8 路灰度传感器数据手册

Powered by LaTeX

感为智能科技

2023 年 7 月 6 日

感为科技

开篇语

首先感谢您对感为的支持！

2019 年我前前后后花了半年时间，推出了第一款灰度传感器，它的设计初衷是解决：校准繁琐、易受光干扰的问题。

我还依然记得，大二那年，我们信心满满的跑去青岛打比赛，出发前我们信誓旦旦的说“程序已经很完美了，又快又稳，一定没问题的!” 就这样，本以为完美无缺的小车封上了封条，带到了比赛现场。

让我意想不到的是，在体育场内，有一扇窗子擦得明亮，正巧一束阳光泼在我们的跑道上。可能赛车在留恋阳光的美，每次经过它都会跑出轨道。站在一旁的我们急的我们满头大汗，每次重来都要跪在赛道上调灰度传感器。

最终也没能挽救回比赛的败局。后来，我毕业了，有了研发能力，于是做出了灰度智能传感器。目的之一，我不要再让我的学弟学妹们拿着螺丝刀跪着调车了，目的二，我不要让车子再出轨了。

如今推出了 8 路灰度传感器，它是继承于与单路灰度传感器而研发的多路传感器，其目的是降低产品价格，增加产品的功能。

8 路灰度传感器集成了并行通讯、串行通讯、I^2C 通讯，以适应不同需求的学弟学妹。在功能上，并/串通讯只能传输通道数字量。I^2C 通讯，可以读取通道数字量、读取通道模拟量、读取错误信息、写入重启命令、ping 网络诊断等功能。是一款能适用于更多人的优秀产品。

重要事项

- 为获得最佳效果，建议您避免在阳光直射的室外校准传感器。

- 尽管传感器有滤光膜防止过曝，但是感光元件具有物理上限，请尽量避免在强烈日照条件下使用。当到达感光上限后，传感器 **ERR** 会亮起，这种情况下传感器可能出现误判。此时只需对传感器进行简单的遮挡，例如手持一把遮阳伞，使之至于阴影下，避免阳光直射即可。

- 在强烈日照条件下，且 **ERR** 亮起时，不允许校准传感器，请将传感器置于弱光下，断电重启，或用 I^2C 发送重启命令后再行校准。

- 传感器校准后，请勿改变安装位置，传感器对于距离十分敏感，检测距离的变化会引发误判。如有必要，重新校准。

- 如果赛道中有上下坡场景，请适当提高一些安装高度，以降低检测距离的波动对检测精度的影响。

- 传感器具有记忆功能，重启后不需要重新校准

- 传感器的电压务必准确且稳定。尤其是有电机应用的场景，务必选用优质的电机驱动器，以防电机在启停机时产生电压尖峰冲击传感器。

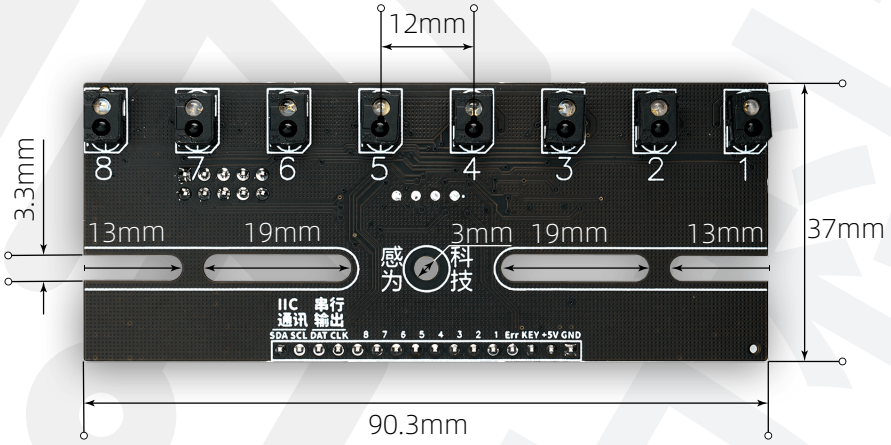
目录

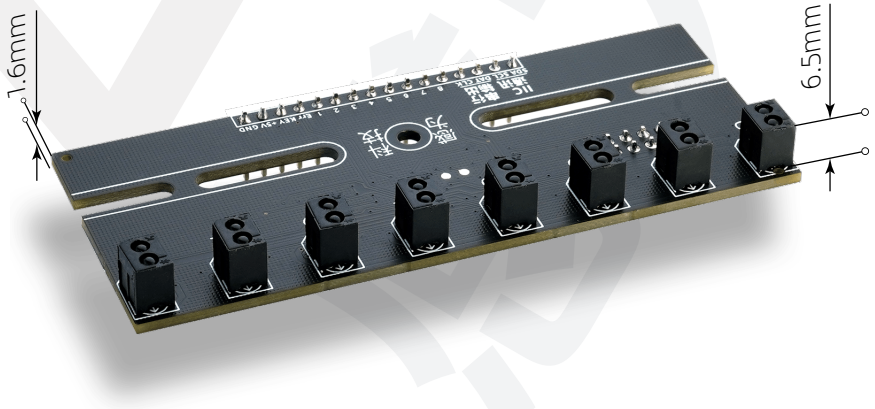
1 产品参数

1.1 产品参数

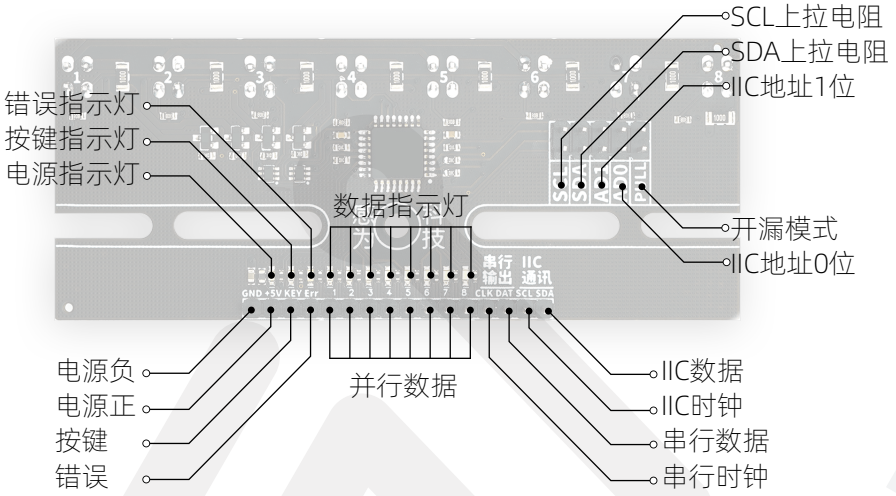
性能参数						
	符号	最小值	典型值	最大值	单位	测试条件
最大端口电压	U_{IO_max}	$0.7 \times U$	3.3 或 5	U+0.3	V	$U = 5V$
工作电压	U	4.5	5	5.5	V	——
工作电流	I	32.62	33	46.5	mA	$U = 5V$
检测高度	h	4	20	100	mm	$U = 5V$
响应速度	t	1.5	1.52	1.55	ms	$U = 5V$
工作温度	T	-25	——	85	℃	——
安装孔尺寸	——	3.3			mm	——
重量	G	15.3			g	——
通讯技术	1、并行通讯；2、串行通讯；3、I^2C 通讯					
I^2C 传输的数据	1、数字量；2、模拟量；3、滞回比较器参数；4、错误信息……					

1.2 传感器尺寸





1.3 名称与功能



指示灯名称及功能：

- 电源指示灯：用于指示供电。
- 按键指示灯：用于设置与交互。
- 错误指示灯：用于指示故障。
- 数据指示灯：用于指示每路开关了信息。

端口名称及功能：

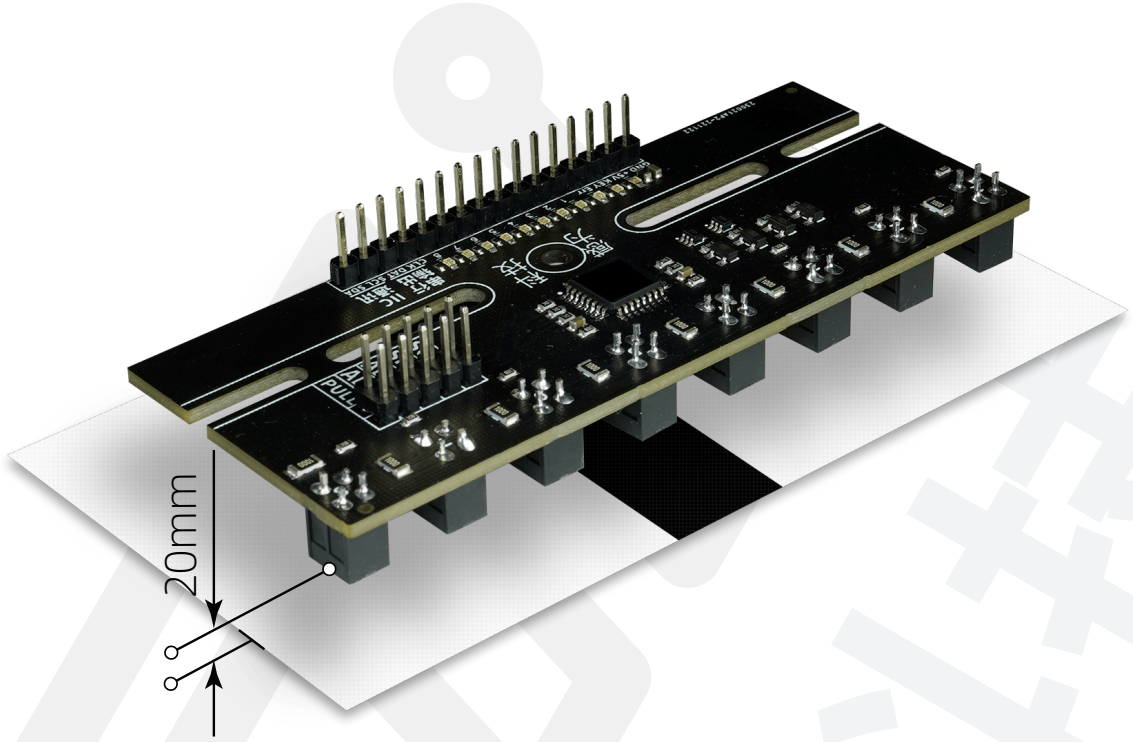
- 电源正、电源负端口：用于接 5V 电源。
- 按键端口：用于外接按键，按键的两个脚需连接至按键端口与电源负极。
- 错误端口：用于向外输出报错信号，该端口与红色 LED 灯相连，可以悬空不用。
- 并行数据：用于向外传输每一路的开关量信息。
- IIC 数据、IIC 时钟端口：用于连接 IIC 主设备，是 IIC 的标准接口，该端口浮空，需主机提供上拉电压。
- 串行数据、串行时钟端口：用于传输开关量的串行端口，与并行端口传输数据相同。

跳线帽名称与功能：

- SCL、SDA 上拉电阻：插入跳线帽，IIC 总线被上拉至 5V，但该功能为应急使用，请谨慎使用。
- IIC 地址位 1、0：插入跳线帽相应的地址位为 1。
- 开漏模式：插入跳线帽后启动设备，并行数据、串行数据（DAT）进入开漏模式，该模式用于兼容 3.3V 用户主控，注意如果您使用 IIC 通讯则无需开启开漏模式。

2 设备的安装

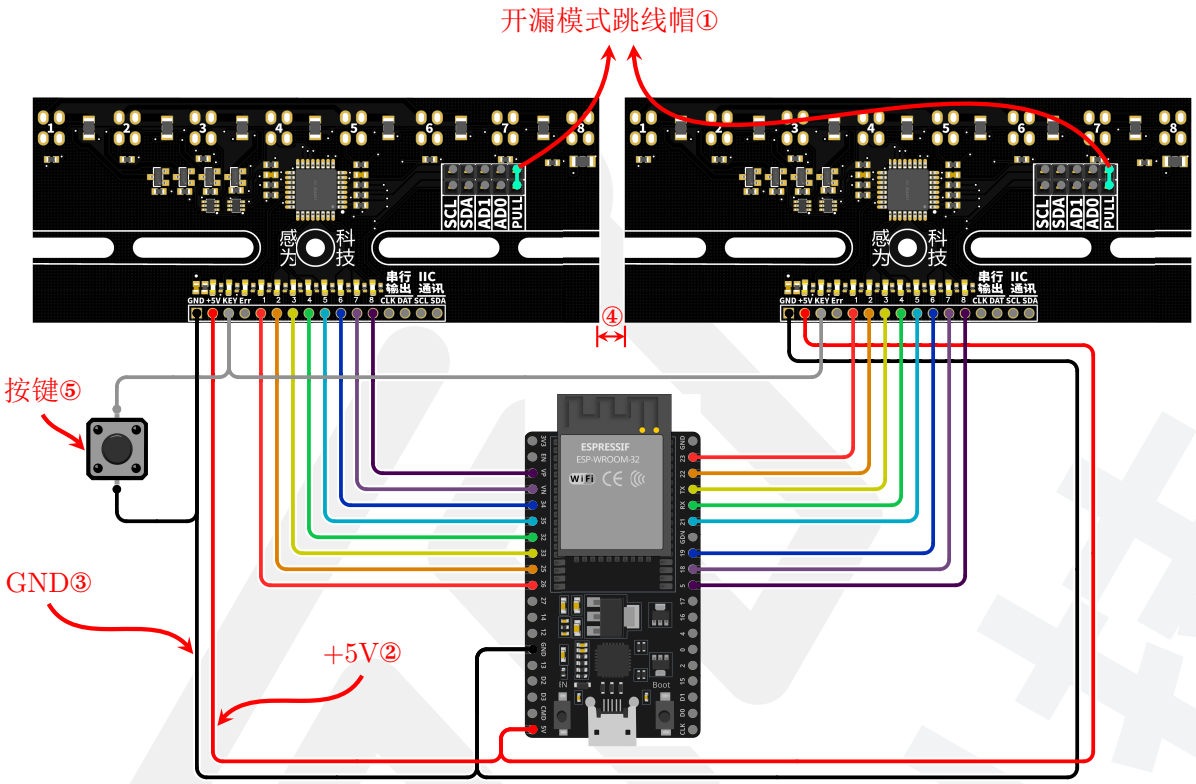
2.1 传感器安装



新手朋友一定要按照以下步骤操作，切勿擅自上机，以免造成不必要的麻烦。安装事项：

- 请务必确保 5V 供电的稳定，请勿与舵机、电机、喇叭等感性负载（有电感属性的负载）共用一个 5V 电源，由于感性负载在电流发生突变时会产生非常高的电压，可以轻松烧毁传感器。
- 初次拿到传感器，需要按照试机步骤试机，以验证产品的质量。
- 试机第一步：请勿安装螺丝到赛车上，首先安装 5V 电源和按键，按键的安装请参考章节2.2（章节号可直接点击），只参考按键接线部分，其余任何线路都不要连接，请勿安装任何跳线帽。
  1、可以避免外置线路、跳线帽影响数据指示灯的正常工作；2、避免盲目上电损毁用户主控。
- 试机第二步：将传感器拿在手上，确保检测高度稳定在 20mm 左右，按照校准步骤进行校准，请参考章节2.3。
- 试机第三步：校准完成后，请用黑白色简单测试传感器的性能，以确保后续的故障排除。
- 测试没问题后，请将传感器安装到车上，探头距地面 20mm 处，传感器设有一排安装槽孔，可以安装上铜柱，以便调试高度。
- 请确保安装稳定牢固，灰度传感器是对安装距离敏感的设备，如果安装不稳定会影响输出质量。
- 安装高度请根据实际情况自由调整，调整高度的相关知识请参考章节4.3（章节号可直接点击）

2.2 传感器接线



①： 实例中的 ESP32 运行电压为 3.3V，而灰度传感器供电为 5V，端口电压不匹配，有烧毁 ESP32 的风险。插上跳线帽，设备运行在开漏模式下，开漏模式请参考章节5.3（章节号可直接点击），此模式下，设备端口悬空，需要外接上拉电阻至 3.3V。而 ESP32 有内部上拉源，仅需将 GPIO 设置为上拉输入，即可完成电压的匹配。

注意: 1、请在安装好跳线帽后启动设备，开漏模式在设备开机阶段完成，在使用途中改动跳线帽无效。

2、在安装好跳线帽后，由于端口进入开漏状态，开漏状态端口仅做导通与断开动作，无电压，数据指示灯也不会亮起，开漏模式请参考章节5.3（章节号可直接点击）。请安装或设置上拉电阻，即可正常亮起指示灯。

3、在使用串行的开漏模式下，只有 DAT 进入开漏。此时需设置 DAT 为上拉输入，SCL 设置推挽输出即可，此时各路指示灯灯不会亮，望悉知，您可以用 9P 的 1K 排阻上拉，即可使之亮起。

4、在使用 IIC 时，无需进入开漏模式。

②： 官方推荐 5V 供电。

③： 灰度传感器要与 ESP32 共地，设备间需要共地才能完成通讯。

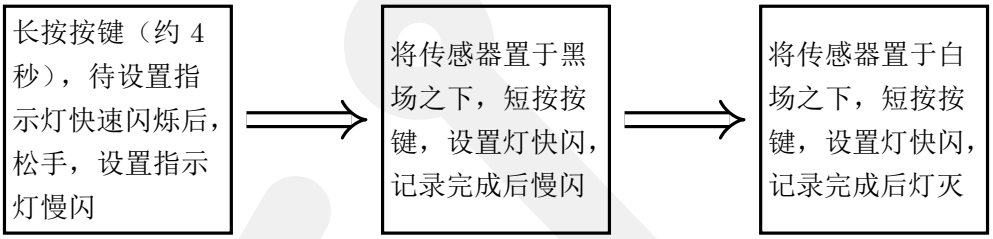
④： 安装距离不能太近，因为两个灰度传感器的时钟不同步，如果太近，设备之间的光信号会相互干扰，造成误判。

安装时请遵循公式：$D_l \geqslant 3 \times \tan(15°) \times D_h$，其中 D_l 为对管中心点间距，D_h 为对管顶点距检测面的高度。

⑤: 请将按键的一个引脚接入灰度传感器的按键端口，另一个引脚请接入电源负端口。此外，当有多个灰度传感器共同使用时，您可以共用按键。

2.3 设置校准

在每一次更改安装高度后，都应该重新校准传感器。传感器校准过程我们拿白底黑线寻迹场景为例，白色为白场，黑色为黑场。黑白场是指一种宏观的概念，白场可以是蓝色，黑场可以是黄色，黑白场是传感器视角下的一种黑与白，是相对量。在校准传感器时，请按下图校准，黑白场无先后顺序。



2.4 输出逻辑

数据端口输出逻辑：

- 当某一路灰度值接近白场 时，这一路的数据端口输出高电平。
- 当某一路灰度值接近黑场 时，这一路的数据端口输出低电平。

错误端口输出逻辑：

- 当设备正常运行时，错误端口输出为低电平，此时无错误指示。
- 当错误 事件发生时，错误端口输出高电平，错误指示灯亮起。随着错误消失，错误端口立即输出为低电平。
- 当校准时发生错误，错误端口会交替输出高低电平方波，错误指示灯闪烁，重启设备方可复位。

2.5 故障排查

2.5.1 基础排错

故障现象	可能原因	解决方法
上电后 15 后错误指示灯常亮	按键端口接地	检查按键端口对地是否电压为 0V 排除故障后重启设备。
上电后错误指示灯无规律闪烁	环境光过强	遮挡检测面，避免过量的光线
校准时错误指示灯有规律闪烁	环境光过强	遮挡检测面，避免过量的光线

注意：环境光过强不会影响传感器使用，但可能出现误检测。在校准时遇到环境光过强，设备将终止校准，并滚回之前的校准值。

2.5.2 高阶排错

故障现象	可能原因	解决方法
传感器除电源灯外无反应	1、外接线路故障	初次拿到传感器时，请勿在并行数据端口安装线路，避免外置线路影响数据指示灯的正常工作。
传感器除电源灯外无反应	2、PULL 档安装有跳线帽	初次拿到传感器时，请勿任何插入跳线帽，如果 PULL 有跳线帽，并行数据端口进入开漏状态，此时端口无电压,数据指示灯无任何反应。
检测面反光导致的误判	黑场反光错认为是白场	安装时稍微倾斜一个小角度，避免直对
数据指示灯不亮	PULL 档安装有跳线帽	如果您是 5V 主控，请移除跳线帽；如果您是 3.3V 主控，请在程序中将 GPIO 设置为上拉，或外置上拉电阻。
用户主控芯片被烧毁	数据端口电压不兼容	3.3V 主控请安装跳线帽至 PULL 档，随后在主控程序中将 GPIO 设置为上拉模式
用 I^2C 无法通讯	地址错误	请详细阅读数据手册中的 I^2C 部分，做相应的地址偏移。
地址正确 I^2C 却无法通讯	总线无上拉	请外置 10K 上拉电阻到 I^2C 总线上。
总线也正确 I^2C 却无法通讯	用户主控 GPIO 电压过低	请尝试用传感器的上拉源上拉，将跳线帽插入 SDA 和 SCL 档，即可完成配置。

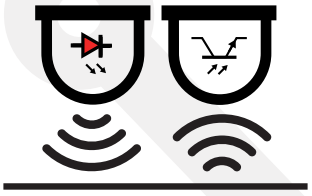
3 更换光电对管

未完待续》》》》》》

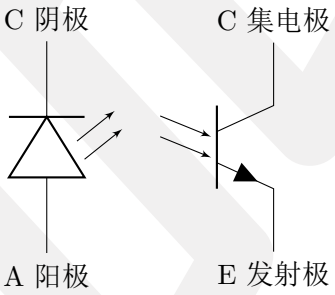
4 灰度测量理论

4.1 灰度测量的原理

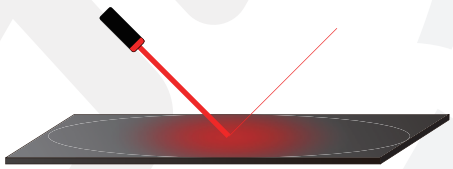
说起灰度传感器不得不提及光电对管，因为它是灰度传感器的重要部件，没有一个稳定的光电对管，无论算法有多先进都是不能实现的。



这是一个光电对管的简易示意图，LED 发出一束光，被检测面反射后，照射到光电三极管中。光电三极管是一个对光线敏感的器件，进光量越多，经过它的电流也就越大。当一束光照射到检测面后，一部分被检测面吸收，一部分反射到光电三极管中，于是产生了电流值。



检测面的灰度越深对于光的吸收越强，而反射的光线也会更弱，能被三极管捕捉的光先也就更少了。于是我们可以的出结论：不同的检测面灰度，产生了不同的电流值。



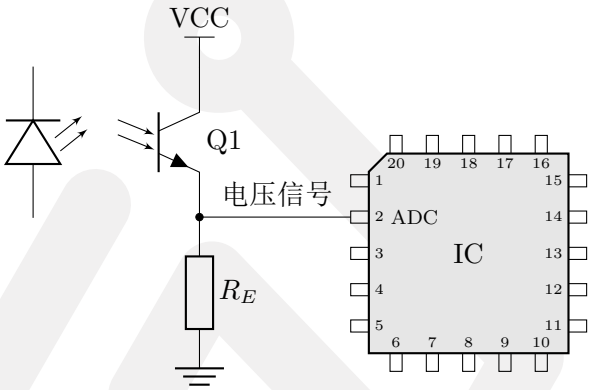
在检测过程中，能被三极管检测到的不是只有 LED 发出的光线，还有环境当中的复杂光线等很多种干扰，所以我们还要对电流信号进行信号处理。在市面上的产品当中，信号处理方式各有不同，对应的价格各不相同，可以简单的归类为三种信号处理方式：直出信号不处理、比较器降敏处理、抗干扰滤波处理。

- 直出信号不处理：直出信号是指直接将三极管的电流信号通过欧姆定律直出为电压信号输出，这种方式确实具有较高的灵活性，用户可以自己进行信号处理，但缺点是不做任何信号处理，使用难度大，容易受干扰。
- 比较器降敏处理：这种方法也不会对干扰信号进行处理，更像是一种小猫靠埋屎除臭的一种方法。三极管的电流信号转为电压信号后，经过一个电压比较器，将电压信号与阈值电压进行对比，从而输出一个高低电平数据，也称为状态信号。它是通过有损压缩信号的方式，将有用信号与干扰信号同时压缩，章节4.1.2中将会详细讲述。
- 抗干扰滤波处理：这种方法是通过算法将干扰信号剔除。这种方法文字叙述最少，工程量与复杂程度巨大，要考虑诸多的干扰因素，逐一剔除。

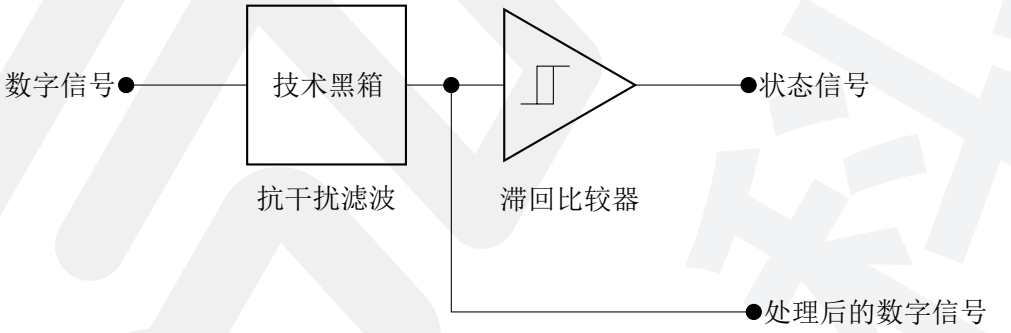
4.1.1 感为信号处理方案

感为灰度传感器的信号处理方案是第三种，抗干扰滤波处理，让我们先从电流信号说起。

光电三极管中的电流信号利用欧姆定律被该电路被转化为一个电压信号，电压信号经 ADC 捕获转化为数字信号存储在芯片的内存里，由此我们完成了灰度值—电流信号—电压信号—数字信号的转变。



当芯片接收到数字信号，经过技术黑箱[1]滤除干扰值后，输出到滞回比较器中，滞回比较器能将数字数据数据与内置的灰度值进行比较，最终输出一个状态信号。该状态信号体现为高低电平，他们分别代表着，此时检测面的黑、白状态。滞回比较器的相关内容请参考章节4.4



除此之外，处理后的数字信号也能被用户读取，拿到数据后，用户可以对数据进行二次加工，例如寻迹时，可以用模拟信号评估黑线的位置，从而引入对 PID 寻迹的支持。当然这部分需要用户来发散思维，创造使用场景，这里就不过多赘述了。

对于技术黑箱的细节，由于知识产权的原因，未能公开，其中包含了对环境光的滤波、日光的滤波、对闪光的滤波、对变色光的滤波等等。

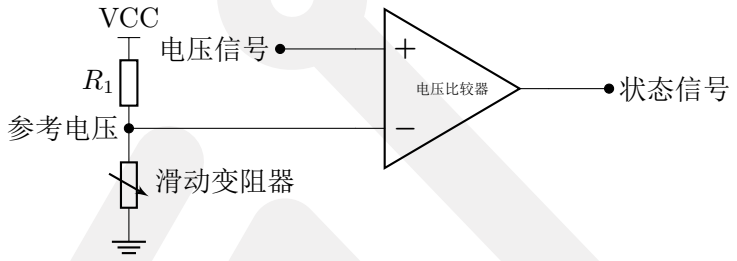
在产品持续更新的 4 年里，积累了相当多的工程经验与实例。该技术黑箱一度将 STM8 的性能跑满，可见其中的复杂程度，最后通过算法优化，分时复用等的方案得以使 STM8 保留了下来，从而降低了产品价格。因为增加算力会大大增加产品的价格，所以不得不在这方面下了很大功夫。

[1]技术黑箱：经过技术黑箱，输入的变化能引起输出的变化，但不必研究其内部结构。内部为全黑状态，使用者仅需考虑其输入、输出数据。

4.1.2 传统信号处理方案

传统的灰度传感器由于成本的原因，并不会应用单片机芯片，因此也不需要将电压信号转为数字信号，而是采用第二种方案：比较器降敏处理。

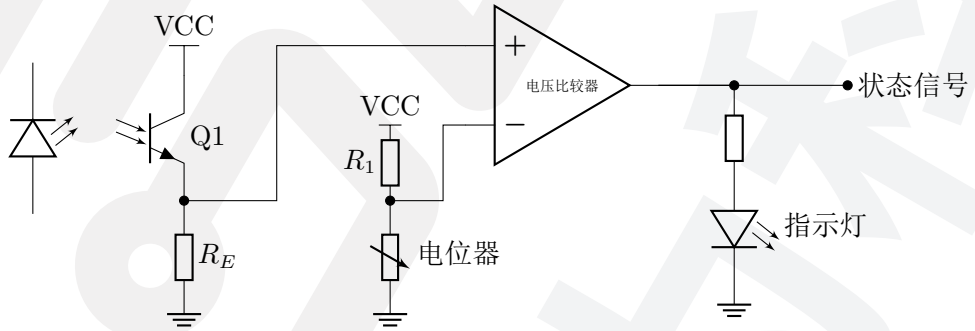
比较器可以理解为将数据有损压缩，将电压信号与参考电压进行比较，从而输出一个状态信号即高低电平。



电压比较器运行原理比较简单，当“+”引脚的电压高于“-”引脚的电压时，输出为高电平；当“+”引脚的电压低于“-”引脚的电压时，输出为低电平。传统灰度传感器将光电对管产生的电压信号直接连接至“+”引脚，而“-”引脚连接至一个电阻分压器[2], 通过两个引脚的电压，产生状态信号。像这种比较器我们称之为单限比较器，对于单限比较器的内容请参考章节4.4

4.1.3 * 传统传感器得校准方法 (拓展)

我们在调试这类传感器时，往往是通过旋转电位器的旋钮，同时观察与电压比较器输出相连的指示灯。将传感器置于白色检测面下，转动电位器，使输出指示灯亮起，即为调试完毕。



实际上这样调试是不稳定的，此时电压比较器“-”引脚电压恰好等于“白色”电压，而非“灰色”电压。尽管“白色”电压也可以正常使用，但是如果白色检测面泛黄或者其他因素导致变暗，则灰度传感器将失灵。

正确的做法是，将灰度传感器置于白色检测面之上，用万用表测量其“白色”电压；再将灰度传感器置于黑色检测面上，同样测量出“黑色”电压。然后将两个数值相加除二，即可得到“灰色”电压，此时我们用电位器将电压旋至该值即可。

$$U_{gray} = \frac{U_{black} + U_{white}}{2}$$

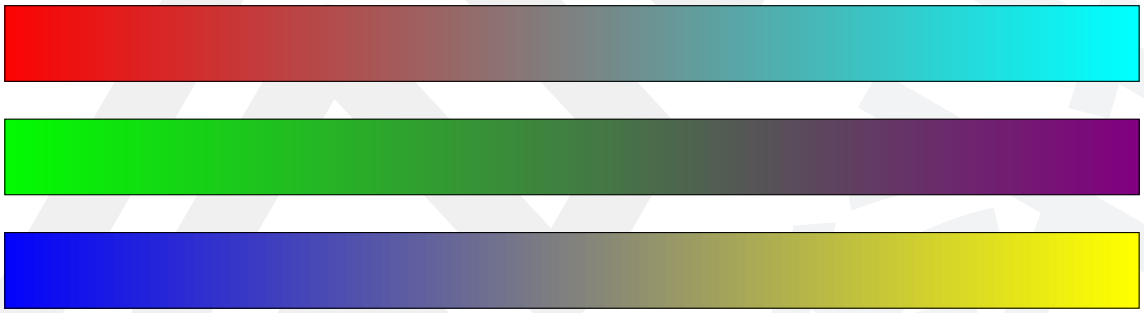
[2]电阻分压器由两个电阻串联组成，调动下面的滑动变阻器，可以改变其分压器的电压。

4.2 传感器的颜色理论

我们通常理解上讲，灰度是一个在白黑之间的数据，是一种不包含色彩的灰度信息。但实际上在灰度传感器当中，灰度只是一个电信号，可以是任意颜色，究竟是怎么怎么回事呢？

灰度传感器中，能感受到的只是电压信号值，而不是人类理解的颜色，这个电压信号仅与三极管接收光线多少有关。在章节4.1的理论中最关键的是检测面对 LED 光的吸收，如果想检测到某一种颜色，只需要保证检测面的颜色与 LED 光线的颜色匹配，即可将大多数光线反射到光电三极管中，从而形成与白色相同的电信号。

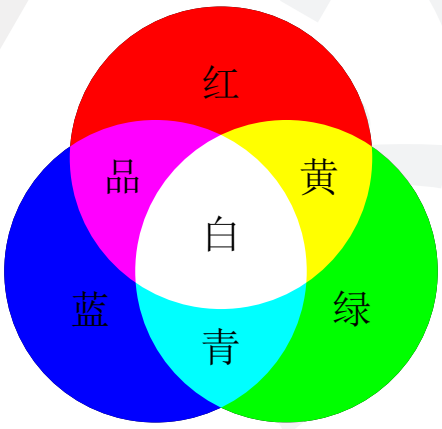
正色的补色被正色的 LED 光照射后，所产生的电信号和黑色一样，因为补色是除正色之外的所有颜色的集合，相当于白色中抠除正色后剩下的部分。例如红色的补色是青色，青色的 RGB 数值是 (0%,100%,100%)，红色的 RGB 数值是 (100%,0%,0%)，两种颜色互为相反。如果用红光的 LED 照射两种颜色的话，青色因为不包含红色，所以红光被吸收，电信号就为黑色；而红色因为可以反射大部分红光，所以电信号显示为是白色。不难发现，我们利用原色和补色的对立性，顺利的将灰度与色彩产生了联系，仅需要改变 LED 的颜色，即可对彩色识别。



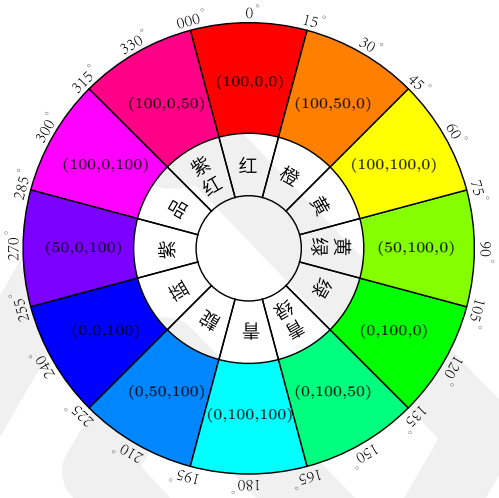
上图为红绿蓝三色与补色的渐变图，实际上这种表示方式是不够准确的一维的图像不足以覆盖更多的颜色，仅因为它与黑白表达类似，能方便理解从而被采用。实际使用中，色彩的灰度远比图中的色彩丰富的多。我们必须要引入更加丰富的二维图像——色相环。

4.2.1 色相环理论

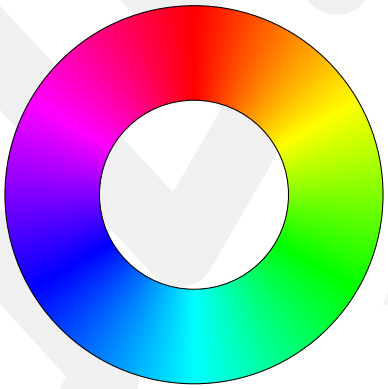
色相环是由红、绿、蓝三原色经过多次融合而形成的环状色相图。首先红 (100,0,0)、绿 (0,100,0)、蓝 (0,0,100) 三原色相互融合生成黄 (100,100,0)、青 (0,100,100)、品 (100,0,100)，这三种颜色称之为三间色。



再将三原色与三间色相邻相容，形成二次间色，橙 (100,50,0)、黄绿 (50,100,0)、青绿 (0,100,50)、靛 (0,50,100)、紫 (50,0,100)、紫红 (100,0,50)。例如红 (100,0,0) 与黄 (100,100,0) 相容成为橙 (100,50,0)；青 (0,100,100) 与蓝 (0,0,100) 相容成为靛 (0,50,100)。



以此类推相邻相容，无限细分，就形成了标准的 RGB 色相环，但通常上，12 个色相就足够满足工程上的需要，无需再度细分。



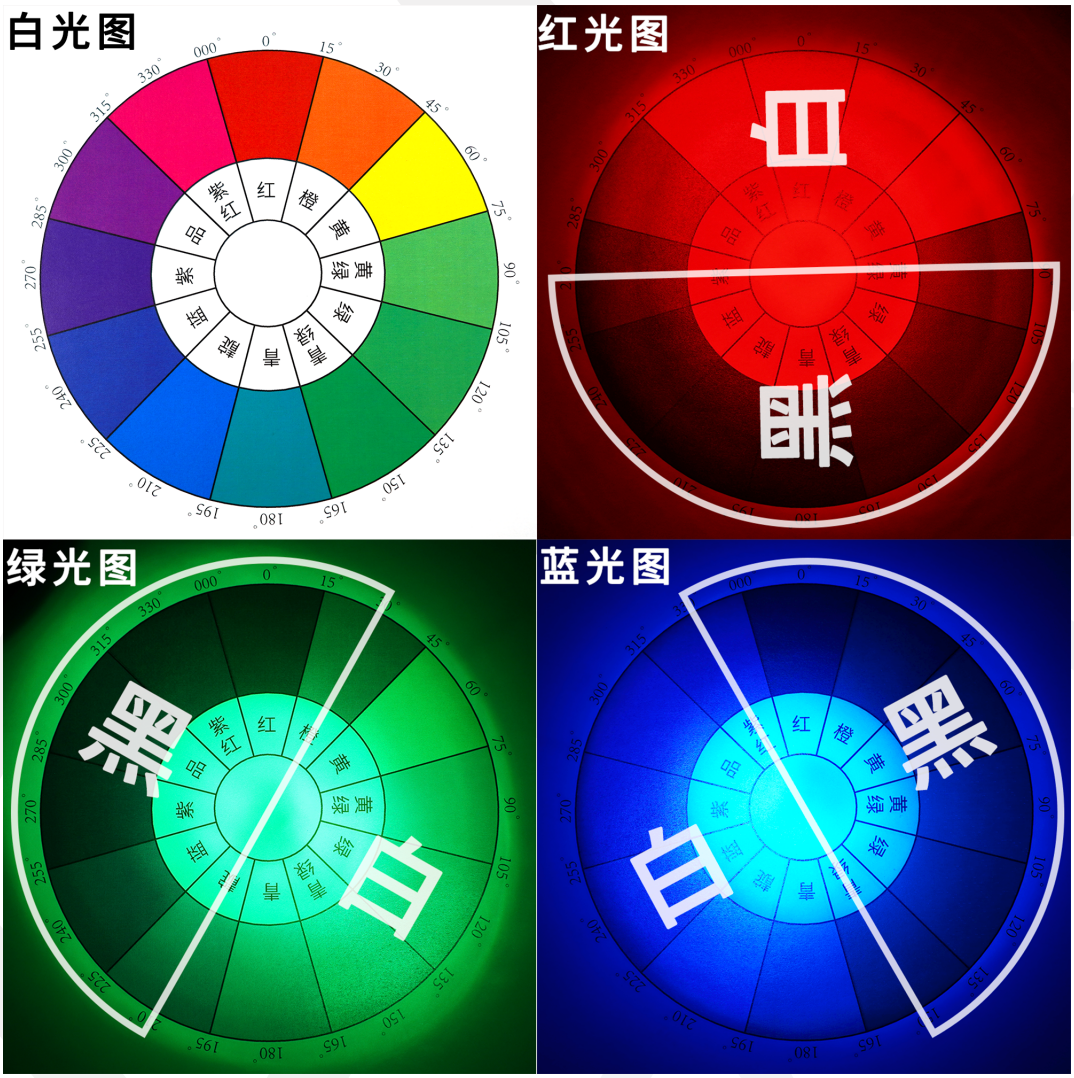
在 RGB 三原色的体系下，色相环中每一种颜色都能由三原色混合而成，就像此时你看的屏幕一样，是由无数的红绿蓝发光单元组合而成。每一种原色都可以认为是一个通道，分别有红、绿、蓝三种，例如橙色 (100,50,0)，意义是红色通道透明度 100%；绿色通道透明度 50%；蓝色通道透明度 0%。将三个原色叠加一起，就得到了橙色。

此时如果我们将色相环的通道分别单独提取出来，由于丢失部分通道数据，失去了作为色彩的意义，就成为了黑白图像，而这个图像就是我们熟悉的灰度图像。

4.2.2 灰色的色相环

为什么要介绍色环与通道呢？因为我们真的能实现检测面的通道提取。其要点是在于光电对管的 LED 发光颜色，假如 LED 发射红光，那么经检测面反射的只有红光，于是绿色或者蓝色的检测面则会吸收红光，所以呈现黑色。

用这种方法，我们可以将所有红色通道提取出来，转成数字信号后进行分析，于是就得到了单通道的灰度，我们在实验中也印证了此结论。

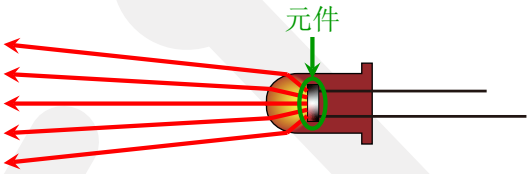


未完待续》》》

4.3 检测距离对检测值的影响

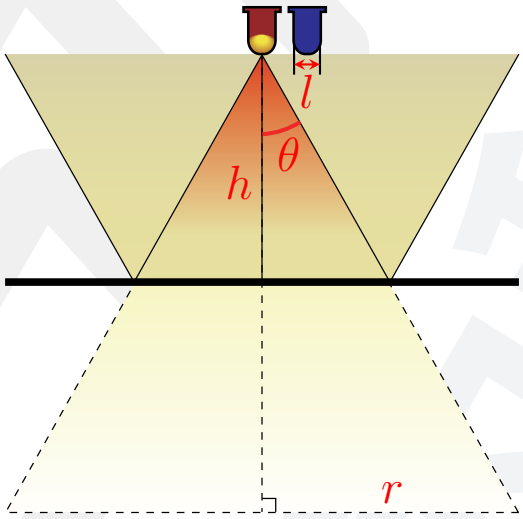
4.3.1 理想模型

要想研究检测距离对检测值的影响，必须了解光电对管的内部构成。其实不管是光电对管的发射管还是接收管，他们内部的发光或受光元件体积很小，且光路都是向外发散的。它们必须经过一个透镜，才能使用。经过透镜后，光路被束缚在一个较小的范围内，这个范围角度我们称之为光角。



加上透镜后，发射管能“照”的更远、“照”的更亮；接收管则在沿光路的方向上受光面积增大，而超出光角范围受光面积减小，从而能在特定的方向上捕捉更多的光线。这就像戴着老花镜的老奶奶，戴上老花眼镜后眼睛会变大一样。

为了进一步研究光在反射后的效果，我们假设检测面是光滑且能全反射的镜面。我们可以把光路的图像[3]绘制为如下所示。其中虚线部分为镜子反射的虚像，实线部分为实像，h 为对管距检测面的高度；r 为光路到达接收管时，光角扫过长度的一半；θ 为光角的一半，l 为接收元件经过透镜放大后的最大直径，即为透镜的直径。



假设反射的光线是均匀的，如图可知，如果 2 倍的 r 是发射光线的总和，则 l 可代表接收光线的总和。将 $\frac{l}{2r}$ 称之为相对进光量，即接收光/发射光，为一个百分数，比值接近 100% 时，证明发射的光线被全部接收。

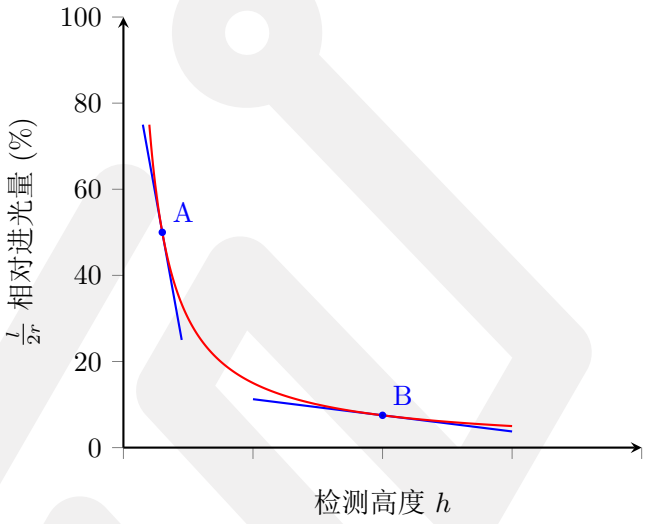
$$\begin{aligned}\frac{l}{2r} &= \frac{l}{2\times(2\times h\times\tan\theta)}\\ &= \frac{l}{4h\tan\theta}\\ &= \frac{l}{4\tan\theta}\times\frac{1}{h}\end{aligned}$$

[3]这是在二维平面上的推导过程，在三维空间中，光角扫过的是一个圆锥体，在检测面上的投影是一个圆形，但这并不影响推导结果，差异被常数 c 吸收，趋势未发生变化，有兴趣的朋友可以自行推导。

其中 $\frac{l}{4\tan\theta}$ 中的 l、θ 为已知常量，可以将整体看做一个常数 c，则：

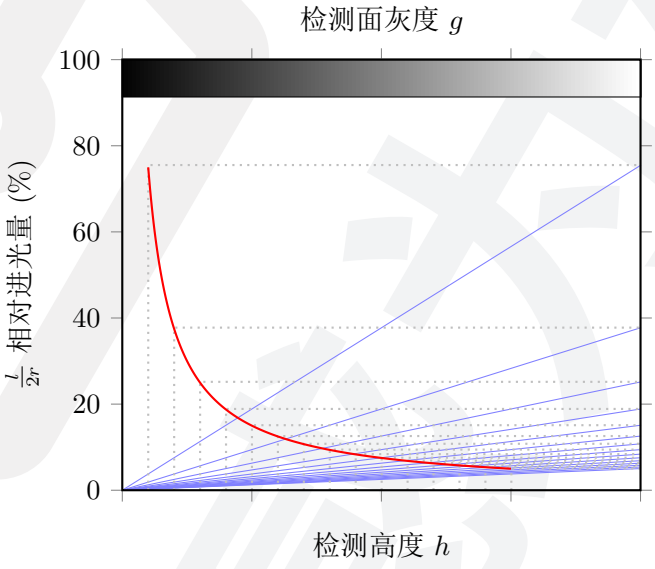
$$\frac{l}{2r} = \frac{c}{h}$$

相对进光量 $\frac{l}{2r}$ 与距检测面高度 h 成反比例，其图像为：



图中标记了在不同检测高度 h 下的斜率。可以看到，在不同高度下，曲线的切线斜率不同的。这意味着在不同的检测高度下，测量稳定性有所差异。

例如检测高度 h 置于 A 点，当检测高度 h 发生了微小的波动，由于 A 点的切线斜率过大，相当于给波动乘以一个十分大的系数，使得相对进光量波动巨大，从而造成灰度值的检测错误。当检测高度 h 置于 B 点，即使检测高度发生了一些波动，由于切线斜率较小，使得波动被削弱，从而减小了相对进光量的波动，增加了灰度传感器的稳定性。

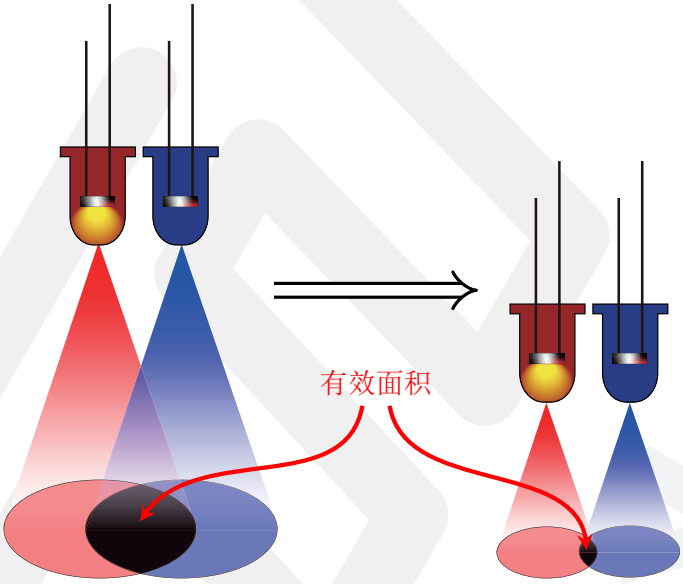


上面的图像中，有 3 个坐标轴，分别为检测高度 h、检测面灰度 g、相对进光量 $\frac{l}{2r}$。红色代表着相对进光量 $\frac{l}{2r}$ 与检测高度 h 的曲线，而蓝色则代表着在不同检测高度 h 下的，相对进光量 $\frac{l}{2r}$ 与检测面灰度 g 的曲线。

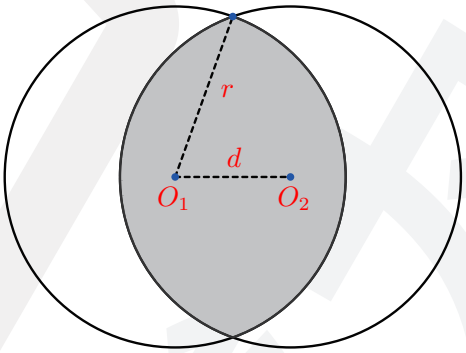
由图中可以看出，随着检测高度 h 的增高，传感器对检测面灰度 g 的灵敏度相应降低，灵敏度体现在蓝色直线的斜率上，在同一检测面灰度 g 的变化下，斜率越大，进光量 $\frac{l}{2r}$ 的变化越大。故检测高度 h 越低，传感器的灵敏度越高。

4.3.2 实际模型

实际上当检测高度 h 低到一个阈值时，灰度的灵敏度、稳定性反而会双双降低，其原因如图所示，红色是发射管，蓝色是接收管。由于光路可逆原理，我们将接收管也表达为发射光束，以便分析。



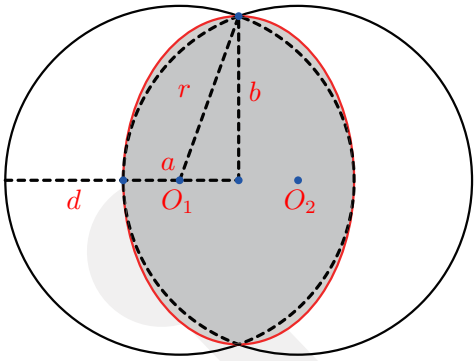
当光电对管距检测面的距离降低时，其有效面积实际上是减小的，因为接收管与发射管在水平方向上有一些距离，当检测距离拉近的时候，投影的半径会减小，但管间距不变，从上图可以看出，有效面积实际上是减小的。究竟有效面积是如何变化的呢，我们可以通过几何运算得出，其示意图如图所示：



其中，d 为发射管与接收管的水平距离，该参数不会发生任何变化；r 为投影圆的半径，该参数为变量，随着检测高度 h 变化而产生变化。

要想计算出阴影面积，有很多种方法可以精确计算出，但演算过程较为复杂。如果我们的目的仅为分析趋势，可以采用一种近似法，简单的求出一个近似解。

我们可以将阴影部分的面积看似一个椭圆形，椭圆形的面积公式为 $s = \pi ab$，其中 a、b 分别为椭圆的短半轴与长半轴。这样我们就可以将两个弓形面积简化为一个椭圆的面积。



可以看到在该图中，d 的位置发生了变化，这是由于两个半径相等的圆，在水平方向上，两边缘的距离与圆心的距离相等。如果我们将圆的直径 $2r$ 减去 d，就可以得到短轴的长度。则短半轴长度 a 的公式为：

$$a = \frac{2r - d}{2}$$

又由于 O_1 至 O_2 的长度为 d，已知半径为 r，b 可以用勾股定理计算。则长半轴长度 b 的公式：

$$b = \sqrt{r^2 - (\frac{d}{2})^2}$$

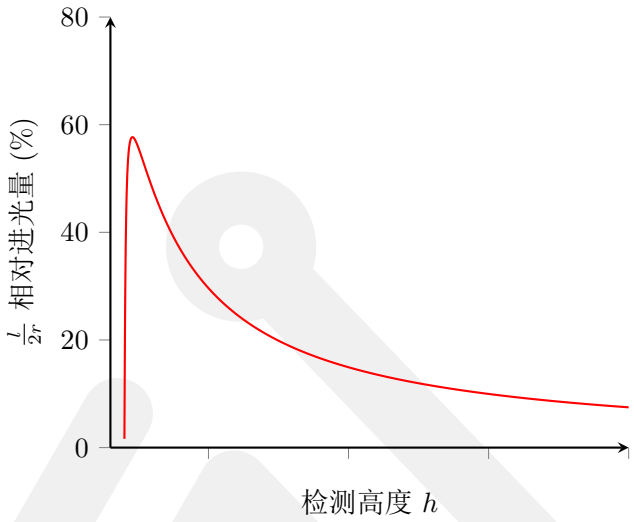
分析这个问题仅计算阴影面积是没有意义的，我们必须得到一个参数—有效面积率 ρ，即计算出阴影面积占圆面积的比值，这样我们用这个比值乘以理想模型中的函数，就可以直观看到实际模型的曲线。有效面积率 ρ 为:

$$\begin{aligned}\rho = \frac{S}{S_O} &= \frac{\pi ab}{\pi r^2} \\ &= \frac{\frac{2r-d}{2}\sqrt{r^2 - (\frac{d}{2})^2}}{r^2} \\ &= \frac{(2r - d)\sqrt{r^2 - (\frac{d}{2})^2}}{2r^2}\end{aligned}$$

由理想模型可知，$r = 2h\tan\theta$，则有：

$$\begin{aligned}\rho &= \frac{(2r - d)\sqrt{r^2 - (\frac{d}{2})^2}}{2r^2} \\ &= \frac{(2(2h\tan\theta) - d)\sqrt{(2h\tan\theta)^2 - (\frac{d}{2})^2}}{2(2\tan\theta h)^2} \\ &= \frac{(4\tan\theta h - d)\sqrt{4\tan\theta^2 h^2 - (\frac{d}{2})^2}}{4\tan\theta^2 h^2}\end{aligned}$$

最终将理想模型乘上有效面积率 ρ，即可得到实际中的模型，其图像为：



可以看到，当检测距离较近时，曲线斜率很大，测量十分不稳定，且灵敏度也会随着距离的拉近变得更差。所以说尽管降低检测距离能带来高的敏感度，我们也不应该让传感器运行在该工况下。

4.3.3 结论

综上所述，检测高度对检测值影响的结论为：

- 检测高度 h 越低，对于灰度的检测越敏感，但是如果传感器使用工况存在颠簸的情况，那么传感器极易受路面影响而发生误判。
- 检测高度 h 越高，对于灰度检测越稳定，但是如果距离太远，由于传感器灵敏度过低，极易出现判别不清，从而失灵的状况。
- 检测高度 h 过低，对于灰度检测是破坏性的，检测极不稳定且灵敏度会变低，我们应该极力避免传感器运行在该工况下。
- 检测高度 h 应当设置在一个合理的点上，做到既稳定又敏感。可以参考章节1.1
- 示例中是以镜面为检测面，在实际使用中，检测面的类别、材质、颜色各不相同，远比镜面复杂的多，但趋势不会发生变化。

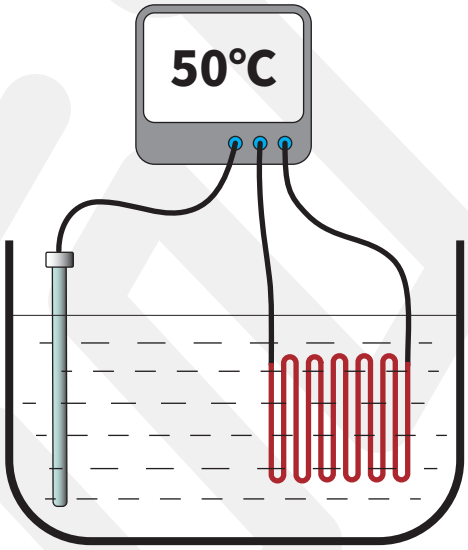
我们在使用传感器时，如果出现问题应该按照以上结论自行判断。

4.4 * 滞回比较器 (拓展)

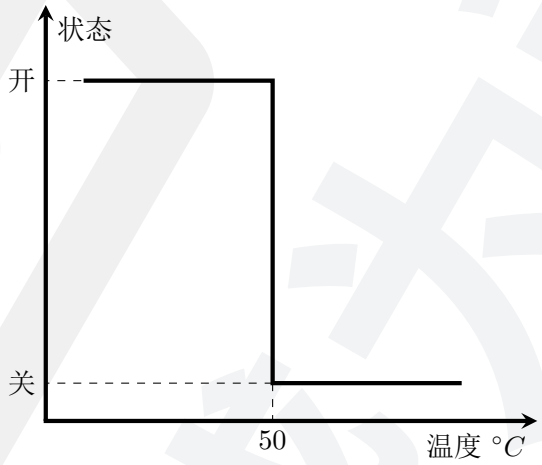
要想由浅入深的了解滞回比较器，我们不得不提及单限比较器，单限比较器一般只有一个阈值[4]，在阈值左右有两种不同的状态。

4.4.1 单限比较器

为了解释单限比较器，我们以水温控制器为例说明。



如果我们将水温用单限比较器控制在 50℃ 左右，那么如果测温棒测得水温低于 50℃ 水时，控制器就打开加热器给水加温，如果高于 50℃ 则关闭加热器停止加温，这样最终温度会稳定在 50℃ 左右。这是单限比较器一个典型的例子，其对应的特性为：

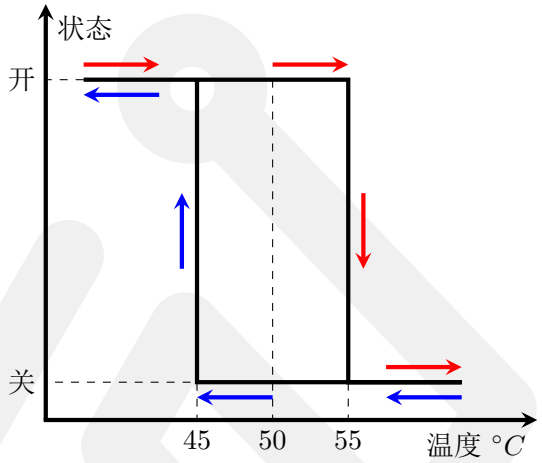


单限比较器比较灵敏、使用简单，但其抗干扰能力差，当温度在 50℃ 附近时，任何的微小变化，都将引起开关状态的跃变，不论是水温变化还是外部干扰。具体表现为：控制器将不停的在开、关两种状态间跳跃，导致设备的寿命降低。

[4]阈值：又称临界值，事物在两种不同状态之间的临界点，是事物状态发生跳变的分水岭

4.4.2 滞回比较器

滞回比较器因此应运而生，解决了这个问题，因为滞回比较器具有滞回特性，即具有惯性，因而具有一定的抗干扰能力。滞回比较器如下图所示：



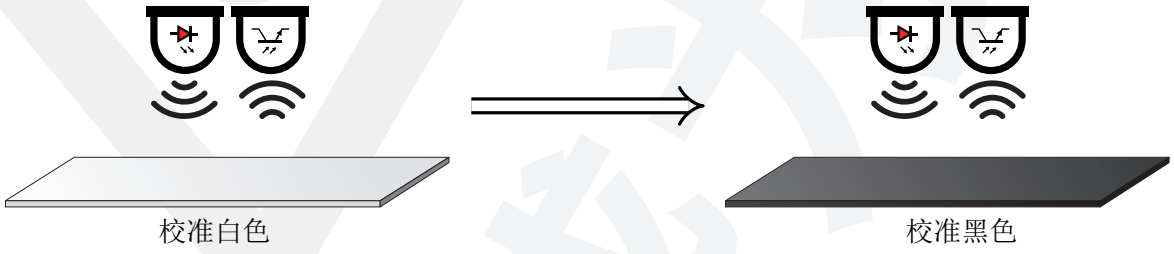
当水温低于 45℃ 时，打开加热器，提高水温，直至温度高于 55℃ 时，关闭加热器，停止加热，如红色箭头所示；当水温高于 55℃ 时，关闭加热器，降低水温，直至温度低于 45℃ 时，打开加热器，开始加热，如蓝色箭头所示。

换言之，由于滞回比较器的惯性特征，当水温加热至 45℃ 阈值时，滞回比较器会让水温再加热到 55℃ 后才停止通电。同理，当水温降至 55℃ 后，由于惯性特征，滞回比较器会等待水温低于 45℃ 才打开加热器。

这样一个循环，可以将水温保持在 45 ～ 55℃ 之间，滞回比较器在任何温度下均不会发生抖动，可以说滞回比较器具有了抗干扰特性。

4.4.3 灰度传感器中的滞回比较器

在以上例子中，我们有两个阈值参数，一个是 45℃ 阈值，一个是 55℃ 阈值。感为灰度传感器中也有两个阈值参数，分别为灰黑阈值 (GrayB) 与灰白阈值 (GrayW)，该参数是在您用按钮校准后，由设备自动计算完成的。



校准完成后，设备会得到两个值：白色采样值与黑色采样值。它会根据这两个值计算出灰黑值 (GrayB) 与灰白值 (GrayW)，它们分别在黑白之间的 $\frac{1}{3}$ 处、$\frac{2}{3}$，其计算公式为：

$$U_{GrayW} = \frac{U_{black} + 2U_{white}}{3}$$

$$U_{GrayB} = \frac{2U_{black} + U_{white}}{3}$$

5 设备的并行通讯

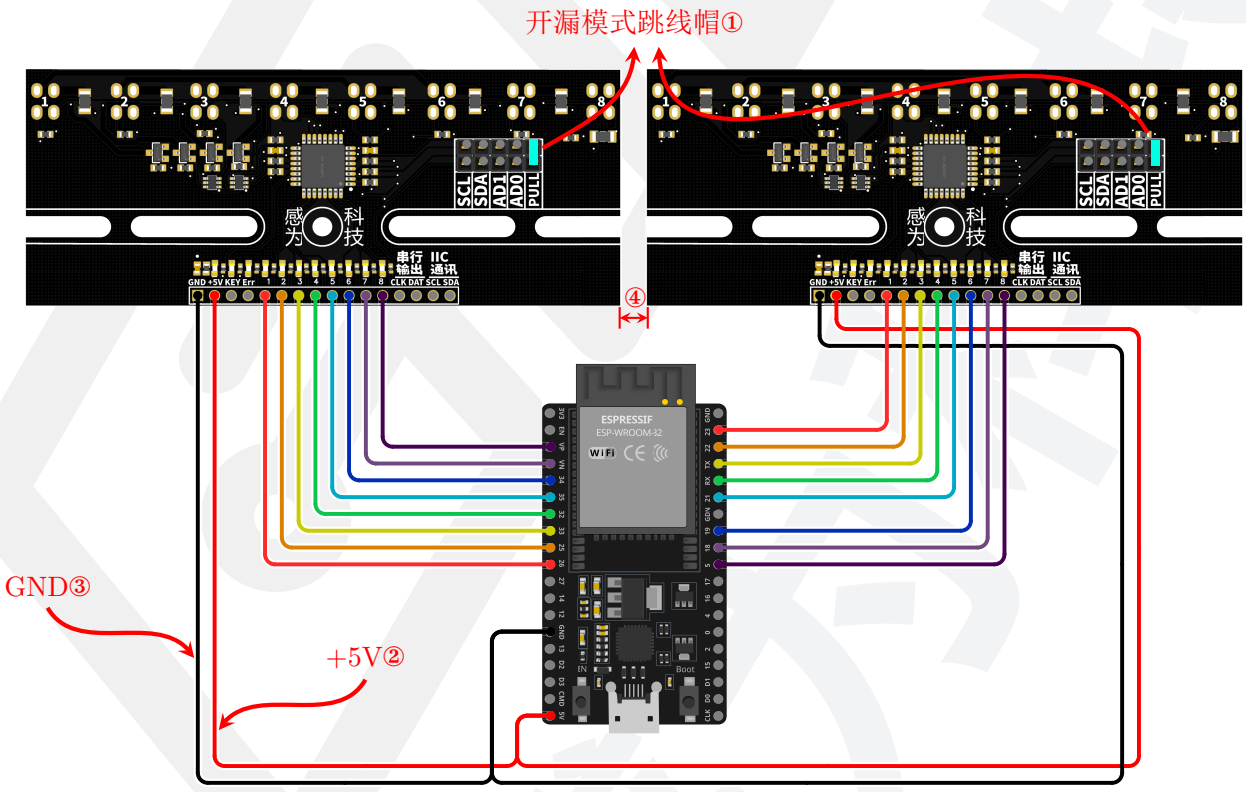
并行通讯是一种最简单的通讯方式，用户仅需要简单掌握单片机的 I/O 功能即可上手操作。在这一章节中，我们将讲述如何使用并行接口。

5.1 描述

在传感器最下排的接口中，有一组从 1~8 标号的接口，这个接口就是并行接口，每一个接口代表一路，例如标号为 4 的接口，其连接的是第 4 路的输出数据。您编程时可以将主控的 I/O 设置为输入模式，与之相连，读取对应 I/O 的电平值，即可完成通讯。

当您的主控运行电压低于 5V 时，常见为 3.3V，此时您需要将灰度传感器的 PULL 跳线帽插接上，再将主控 I/O 设置为上拉输入模式，这个操作，可以有效的保护主控的安全，我们将在章节5.3中详细讲解其中的原理，帮助您理解为什么这个方法可以保护主控的安全。

5.2 设备接线



①： 实例中的 ESP32 运行电压为 3.3V，而灰度传感器供电为 5V，端口电压不匹配，有烧毁 ESP32 的风险。插上跳线帽，设备运行在开漏模式下，此模式下，设备端口悬空，没有电压。此时需要外接上拉电阻，将电压拉至 3.3V，不过 ESP32 有内部上拉源，所以仅需将 GPIO 设置为上拉输入，即可完成配置。注意：安装好跳线帽后，需要断电重启设备，才能进入开漏模式，否则将维持原状。

②： 官方推荐 5V 供电。

③： 灰度传感器要与 ESP32 共地，设备间需要共地才能完成通讯。

④： 安装距离不能太近，因为两个灰度传感器的时钟不同步，如果太近，设备之间的光信号会相互干扰，造成误判。
安装时请遵循公式：$D_l \geqslant 3 \times \tan(15°) \times D_h$，其中 D_l 为对管中心点间距，D_h 为对管顶点距检测面的高度。

5.3 开漏输出

在使用场景中，很常见一种电压不匹配的现象。例如您的主控运行电压为 3.3V，而灰度传感器的运行电压为 5V，如果您将灰度传感器的端口直接连接至您的主控，可能存在主控烧毁的风险。在这里我们提供了一种电压匹配的解决方案——开漏模式。

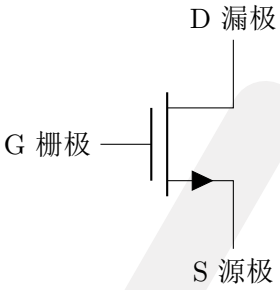


图 1

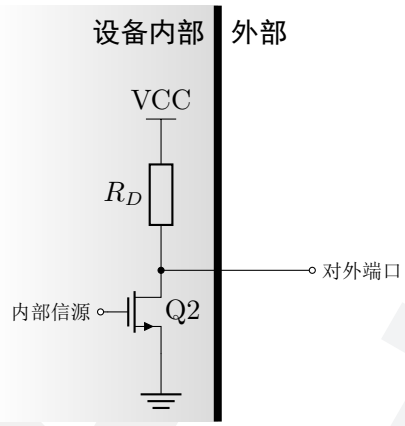


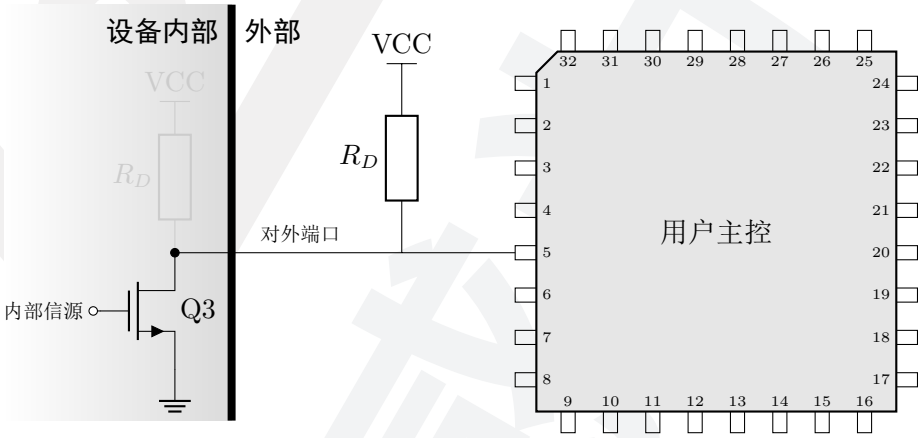
图 2

MOSFET 有 3 个端口，分别为栅极 (Gate)、源极 (Source)、漏极 (Drain)，如图 1 所示，而开漏模式的意思是开放漏极 (Open Drain)。

一般情况下，灰度传感器 (以下称之为设备) 的内部模型[5]可以抽象为共源极上拉输出，如图 2 所示。当设备想要向外输出“1”时，内部信源关闭 MOSFET，此时 MOSFET 断路，对外端口电压为 VCC；当输出“0”时，内部信源打开 MOSFET，此时 MOSFET 短路，对外端口电压值为 GND。由于设备内部的 VCC 直接连至电源，如果设备的供电电压为 3.3V，VCC 则为 3.3V，同理供电电压为 5V，VCC 则为 5V。不难发现，想要更改对外端口高电平的输出电压，只能改变设备的供电电压。

解决“既要设备 5V 供电，又要端口电压匹配”问题的关键点就在于，对外端口的高电平电压与设备的供电电压强绑定且不可更改。

有没有一种可能，我们可以绕开电压强绑定，换个思路来解决这个问题。



我们将上拉电阻 R_D 从设备内部移出到设备外部，这样上拉电压 VCC 就交给了用户配置。我们就可以实现“既要设备 5V 供电，又要端口电压匹配”了。

[5]注：在实际工程应用中，内部结构要比图中模型复杂的多，在这里为了解释方便，仅抽象为一个简单的共源极放大电路。

在这个方案中，设备的对外端口设置为开漏模式，即开放漏极 (Open Drain)。用户只要合理配置上拉电阻、VCC 即可。例如您的主控 I/O 端口的耐受为 3.3V，而灰度传感器官方推荐的供电电压为 5V。这种情况下，您仅需要用跳线帽插入 PULL 口，就可以将设备的开漏模式打开，之后重启设备[6]。配置您主控的 GPIO 为上拉输入模式即可。如果您主控的 GPIO 没有上拉输入功能，您也可以在对应的 I/O 上，加置 10k-60kΩ 的电阻连接至 3.3V 即可。

值得注意的是，通常情况下，这种方案是在解决如何向下兼容用户电压的问题，一般用在用户电压低于灰度传感器电压的情况下。换句话说，图中的 VCC 只能小于 5V，而不得高于 5V。因为对外端口的耐受电压上限是 5V，如果高于这个电压，设备或将永久性损坏。

其次是，在开漏模式下，如果您未接上拉电阻，板载的并口上的 LED 灯是不会亮起的，因为开漏模式下，端口无电压，LED 没有电源供电。

5.4 用法及示例

在并行通讯中，一般有两种用法，分别为：普通模式与开漏模式，其两者的用法不尽相同。唯一的区别是在于初始化阶段，普通模式的 GPIO 需要初始化为悬浮输入；开漏模式则上拉输入模式。

- STM32: 在 STM32CubeMX 工具里，可以通过 UI 界面配置上下拉。也可在 MX_GPIO_Init() 里修改 GPIO_InitStruct.Pull = GPIO_NOPULL;//GPIO_PULLUP
- ESP32:gpio_set_pull_mode(GPIO_PIN_NUMBER, GPIO_PULLUP_ONLY); //GPIO_FLOATING
- Arduino: 在 setup() 里配置上拉 pinMode(GPIO_PIN, INPUT_PULLUP); 或者 pinMode(GPIO_PIN, INPUT);

5.4.1 普通模式示例

以 ESP32-ARDUINO 为例：

```
void setup(){
    //以章节：“设备接线”图为例，PULL不插入跳线帽时为普通模式
    // 配置主控IO为悬浮输入
    pinMode(23, INPUT); //接传感器并口1
    pinMode(22, INPUT); //接传感器并口2
    //...
    pinMode(5, INPUT); //接传感器并口8
}
```

5.4.2 开漏模式示例

以 ESP32-ARDUINO 为例：

```
// 配置函数
void setup(){
    //以章节：“设备接线”图为例，PULL插入跳线帽时为开漏模式，需要用户主控主动上拉。
    // 主控配置IO上拉
    pinMode(23, INPUT_PULLUP); //传感器并口1
    pinMode(22, INPUT_PULLUP); //传感器并口2
    //...
    pinMode(5, INPUT_PULLUP); //传感器并口8
}
```

[6]注意：必须重启设备，因为设备仅在初始化时扫描 PULL 跳线帽

6 设备的串行通讯

上一章中，我们讲述了如何并行通讯，并行通讯的优点是使用简单，简单掌握单片机的 I/O 功能即可上手操作，大大降低了用户的使用难度。但是它有一个缺点，就是浪费 I/O 资源。这个问题其实有很多种解决方案，可以用 I^2C、Uart、SPI 等协议通讯。他们所需 I/O 都不多，可以解决问题，但是它们的使用难度相对较大，有些协议可能需要专业仪器仪表来帮助，这对于大多数人来说，门槛就高多了。

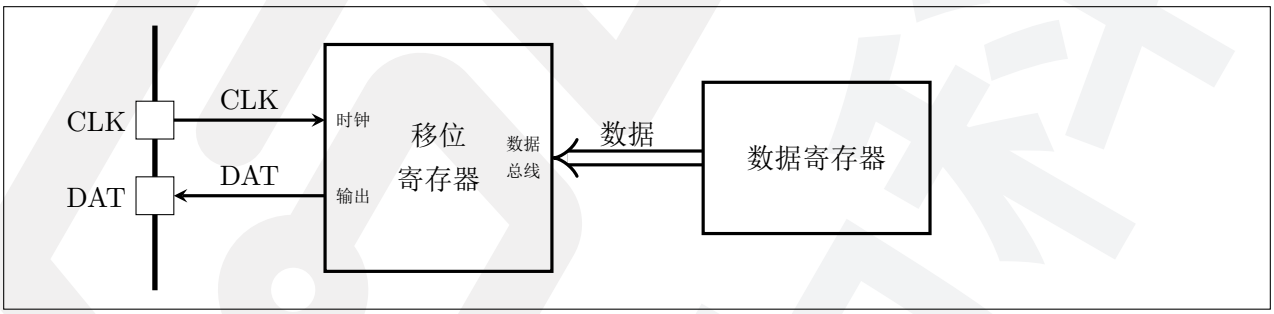
这一章节，我们将讲述一种折中的串行通讯方案，它不需要协议，仅需简单掌握单片机 I/O 功能即可上手操作。灵感来源于 74HC165 移位寄存器的操作逻辑，即每个时钟移一位数据。用户仅需要两根线，即 CLK 时钟线，DAT 数据线，即可读取并行端口的所有数据，简单实用。当然，如果您在一个主控中，不止使用一台灰度传感器，那么也可以让他们共用一条 CLK，如果您的主控连有 n 个灰度传感器，那么您所需 I/O 数为 n+1，相比于并口节约了很多 I/O 资源。

6.1 思路及原理

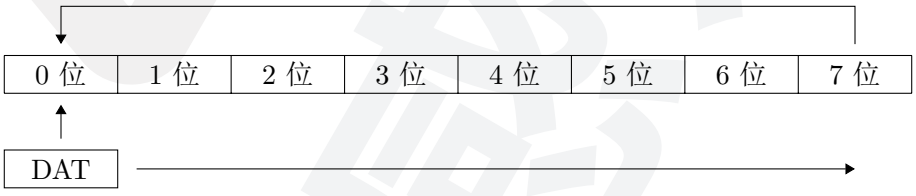
如果想要让用户“仅掌握单片机 I/O 的功能即可上手操作”，那么我们需要从基础的串行理论出发来设计，这样，您仅需要简单了解原理就可以直接应用。接下来我将由浅入深的带您了解我们是如何一步一步设计的，带您全面了解这个功能。

6.1.1 设计思路

我们设计该功能的灵感来自于移位寄存器，其中移位寄存器的左侧为对外连接的是时钟和输出；右侧的是 8 位的并行数据总线。



当开始传输数据时，移位寄存器每接收一个时钟，就会将 8 位数据中的一位输出出去。例如用户发送第一个时钟后，移位寄存器会发送第 0 位数据到 DAT，此时 DAT 的电平等于第 0 位数据，用户读取 DAT 电平后，再发送下一个时钟，移位寄存器会将第 1 位数据发送到 DAT，用户读取 DAT 电平，以此类推。每一个时钟输出一位，直至将 8 个并行数据轮流输出。当移位寄存器收到第 9 个时钟时，移位寄存器循环回第 0 位，然后第 1 位，第 2 位，第 7 位，以此类推，循环往复。



这个方法简单易懂，用户仅需要两条线就可以输出所有的并行数据。但是它有一个致命的缺点：数据帧之间无法解耦，上一帧与下一帧之间有关联。如果数据在传输过程中发生了错位，那么今后所有的数据都将是错的。这就像在抄同学作业，选择题少抄一个，接下来抄的答案全是错的。

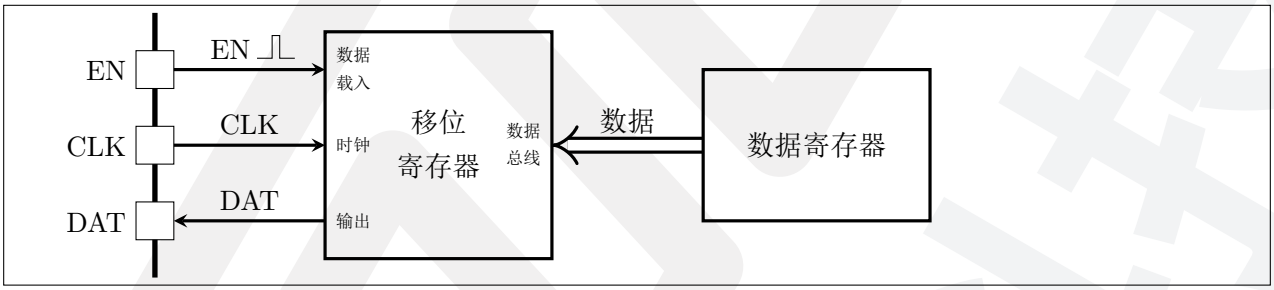
将数据帧解耦实际上不难，加上一个“归零”即可，即每传输完 8 个数据双方都要“归零”。彼此同步一下，就能将数据帧之间的耦合打开。在很多通讯协议中都有类似的方案，例如 I^2C 总线中的 ACK，就是在接收完一帧数据后，回应对方“收到”，这个在章节7中将会讲述。

S	从机地址	ACK	数据	ACK	数据	ACK	P

注意：灰色格子为主机行为，白色格子为从机行为

但是 I^2C 总线中的解耦方式，需要双方交换着操作总线，用起来相对复杂，这背离了“仅需简单掌握单片机 I/O 功能即可上手操作”的设计初衷。我们有没有其他方法可以解耦的呢？我们认为有两种方案比较适合使用：

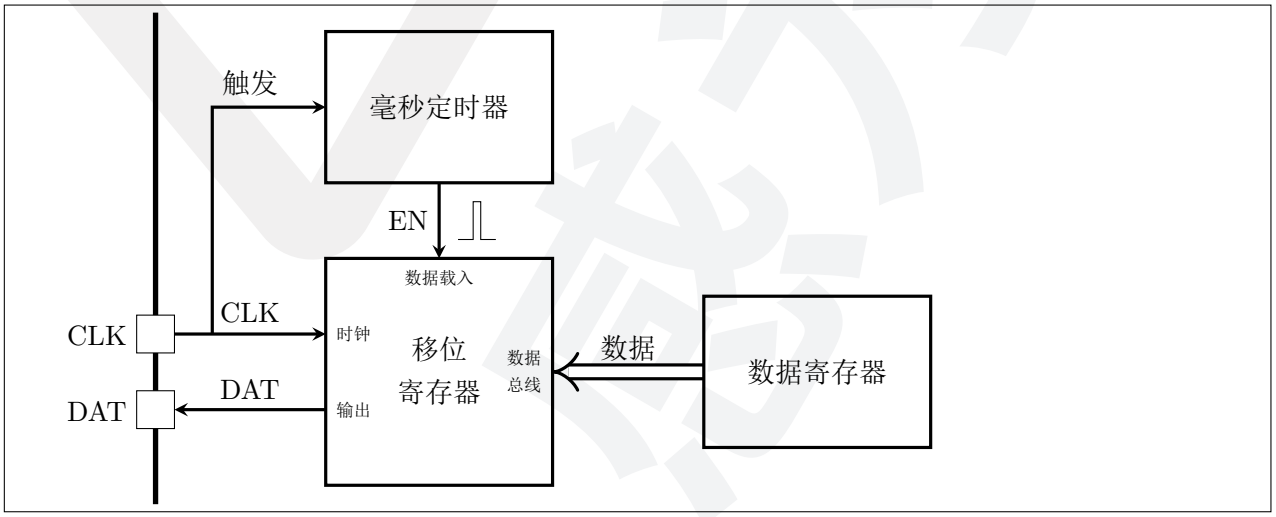
方案一，我们可以给设备加一个新的端口 EN，用于控制总线数据载入开关。这样，在开始通讯时数据被锁存，当传输完一帧数据后，外接主控使能 EN，重新打开数据总线的通道，重新载入数据，移位寄存器会自动地回到第 0 位。这样即便上一帧中，缺失一位数据，第二帧都由 0 位开始传输，不会影响下一帧。这是个可行的方案，但是用户需要用 3 个 I/O 与设备进行通讯，浪费了用户的资源。



方案二，我们不妨换个思路，其实我们需要的是一个触发“归零”的条件, 这个条件未必是一个 I/O，它也可以是时间。当用户不再读取数据一段时间后，将会触发“归零”条件，双方内部都有计时器，用时间进行同步，也不妨是一个两全其美的办法。灰度传感器用的就是这种，接下来详细讲解这种方案。

6.1.2 运行原理

在这个方案中，我们将方案一中的“数据载入”端口连接上了一个毫秒定时器。毫秒定时器只被 CLK 上升沿触发，每次触发时都从零开始计时，如果时间超过 1ms，它将产生一个脉冲到“数据载入”端口，使得总线被重新打开，移位寄存器“归零”，可以达到方案一一样的效果。



您在使用过程中，仅需要将连接至 CLK 的 I/O 设置为输出，DAT 的 I/O 设置为输入，即可完成通讯。对于“归零”的操作，您可以使用简单的 delay_ms(1) 函数产生。如果您觉得 delay_ms(1) 函数损失主控算力，您也可以通过定时器产生，在 1ms 的中断中读取数据，即可完成该功能。

在这个方案中，如果您在一帧数据后“归零”，理论上您能读取的最高速度是 1Kb/s。如果您在多帧后“归零”，理论上您读取速度会更高，但是，并口的刷新速率为 1.5kHZ，如果您通讯的速度大于 1.5Kb/s，理论上是没意义的，即便您读取速度更高，您读到的数据，一部分也是旧的。

值得注意的是，您需要确保您的 delay_ms(1) 尽可能准确、稳定。当然您也没有必要为了提高稳定性而增加延时的时间，因为内部的毫秒计时器实际上在约 0.8ms 左右就已经产生了脉冲。如果您设置了 1ms 的延时，对于灰度传感器来说，不存在竞争与冒险的可能。

您没必要担心延时超过 1ms 后，读到陈旧的脏数据，因为当移位寄存器收到毫秒定时器的脉冲后，数据总线一直是使能的，直到您发送一个时钟到移位寄存器，数据才会被锁存发送，所以您在任何时候，读到的数据均是新的。

移位寄存器在收到第 9 个时钟后，会自动使能总线，更新下一帧数据。假如您一直不“归零”，反复读取数据，理论上是可行的。但延时 1ms 的意义是让通讯双方共同“归零”，以此同步。这能有效防止数据错位的风险发生，为避免数据错误，应避免不“归零”读取数据的操作。

6.2 信号时序

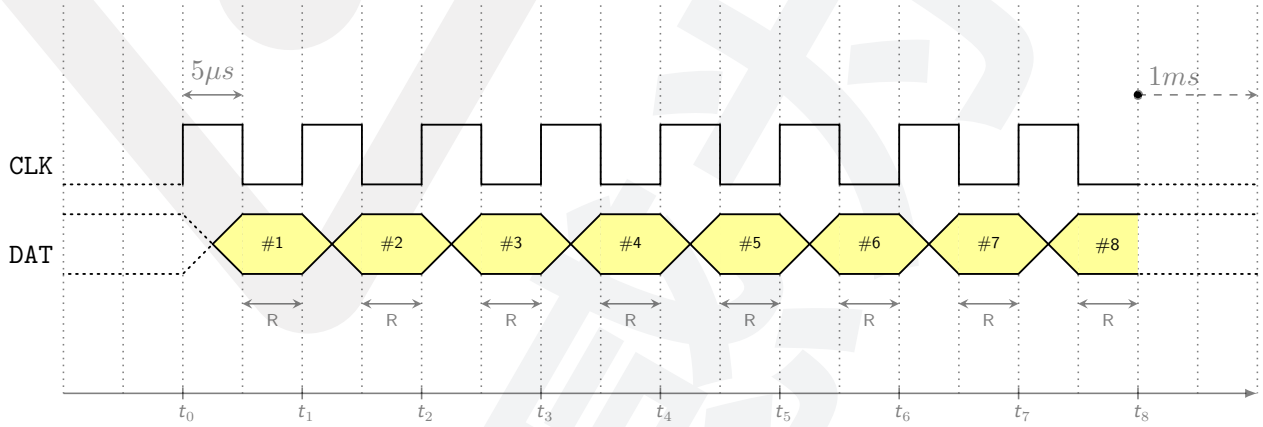
在上一小节6.1.2中，主要讲述了“归零”操作。这一小节中，将讲述如何读取一帧数据，即读取一帧的时序是什么样的。

对于一帧 8 位数据的读取，方法很简单，仅需要注意三点：

第一点，遵循高电平写，低电平读的原则。当 CLK 为高电平时，灰度传感器会将数据写入到 DAT 上；当 CLK 为低电平是，您才能读取 DAT 的电平数据。

第二点，CLK 的高电平一定要维持至少 5μs，因为移位寄存器的各种运算需要时间，如果您抢在移位寄存器工作结束前读取 DAT，将会读到错误的数据。而在 CLK 低电平段，读 DAT 不需要延时，可以在 CLK 拉低后，立刻读取 DAT 的电平。

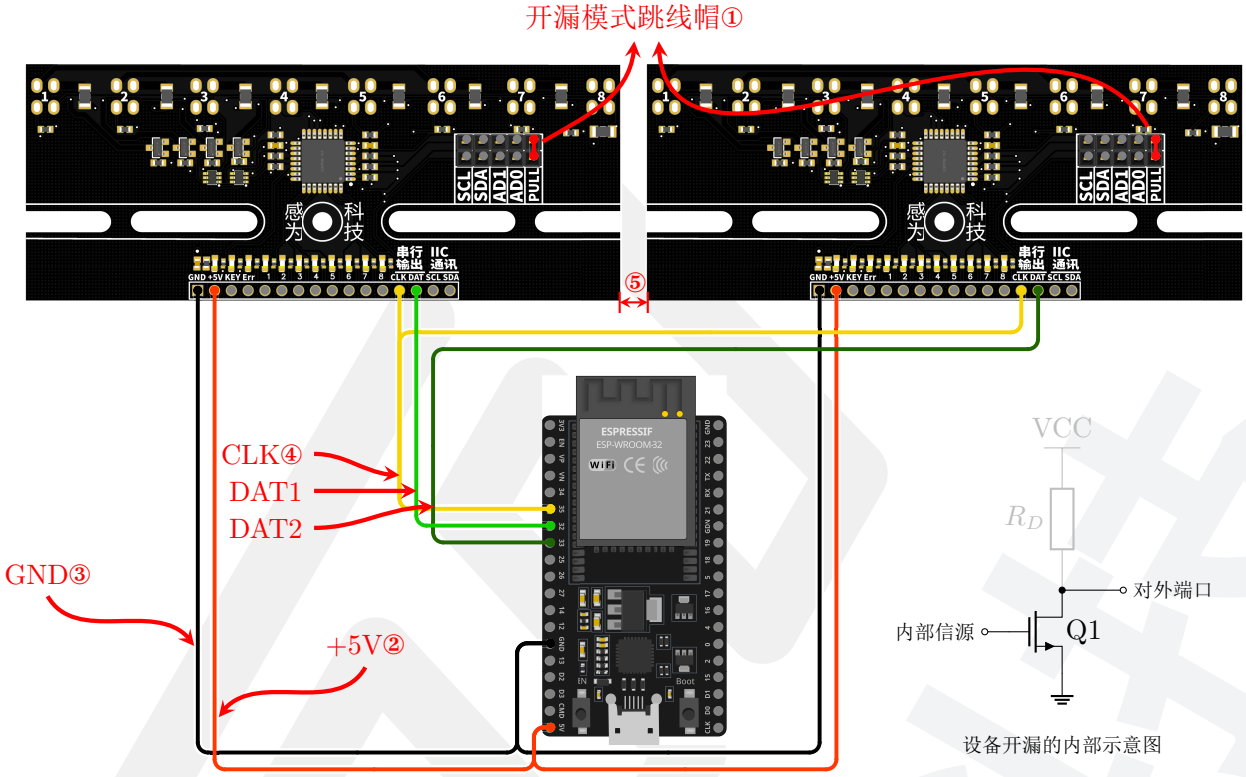
第三点，每个时钟的间隔需要远小于 1ms，如果时钟间隔长于 1ms，内部毫秒计时器就会将数据“归零”，您将只能读到第 1 路的数据，而其他数据将会丢失。



注：图中时序为用户操作时序，与传感器内部的时序略有不同，该图仅供编程参考之用。其差异在于，在传感器内部 1ms 定时是从第 8 帧的时钟上升沿开始算起，而非图中第 8 帧结束开始计时；

R 为读取段，读取时不需要延时，直接读取即可；

6.3 设备接线



设备开漏的内部示意图

①： 实例中的 ESP32 运行电压为 3.3V，而灰度传感器供电为 5V, 端口电压不匹配，有烧毁 ESP32 的风险。插上跳线帽，设备运行在开漏[7]模式下，此模式下，DAT 端口悬空，没有电压，需要外接上拉电阻至 3.3V。而 ESP32 有内部上拉源，仅需将 DAT 的 GPIO 设置为上拉输入，即可完成电压的匹配，CLK 端口无论 5V 与 3.3V，均设置为推挽输出。注意：安装好跳线帽后，需要断电重启设备，才能进入开漏模式，否则将维持原状，具体请参考章节5.3.

②： 官方推荐 5V 供电。

③： 示例中，灰度传感器要与 ESP32 共地，设备间需要共地才能完成通讯。

④： 串行时钟可公用。

⑤： 安装距离不能太近，因为两个灰度传感器的时钟不同步，如果太近，设备之间的光信号会相互干扰，造成误判。
安装时请遵循公式：$D_l \geqslant 3 \times \tan(15°) \times D_h$，其中 D_l 为对管中心点间距，D_h 为对管顶点距检测面的高度。

[7]开漏：含义是场效应管 (MOSFET) 的漏极开放，不接上拉电阻。参阅上图中的“设备开漏的内部示意图”，将 R_D 电阻去掉即为开漏

6.4 用法与例程

由于串行输出通讯是单向的，时钟仅为用户主控发出，所以不论 PULL 跳线帽是否安装，所接 CLK 端口的 GPIO 仅需设置为输出模式。而 DAT 是接收数据，需要电压兼容，如果您设备为 3.3V，则应该安装 PULL 跳线帽，并按照章节5.4中的示例初始化 GPIO。

一、单个传感器通讯

以 ESP32-ARDUINO 为例：

```
#define GW_GRAY_GPIO_CLK 32
#define GW_GRAY_GPIO_DAT 35
/*读取传感器8bit数据*/
static unsigned char gw_gray_serial_read_simple ()
{
    unsigned char ret = 0;
    for (int i = 0; i < 8; ++i) {
        digitalWrite(GW_GRAY_GPIO_CLK, 0);//输出时钟下降沿
        ret |= digitalRead(GW_GRAY_GPIO_DAT) << i; //读取数据并存到ret的第i位bit
        digitalWrite(GW_GRAY_GPIO_CLK, 1);//输出时钟上升沿
        delayMicroseconds(5);
    }
    return ret;
}
void setup() {
    pinMode(GW_GRAY_GPIO_CLK, OUTPUT); //设时钟为输出
    pinMode(GW_GRAY_GPIO_DAT, INPUT_PULLUP); //设数据为输入
    digitalWrite(GW_GRAY_GPIO_CLK, 0);
    //初始化串口，方便查看数据
    Serial.begin (115200);
}
void loop() {
    unsigned char sensor_status = 0;
    //读取传感器串行输出
    sensor_status = gw_gray_serial_read_simple();
    //把读取到的传感器数据打印到公屏上
    for (int i = 0; i < 8; ++i) {
    //获取第N位bit
        Serial.print(( sensor_status >> i) & 0x1);
        Serial.print(" ");
    }
    Serial.println();
    delay(500);
}
```

二、两个传感器通讯

以 ESP32-ARDUINO 为例：

```
#define GW_GRAY_GPIO_CLK 32
#define GW_GRAY_GPIO_DAT1 35
#define GW_GRAY_GPIO_DAT2 34
/*读取传感器8bit数据*/
static void gw_gray_serial_read_double(unsigned char *sensor1_status , unsigned char *sensor2_status)
{
    unsigned char ret1 = 0;
    unsigned char ret2 = 0;
    for (unsigned int i = 0; i < 8; ++i) {
        digitalWrite(GW_GRAY_GPIO_CLK , 0);//输出时钟下降沿
        ret1 |= digitalRead(GW_GRAY_GPIO_DAT1) << i; //读取传感器1
        ret2 |= digitalRead(GW_GRAY_GPIO_DAT2) << i; //读取传感器2
        digitalWrite(GW_GRAY_GPIO_CLK, 1);//输出时钟上升沿
        delayMicroseconds(5);
    }
    *sensor1_status = ret1;
    *sensor2_status = ret2;
}

void setup() {
    pinMode(GW_GRAY_GPIO_CLK, OUTPUT); //设时钟为输出
    pinMode(GW_GRAY_GPIO_DAT1, INPUT); //设传感器1数据为输入
    pinMode(GW_GRAY_GPIO_DAT2, INPUT); //设传感器2数据为输入
    digitalWrite(GW_GRAY_GPIO_CLK, 0);
    Serial.begin (115200);//初始化串口，方便查看数据
}

void loop() {
    unsigned char sensor1_status = 0, sensor2_status = 0;
    //读取传感器1和传感器2串行输出
    gw_gray_serial_read_double (&sensor1_status , &sensor2_status);
    //传感器1数据
    for (unsigned int i = 0; i < 8; ++i) {
        Serial.print(( sensor1_status >> i) & 0x1);//获取第N位bit
        Serial.print(" ");
    }
    Serial.print(" ");
    //传感器2数据
    for (unsigned int i = 0; i < 8; ++i) {
        Serial.print((sensor2_status >> i) & 0x1);//获取第N位bit
        Serial.print(" ");
    }
    Serial.println();
    delay (500);
}
```

7 设备的 I^2C 通讯

7.1 简介

灰度传感器使用 I^2C 总线通讯, 意在节约用户 I/O 资源。I^2C 总线仅需要两个 I/O 即可完成对 127 个设备的通讯。

该设备运行在 7 地址从模式下，支持配置 2 个硬件地址位、5 个软件地址位，全地址可设置，最多可配置 127 个灰度传感器。

该设备可以通过跳线帽自由配置上拉电阻，注意：一般情况下 I^2C 总线应由主设备配置上拉，以兼容主设备电压，所以 SCL、SDA 两个跳线帽通常情况不安装，只有在主设备没有上拉选择时应急使用。

7.2 设备的功能

- 读取数字数据 7.7
- 单通道直接读取 7.8
- 连续多通道读取 7.9
  - 传输通道使能7.9.4
- 读/写滞回比较器参数 7.10
  - 读取滞回比较器参数信息 7.10.2
  - 写入滞回比较器参数信息 7.10.3
- 配置软件地址 7.11
- ping 网络诊断工具 7.13
- 读取错误信息 7.14
- 设备软件复位重启 7.15

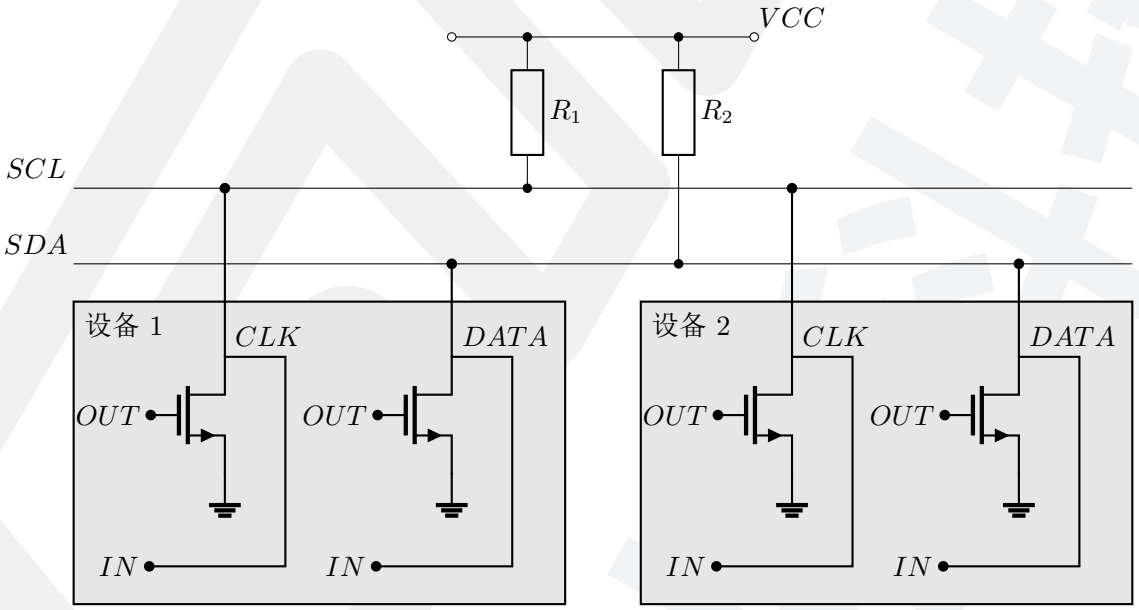
7.3 I^2C 协议回顾

这一章节中，将笼统的讲述 I^2C 的预备知识，为了方便速读或翻阅，我将这一章节的重点画出了出来，供大家参考。如果您对 I^2C 是零基础，阅读这些内容是远远不够的，您还需要自行查阅协议的具体内容，或者去互联网搜索相关的视频教程。

7.3.1 I^2C 硬件

I^2C 总线是 由飞利浦 (Philips) 公司开发的一种半双工[8]总线。它只需要两根线即可在连接于总线上的器件之间传送信息，这两根线分别是 SDA(串行数据线) 和SCL(串行时钟线)。一般情况下，在总线上的设备，I/O 均为开漏[9]状态，需要外接上拉电阻来实现通讯。

为什么是开漏的呢？因为设备间的运行电压可能不同，例如，有些设备是 5V 供电，有些设备是 3.3V 供电，这时候双方 I/O 输出的电压不相同，可能造成通讯失败。而 I/O 开漏仅会对总线进行拉低操作，也就是说逻辑“1”为总线电压，“0”为 0V，电平得到了统一。所以只要调配好总线为 5V，设置好总线的限流电阻 R_1, R_2(一般情况下为 $10k\Omega$)，即使总线电压略有差异，由于总线的总电流仅为 $500\mu A$，也可以安全通讯。



灰度传感器的电路板上有板载的 10K 上拉源，用跳线帽连接SCL 与 SDA 双排针即可完成上拉。注意：上拉电压为 5V，请详细查阅主控的数据手册以确定是否兼容。

7.3.2 I^2C 基础

I^2C 总线是一种半双工通讯协议，同一时间只能由总线中的一个设备发送，为保证双向通讯，I^2C 采用分时复用的原理设计，所以它的通讯协议相较于其他协议，稍稍复杂些。为了准确书写 I^2C 代码，我们需要简单回顾一下。

[8]半双工：数据可以在一个信号载体的两个方向上传输，但是不能同时传输。

[9]开漏：含义是 mosfet 的漏极开放，不接上拉电阻。这里的 mosfet 漏极类似于三极管的集电极，为了方便理解，请查阅“共射放大电路”，将 R_c 电阻去掉即为开集 (mosfet 的开漏)

I^2C 总线的通讯方，分为主设备 (Master) 与从设备 (Slave)，我们在日常应用中，单片机作为一个设备的控制中枢，常设置为主设备，因为主设备有权发起一次通讯，可以随时向从设备要数据。而作为一个提供服务的设备，则需要运行在从设备下，为主设备提供服务。

I^2C 协议规定，仅主设备有权发起、结束一次通讯，其向总线发送的发起标志，称为起始位，发送的结束标志，称为停止位。每一次通讯必须含有起始位与停止位, 但不限制起始位与停止位之间传输的帧数。

在 I^2C 通讯中，主设备是通过地址，呼叫从设备的。这就像打电话，电话号码为手机呼叫的身份标志，建立通讯。I^2C 也需要一个身份标志完成通讯，这个身份标志就是地址。地址为 char 型[10]，其组成为：7bit[11] 地址位 +1bit 读写位，其中读写位决定了主从设备的传输方向，“0”表示：主机发送数据，“1”表示：主机接收数据。

char型bit位	7	6	5	4	3	2	1	0
作用及名称	**从机地址**							**读/写**

I^2C 通讯数据长度为 8bit，即每一帧数据长度为一个 char 型。在收到数据后，对方要回应一句“收到”，这个回应标志，称为 ACK 位。

7.3.3 I^2C 时序

I^2C 的时序，对于通讯来讲可分为为三种：主机向从机发送数据、从机向主机发送数据、主机先向从机发送数据然后从机再向主机发送数据，在下图中，展示了详细的时序图，其中，灰格子为主机行为，白格子为从机行为。

1、 主机向从机发送数据

S[i]	从机地址	0[ii]	A[iii]	数据	A	数据	$A/\overline{A}$	P[iv]

2、 从机向主机发送数据

S	从机地址	1	A	数据	A	数据	$A/\overline{A}$	P

3、 主机先向从机发送数据，然后从机再向主机发送数据

S	从机地址	0	A	数据	$A/\overline{A}$	S	从机地址	1	A	数据	$A/\overline{A}$	P

[i] **S**：I^2C 起始位。表格中灰色格子为主机发送。
[ii] **读写位**：“0”表示主机发送数据，’1’ 表示主机接收数据。
[iii] **A**：I^2C 应答位。
[iv] **P**：I^2C 停止位。

1(主–从)、2(从–主) 完全按照章节：7.3.2 中描述的样子运行，不难理解。而第 3 种(主–从–从–主) 的形式比较难以理解，其实这种形式，也仅仅是将 1(主–从)、2(从–主) 两种形式，“复制–粘贴”式的拼合起来。

唯独不同的是，中间切换过程并未发送停止位，原因是，一旦中途发送停止位，总线 就从繁忙变为空闲状态，这时如果有其他设备来抢总线，极易出现传输失败的情况。

这三种都是标准形式，我们应该养成按照标准写程序的习惯，尽管您的总线中可能只有一个主设备。按 照标准写程序，在很大程度上，能避免 bug 的出现。

[10] char：C 语言中的一个数据类型，由 8 个比特组成。
[11] bit：二进制的一位为 1 比特，名称为 bit，与十进制的个十百千万类似

7.4 通讯语法及结构

7.4.1 语法简介

通讯语法，是为了丰富通讯而建立的。如果单单读数据或是写数据，双方没有约定内容，那么设备就不清楚要读什么数据，或者要写什么数据了。这时候我们需要建立出一个会话协议出来，让对方知道在说些什么，这样就明白对方要传输什么样的数据了。

在一开始设计中，我们仅考虑了 I^2C 传输数字数据[12]的功能，在该情况下，I^2C 仅需要：起始位 +7bit 地址位 +1bit “读”，就可以不间断的读数字数据了，直至发送停止位通讯结束。

后来，随着功能的不断扩展，无会话的 I^2C 通讯协议，就显得为捉襟见肘了。基于这个问题，我们规定了通讯的语法。

7.4.2 语法

语法十分简单：先传输一个命令，再传输命令所需的任意个的数据，其中“命令”仅允许主设备发往从设备，“数据”主、从设备均可发送。

命令 (主设备发送)	数据 (从、主设备发送)	数据 (传输方向与上一帧相同)	...

举个例子：以老板命令员工扫落叶为例，老板先告诉员工：“把落叶扫了，然后装垃圾袋里交给我”员工收到命令后，把地上的落叶扫了起来，用垃圾袋打包好，递交给老板，从而执行完一次命令。这里老板就是主设备，员工就是从设备。这时候我们就可以抽象为，命令 [主设备发送]+ 数据 [从设备发送] 的语句了。

把例子等价为灰度传感器的通讯：以传输数字数据为例，首先主设备发送一个命令给从设备，代号为 0xDD，告诉灰度传感器：“把每一路的高低电平收集起来，打包在一个 char 型数据中，发送给我”，从设备接收到 0xDD 后, 立即将该数据打包上传给主设备，这就完成了一次简单的通讯。

这里应该记住的是，通讯的语法：命令 [主设备发送]+ 数据 [从、主设备发送]。当然往后阅读你会发现，为了提高通讯的效率，语法并不局限于此。

7.4.3 结构

那么语法在 I^2C 协议中结构是如何的呢？或者说我们该如何写程序呢？我们来探究一下，以防编程出错。

在前一节：I^2C **时序7.3.3**中提到，I^2C 通讯传输分为三种，分别是：主设备读、主设备写、主设备先写再读。在本章节中，我们将仅用到主设备写、主设备先写后读的功能。

但是实际上，主设备读，也经常被用到，那是因为它能有效的提高通讯效率。这里先按下不表，您在随后的阅读中，将会了解到这个功能。

1、主设备写：

主设备写数据，常用在类似于更改软件地址、传输通道使能等参数中，其语法为: 命令 + 数据，结合 I^2C **时序7.3.3**中的内容，其通讯结构为：

S	从机地址	0	A	命令	A	数据 1	A	数据 2	A	...	数据 N	$A/\overline{A}$	P

注：图中灰色为主机行为，白色为从机行为。从机地址后紧跟着读写位，主机写为“0”，主机读为“1”。

[12]数字数据：是指传感器 8 路的高低电平数据，打包成一个 char 型，此数据称之为数字数据。后续内容将会详细讲解

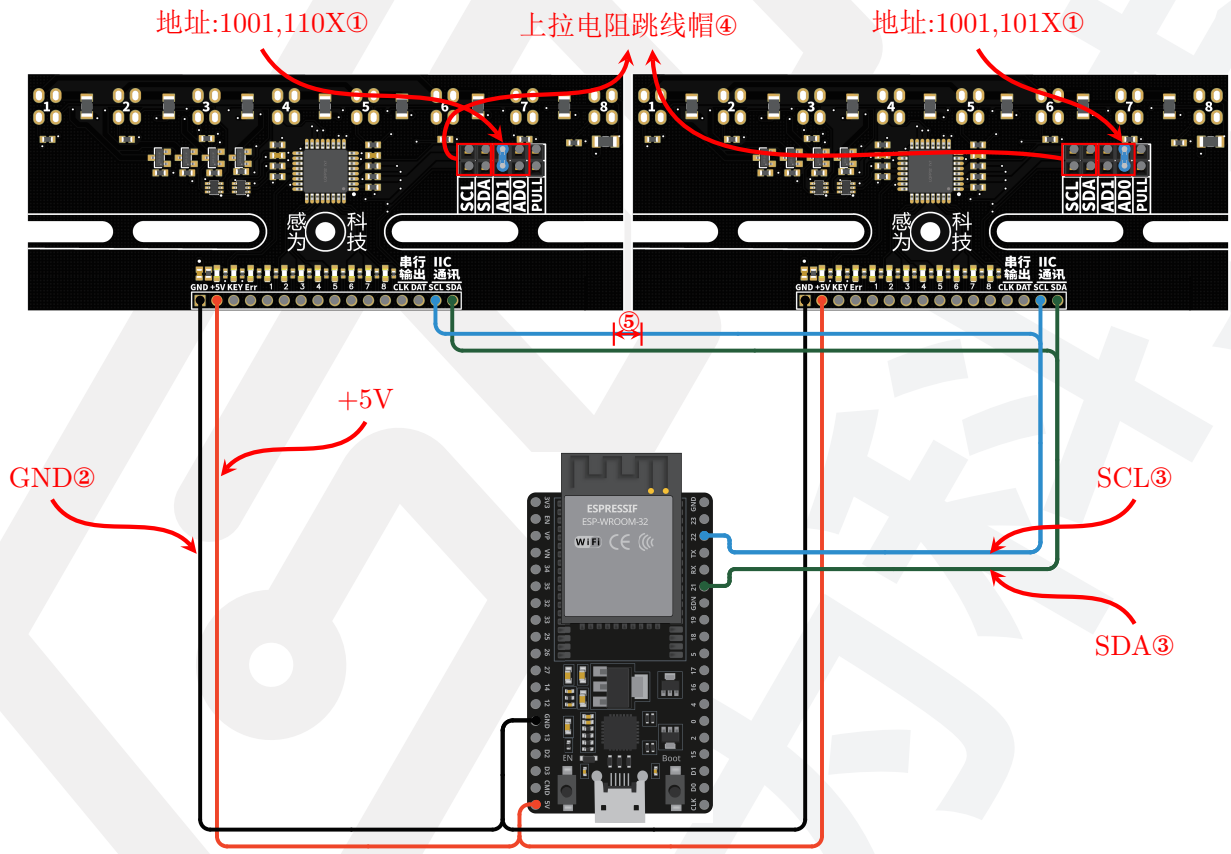
2、主设备先写再读：

主设备先写再读，常用在类似于传输数字数据，模拟数据等应用中，其语法为: 命令 + 数据，其通讯结构为：

S	从机地址	0	A	命令	A	S	从机地址	1	A	数据 1	A	...	数据 N	$A/\overline{A}$	P
	↑ 写数据						↑ 读数据								

需要注意的是，在这个模式中，我并没有在写数据与读数据之间加停止位，原因是，如果加停止位，总线将被释放，如果总线被其他设备抢走，这时候需要等待总线空闲后，才能读取后续数据。尽管我们修复了加停止位总线被抢走后出现的 bug，但是出于写程序的标准性考虑，我们还是列出了这条注意事项。

7.5 设备接线



①： 地址设置请参阅章节7.6;

②： 主控必须与灰度传感器共地，共地才能建立通讯。

③： SCL、SDA 总线需要上拉电压，这是 I^2C 通讯协议规定的。

④： 安装跳线帽后，总线为上拉状态。

- 板载的上拉源为 $5V, 10k\Omega$，请详细查阅您所用主控数据手册，以验证是否兼容。
- 如果外置上拉源，灰度传感器要求总线最低上拉源为 72% 的供电电压，如果低于这个电压阈值通讯极易失败。
- 如果外置上拉源，上拉电阻应选择 $1k$ $10k$ 范围内，如果电阻过大将影响总线的通讯效果。

⑤： 灰度传感器间的安装距离不能太近，因为两个灰度传感器的时钟不同步，如果太近，设备之间的光信号会相互干扰，造成误判。
安装时请遵循公式：$D_l \geqslant 3 \times \tan(15°) \times D_h$，其中 D_l 为对管中心点间距，D_h 为对管顶点距检测面的高度。

7.6 地址位设置及实例

7.6.1 地址位设置

在这个章节中，我们探讨硬件地址的设置方法，如需软件配置地址请参考章节:7.11

在 I^2C 总线协议中，设备间的通讯，是通过主设备呼叫从设备的地址实现的。使用该传感器时，需要将您的设备设置在主模式下，而灰度传感器则运行在从设备模式下。从设备的地址是由 7bit 地址位 +1bit 读写位组成的，在地址位中高 5bit 为软件地址位，由软件配置，出厂数据为 0b10011，低 2bit 为硬件地址位，由安装跳线帽[13]配置，读写位决定了传输的方向。其组成形式如下所示：

S地址位7[i]	S地址位6	S地址位5	S地址位4	S地址位3	H地址位2[ii]	H地址位1	读写位0
1	0	0	1	1	AD1[iii]	AD0[iv]	X[iv]

[i] S 地址位: 含义为软件地址位 (software address)，由软件配置, 出厂时其地址位 1001 1XXX, 具体细节参考章节:7.11 。

[ii] H 地址位: 含义为硬件地址位 (hardware address), 由板载插针、跳线帽配置，不装跳线帽时为“0”。

[iii] AD1: 电路板上跳线帽安装位置，有对应的丝印层标记，其名称为 AD1，安装跳线帽后，由“0”变为“1”。

[iv] AD0: 电路板上跳线帽安装位置，有对应的丝印层标记，其名称为 AD0，安装跳线帽后，由“0”变为“1”。

[v] X: 其含义为任意数值，不管该位是“0”或“1”。这一节讲地址位，与读写位无关，但其与地址位组成一帧数据，故必须示意，但不予理睬。

7.6.2 地址程序实例

上一节中，我们讲述了如何用硬件配置 I^2C 地址，即通过安装跳线帽的方式，来改变设备的地址。在这一节中，我们将讲述如何在不同的编程习惯下，正确写入设备的地址，首先我们假设插针 AD1、AD0 均安装有跳线帽，这时 H地址2=1，H地址1=1。按照上述表格，地址为：0b1001111X[14]

在编写程序时，如果您用寄存器编程，那么请在对应的寄存器写入该地址 0x9E[15], 即 0b1001111X，以给 stm8 编程为例:

```
/*I2C_OARL为stm8单片机的地址寄存器，需要将设备地址写入该寄存器。*/
I2C_OARL &= 0x01;//地址位清零,为了不影响0位，先将高7位清零。按位与0x01=0b0000 0001,结果为：0b0000 000X
I2C_OARL |= 0x9E;//写入地址,为了不影响0位，按位或0x9E=0b1001 1110。结果为：0b1001 111X
```

在程序编写时，如果您用库函数编程，那么地址为 0x4F。请注意, 由于大多数 I^2C 库函数的内部会将地址左移，目的是，在左移后的第 0 位插入读写位，以建立通讯方向，所以在提供地址时，应该相应的，要右移一位，即 0b1001 111X>>=1; 结果为 0b01001111=0x4F，以 ESP32 写法为例：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f //本应地址位0x9E，但Wire.h库会对地址左移一位，故要右移1位，即0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数，运行它能让esp32使能IIC
}
void loop(){
    Wire.beginTransmission(GW_GRAY_ADDR);//通讯开始函数，它会对输入的地址进行左移操作。
    Wire.write(0xAA);//Wire.h库中的写函数，向从设备发送数据0xAA。
    Wire.endTransmission(1);//Wire.h库中的停止位函数，写入1为真，即产生停止位。
}
```

[13]跳线帽：默认不装地址跳线帽的情况下，为“0”，安装地址跳线帽后其地址位为“1”。

[14]0b：二进制标识符，与 0x 为十六进制标识符类似

[15]0x: 十六进制标识符，在 C 语言中，char 型有 8bit，可以由两个十六进制数表示。

7.7 读取数字数据

7.7.1 功能描述

读取通道数字量功能（命令符：0xDD，即Digital Data 数字数据），是将传感器 8 路的高低电平数据，打包成一个 char 型，通过 I^2C 的一帧发送出去，此数据称之为数字数据。该功能为只读模式。

通俗来讲，就是将 1~8 路 LED 灯的亮灭，编辑为一个 Char 型发送出去。值得注意的是：LED 亮为“1”、LED 灭为“0”。同时 Char 的第 0bit 为第 1 路数据，第 1bit 为第 2 路数据，以此类推，第 7bit 为第 8 路数据。

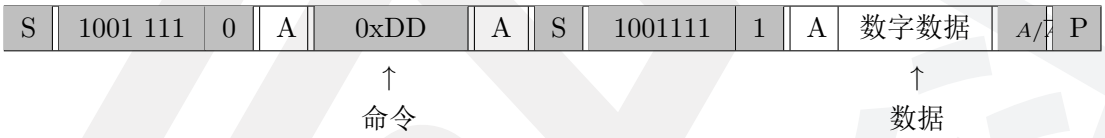
7bit	6bit	5bit	4bit	3bit	2bit	1bit	0bit
OUT8	OUT7	OUT6	OUT5	OUT4	OUT3	OUT2	OUT1

7.7.2 通讯方法

在上一章节: **语法7.4.2**中，我们提到，一条语句是由命令 + 数据组成的。如果您想获取数字数据，需要在一条语句中先发送一个命令符：0xDD，再去读数据，则所读得的数据即为数字数据。

方法 1：

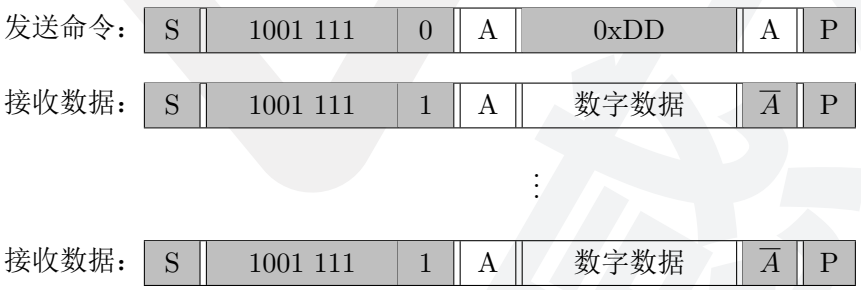
在以下说明中，我们都将从设备地址设置为 0b1001 111，以保证文章的一致性。方法 1，用的是标准的命令语句，是一种最基础的通讯方案。其结构为命令 + 数据的形式：



在整个语句中，在 I^2C 通讯中，占用了共计 39 个时钟时间，这个写法是没有问题的，完全能运行。但是我们所需的 char 数据，却仅仅需只占 8 个时钟传输，外加一个应答位，共计 9 个时钟。如果效率这么低下，那么将会大大增加我们的通讯时间。

方法 2：

为了提高通讯效率，我们允许用户在只发一次该命令后反复读取该值，而不需要重复的发送命令。当然，前提是不要有新的命令覆盖 0xDD，所以如果您有覆盖该命令，请一定要在读取数据前发一条命令 0xDD，用于更新命令。方法 2 的结构为：



就这样，写一次命令，设备会记录当前命令，后续读取的数据皆为 0xDD 下的数据, 所需时钟为 20 个，增加了近乎 2 倍的效率。

方法 3：

命令所在的内存在初始化后，默认为 0xDD。所以您开机后, 命令默认是 0XDD，这时候，如果您仅仅要读取数字数据，可以直接向设备要数据即可，结构为：

接收数据：	S	1001 111	1	A	数字数据	$\overline{A}$	P

7.7.3 代码示例

方法 1：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。文档在方法3之后有代码，请添加到您的程序中。
}
void loop(){
    char recv_value;//数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xDD);//写函数，发送命令0xDD
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

方法 2：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。文档在方法3之后有代码，请添加到您的程序中。
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xDD);//写函数，发送命令0xDD。
    Wire.endTransmission(1);//停止位函数，写入1为真，即产生停止位。
}
void loop(){
    char recv_value;//数据存放点。
    //读数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

方法 3：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。文档在方法3之后有代码，请添加到您的程序中。
}
void loop(){
    char recv_value;//数据存放点。
    // 开机默认0xDD模式
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

在上述例程中，均在初始化时，有一个 ping 函数，详情请参阅7.13。其目的是解决一个 bug，当您的主控与灰度传感器同时上电时，您的主控初始化可能会比灰度传感器初始化快，这时候如果发命令，命令可能传达不到，收数据可能数据不对。所以需要在您主控的初始化函数中，写入 ping 函数。此函数会不停的发送命令 0xAA，然后去读取数据。当灰度传感器初始化完成后，收到了 0xAA 命令，它会理解为 ping 命令，这时如果它在线，它会返回 0x66。此时您的主控如果读到 0x66，则会跳出循环。这个方法可以确保您的主控与灰度传感器是同步初始化的，这样可以减少了数据错误的发生，其函数为：

```
void ping(void)
{
    char sensors_status;
    while(sensors_status!=0x66)//开机判断，如果读出的数据为0x66，则停止循环。
    {
        Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
        Wire.write(0xAA);//写函数，发送命令0xAA
        Wire.endTransmission(0);//停止位函数，0为不产生停止位。
        Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
        sensors_status = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
```

7.8 单通道读取模拟数据

7.8.1 功能描述

单通道读取模拟数据 (命令符：0xB1~8) 功能，是将传感器对管读取的模拟数据值，经过抗干扰滤波后，经 I^2C 发送出去。该功能为只读模式，命令符为 0xB1~8，其中命令 0xB1 为第 1 路的模拟信号，0xB2 为第 2 路，以此类推，0xB8 为第 8 路。

命令	0xB1	0xB2	0xB3	0xB4	0xB5	0xB6	0xB7	0xB8
模拟数据	1 路	2 路	3 路	4 路	5 路	6 路	7 路	8 路

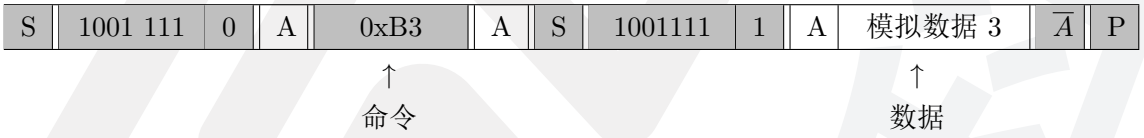
灰度传感器所用的 ADC 为 10bit 数据，为了传输简单快捷，我们将 10bit 压缩至 8bit，这意味着数据的分辨率会变差，但也仅仅丢失了一位数据，精度与设备内部所用的差 2 倍。但是能保证 I^2C 仅传输 1 次，提高了通讯的效率，是一种两全其美的折中方案。传输的每一帧数据都可以拿来直接用，而不需要进行移位叠加操作，方便快捷。

7.8.2 通讯方法

单通道读取数据的通讯方法与数字数据通讯方法相同。我们拿读取第 3 路模拟数据为例，讲述如何使用该命令。

方法 1：

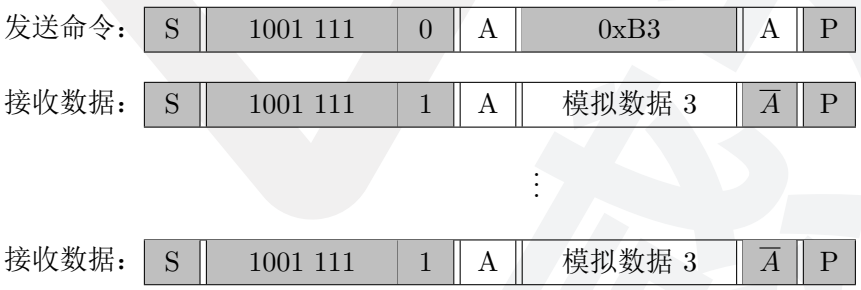
在该方法中，我们将使用标准的命令 + 数据的通讯方案进行讲述，其结构为：



该方法适用于读取特定的某几路数据，我们读哪路就需要发送哪路的命令。如果您想追求更高的效率，请看章节：**7.9**

方法 2：

如果仅仅需要读取一路的模拟数据，我们允许用户在只发一次该命令后反复读取该数据，而不需要重复的发送命令。其结构为：



就这样，写一次命令，设备会记录当前命令，后续读取的数据皆为 0xB3 下的数据。如果您想追求更高的效率，请看章节：**7.9**

7.8.3 代码示例

方法 1：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
}
void loop(){
    char recv_value;//你的数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xB3);//写函数，发送命令0xB3
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

方法 2：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。文档在方法3之后有代码，请添加到您的程序中。
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xB3);//写函数，发送命令0xB3。
    Wire.endTransmission(1);//停止位函数，写入1为真，即产生停止位。
}
void loop(){
    char recv_value;//你的数据存放点。
    //读数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

7.9 连续通道读取模拟数据

在上一章节7.8中，我们讲述了如何单路读取模拟数据，如果使用这种方案读取 8 路模拟数据，每次就要发 8 个命令，整个过程十分消耗时间，大大降低了通讯效率。在这一章节我们将讲述如何一次性的取出所有通道的模拟数据。

7.9.1 功能描述

连续通道读取模拟数据 (命令符:0xB0) 功能，提供一种更优的方案，该功能可以将 8 路模拟数据一次性提取出来。该功能为只读模式。

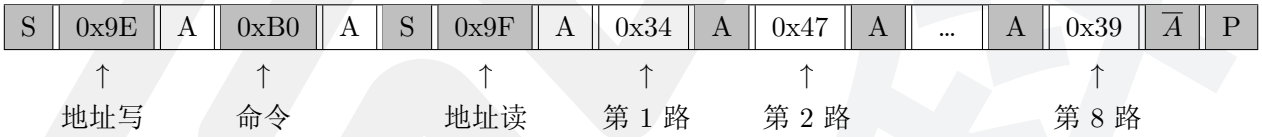
除此之外，该功能还有一个通道使能寄存器，您可以通过该寄存器，关闭传输通道，滤除您不需要的数据。

例如假设您仅需要 1、3、5、7 路的数据，如果按照上一章节：7.8的方法，您需要轮替 4 次发送命令、接收数据，效率十分的低。如果使用此功能，您仅需要先配置通道使能寄存器，使之关闭 2、4、6、8 路通道数据，这样就可以连续读取到 1、3、5、7 这 4 路信息了。当然了您也可以结合此功能，达到传输单通道的作用。

7.9.2 通讯方法

方法 1：

第一种方法使用标准命令 + 数据的语法，当我们发送命令符：0xB0 后，紧接着读取 8 帧数据，这样我们可以不间断的读取每一路数据。

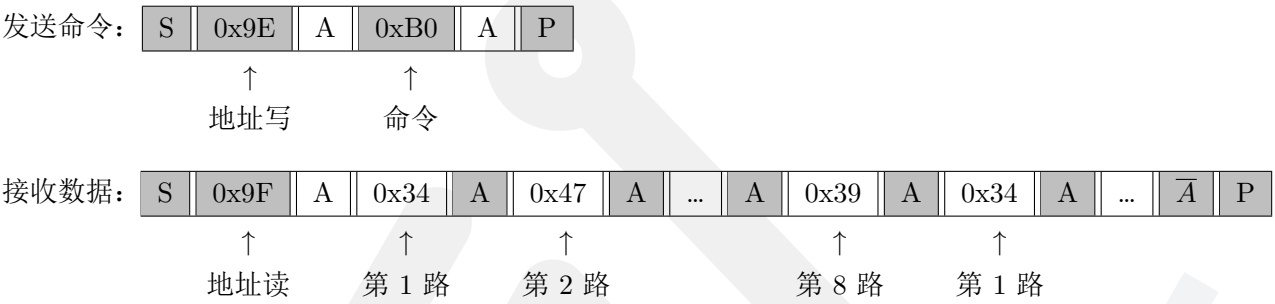


方法 2：

您可以先发命令，再反复接收数据。这样您仅需要发送一次命令，即可反复读取该数据，这种方法适合于多台灰度传感器共同使用。

方法 3：

当然，如果您想要更高的效率，您也可以先发送命令，再连续读取该数据。这种情况下，当读取到第 8 路数据后，设备将从头读取。这样当读到第 9 帧时，该数据为第 1 路数据。这个方法适合与单台灰度传感器的连续读取。



7.9.3 代码示例

方法 1：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
}
void loop(){
    unsigned char recv_value[8];//数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xB0);//写函数，发送命令0xB0
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,8/*读8个char*/,1/*停止位*/);//读数据的集成函数。
    for (unsigned int = 0; i < 8; ++i) {
        recv_value[i] = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
```

方法 2：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xB0);//写函数，发送命令0xB0。
    Wire.endTransmission(1);//停止位函数，写入1为真，即产生停止位。
}
void loop(){
    unsigned char recv_value[8];//数据存放点

    // 读取数据 有STOP
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,8/*读8个char*/,1/*停止位*/);//读数据的集成函数。
    for (unsigned int = 0; i < 8; ++i) {
        recv_value[i] = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
```

方法 3：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xB0);//写函数，发送命令0xB0。
    Wire.endTransmission(1);//停止位函数，写入1为真，即产生停止位。
}
void loop(){
    unsigned char recv_value[64];//数据存放点

    //一次性连续读取8组8路数据，共计64个，存放在recv_value中
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,64/*读64个char*/,1/*停止位*/);//读数据的集成函数。
    for (unsigned int = 0; i < 64; ++i) {
        recv_value[i] = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
```

7.9.4 传输通道使能

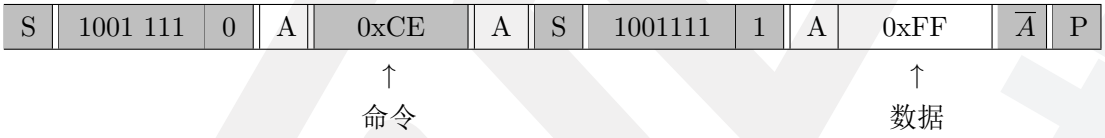
传输通道使能 (命令符：0xCE，即 Channel Enable) 寄存器，它与连续通道读取功能共同使用，起到关闭传输通道的作用。该寄存器每一位控制一路数据的传输，置位时使能，默认的复位值为 0xFF，即全通道打开。该功能为读写模式。

寄存器	7bit	6bit	5bit	4bit	3bit	2bit	1bit	0bit
使能位	8 路	7 路	6 路	5 路	4 路	3 路	2 路	1 路

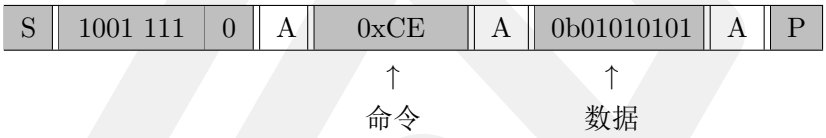
该寄存器不具备存储功能，在重启或者断电后，数据将复位为 0xFF，请将通道数据写入您的主控中，初始化时写入。此外，该寄存器仅仅控制着数据传输通道，并不会实际关闭传感器对管的实时测量，所以不会影响并/串行输出的使用。

通讯方法：

读取该寄存器时的结构为：



写入该寄存器的结构为：



代码示例：

```
/*该示例为修改使能寄存器后读取连续通道模拟数据，已验证传输通道使能功能*/
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。//模拟通道使能
    Wire.write(0xCE);//写函数，发送命令0xCE。
    Wire.write(0b01010101);//只要探头1,3,5,7的数据
    Wire.endTransmission (1);//停止位函数，写入1为真，即产生停止位。
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。 //打开模拟数据模式
    Wire.write(0xB0);//写函数，发送命令0xCE。
    Wire.endTransmission (1);//停止位函数，写入1为真，即产生停止位。
}
void loop(){
    unsigned char recv_value[4];//数据存放点。
    //读取探头1，3，5，7数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,4/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    for (int i = 0; i < 4; ++i) {
        recv_value[i] = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
```

7.9.5 通道数据归一化使能 (仅 V3.6 及以上可用)

通道数据归一化使能 (命令符：0xCF，即 Channel Flat) 寄存器，该功能仅 V3.6 及以上版本可用，您可以通过传感器的丝印查看，或者您可以通过 IIC 读取固件版本号查询。

正所谓世上没有相同的两片叶子，探头在生产过程中，由于各种因素，接收管的光感度，发光管的亮度都有不同。在读取模拟量数据时，即使 8 个探头都同为探测白色，他们的数据也不统一，这种情况下，我们就需要归一化，即让所有的探头探测白色/黑色时，数据是相同的。

归一化结果与校准相关，校准的过程可以参考章节：2.3，校准过程实际上就是学习黑白的过程，读取归一化后的模拟量时，当检测到白色时，输出结果将为 255，当检测黑色时，输出结果将为 0。

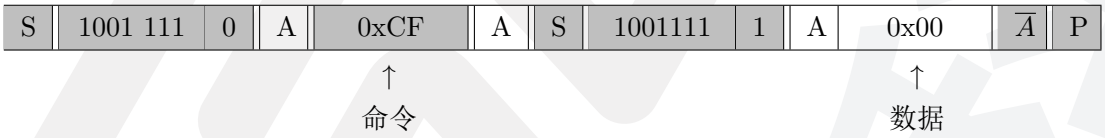
它与单、连续通道读取功能共同使用，起到关闭通道归一化的作用。该寄存器每一位控制一路数据的归一化，置位 (bit=1) 时使能，默认值为 0x00，即未开启归一化。请在读取模拟量前写入该数据，以便进入归一化，为了防止通讯错误等故障，请时不时的读取一下该命令下的数据，以确定模拟通道是否进入归一化。该功能为读写模式。

寄存器	7bit	6bit	5bit	4bit	3bit	2bit	1bit	0bit
使能位	8 路	7 路	6 路	5 路	4 路	3 路	2 路	1 路

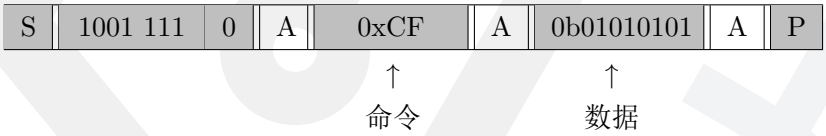
该寄存器不具备存储功能，在重启或者断电后，数据将复位为 0x00，请将通道数据写入您的主控中，初始化时写入，写入前请先发送 ping 命令，请参考章节：7.13 , 以确保线路以及传感器正常工作。

通讯方法：

读取该寄存器时的结构为：



写入该寄存器的结构为：



代码示例：

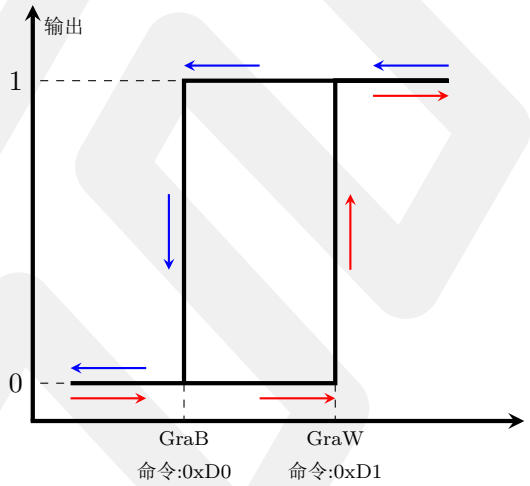
```
1 /*程序请参考传输通道使能*/
```

7.10 读/写滞回比较器参数

7.10.1 功能描述

读/写滞回比较器参数 (命令符：0xD0、0xD1) 功能，是通过 I^2C 读取或者改变灰度传感器内部的两个阈值参数，起到自由控制灰度传感器输出的功能。但值得注意的是，写入的数据仅存在于内存中，不会保存，断电即消失，请将此数据保存在您主控中，在初始化阶段写入。

在灰度传感器的滞回比较器中，有两个参数，一个是红色箭头方向，输出从 0 至 1 的阈值参数，我们将其称之为灰黑 (GrayB) 参数，GrayB(GrayBlack) 参数读写命令为 0xD0；另一个是蓝色箭头方向，输出从 1 至 0 的阈值参数，同理我们将其参数称之为灰白 (GrayW) 参数，GrayW(GrayWhite) 参数读写命令为 0xD1，其图像为：



7.10.2 读滞回比较器参数

滞回比较器有两个参数，一个是 GrayB 参数，其命令为 0xD0，读取其数据的结构为：

S	0x9E	A	0xD0	A	S	0x9F	A	0x34	A	0x47	A	...	A	0x39	$\overline{A}$	P
	↑		↑			↑		↑		↑				↑		
	地址写		命令			地址读		第 1 路		第 2 路				第 8 路		

另一个为 GrayW 参数，其命令为 0xD1，读取其数据的结构为：

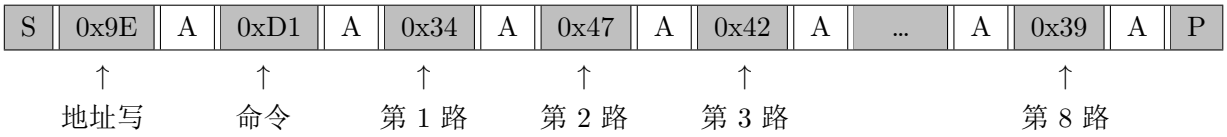
S	0x9E	A	0xD1	A	S	0x9F	A	0x95	A	0xA3	A	...	A	0x89	$\overline{A}$	P
	↑		↑			↑		↑		↑				↑		
	地址写		命令			地址读		第 1 路		第 2 路				第 8 路		

7.10.3 写滞回比较器参数

写滞回比较器参数与读类似。写 GrayB 参数，其命令为 0xD0，其结构为：

S	0x9E	A	0xD0	A	0x34	A	0x47	A	0x42	A	...	A	0x39	A	P
	↑		↑		↑		↑		↑				↑		
	地址写		命令		第 1 路		第 2 路		第 3 路				第 8 路		

写 GrayW 参数，其命令为 0xD1，其结构为：



上述结构，无论读写，我们必须一次性配置 8 路，而不能单独配置某一路。如果您想仅更改某一路数据，可以先将原始数据读出来，再改变其中一路信息后，重新写入。

7.10.4 代码示例

一、读滞回比较器参数

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    unsigned char recv_value[8];
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xD0);//发送命令0xD0。
    Wire.endTransmission(0);//不发送停止位
    // 读取滞回比较器参数
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,8/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    for (int i = 0; i < 8; ++i) {
        recv_value[i] = Wire.read();//取出Wire.requestFrom运行后的数据。
    }
}
void loop(){
    //...
}
```

二、写滞回比较器参数

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    unsigned char setup_value[8] = {/* 滞回比较器配置 */};
    Wire.begin();//初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数。
    Wire.write(0xD0);//发送命令0xD0。
    Wire.write(setup_value,8);  // 写入滞回比较器参数
    Wire.endTransmission(1);//发送停止位
}
void loop(){
    //...
}
```

注意：GrayB 参数需要比 GrayW 数值要小，否则比较器将产生混乱，OUT 灯可能会闪动。

7.11 软件地址配置

在前一节7.6我们提到如何设置硬件地址，在这一节中，我们将讲述软件地址如何配置。

7.11.1 功能描述

配置软件地址 (命令符：0xAD，即 Address)(V3.6 及以上型号命令符则需要连续发送两个 0xAD，其目的是解决命令误触导致地址无意中被改掉，加长命令有助于容错)，是将设备的 I^2C 地址，通过软件代码进行更改的一种功能。在 I^2C 协议中规定，地址帧由 8bit 组成，其中 7bit 为地址位，1bit 为读写位。7bit 地址位中，设备规定 5bit 高位为 S(软件) 地址位，2bit 低位为 H(硬件) 地址位，其中 H 地址位与 S 地址配置方式不同，相互独立，S 地址的默认值为 0b10011，其结构为：

S地址位7	S地址位6	S地址位5	S地址位4	S地址位3	H地址位2	H地址位1	读写位0
1	0	0	1	1	AD1	AD0	X

配置地址的方式很简单，仅需要先发送命令 0xAD(V3.6 及以上版本需要发送 0xAD,0xAD)，再发送软件地址位信息即可。设备将只读取 8bit 之中的 5bit 高位数据，自动忽略其中的 3bit 低位数据。如果高 5bit 皆为 0，则此次传输失效，回滚为原地址，其结构为：

S	1001 111	0	A	0xAD	A	0b1010 0XXX	A	P
				↑		↑		
				命令		S地址		

发送完 S 地址位信息后，需要将设备重启，以生效该地址, 可以发送命令:0xC0，即可重启设备，否则设备将延续旧的地址，直至重启，或者重新开机。

请注意：在改地址时，我们应该避免多个设备的地址相同，因为那样，可能在更改地址时，一次更改多个设备的S地址，造成设置混乱。所以在改地址时，我们应该将设备分配不同的H地址，超过 4 个后，需要分批挂载总线配置。

7.11.2 用法及例程

用法：

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器。
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xAD);
    Wire.write(0x98); // 0x98~0x9B的地址都会变成0x4C（=清除低3位+右移一位读写位）
    Wire.endTransmission(1);//终止通讯，发送停止位
}
void loop(){
    // ...
}
```

7.12 广播重置地址与扫描找回地址

如果您在使用过程中，忘记已经设置好的地址，或者是误发了设置地址命令，导致地址丢失。这时候您不必担心，我们提供两种方法解决此问题。

方法 1：通过广播重置地址

您可以通过呼叫广播地址 0x00，来重置出厂地址 (0b1001 1XXX)，要重置地址，您仅需要广播一个重置命令即可。不过 I^2C 总线的广播功能，应该被谨慎对待，因为他可以跳过设备地址操作设备，从而传输一些不属于自己的数据，常常会出一些意想不到的 bug。出于此原因，我们将命令规定为一串长度为 64bit 的数据，以增加命令的复杂度，防止误触发，其数据为：

0xB8	0xD0	0xCE	0xAA	0xBF	0xC6	0xBC	0xBC

传输这段数据串时，您不需要按照“命令+数据”的语法操作，您仅需要将这一串数据通过广播的方式，发送给所有设备，当总线上所有的感为灰度传感器收到这一串数据后，将全部自动为您重置地址，并重启。其结构为：

S	0x00	A	0xB8	A	0xD0	A	0xCE	A	0xAA	A	...	A	0xBC	A	P
	↑		↑												
	地址		数据串												

即便使用 64bit 的命令符，也应该谨慎对待广播功能，因为 64bit 的方式仅能防御设备自己，而不能做到挂在总线上的其他设备的安全。请在重置时，将其他除感为灰度传感器之外的设备移除；或将感为灰度传感器挂载到未连接任何从设备的主控身上，单独重置该设备。

```
#include <Wire.h>
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
// 广播重置地址所需的字符串
unsigned char reset_magic_number[8] = {
    0xB8,
    0xD0,
    0xCE,
    0xAA,
    0xBF,
    0xC6,
    0xBC,
    0xBC,
};
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    Wire.beginTransmission(0x00);//起始位函数
    Wire.write(reset_magic_number,8);
    Wire.endTransmission(1);//停止位函数。
    ping();
}
void loop(){
    //...
}
```

方法 2：扫描地址

I^2C 总线的从设备地址，一共有 7bit，排除 0x00 广播地址外，共有 127 个地址。您可以将灰度传感器单独挂载到总线上，将 127 个地址全部扫描一遍，如果哪个地址下有设备回应 (ACK)，那么这个地址就是该设备的地址。

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f

void i2c_scan()
{
    byte error, address;
    int nDevices;

    Serial.println("扫描中...");

    nDevices = 0;
    for(address = 1; address < 127; address++ )
    {

        Wire.beginTransmission(address);
        error = Wire.endTransmission();

        if (error == 0)
        {
            Serial.print("找到I2C设备, 地址为 0x");
            if (address<16)
                Serial.print("0");
            Serial.print(address,HEX);
            Serial.println("");

            nDevices++;
        } else if (error==4) {
            Serial.print("发生错误:  地址为 0x");
            if (address<16)
                Serial.print("0");
            Serial.println(address,HEX);
        }
    }
    if (nDevices == 0)
        Serial.println("无i2c设备\n");
    else
        Serial.println("完成\n");
}

/*函数setup与loop是arduinoIDE标准框架, setup是程序的入口, 只运行一次, loop则是循环运行, 类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    Serial.begin(115200);
}

void loop()
{
    i2c_scan(); //扫描i2c地址
    delay(5000); //每5秒扫描一次
}
```

7.13 ping 网络诊断工具

7.13.1 功能描述

ping（读音“乒”）网络诊断工具 (命令符：0xAA)，用于 I^2C 调试时使用，或是用于初始化中使用。实际上，该命令并不具备网络诊断功能，这个名称的意义在于帮助软件工程师快速理解其命令的用法，就是我们常说的：“能不能 ping 通?”

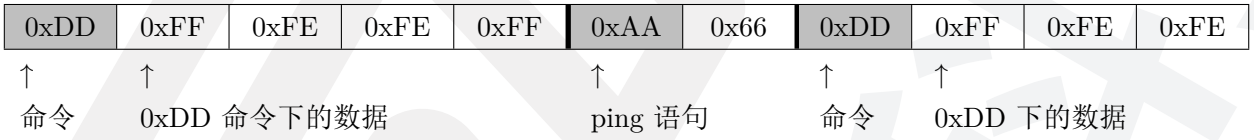
如果总线连接完整，地址正确，I^2C 程序无误，给灰度传感器发送 0xAA 命令，灰度传感器将返回数据 0x66 以证明其工作正常、总线完整。这个功能为只读模式。其语法结构只能为：

7.13.2 命令特性

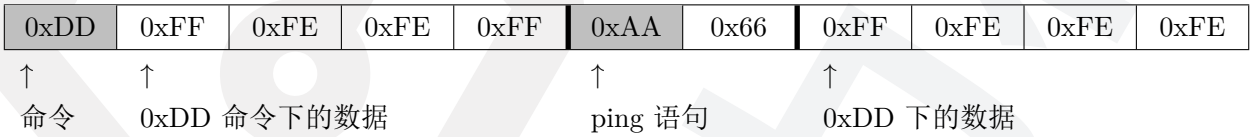
特性 1：回滚特性

ping 命令可以随时被插入，而不会影响当前命令的进程，此功能仅限于读数据时应用。

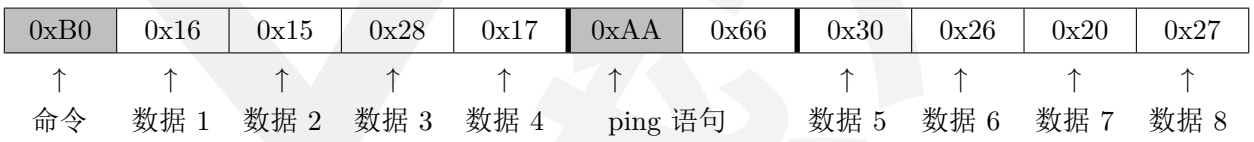
按照标准语法，当 ping 命令执行完毕后，为了不打断当前的进程，我们需要重新发送 ping 之前的命令，使之恢复并继续执行。例如，当前使用章节：数字数据7.7中的方法 2 读数字数据 (命令符：0xDD)，此时 ping 了一下。随后需要再发送数字数据命令 (命令符：0xDD)，使之继续读取数字数据。



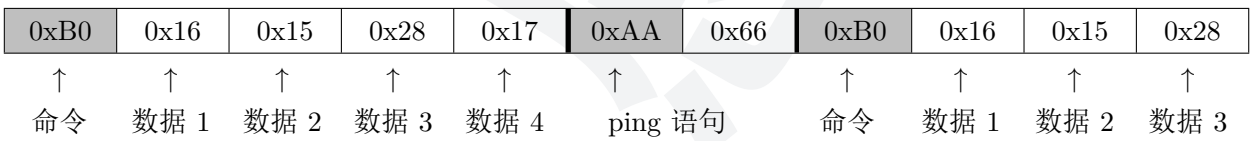
但您不需要这样做，ping 有命令回滚的特性。返回 0x66 后，您将不用重新发送命令 0xDD，灰度传感器将自动回滚为之前的命令 (0xDD)，并继续执行。



当您在连续读模拟数据 (命令符：0xB0) 时，ping 一下后，灰度传感器会按照之前的序号继续读数据，而不是从头开始传输。

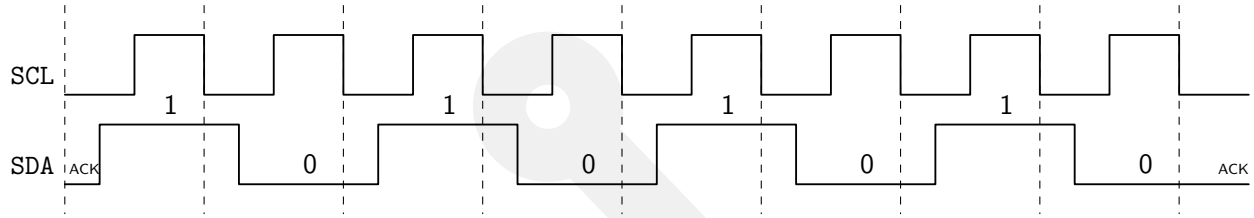


但，如果您要是在 ping 过后，重新发命令 0xB0，灰度传感器将会从头发送。需要注意的是，如果您向传感器写数据，这种操作是不被允许的，ping 后不可重新

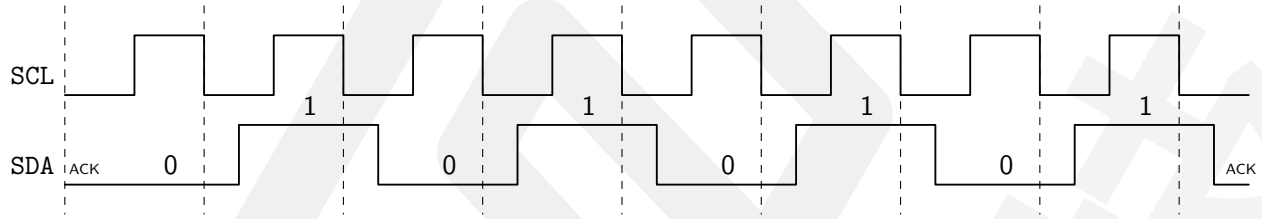


特性 2：信号校验特性

命令符：0xAA，它的二进制形式为，0b1010 1010。恰好为稳定的方波，方便示波器观看，以便调试总线。



返回值 0x66，它的二进制形式为，0b0101 0101。也为稳定的方波：



7.13.3 用法及程序实例

ping 功能常用在设备初始化中，用于同步您所用的主控与灰度传感器，由于灰度传感器开机到正常工作也需要一定的时间，如果您在未正常工作之前发送命令，可能会丢失通讯数据，或者通讯失败，所以需要用 ping 工具在用户主控程序初始化时，反复 ping，如果有正常返回值，则证明可以进行后续工作。

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    Serial.begin(115200);
}
void loop(){
    char recv_value;//数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xAA);//写函数，发送PING命令0xAA
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
    if (recv_value == 0x66) {
        Serial.println("PING OK");
    } else {
        Serial.println("PING NOT OK");
    }
    delay(1000);
}
```

7.14 读取错误信息

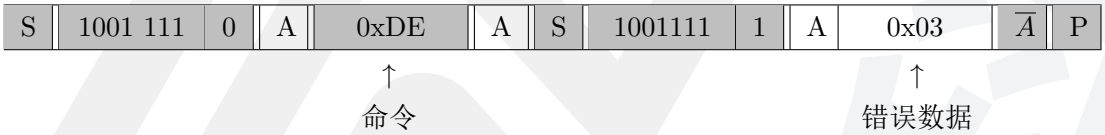
7.14.1 功能描述

读取错误信息（命令符：0xDE，即Data of Error），是将传感器使用过程中，出现的错误，将其汇总在 8bit 错误寄存器中。其中每 bit 代表一种错误，其详细构成为：

寄存器	7bit	6bit	5bit	4bit	3bit	2bit	1bit	0bit
使能位	保留						按键短路	对管过曝

位7:2	保留位，读出 0
位1	按键错误： 按键按钮按住时长超过 15 秒，系统判定为用户按键 pin 一直短接在 GND 上，此时置此位为 1。其目的是防止用户按键 pin 不小心搭接到 GND，而干扰传感器正常使用。
位0	对管过曝： 对管接收光线量是有物理上限的，假如对管受日光或者环境光影响而过曝，此位置 1。 注意：读取模拟数据不能判定环境光是否过曝，因为交给用户的模拟数据已经将环境光滤除，您收到的仅为对管反射光，当过曝发生时，对管达到物理阈值，此时模拟数据数值会降低，从而导致设备检查不准确。

该错误寄存器为只读寄存器，不可修改其中的内容。当有错误发生时，相应的位将被置位，并保持为置位状态，等待用户读取。当用户将数据读取后，整个寄存器将被自动清零。其通讯结构仅为：



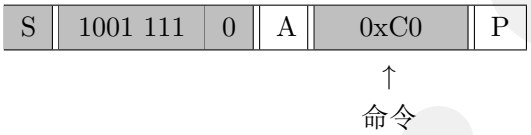
7.14.2 用法及程序实例

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器
}
void loop(){
    char recv_value;//数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xDE);//写函数，发送读取错误信息命令0xDE
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

7.15 设备软件重启

7.15.1 功能描述

设备软件重启命令 (命令符：0xC0, 即 Command top)，当您的主控程序出现 bug，或是更改设备软件地址后，需要对设备进行重启操作，这时候就可以发送命令 0xC0，即可完成设备重启。其通讯结构仅为：



7.15.2 用法及示例

当我们重启设备后，设备将首先初始化，这需要一些时间。如果您的主控在重启后，紧接着就发命令读写数据，那么大概率会漏掉信息，导致通讯错误。所以需要您使用 ping 函数，等待设备回应。

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器

    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xC0);//写函数，发送重启命令0xC0
    Wire.endTransmission(1);//停止位函数，0为不产生停止位。

    ping();//ping函数，用于同步esp32与灰度传感器
}
void loop(){
    // ...
}
```

7.16 固件版本号查询

7.16.1 功能描述

固件版本号 (命令符:0xC1)，用于查询设备固件的型号，该型号由 8bit 组成，其中前 4bit 为一组 16 进制数，后 4bit 为 16 进制数，例如：版本 V3.14 的版本号为 0x3E

数据位	7bit	6bit	5bit	4bit	3bit	2bit	1bit	0bit
	0	0	1	1	1	1	1	0
版本号	高位版本：3				低位版本：14			

您在查询固件版本号时，请先发送 0xC1，灰度传感器将返回数据到您的主控上，其语法结构只能为：

S	1001 111	0	A	0xC1	A	S	1001111	1	A	0x3E	$\overline{A}$	P
				↑						↑		
				命令						返回值		

7.16.2 用法及示例

```
#include <Wire.h>
#define GW_GRAY_ADDR 0x4f
/*函数setup与loop是arduinoIDE标准框架，setup是程序的入口，只运行一次，loop则是循环运行，类似while(1)*/
void setup(){
    Wire.begin();//Wire.h库中的初始化函数
    ping();//ping函数，用于同步esp32与灰度传感器
}
void loop(){
    char recv_value;//数据存放点
    //写命令
    Wire.beginTransmission(GW_GRAY_ADDR);//起始位函数
    Wire.write(0xC1);//写函数，发送读取固件版本命令0xC1
    Wire.endTransmission(0);//停止位函数，0为不产生停止位。
    // 读取数据
    Wire.requestFrom(GW_GRAY_ADDR/*地址*/,1/*读1个char*/,1/*停止位*/);//读数据的集成函数。
    recv_value = Wire.read();//取出Wire.requestFrom运行后的数据。
}
```

7.17 命令符

读取数字数据	0xDD	读
单通道读取模拟数据	0xB1~8	读
连续通道读取模拟数据	0xB0	读
传输通道使能	0xCE	读写
通道数据归一化使能	0xCF	读写
读/写滞回比较器参数 GraB	0xD0	读写
读/写滞回比较器参数 GraW	0xD1	读写
软件地址配置	0xAD	写
广播重置地址	字符串:0xB8 0xD0 0xCE 0xAA 0xBF 0xC6 0xBC 0xBC	写
ping 网络诊断工具	0xAA	读
读取错误信息	0xDE	读
设备软件重启	0xC0	无
固件版本号查询	0xC1	读