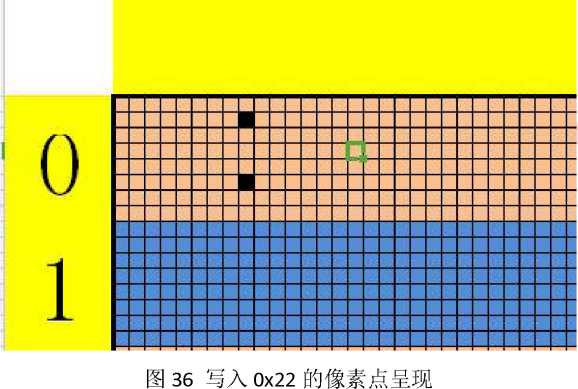
写页地址：写命令（0xb0);

写列地址的高4位：写命令（0x10);

写列地址的低4位：写命令（0x08);

写入数值：些数据（0x22)。

完成以上操作后，我们的液晶上面就会留下两个黑点，如图36所示。



到这里，细心的同学可能发现了，每次写入的数值，即为一页数据的8位二 进制数，且低位在前，高位在后。

如果使用我们开发板的同学，按照我们这个逻辑去把程序烧录进去，肯定会 发现怎么样这个液晶都是不会有任何显示的，为什么呢？我们来看下 STM32F103C8T6的数据手册，原来PB3，PB4的初始默认功能是JTAG 口的相关

引脚，如图37所示，因此想要用其作为普通I/O,需要先开启复用功能。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A7 | 39 | 55 | 89 | 30 | PB3/JTDO | I/O | FT | JTDO | PB3/TRACESWO |
| A6 | 40 | 56 | 90 | 31 | PB4/JNTRST I/O |  | FT | JNTRST | PB4 |

图37PB3，PB4复用功能

我们可以使用如代码片段34的方式来关闭它。

RCC\_APB2Peripha〇ckCmd(RCC\_APB2Periph\_AFIO,ENABLE);

GPIO一PinRemapConfig(GPIO一Remap一SWJ\_JTAGDisable,ENABLE);

程序片段34

好了，写到这里，你就可以动手写我们的程序了，将0x22写到第1页，第 8列上去，可以用代码片段35的程序去写。这么写完之后，估计令你崩溃的事 情产生了，整个屏幕出现了花屏，你根本就分不清哪几个点是你自己写的，那是 因为这个屏幕开始时，是不会自动清屏的，因此一些非常随机的黑点就会出现在

这个屏幕上面。因此，我们要用程序片段36的代码，先进行清屏，就是对所有

页，所有列写入“0x00” .

while⑴

{

write一lcd一cmd一spi(0xb0 ); write」cd\_cmd一spi(0x10); writejcd一cmd一spi(0x08); write一led一data一spi(0x22);

}

程序片段35

for(i = **0;i<** 8; i++)

{

write一lcd一cmd一spi(0xb0+i); for(j = 0; j < 128; j++)

{

write一lcd一cmd一spi(0x10 + ((j & 0xf0) >> 4)); write一lcd一cmd一spi(0x00 + (j & 0x0f)); write一lcd一data一spi(0x00);

}

}

程序片段36

以上代码加入之后，我们可以很清楚的看到图36所示的图像。因此，我们 在这里总结一下，页地址，高位列地址和高位列地址的执行方式，我们都能理解 了，那么写入的数据该如何编排呢？请看图37。

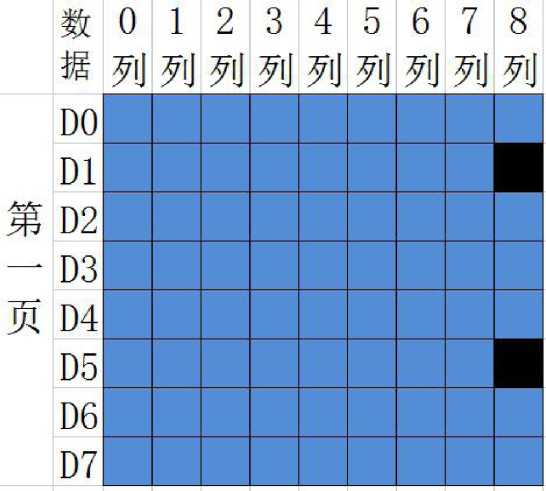
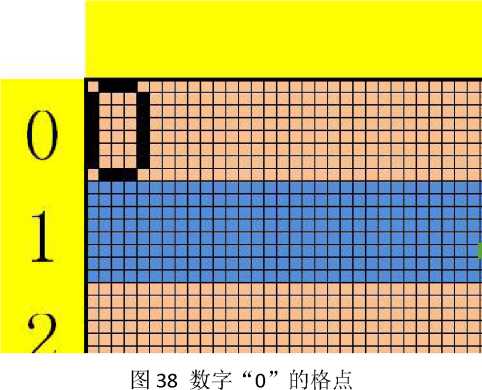


图37些数据的编排

我们可以看出，写入的数据DATA，就是将相应页，相应列的8个像素设置 成DATA (D[7..0])。如果正常模式，“1”为黑色，“0”为不黑。反转模式就相 反。

讲到这里，我想对于这种页寻址的LCD已经讲的非常透透彻了。接着我们来 讲讲如何来写一个字，假设我们要写“〇”，我们先对图4里面的液晶格点进行 填充，因此可以得到如图38的图片。



那我们构建这个字的方式可以表示为：第0页第0列为0x7e，第0页第1 列为0x81，第0页第2列为0x81，第0页第3列为0x81，第0页第4列为0x7e。

因此，我们可以将代码片段35的代码替换成代码片段37里面的内容。现在可以

在第0页开始的前5列显示“0”，如图39所示。

while⑴ {

write\_lcd\_cmd\_spi(0xb0);

write」cd\_cmd\_spi(0x10);

write」cd\_cmd\_spi(0x00);

write\_lcd\_data\_spi(0x7e);

write」cd\_cmd\_spi(0x10);

write」cd\_cmd\_spi(0x01);

write」cd\_data\_spi(0x81);

write」cd\_cmd\_spi(0x10);

write\_lcd\_cmd\_spi(0x02);

write」cd\_data\_spi(0x81);

write」cd\_cmd\_spi(0x10);

write\_lcd\_cmd\_spi(0x03);

write」cd\_data\_spi(0x81);

write」cd\_cmd\_spi(0x10);

write」cd\_cmd\_spi(0x04);

write\_lcd\_data\_spi(0x7e);

}

程序片段37



图39显示“0”字符

虽然现在可以显示出来了，不过作为我们的程序员，写一个“〇”就需要这 么多代码，这是非常不可取的，我们可以来精简一下，我们注意到，程序里面的

列其实是有归一性质的，即列地址高四位右移4位后，与“0x10”相与，就可以 将列地址高四位写入，而将列地址的高四位清0之后，即可以将列地址的低四位 写入。这样，我们可以将这个函数包装起来。

首先，先将“0”这个字符需要现实的内容数据存储在一个数组里面，我们 把它称作为“0”这个字体的字模。我们可以用如下的数组表示。

u8 num0[5] = {0x7e, 0x81, 0x81, 0x81, 0x7e};

然后在写入的时候，我们需要一个函数，可以实现在指定的页和指定的列上

面进行显示，因此我们可以用如下的函数，参考代码片段38所示。

void show\_number(u8 start\_page, u8 start\_col)

{

u8col\_address\_msb,col\_address\_lsb;

u8 i, temp\_page, temp\_col;

temp\_page = start\_page; //起始页

temp\_col = start\_col; //起始列

write\_lcd\_cmd\_spi(0xb0 + temp\_page);

for( i = 0;i< 5; i++)

{

col\_address\_msb= ((temp\_col >>4) & 0x0f) | 0x10;

col一address\_lsb = temp一col&0x0f;

write\_lcd\_cmd\_spi(col\_address\_msb);

write一lcd一cmd一spi(col一address一lsb);

write一lcd一data一spi(num0 [i]);

temp一col ++;

}

}

程序片段37

接下来，我们该考虑一个问题，如果我们需要一下子写好多页，即将这个数

字放大，那我们该怎么做呢？假设我们要写图40中的那个数字。

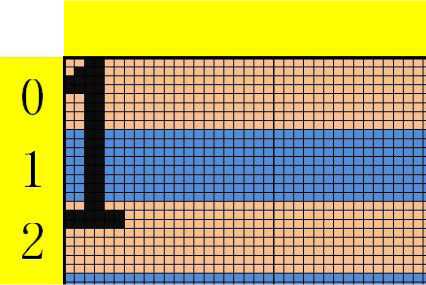


图40跨页写数据

在这里，我们首先肯定像之前一样些数据是不可能了，因为这个显示的字符 比较庞大。那在这里，我们肯定不会对它一个点一个点地手动取模了，这里给大 家推荐一个软件，即“字模提取V2.2”。其界面如图41所示。

V2.2 CopyLeft By Horse2000 - □

+基本操作

+取植方式 +修改图像 ♦镇拟动画 +参數设S +退出程序

图41字模提取软件V2.2

首先要点击“新建图像”；

然后设置你需要的字符尺寸，如图42所示，注意这里你所需要设置的宽度 和高度指的是像素点，以图40里面的数字“1”为例，我们需要设置宽度为“6”， 高度为“19 ”；

图42新建图像尺寸

然后就会生成如图43所示的界面。我们在这里可以放大图43的格点，只要 点击图44中所示的选项即可进行放大。

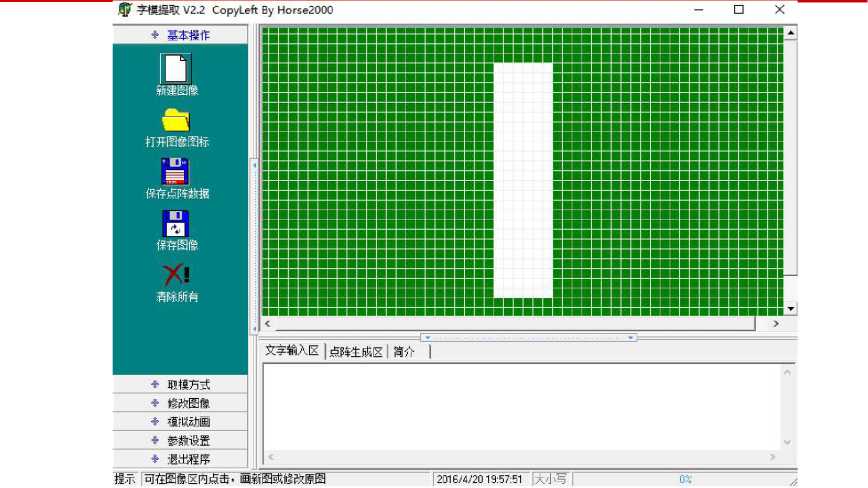


图43生成6\*19的点阵格

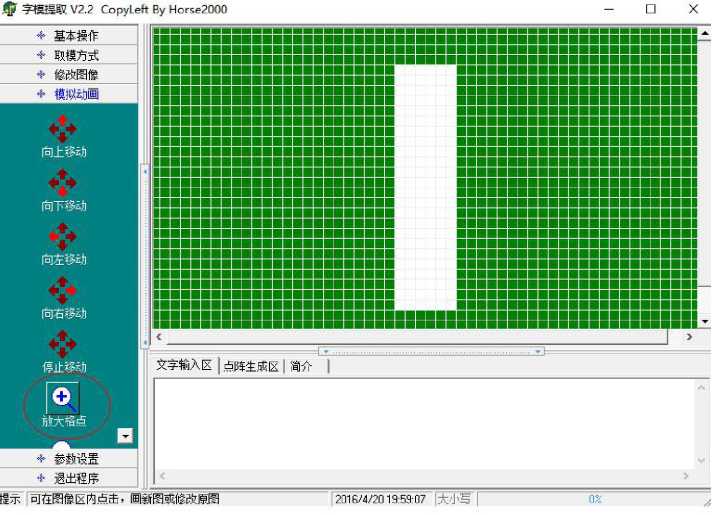


图44放大格点

到这里之后，我们就可以按照图40中所示的格点进行取模了，只需要单击 相应的格点，即可实现描出黑色的部分。如图45所示。

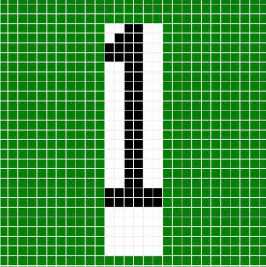


图45描点取模

接着，点击图46中所示的取模方式，我们将其设置成C语言，就可以看到 相应的字符数组了。

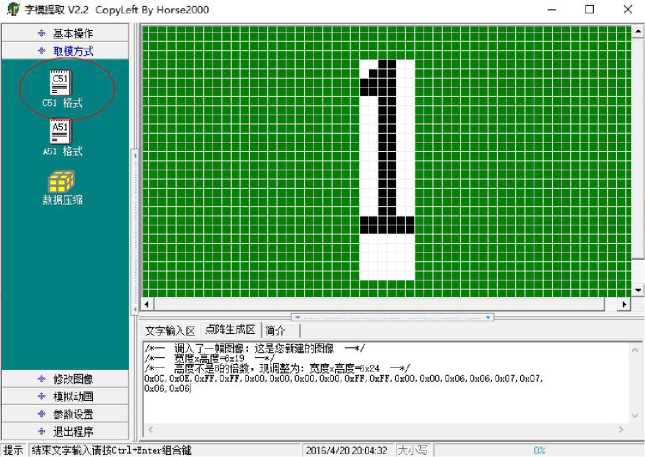


图46生成字符数组

由于这个字符是横跨3页，因此在写入的时候，应先写第0页的6列，再写 入第1页的6列，最后写入第2页的6列。我们观察一下这个数组，一共有18 个字节，正好对应我们上述的逻辑顺序，因此我们可以得到代码片段38中的代 码。

void show\_number(u8 start\_page, u8 start\_col)

{

u8 col\_address\_msb, col\_address\_lsb;

u8 i, j, temp\_page, temp\_col;

temp\_page = start\_page; //起始页

f〇r(j = 0； j < 3; j++)

{

write\_lcd\_cmd\_spi(0xb0 + temp\_page); temp\_col = start\_col; //起始列

for( i = 0;i< 6; i++)

{

col\_address\_msb = ((temp\_col >> 4) & 0x0f) | 0x10; col\_address\_lsb = temp\_col & 0x0f; write\_lcd\_cmd一spi(col\_address一msb); write\_lcd\_cmd一spi(col\_address一Isb); write一led一data一spi(num1 [i + 6\* j]); temp一col ++;

}

temp一page ++;

}

}

程序片段38

以此类推，我们可以将所有需要的0~9写在一个二维数组里面，第一维数组 的元素代表某一个数字，第二维数组的元素，代表字模。其实我们可以将所有 ASCII码的字符进行取模，编排在一个二维数组里面。具体的实现方法，我们就 不一一说明，详情请见源码“SourceCode”里面，在这里面，我为大家写好了很 多字模以及显示函数，大家自行阅读研宄。

1. 编写一些简单的图片型UI

这个液晶可以显示最大128\*64的图片，假设我们要对一张图片进行显示， 首先，我们需要将此图片设置128\*64分辨率，然后将其转换成单色的位图。以 上这些步骤，用画图即可完成。然后保存成“.bmp”格式的文件。

根据图47里面的方法，将此图像载入取模软件。

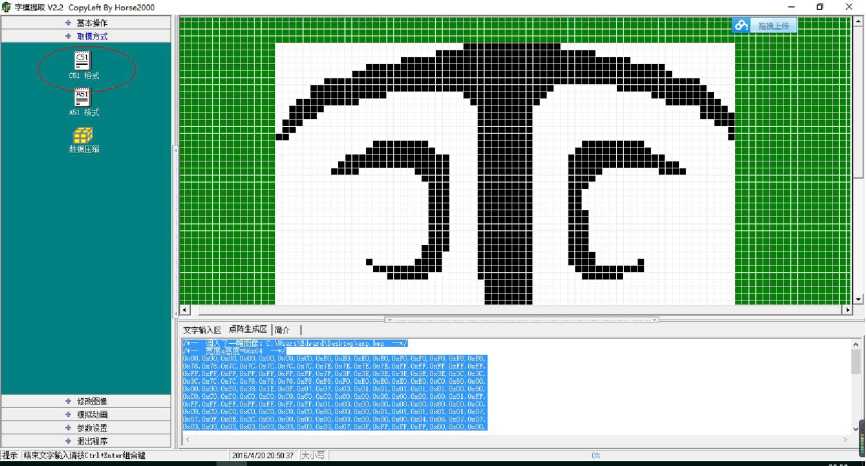


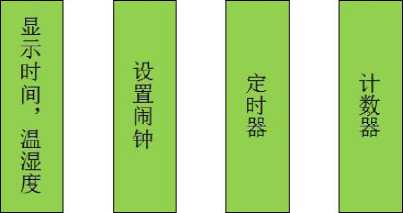
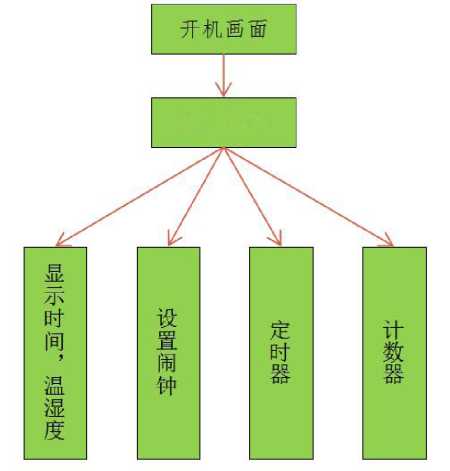
图47载入图像

好了，我们接下来就可以将这一组图像显示出来了，看取模上面的说明，这 幅图像宽度66,高度64,这里应该不需要我教大家也明白了，只要从第0页的 第〇列扫码到第7页的第66列就行，每页扫描66列。讲宄一点，我们可以学着 我们的给出的例子源码，将所有的字模数据存储在“font.h”里面。

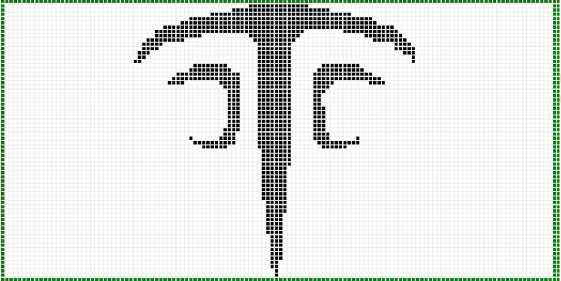
1. 编写一些复杂的综合性UI

到此为止，我相信大家已经可以完成单个字符和图片的写入了，有了以上的 基础，我们继续写下去，这次我们的目标是建立起我们需要的UI，并且为我们 后续的产品留出API，以便于我们后续可以扩展开发，同时，我们这一部分内容 还包括具体的模式设置。我们本小节的思路分成两部分来讲：各模式下的UI规 划以及具体代码实现。

首先，我们来规划下各模式下的UI。我们代码这里，给大家预定以下几种模 式如图48所示。

图48模式设置

模式选择

开机画面。开机画面如图49和图50所示。

图**49**开机画面（**start1.bmp)**

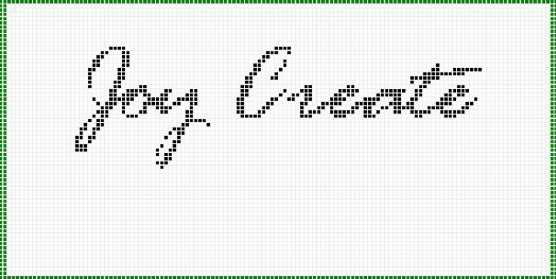


图50开机画面（start2.bmp)

我们这样安排：开机-> startl (2秒）-> start2 (4秒）-> start2反色（2秒），

进入设置界面。

其他模式。由于我们这里模式选择在后期需要通过按键去动态的选择变化， 因此这里不采用图片的形式去实现，而是采用不同的英文字母去配合，当然如果 你想做成中文，也可以对你所需要的中文字母取模。我们在这里选用英文模式。 由于我们这里有四个菜单，要显示四行，因此每个在此页面的字母占用2页。再 多说一点，在这里，为了便于大家以后很方便地使用此液晶，帮大家进行了三种 字体大小的取模，分别是占1页，占2页和占3页。由此我构成了三个组数，并 且每个数组都是按照ASCII码表排列。分别对于ASCII码表的48-57, 65-90。其余 部分为空白部分，大家自己可以添加。另外由于我们这里还加入一些其他符号， 比如百分号，摄氏度等等。

接着，我们来实现代码，参照之前所说的，先建立一个“font.h”专门用于 存放字模，然后将我们刚刚所取模的字模写入其中。在这里注意一下，为了便于 我们的程序编写，我们在这里把数据和字母分开，并且按照ASCII码表排列，因 此数字是从48 (ASCII码对应0)开始到57 (ASCII码对应9)结束，而字母是从 65 (ASCII码对应A)开始到90 (ASCII码对应Z)，因此在后面写程序的时候， 字母需要减65。这里注意一下。

接下来我们来编写这些不同大小字符的显示函数。

小数字的显示驱动，如代码片段39所示。

void show\_number\_small(u8 start\_page, u8 start\_col, u8 number)

{

u8col\_address\_msb,col\_address\_lsb;

u8 i, temp\_page, temp\_col;

temp\_page = start\_page; //起始页

write\_lcd\_cmd\_spi(0xb0 + temp\_page);

temp\_col = start\_col; //起始列

for( i = 0;i< 6; i++)

{

col\_address\_msb= ((temp\_col >>4) & 0x0f) | 0x10; col\_address\_lsb = temp\_col & 0x0f; write\_lcd\_cmd\_spi(col\_address\_msb); write\_lcd\_cmd\_spi(col\_address\_lsb); write\_lcd\_data\_spi(num\_s[number][i]); temp\_col ++;

}

}

程序片段39

小字母的显示驱动，如代码片段40所示。

void show\_letter\_smaN(u8 start\_page, u8 start\_col, u8 letter) {

u8 col\_address\_msb, col\_address\_lsb;

u8 i, temp\_page, temp\_col, letter\_ascii;

temp\_page = start\_page; //起始页

letter\_ascii = letter - 65; //取 ASCII 码对 ‘A’ 的偏移

write\_lcd\_cmd\_spi(0xb0 + temp\_page);

temp\_col = start\_col; //起始列

for( i = 0;i< 6; i++)

{

col一address一msb=((temp一col>>4)&0x0f) |0x10;

col\_address\_lsb = temp\_col & 0x0f;

write一led一cmd一spi(col一address一msb);

write一led一cmd一spi(col一address一Isb);

write一led一data一spi(letter一s[letter一ascii][i]);

temp一col ++;

}

}

程序片段40

中号数字显示驱动程序如程序片段41所示。

void show\_number\_middle(u8 start\_page, u8 start\_col, u8 number)

{

u8col一address\_msb,col一address一Isb;

u8i, j,temp\_page,temp\_col;

temp一page = start一page; //起始页

for(j = 0;j<2;j++)

{

write一led一cmd一spi(0xb0 + temp一page); temp一col = start一col; //起始列

for( i = 0; i< 8; i++)

{

col一address一msb = ((temp一col >> 4) & 0x0f) | 0x10; col一address一lsb = temp一col & 0x0f; write一lcd一cmd一spi(col一address一msb); write一lcd一cmd一spi(col一address一lsb); write」cd—data一spi(num一m[number][i + 8\*j]); temp一col ++;

}

temp一page ++;

}

}

程序片段41

中号字母显示驱动程序如程序片段42所示。

void show\_number\_middle(u8 start\_page, u8 start\_col, u8 number)

{

u8col一address\_msb,col一address一Isb;

u8i, j,temp\_page,temp\_col;

temp一page = start一page; //起始页

for(j = 0;j<2;j++)

{

write一led一cmd一spi(0xb0 + temp一page); temp一col = start一col; //起始列

for( i = 0; i< 8; i++)

{

col一address一msb = ((temp一col >> 4) & 0x0f) | 0x10; col一address一lsb = temp一col & 0x0f; write一lcd一cmd一spi(col一address一msb); write一lcd一cmd一spi(col一address一lsb); write」cd—data一spi(num一m[number][i + 8\*j]); temp一col ++;

}

temp一page ++;

}

}

程序片段41

小号和中号字符的显示如图51所示。



图51小号和中号字体

大号数字显示驱动程序如程序片段42所示。

void show\_number\_large(u8 start\_page, u8 start\_col, u8 number)

{

u8col一address\_msb,col一address一Isb;

u8i, j,temp\_page,temp\_col;

temp一page = start一page; //起始页

for(j = 0;j<3;j++)

{

write一led一cmd一spi(0xb0 + temp一page); temp一col = start一col; //起始列

for( i = 0; i< 10; i++)

{

col一address一msb = ((temp一col >> 4) & 0x0f) | 0x10;

col一address一lsb = temp一col & 0x0f;

write一lcd一cmd一spi(col一address一msb);

write一lcd一cmd一spi(col一address一lsb);

write一lcd一data一spi(num一l[number][i + 10 \*j]);

temp一col ++;

}

temp一page ++;

}

}

程序片段42

大号字符显示驱动程序如程序片段43所示。

void show\_letter\_large(u8 start\_page, u8 start\_col, u8 number)

{

u8col一address\_msb,col一address一Isb;

u8i, j,temp\_page,temp\_col;

temp一page = start一page; //起始页

for(j = 0;j<3;j++)

{

write一led一cmd一spi(0xb0 + temp一page); temp一col = start一col; //起始列

for( i = 0; i< 10; i++)

{

col一address一msb = ((temp一col >> 4) & 0x0f) | 0x10;

col一address一lsb = temp一col & 0x0f;

write一lcd一cmd一spi(col一address一msb);

write一lcd一cmd一spi(col一address一lsb);

write一lcd一data一spi(num一l[number][i + 10 \*j]);

temp一col ++;

}

temp一page ++;

}

}

程序片段43

大号字符的显示如图52所示。



图52大号字体

LOGO显示驱动程序如程序片段44所示。

void show\_logo(u8 start\_page, u8 start\_col, u8 pic)

{

u8col一address\_msb,col一address一Isb;

u8i, j,temp\_page,temp\_col;

temp一page = start一page; //起始页

for(j = 0;j<8;j++)

{

write一led一cmd一spi(0xb0 + temp一page); temp一col = start一col; //起始列

for( i = 0; i< 128; i++)

{

col一address一msb = ((temp一col >> 4) & 0x0f) | 0x10;

col一address一lsb = temp一col & 0x0f;

write一lcd一cmd一spi(col一address一msb);

write一lcd一cmd一spi(col一address一lsb);

write」cd\_data\_spi(logo[pic][i + 128\* j]);

temp一col ++;

}

temp一page ++;

}

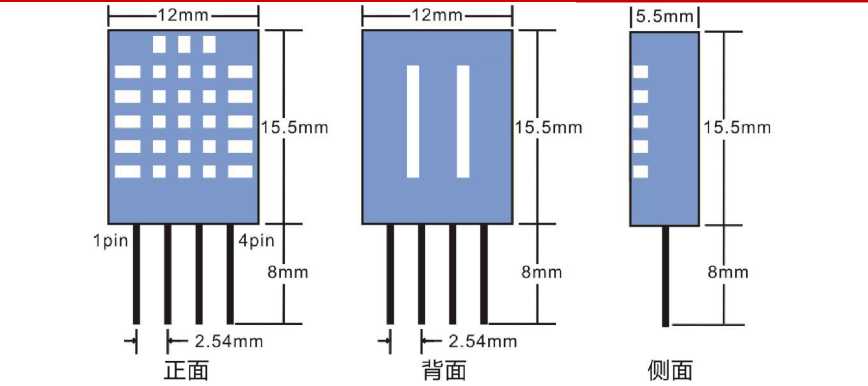
}

程序片段44

好了，到此为止，液晶的全部基础内容已经讲完，后面我们肯定会不断地遇 到液晶显示器，我们遇到时候再来详细地和大家来说。

3.1.6 DHT11温湿度传感器实验

对于温湿度采集部分，我们在这里使用了一个名叫“DHT11”的温湿度采集 传感器，DHT11[数字温湿度传感器](http://baike.baidu.com/view/3457800.htm)是一款含有已校准数字信号输出的温湿度复合 传感器，它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高 的可靠性和卓越的长期稳定性。传感器包括一个电阻式感湿元件和一个NTC测 温元件，并与一个高性能8位单片机相连接。因此该产品具有品质卓越、超快响 应、抗干扰能力强、性价比极高等优点。模块本体一共有四个引脚，其中两个是 电源引脚（VCC和GND)，一个是单总线数据引脚，还有一个为空引脚，关于 DHT11的I/O端口功能如图53所示。



|  |  |  |
| --- | --- | --- |
| Pin | 名称 | 注释 |
| 1 | VDD | 供电 3 — 5.5VDC |
| 2 | DATA | 串行数据，单总线 |
| 3 | NC | 空脚，请悬空 |
| 4 | GND | 接地，电源负极 |

图53 DHT11引脚功能

接下来我们来看看DHT11的数据特性，如图54所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 参数 | ^条件 | min | typ | max | »钃单位 |
| 供电 | DC | 3 | 5 | 5.5 | V |
| 供电电流 | 测量 | 0.5 |  | 2.5 | mA |
|  | 平均 | 0.2 |  | 1 | mA |
|  | 待机 | 100 |  | 150 | uA |
| 采样周期 | 秒 | 1 |  |  | 次 |

图54 DHT11产品特性

好了，在了解了这些基本的知识点之后，我们正式来讲课，讲课的第一点不 是别的，而是就我们上述提到的一些零散的知识点做一个扩展性质的补充。

何为NTC测温元件？我们知道，电阻其实是一个很庞大的组织，我们常用 的是阻值固定的电阻，除此之外还有可以手动调节阻值的电位器，以及一些根据 环境被动调节阻值的电阻，例如热敏电阻和湿敏电阻。热敏电阻对热会有两种变 化，第一种称为NTC热敏电阻，即温度越高，电阻阻值越小，还有一种称为PTC 热敏电阻，即温度越高，电阻阻值越大。

何为单总线？美国的达拉斯半导体公司（DALLASSEMICONDUCTOR)推出了 一项特有的单总线（1 一Wire Bus)技术。它采用单根信号线，既可传输时钟， 又能传输数据，而且数据传输是双向的，因而这种单总线技术具有线路简单，硬

件开销少，成本低廉，便于总线扩展和维护等优点。单总线适用于单主机系统， 能够控制一个或多个从机设备。主机可以是微控制器，从机可以是单总线器件， 它们之间的数据交换只通过一条信号线。当只有一个从机设备时，系统可按单节 点系统操作；当有多个从机设备时，系统则按多节点系统操作。由上述内容我们 可以知道，单总线其实归根到底也是一种异步的总线（收发双方时钟不一样）。 我们先来看下DATA这根线。DATA用于微处理器与DHT11之间的通讯和同 步，采用单总线数据格式,一次通讯时间4ms左右,数据分小数部分和整数部分，具 体格式在下面说明,当前小数部分用于以后扩展，现读出为零.操作流程如下：

一次完整的数据传输为40bit,高位先出。

数据格式：bit湿度整数数据+8bit湿度小数数据+8bi温度整数数据+8bit 温度小数数据+8bit校验和。

数据传送正确时校验和数据等于“8bit湿度整数数据+8bit湿度小数数据+ 8bi温度整数数据+8bit温度小数数据”所得结果的末8位。

用户MCU发送一次开始信号后，DHT11从低功耗模式转换到高速模式，等待 主机开始信号结束后，DHT11发送响应信号，送出40bit的数据，并触发一次信号米 集，用户可选择读取部分数据。从模式下，DHT11接收到开始信号触发一次温湿 度采集，如果没有接收到主机发送开始信号，DHT11不会主动进行温湿度采集。

采集数据后转换到低速模式。

我们来看下单总线的时序图，如图55所示。

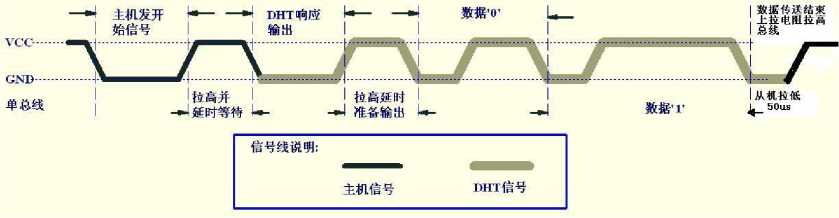


图55单总线时序图

总线空闲状态为高电平,主机把总线拉低等待DHT11响应，主机把总线拉低必

须大于18毫秒，保证DHT11能检测到起始信号。DHT11接收到主机的开始信号后,

等待主机开始信号结束，然后发送80us低电平响应信号.主机发送开始信号结束

后，延时等待20-40us后，读取DHT11的响应信号,主机发送开始信号后，可以切换

到输入模式,或者输出高电平均可，总线由上拉电阻拉高。

由上图可知，我们在做单总线发送的时候，需要先用单片机主动地将总线拉 低18ms，然后再将总线拉高保持20-40us，此时DHT传感器就开始会启动数据 的准备传输。总线为低电平，说明DHT11发送响应信号，DHT11发送响应信号 后，再把总线拉高80us,准备发送数据，每一 bit数据都以50us低电平时隙开 始，高电平的长短定了数据位是0还是1。如果读取响应信号为高电平，则DHT11 没有响应，请检查线路是否连接正常。当最后一 bit数据传送完毕后，DHT11拉 低总线50us,随后总线由上拉电阻拉高进入空闲状态。如图56所示。

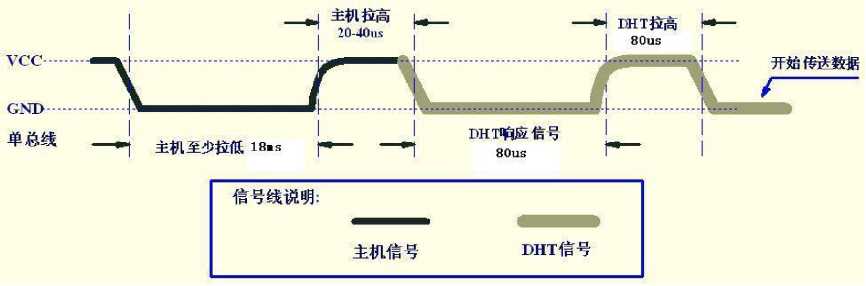


图56启动总线传输 数字0信号表示方法如图57所示。

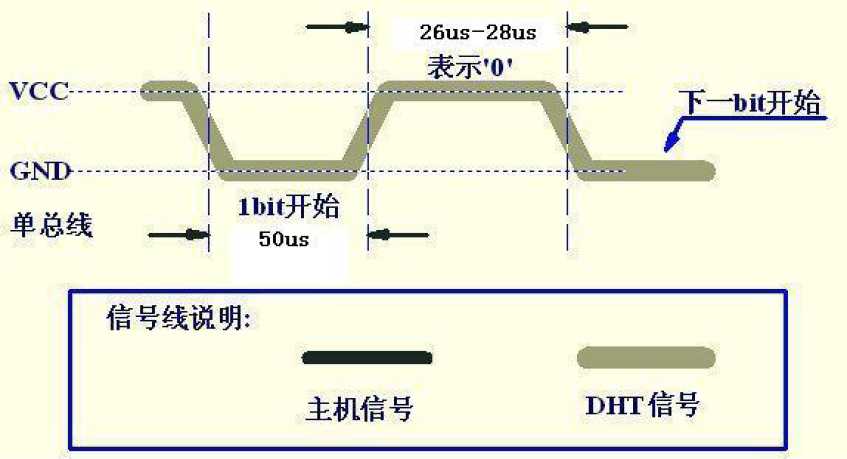


图57数字“0”的表示方法

数字1信号表示方法.如图58所示。

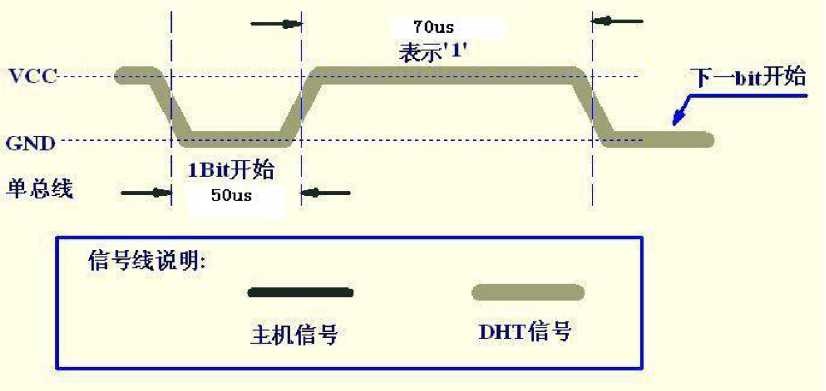


图58数字“1”的表示方法

基于以上知识，我们就可以得出驱动DHT11的方法了。在这里注意一下，

由于我们STM32和DHT11连接的端口需要变化输入输出的方式，因此不需要一 开始就去初始化它，而是在需要使用的过程中去改变端口的输入输出状态。另外， 由于单总线是一种基于异步时间的协议总线，因此需要我们为其提供严格的时间 延迟。同样建立“dht11.h”和“dht11.c”文件。

对于单总线的驱动编写，主要分成以下几个步骤：

\*编写精确延时函数；

\*编写单总线的方向改变函数；

\*编写单总线时序函数；

\*编写数据处理函数。

第一步，编写精确延时函数。由图56可以看出来，在做单总线操作时，对 于延时这一部分要求是比较高的。在这里的延时，需要两个数量级，毫秒延时和 微秒延时。在这里我们可以使用Cortex-M3的Systick去做精确的延时。前面我们 已经给大家说明了，如何去用Systick做毫秒级的延时函数，在这里我们给大家 引入其微秒延时函数的编写，如代码片段45所示。在这里，我们需要对之前写 的代码进行改写，由于这里既有毫秒级的延时，也有微秒级的延时，因此我们可 以将Systick的最小中断时间改成1us。

#include "timer.h"

\_\_IO uint32\_t TimingDelay; void systick\_init(void)

{

T配置Systick重载值，系统时钟为72MHz\*/

厂设置 72000，中断时间：72\* (1/72000000) = 1us\*/

if(SysTick\_Config(72)==1)

{

while(1);

}

}

voidTimingDelay一Decrement(void)

{

if (TimingDelay != 0x00)

{

TimingDelay--;

}

}

void SysTick一Handler(void)

{

TimingDelay一Decrement。；

}

void delay一us(—IO uint32一t nTime)

{

TimingDelay = nTime;//时钟滴答数 while(TimingDelay!= 0);

}

程序片段45

第二步，编写单总线的方向改变函数。由于我们这里的单总线需要做双向的 通讯，因此需要根据情况实时改变输入输出方向。如程序片段46所示。这两个 程序，一个是把PA4这个端口设置成浮空输入模式，一个是把PA4设置成推挽输 出模式。并且我们可以将PA4这个端口的输入输出定义成宏定义的形式，如程序 片段47所示。

voiddhtll一portin(void)

{

GPIO一InitTypeDef GPIO」nitStructure;

RCC一APB2PeriphClockCmd(RCC一APB2Periph一GPIOA, ENABLE);

//Configure pin as input GPIO」nitStructure.GPIO一Pin = GPIO一Pin一4;

GPIO」nitStructure.GPIO一Speed = GPIO一Speed一50MHz;

GPIO一InitStructure.GPIO一Mode = GPIO一Mode一IN一FLOATING; //浮动输入 GPIO一Init(GPIOA,&GPIO一InitStructure);

}

void dhtll一portout(void)

{

GPIO一InitTypeDef GPIO 一InitStructure;

RCC一APB2PeriphClockCmd(RCC一APB2Periph一GPIOA, ENABLE);

//Configure pin as input

GPIO一InitStructure.GPIO一Pin = GPIO一Pin一4;

GPIO一InitStructure.GPIO一Speed = GPIO一Speed一50MHz;

GPIO一InitStructure.GPIO一Mode = GPIO一Mode一Out一PP; //推挽输出

GPIO一Init(GPIOA,&GPIO一InitStructure);

}

程序片段46

#define DHT11一OUT一H GPIO一SetBits(GPIOA, GPIO一Pin一4)

#define DHT11一OUT一L GPIO一ResetBits(GPIOA, GPIO一Pin一4)

#define DHT11一IN GPIO一ReadInputDataBit(GPIOA, GPIO一Pin一4)

程序片段47

第三步，编写单总线时序函数。以上准备工作做好之后，我们开始来编写单

总线的驱动函数，阅读时序图，这个程序的大致原理是，发送命令启动转换。接

着，等待DHT11转换响应并且输出数据。最后，读取数据。

启动总线函数“dhtll一reset();”如程序片段48所示。

void dhtll一reset(void)

{

dhtll一portout();

DHT11一OUT一L;

Delay一us(l9000);

DHT11一OUT一H;

Delay一us(25); dhtll一portin();

}

程序片段48

然后，实时监控DHT11的数据线，直至其产生出低电平，表示DHT11响应 单片机请求，开始传输数据。如程序代码49所示。

u16dht11一scan(void)

{

return DHT11一IN;

}

程序片段49

写完等待函数之后，我们来写一下读取字节的函数。由于DHT11每次输出

40位数据，因此转换成字节，就是5个字节。我们可以分成两个函数来写，一 个是读字节函数，在读字节函数里面，我们再来调用一下读位函数。如程序片段 50所示。

u16 dht11一read一bit(void)

{

while(DHT11一IN == RESET);

Delay一us(40); if(DHT11一IN == SET)

{

while(DHT11」N == SET); return 1;

}

else

return 0;

}

u16dht11一read一byte(void)

{

u16i;

u16 data = 0; for(i=0;i<8;i++)

{

data <<= 1;

data | = dht11一read一bit();

}

return data;

}

程序片段50

有了上述准备之后，我们最后来完成DHT11的驱动程序。这里，我们再来 仔细看一下其数据格式。数据格式：

8bit湿度整数数据+8bit湿度小数数据+8bi温度整数数据+8bit温度小数数据 +8bit校验和

数据传送正确时校验和数据等于“8bit湿度整数数据+8bit湿度小数数据+8bi 温度整数数据+8bit温度小数数据”所得结果的末8位。

其驱动程序如代码片段51所示。

u16dht11一read一data(void)

{

u16i = 0; dht11一reset();

if(dht11一scan()==0)//当检测到DHT11给STM32主控芯片的输入压为低电压时， 说明DHT11开始响应 {

while(DHT11\_IN==RESET);//等待80us的低电平响应信号结束 while(DHT11」N==SET);//等待80us的高电平结束，开始接收数据 for (i = 0; i< 5; i++)

{

Data一Buff[i] = dht11一read一byte();

}

while(DHT11」N==RESET);//等待40位数据输出完后的50us低电平输出结

束

dht11一portout();

DHT11一OUT一H;

if ((Data一Buff[0]+Data一Buff[1]+Data一Buff[2]+Data一Buff[3]) == Data一Buff[4])

{

return 1;

}

else//校验出错 {

return 0;

}

}

else

{

return 0;

}

}

程序片段51

最后，我们所有数据都放在Data一Buff[5]里面，只要对里面的数据进行计算，

处理，就可以得到我们最终的温度和湿度了。如程序片段52所示。

if(dht11\_read\_data())

{

temp = Data\_Buff[2]; hum = Data\_Buff[0];

}

display\_dht11\_temp(temp); display\_dht11一hum(hum);

程序片段52

我们在这里只取其整数部分，小数部分不做处理。到这里之后，我们可以利 用之前写的显示程序，把温湿度发送出来。实现结果如图59所示。大家可以结 合我们之前所讲的知识，给它添加上符号，单位等等，这里就不再演示。



图59 DHT11实验结果

3.1.7定时计数器实验 **(1)基本定时器功能**

STM32中一共有11个定时器，其中2个高级控制定时器，4个普通定时器 和2个基本定时器，以及2个看门狗定时器和1个系统嘀嗒定时器。其中系统嘀 嗒定时器是前文中所描述的SysTick，看门狗定时器以后再详细研宄。今天主要 是研宄剩下的8个定时器。关于这8个定时器的简要说明，如图60所示。

其中TIM1和TIM8是能够产生3对PWM互补输出的高级定时器，常用于三 相电机的驱动，时钟由APB2的输出产生。TIM2-TIM5是普通定时器，TIM6和TIM7 是基本定时器，其时钟由APB1输出产生。由于STM32的TIMER功能比较复杂，

所以只能一点一点的学习。因此今天就从最简单的开始学习起，也就是 TIM2-TIM5普通定时器的定时功能。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 定时器 |  | 计数器輕 | 预5m系数 | 产生DMA请求 |  | 互补输出 |
| TIM1  TIM8 | 16位 | 向上，向下，向上/向下 | 1-65536之间的任意数 | 可以 | 4 | 有 |
| TIM2  TIM3  TIM4  TIM5 | 16位 | 向上，向下，向上/向下 | 1-65536之间的任意数 | 可以 | 4 | 没有 |
| TIM6  TIM7 | 16位 | 向上 | 1-65536之间的任意数 | 可以 | 0 | 没有 |

图60 STM32定时器的简要说明

我们由于这一部分是让大家来快速上手的，因此，我们在这里仅仅先介绍一 下关于通用定时器普通定时模式的使用，关于捕获比较模式以及一些其他高端的 应用，我们将在以后的实例中来讲解。<

接下来，我们来梳理一下STM32TIMX的时钟源，从《STM32中文手册》上 面，可以知道，TIM1和TIM8是挂载在APB2总线上的，而TIM2~TIM7是挂载在 APB1总线上的，因此根据其时钟特性可知，TIM1和TIM8的时钟元为最大72MHz， 而TIM2~TIM7的最大输入时钟源为36MHz。但是，请仔细留意一下图61。

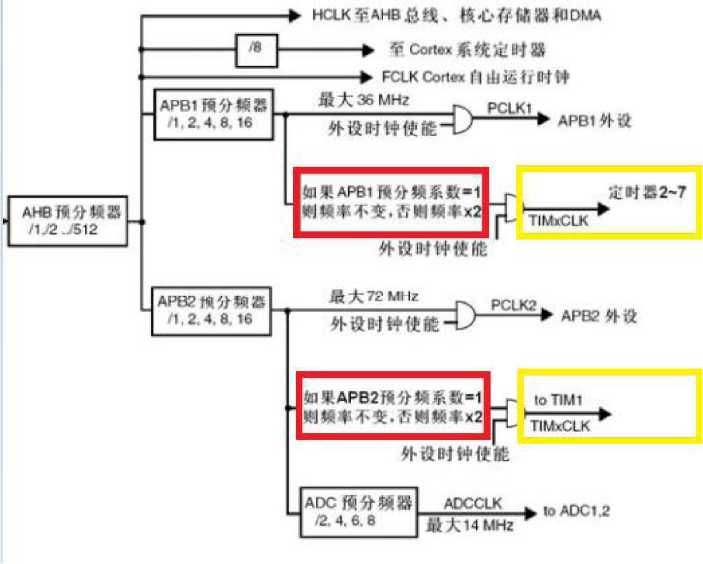


图61定时器的时钟源

图61中，每个时钟源接入之后，都会有一个名为“预分频系数”变量，如 果对这个变量进行修改，可以使接入时钟源双倍频，或者不改变。这是为了满足 APB总线上面其他外设最高工作频率的同时，使得定时器可以工作在更高的频率 上面。因此，就我们现在所学习的通用定时器来说，其完全可以以72MHz的方 式去工作。

关于定时器中断的配置步骤，主要分成以下三步：

\_配置嵌套中断控制器NVCI ；

\*定时器初始化配置；

\*定时器中断函数处理。

假定我们已经配置好了系统时钟，和我们以前的代码一样，使用System」nit() 函数，可以设置默认时钟为72MHz。并且，我们以定时器TIM2为例。我们把定 时器的操作，放在之前建立的“timer.c”文件里面。

第一步，配置嵌套中断控制器。嵌套中断控制器的配置方式，我们在前面已 经详细阐述过，这里就不再赘述，只需要设置“NVICJnitTypeDef”这个结构体 类型里面的每一个参数就可以了。其代码实现如程序片段53所示。

void nvci\_config(void)

{

NVIC\_InitTypeDef NVIC\_InitStructure;

NVIC—PriorityGroupConfig(NVIC—PriorityGroup\_0);//设置组优先级

NVICJnitStructure.NVICJRQChannel=TIM2JRQChannel;"设置定时器 2 中断 NVIC—InitStructure.NVIC—IRQChannelPreemptionPriority = 0;//设置抢占优先级 NVIC—InitStructure.NVIC—IRQChannelSubPriority = 0;//设置子优先级 NVIC—InitStmcture.NVIC—IRQChannelCmd = ENABLE;"使能 IRQ 中断 NVIC—Init(&NVIC—InitStructure);

}

程序片段53

第二步，定时器初始化配置。在这里，我们要做的工作和之前的大部分外设 配置一样，首先开启相应的时钟，然后完成一些参数的设置就可以了。

不过这里有个定时器的核心内容，就是定时中断的时间计算。我们先来看一 个结构体“TIM—TimeBaseInitTypeDef”。这个结构体里面有以下几个元素。

uint16—tTIM—Prescaler： TIM—Prescaler 设置了用来作为 TIMx 时钟频率除数 的预分频值。它的取值必须在0x0000和0xFFFF之间。

uint16—tTIM—CounterMode： TIM—CounterMode 选择了计数器模式。该参数 取值见下表。

|  |  |
| --- | --- |
| TIM\_CounterMode | 描述 |
| TI M\_CountcrMode\_U p | TIM向上计数模式 |
| TI M\_C ounterModeDo wn | TIM向下计数模式 |
| TIM\_CounterMode\_\_CenterAligncd 1 | TIM中央对齐模式丨计数模式 |
| TI MCounterModeCenter A1 i gned2 | TIM中央对齐模式2计数模式 |
| TIM\_CounterModc\_CenterAligncd3 | T1M中央对齐模式3计数模式 |

uint16—tTIM—Period： TIM—Period设置了在下一个更新事件装入活动的自动

重装载寄存器周期的值。它的取值必须在0x0000和0xFFFF之间。 uint16—tTIM—ClockDivision：与捕获比较模式有关，暂时不去管他。

|  |  |
| --- | --- |
| TIM^ClockDivision | 描述 |
| TIM CKD DIV1 | TDTS=Tck tim |
| TIM CKD DIV2 | TDTS=2Tck tim |
| TIM CKD DIV4 | TDTS=4Tck tim |

uint8—t TIM—RepetitionCounter：仅用于 TIM1 和 TIM8。

这里面，我们可以简单的理解成，溢出中断的时间是由TIM—Period和 TIM—Prescaler 共同决定。

假设HCLK为72MHz，那么接入TIM2的时钟为72MHz。

而TIM\_Prescaler可以理解为时钟预分频数，即TIM\_Prescaler-1个TIM2的

时钟频率，可以为定时器提供一个定时的步进信号。

TIM\_Period可以理解为定时器的初值，如果向上计数模式来说，定时器从0 开始进行累加，直到加到TIM\_Period，进行溢出，产生中断。

因此，我们可以得到定时器的定时公式：

发生中断时间=(TIM\_Prescaler+1)\* (TIM\_Period+1)/FLK 以定时 1s 为例 TIM\_Period=2000-1，TIM\_Prescaler=(36000-1)。

定时器配置的代码，如程序片段54所示和程序片段55所示。

void nvci\_config(void)

{

NVIC\_InitTypeDef NVIC\_InitStructure;

NVIC—PriorityGroupConfig(NVIC—PriorityGroup\_0);//设置组优先级

NVIC\_InitStructure.NVICJRQChannel=TIM2JRQn;"设置定时器 2 中断 NVIC—InitStructure.NVIC\_IRQChannelPreemptionPriority = 0;//设置抢占优先级 NVIC—InitStructure.NVIC\_IRQChannelSubPriority = 0;//设置子优先级 NVIC—InitStructure.NVIC—IRQChannelCmd = ENABLE;"使能 IRQ 中断 NVIC—Init(&NVIC—InitStructure);

}

程序片段54

voidtimer2—config(void)

{

TIM—TimeBaseInitTypeDefTIM—TimeBaseStructure;

nvci—config();

RCC—APB1PeriphClockCmd(RCC—APB1Periph—TIM2,ENABLE);

TIM—DeInit(TIM2);

TIM—TimeBaseStructure.TIM—Period=2000-1; //自动重装载寄存器的值 TIM—TimeBaseStructure.TIM—Prescaler=(36000-1); //时钟预分频数

TIM—TimeBaseStructure.TIM—ClockDivision=TIM—CKD—DIV1; //采样分频 TIM—TimeBaseStructure.TIM—CounterMode=TIM—CounterMode—Up;//向上计数模式 TIM—TimeBaseInit(TIM2,&TIM—TimeBaseStructure);

TIM—ClearFlag(TIM2,TIM—FLAG—Update); //清除溢出中断标志

TIM」TConfig(TIM2,TIM」T—Update,ENABLE);

TIM—Cmd(TIM2,ENABLE); //开启时钟

}

程序片段55

第三步，定时器中断函数处理。与前面的中断方式一样，这里的中断处理函 数的入口地址为TIM2」RQHandler。同时，我们需要用到“ if(TIM\_GetlTStatus(TIM2, TIM\_IT\_Update)!= RESET)”语句，去确认中断是否发生，并且退出时候使用 “TIM\_ClearITPendingBit(TIM2,TIM\_FLAGJJpdate);” 进行标志位清除。我们这里

需要让LED灯闪烁，因此我们只需要使用程序片段56里面的代码即可。到此为 止，即可实现一个功能：使开发板上的一个LED进行闪烁，1s亮，1s灭。示波 器波形如图62所示。

void TIM2」RQHandler(void)

{

if(TIM\_GetITStatus(TIM2,TIM」T\_Update) != RESET)

{

led0\_state = (led0\_state >= 1) ? 0 : led0\_state + 1; if(led0\_state == 0)

{

LED0\_ON();

}

else

{

LED0\_OFF();

}

}

TIM\_ClearITPendingBit(TIM2,TIM\_FLAG\_Update);

}

程序片段56

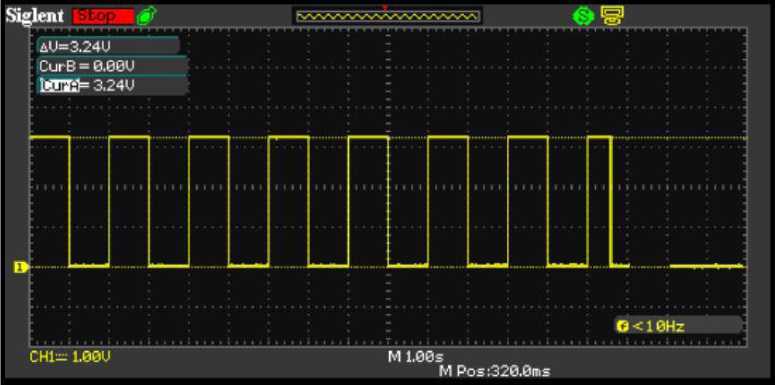


图62 LED闪烁控制波形

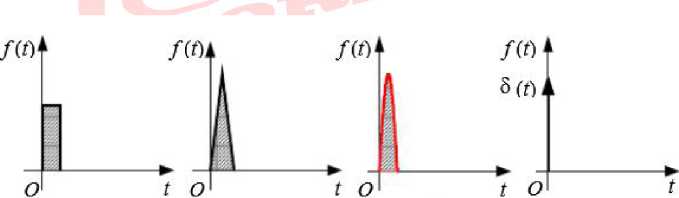
**(2)利用软件算法产生**PWM**波**

在我们前文提到了一点，STM32的定时器可以设置捕获比较模式，产生PWM 波，这是个非常有用的功能。然而，我们在这里没有用到这些功能相关的引脚去 做设计。同时，在我们使用一部分芯片里面，可能也没有包含此功能。那如果以 后要用到，肯定需要自己通过软件来实现。所以我们在这里就介绍一下，关于利 用软件算法，来产生一个PWM波。

我们平时所说的PWM波，这个说法包含了两层意思：首先，PWM是一种 技术；其次，它是以电压或者电流波形的形式展现出来的。

请允许我用不少的篇幅来介绍一下何为PWM技术。PWM英文全拼为“Pulse Width Modu丨ation”即脉宽调制的意思。对于PWM波形原理，都是从比较复杂 的采样控制原理里面推导出来的，即“冲量相等而形状不同的窄脉冲加在具有惯 性的环节上时，其效果基本相同。”冲量指窄脉冲的面积；效果基本相同，是指 环节的输出响应波形基本相同。如图63所示。我们可以知道，图63中的波形， 不管是第一张里面的方波，第二张里面的三角波，第三张里面的正弦，以及第四 张里面的脉冲，只要其面积相等，那么如果其以一种惯性环节输出（可以简单地 理解成系统输出具有一个储能元件），这四张图的波形相应基本相同。

因此由于以上的内容铺垫，我们可以简单地以SPWM技术为例，请看图64。 脉冲宽度按正弦规律变化而和正弦波等效的PWM波形，要改变等效输出正弦波 幅值，按同一比例改变各脉冲宽度即可。图64里面，我们假设PWM的每个方 波的幅度相同，并且正弦波被分割成相同的等份，那因此，只需要让每一份的正 弦波面积，等于方波面积，那其效果就相同了。因此只要让方波的时间进行改变， 即可以改变输出的电压幅度了。因此我们将正弦波推广到直流电压里面，如果我 们要将直流电压降低n倍，只需要让方波的占空比为1/n即可。这就是PWM的 原理。



a) b) c) d)

图63采样控制定量

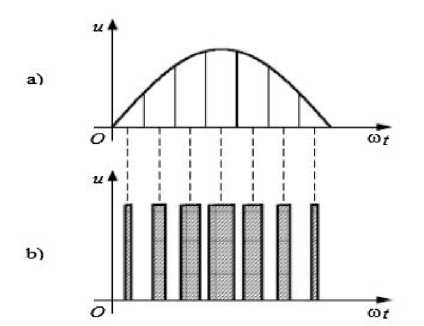
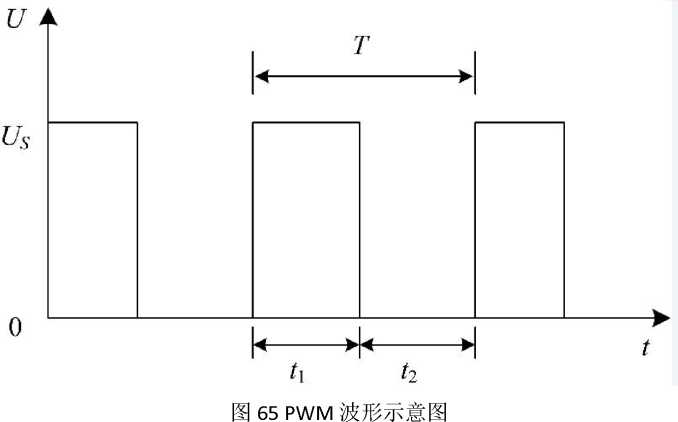


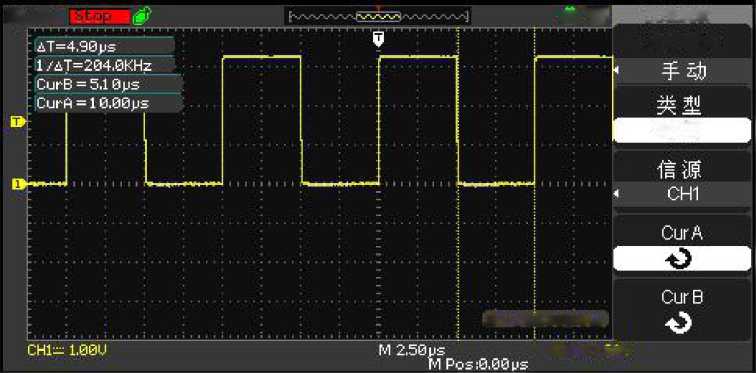
图64正弦等效PWM

接下来，我们就开始自己动手写PWM的程序，我们这次实验的目标是产生 一个PWM波来调节LED的亮度。

首先，我们来概括一下PWM的两个最基本的要素，第一个要素，即为PWM 的频率，第二个要素，即为PWM的占空比。如图65所示。我们将T的倒数， 称为PWM波的频率，t1与T的比值，称为PWM波的占空比。



接着，我们来编写一个函数，这个函数的作用即是来输出PWM波的频率。 同样的，我们利用基本定时器T2来产生一个基础最小频率，为了方便计算，我 们在这里将这个最小的频率设置为100kHz。通过上节课学习的知识，我们只需 要对tim2\_config函数做出如程序片段57里面修改的内容即可。波形图如图66 所示。

voidtim2一config(void)

{

TIM一TimeBaselnitTypeDef TIM一TimeBaselnitStruct;

tim2一nvic一config();

RCC一APB1PeriphClockCmd(RCC一APBIPeriph一TIM2,ENABLE); TIM一DeInit(TIM2);

TIM一TimeBaselnitStruct.TIM一Prescaler = 36-1;

TIM一TimeBaselnitStruct.TIM一CounterMode = TIM一CounterMode一Up; TIM一TimeBaselnitStruct.TIM一Period = 10-1;

TIM一TimeBaselnitStruct.TIM一ClockDivision = TIM\_CKD一DIV1;

TIM一TimeBaseInit(TIM2, &TIM一TimeBaseInitStruct); //初始化 TIM2 配置

TIM一aearFlag(TIM2,TIM-FLAG一Update);//清除溢出中断标志 TIM」TConfig(TIM2,TIM」T\_Update,ENABLE);

TIM\_Cmd(TIM2,ENABLE);//开启时钟

}

程序片段56

mW cursor 舰草式

**Sclent**

时间

G = 10B.000KHZ

'chi yi.btiu

图66 100K PWM波50%占空比波形

我们来分析一下这个波形，由于它是一个100K的方波，因此，我们可以知 道，定时器在1s内中断了 200k次。假设我们的占空比变化率在0-100%内任意 可调，因此我们可以将没100次中断作为一个基准，所以在我们这个基础上面可 以将PWM波的频率设置成2K。

最后，我们PWM波的基本思路就是，定义一个变量作为“占”（即高电平) 的时间，随后我们可以得到‘‘空”（即低电平）时间就是100-“占”的时间。以

上就是PWM波的大致思路，具体程序的实现如程序片段57所示。波形如图67 所示。

u8 pwm\_high\_counter; //定义的存放PWM ‘‘占”的计数器

void set\_pwm\_high(u8 high\_pd)

{

pwm\_high\_counter = high\_pd;

}

void TIM2」RQHandler(void)

{

if(TIM\_GetITStatus(TIM2,TIM」T\_Update) != RESET)

{

led0\_state ++;

if(led0\_state <= pwm\_high\_counter)

{

LED1\_ON();

}

else if((led0\_state > pwm\_high\_counter) && (led0\_state <= 100))

{

LED1\_OFF();

}

else

{

led0\_state = 0;

}

}

TIM\_ClearITPendingBit(TIM2,TIM\_FLAG\_Update);

}

程序片段57

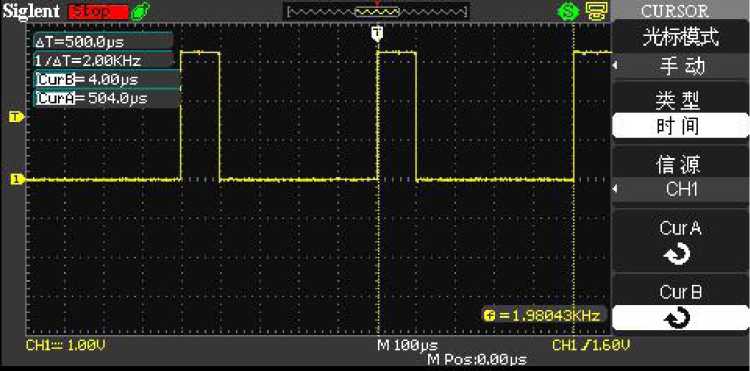


图67频率为2K，占空比为20%的波形

我们再补充一点，关于LED呼吸灯的应用。其实LED呼吸灯就是由PWM波 控制的，它的大致思路是，每隔一定时间，将PWM波的占空比从0开始加到100, 再从100减到0。程序如程序片段58所示。

void set\_pwm\_high(void)

{

if(breath」ed2.breath\_step\_counter >= 4000)

{

if(breath」ed2.high\_flag == 0x00)

{

pwm\_high\_counter ++; if(pwm\_high\_counter >= 99)

{

breath\_led2.high\_flag = 0xff;

}

}

else if(breath」ed2.high\_flag == 0xff)

{

pwm\_high\_counter --; if(pwm\_high\_counter <= 0)

{

breath\_led2.high\_flag = 0x00;

}

}

else

{

breath\_led2.high\_flag = 0x00;

}

}

}

void TIM2」RQHandler(void)

{

if(TIM\_GetITStatus(TIM2,TIM」T\_Update) != RESET)

{

led0\_state ++;

if(led0\_state <= pwm\_high\_counter)

{

LED1\_ON();

}

else if((led0\_state > pwm\_high\_counter) && (led0\_state <= 100))

{

LED1\_OFF();

}

else

{

led0\_state = 0;

breath\_led2.breath\_step\_counter ++;

}

}

TIM\_ClearlTPendingBit(TIM2,TIM\_FLAG\_Update);

}

u8 pwm\_high\_counter = 0; //定义的存放PWM “占”的计数器

struct breath\_type {

u16breath一step一counter;

u8high\_flag;

}breath」ed2;

程序片段58

3. 1. 8 RTC时钟实验

在我们平时的设计工作当中，时钟芯片的应用场合非常多。在我们平时接触 的电子产品里面，大到个人电脑，小到电子表，无不依赖于时钟的功能。有些读 者会存在一个疑问，时钟芯片不就是定时器芯片吗？这个疑问的答案当然是否定 的，如果按照实际的内部结构来说，时钟芯片内部的确主要是由定时器构成。只 不过在定时器的基础上面还加了独立的电源模块（一般用3V的纽扣电池），晶 振模块（为时钟提供基准），非易失性存储器（保存一些常用的数据）以及常用 的一些数据接口（方便用户读取和写入时间）。说到这里，我想大家大致对时钟 芯片的内部结构应该有个了解了，如果有兴趣的同学，可以试着用数字逻辑芯片 去自己搭建一个分立元件的时钟“芯片”，只需要很简单的74161或者74163 芯片就可以当作计数器来用。回过来，我们以前常用的时钟芯片有DS1302、 DS1307、PCF8454等等，这些芯片价格低廉，使用方便，被广泛采用。

我们这节课主要讲的是STM32内部的RTC资源的使用。STM32内部自带一 个RTC时钟，关于其RTC的特性，我们稍后来看《STM32中文参考手册》。在 这里，需要说明一点，尽管STM32手册上面讲了它的RTC时钟有很多优越的特 性，但是我们在这里还是推荐大家在以后的设计制作当中，尽可能使用外部独立 的RTC芯片，因为在以往STM32使用RTC时钟的时候，出现过很多问题，比如

时间误差偏大，LSE (外部低速时钟）间歇性无法起振，LSI (内部低速时钟）误 差偏大等等问题。我们在这里讲述这个内容，仅仅是为了演示给大家观看而已。

关于STM32RTC时钟的使用，可能很多同学在使用时候，都会非常迷茫，明 明按照固件库手册上面使用的函数，却怎么也无法正确使用起来，原因是因为 STM32 RTC和很多其他资源都有紧密的关联，而STM32固件库提供的函数有太 过于分散，因此在使用时候，就出现了各种各样的问题。我们现在就对这个部分 进行详细的解说。

首先，先来看下STMRTC所牵涉到的其他内部资源，这些资源有备份寄存器 (BKP)以及功耗控制（PWR)。

鲁RTC简介：正如我们上文所讲的，RTC模块的核心部件是一个32位的可编程 计数器，并且有一个独立的时钟晶振与其配套。它提供给我们一系列的接口 来设置这个计数器，以及读取我们所需要的当前计数器值等等。

* PWR为电源管理模块，我们需要使能电源控制寄存器（PWR\_CR)的DBP位

来取消后备取余BKP的写保护来允许写入RTC和后备寄存器。 £

* BKP为备份存储器模块，我们需要通过它来存储RTC的配置参数，可以在重 启或者复位之后，RTC里面的配置数据不变。

接着，我们来看下RTC内部的具体框图。如图68所示。

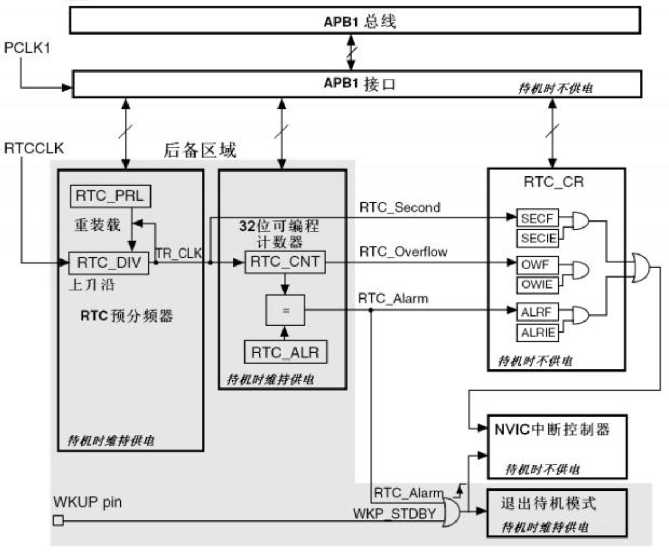


图68RTC内部框图

这里，我们需要说明一下的是，RTC的时钟源RTCCLK是可以通过配置RCC 来选择是用内部的LSI还是外部的LSE (前提是焊接了）的。我们在这里使用LSE。 接着，我们来给出整个配置过程。

第一步，开启BKP和PWR模块的时钟，使用函数 “ RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_PWR | RCC\_APB1Periph\_BKP ,ENABLE)

/

第二步，使能电源控制寄存器的DBP位来允许对备份寄存器和RTC的访问。 使用函数 “ PWR\_BackupAccessCmd(ENABLE); ”

第三步，初始化备份寄存器，使用函数“BKP\_Delnit();”

第四步，使能LSE,并且等待其起振。

RCC\_LSEConfig(RCC\_LSE\_ON);

while (RCC\_GetFlagStatus(RCC\_FLAG\_LSERDY) == RESET)

{

；

}

第五步，选择LSE作为RTC的时钟，并且使能RTC,等待RTC和APB1时钟

同步。

RCC\_RTCCLKConfig(RCC\_RTCCLKSource\_LSE);

RCC\_RTCCLKCmd(ENABLE);

RTC一WaitForSynchro();

第六步，配置RTC的参数。

RTC一WaitForLastTask();

RTC\_ITConfig(RTC」T\_SEC,ENABLE);

RTC一WaitForLastTask();

RTC一SetPrescaler(32767);

RTC一WaitForLastTask();

第六步里面的函数，在固件库手册里面都可以查阅到其含义，在大家以后的 使用中，可以不需修改直接使用，这里需要说明一下的是函数 “RTC一SetPrescaler(32767);”是设置RTC分频系数的，由于我们这里使用了中断， 因此大家可以近似认为是设置中断时间的。但是，即使你不用中断也需要进行设 置，因为这个参数直接影响到技术的时间间隔，有公式：fTR\_CLK = fRTCCLK/(PRL[19:0]+1)，函数的函数，即为PRL[19:0]的值，因为我们这里的LSE 是32768Hz，因此可以计算，当我们设置32767的时候，fTR\_CLK正好是1s。 第七步，配置RTC的中断（如果需要）。 void NVIC\_Configuration(void)

{

NVIC」nitTypeDefNVIC」nitStructure;

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_1);

/\* Enable the RTCInterrupt \*/

NVIC」nitStructure.NVIC」RQChannel = RTC」RQn; NVIC」nitStructure.NVIC」RQChannelPreemptionPriority=1; NVIC」nitStructure.NVIC」RQChannelSubPriority = 0; NVIC」nitStructure.NVIC」RQChannelCmd = ENABLE; NVIC」nit(&NVIC」nitStructure);

}

第八步，编写中断处理函数。

void RTC\_IRQHandler(void)

{

if (RTC\_GetITStatus(RTC\_IT\_SEC) != RESET)

{

/\* Clear the RTC Second interrupt \*/ RTC\_ClearITPendingBit(RTC\_IT\_SEC);

RTC\_WaitForLastTask();

}

}

以上就是我们对RTC配置流程的说明。

实际读者在操作的过程中，可以发现，以上的步骤是有非常大的问题的，就

是每次数据在断电之后，都不能保留下来，原因是因为第三步中“BKP\_Delnit();”

函数的不当使用所导致的，我们应该将其去掉，去掉后方能正常进行读取。如程

序片段59所示。

int main(void)

{

u8 i,j;

^禁止使用JTAG引脚，开启复用，当作普通I/O\*/

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_AFIO,ENABLE);

GPIO\_PinRemapConfig(GPIO\_Remap\_SWJ\_JTAGDisable,ENABLE);

//初始化系统，使得系统频率为72M //配置Systick，使得1ms产生中断

**SystemInit(); systick\_init(); tim2**一**config(); led\_gpio\_init (); key**一**gpio**一**init (); usart**一**config(); Init**一**ST7567(); LCD**一**BK**一**ON;**

//配置LED驱动管脚 //配置按键输入引脚

RCC一APB1PeriphClockCmd(RCC一APBIPeriph一PWR | RCC一APBIPeriph一BKP ,ENABLE); PWR一BackupAccessCmd(ENABLE);

//BKP一DeInit();

RCC一LSEConfig(RCC一LSE一ON);

while (RCC一GetFlagStatus(RCC一FLAG一LSERDY) == RESET)

{ }

RCC一RTCCLKConfig(RCC一RTCCLKSource一LSE); RCC一RTCCLKCmd(ENABLE);

RTC一WaitForSynchro();

RTC一WaitForLastTask();

RTC一ITConfig(RTC一IT一SEC, ENABLE);

RTC一WaitForLastTask();

RTC一SetPrescaler(32767);

RTC一WaitForLastTask();

NVIC一InitTypeDef NVIC一InitStructure;

/\* Configure one bit for preemption priority \*/

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_1);

/\* Enable the RTC Interrupt \*/

NVIC」nitStmcture.NVIC」RQChannel = RTC」RQn; NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 1; NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0; NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE; NVIC\_Init(&NVIC一InitStructure);

for(i = **0;i<** 8; i++)

{

write\_lcd\_cmd\_spi(0xb0+i); for(j = **0;j<** 128; j++)

{

write\_lcd\_cmd\_spi(0x10 + ((j & 0xf0) >> 4)); write」cd\_cmd\_spi(0x00 + (j &0x0f)); write」cd\_data\_spi(0x00);

}

}

temp = BKP\_ReadBackupRegister(BKP\_DR1); while⑴

{

//temp = BKP\_ReadBackupRegister(BKP\_DR1); display\_num(1, 1, 1,RTC\_GetCounter()); breath一pwm一change();

}

}

void RTC一IRQHandler(void)

{

if(RTC一GetITStatus(RTC一IT一SEC) != RESET)

{

/\* Clear the RTC Second interrupt \*/

RTC一ClearITPendingBit(RTC一IT一SEC);

//temp = RTC\_GetCounter();

//BKP一WriteBackupRegister(BKP一DR1, temp);

/\* Wait until last write operation on RTC registers has finished \*/ RTC一WaitForLastTask();

}

}

程序片段59

另外提一点，关于备份寄存器BKP的使用，有很多同学可能会有疑问，为什 么STM32这么一块先进的芯片，整个系列都找不到片内的EEPROM?关于这个问 题，我的解释是这样的，由于STM32内部所有的存储器都是FLash,并且按照 ARM的存储器架构，我们可以任意获取STM32的Flash Section作为我们的非易 失性存储器来长期保存我们的数据，这样的话，即使你有几十K的数据也可以很 好的放下，这是第一个理由。第二个理由是因为STM32内部具有一个BKP寄存 器可以共用户存储一些较小的数据，并且这个寄存器在掉电的时候还可以正常被 我们的RTC所操作。这和EEPROM相比，具有更高的性价比。

我们这里再来给大家说一下关于STM32 BKP寄存器的使用，在使用之前，

也需要和RTC—样去设置PWR，BKP这两个模块。但是在我们使用的时候，只要 使用 “void BKP\_WriteBackupRegister(u16 BKP\_DR, u16 Data)” 进行写操作，u16 BKP—ReadBackupRegister(ul6BKP—DR)进行读操作。关于BKP—DR这个参数，根据不同 容量的芯片有不同的大小，如图69所示。

|  |  |
| --- | --- |
| BKP^DR | 描述 |
| BKP一 DR 1 | 选屮数据寄存器i |
| BKP-DR2 | 选中数据寄存器2 |
| BICP-DR3 | 选中数据奇存器3 |
| BKP\_DR4 | 选中数据寄存器4 |
| BKP\_DR5 | 选中数据寄存器5 |
| BKP\_DR6 | 选中数据寄存器6 |
| BKP\_DR7 | 选中数据寄存器7 |
| BKP\_DR8 | 选中数据奇存器8 |
| BKP\_DR9 | 选中数据寄存器9 |
| BKP\_DR10 | 选中数据寄存器10 |

例：

/\* **Write 0xA587 to Data Backup Registerl** \*/ **BKP\_WriteBackupRegister(BKP\_DR1# 0xA587**)；

图69 BKP\_DR的参数列表