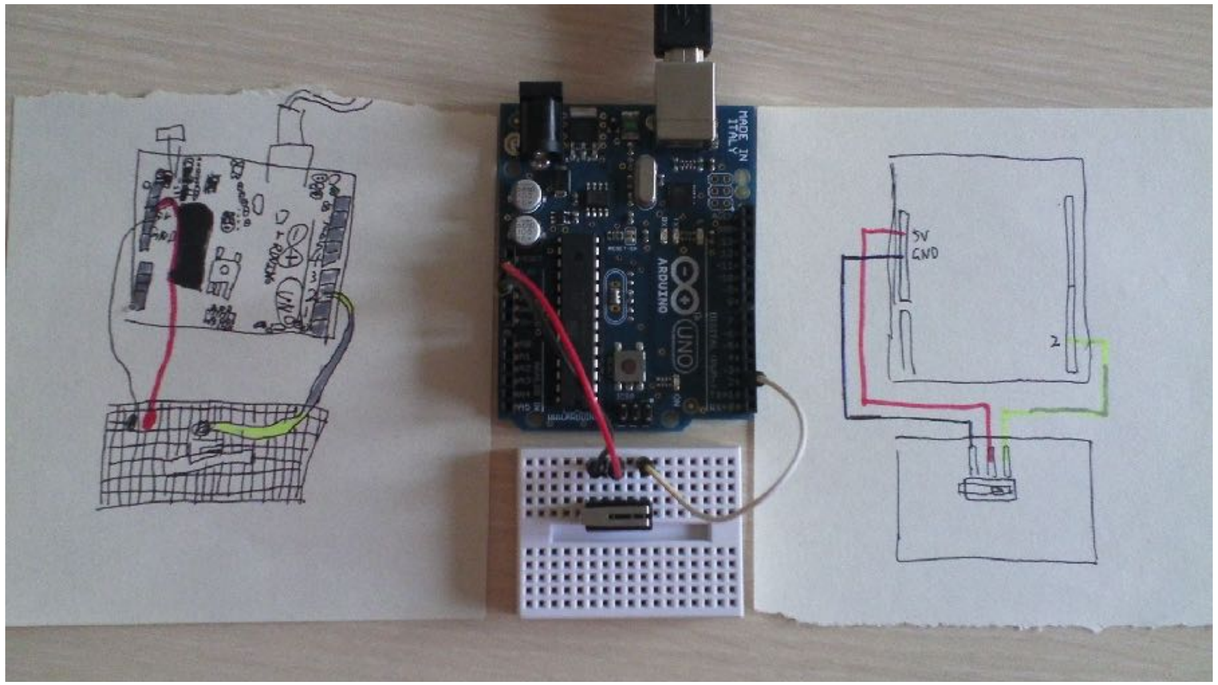
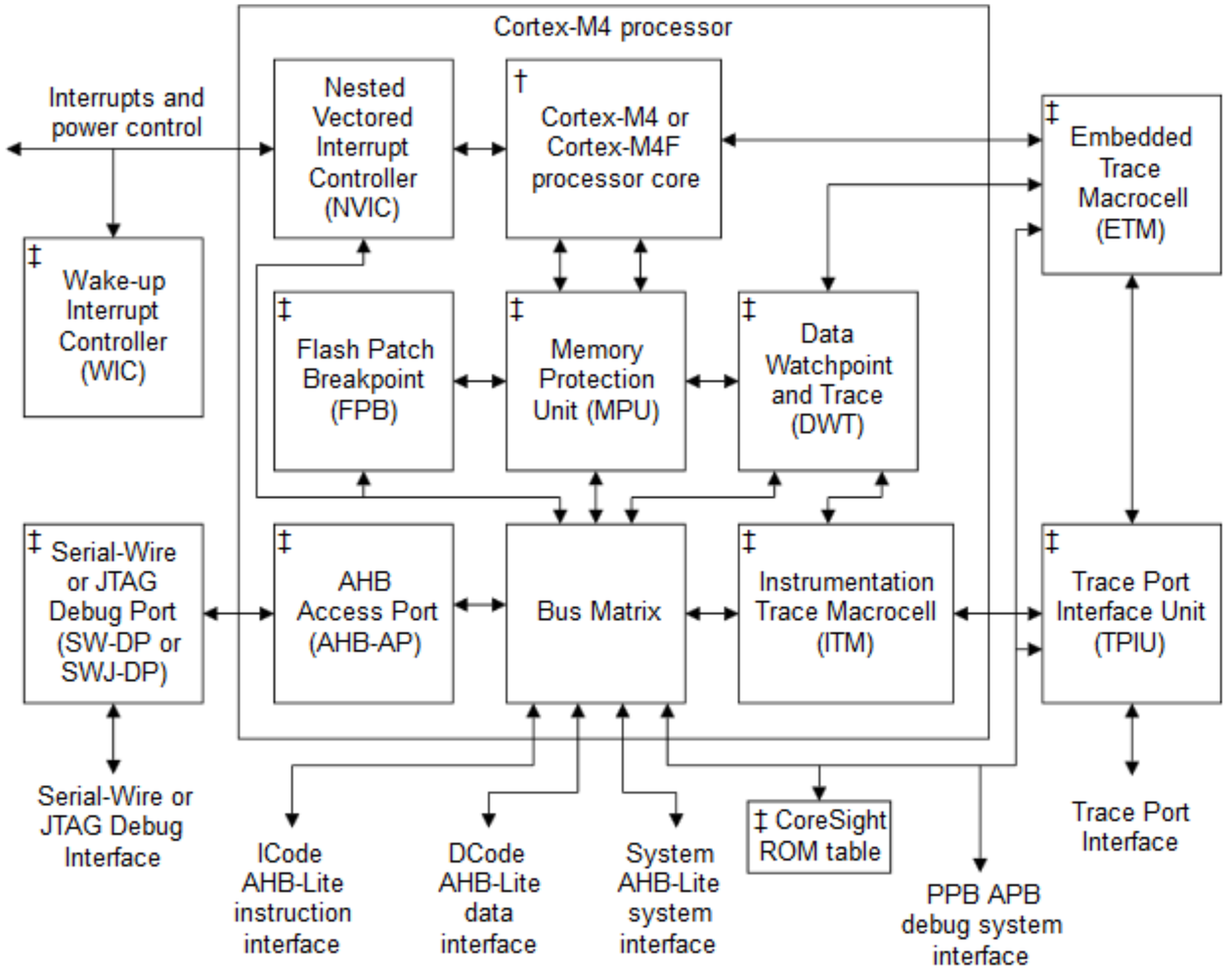



# 嵌入式系统的接口与外设

快速入门的物理计算

# Cortex-M4 Core



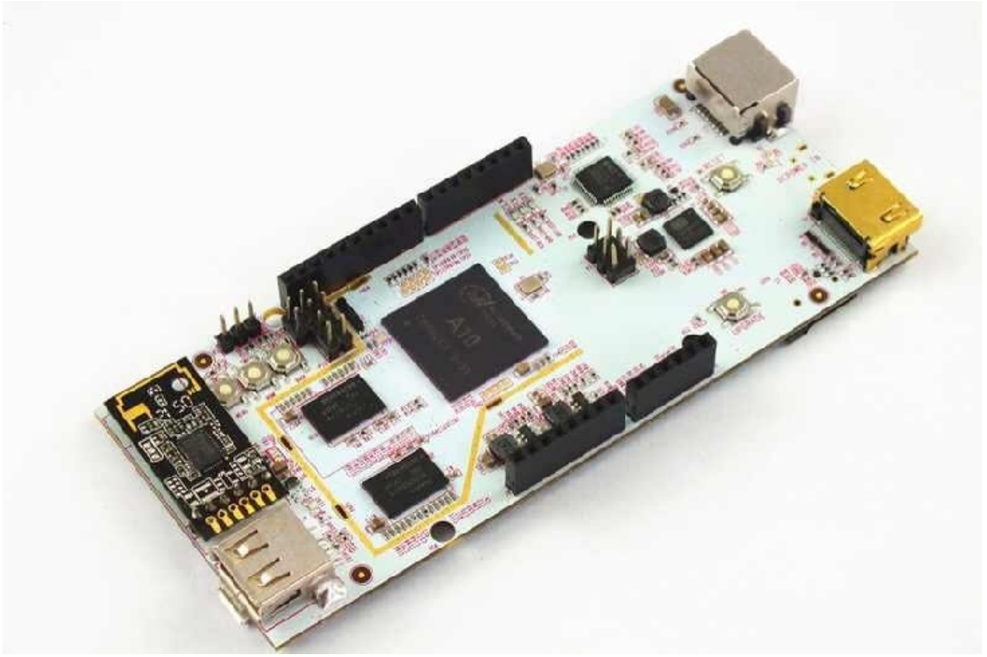

† For the Cortex-M4F processor, the core includes a Floating Point Unit (FPU)

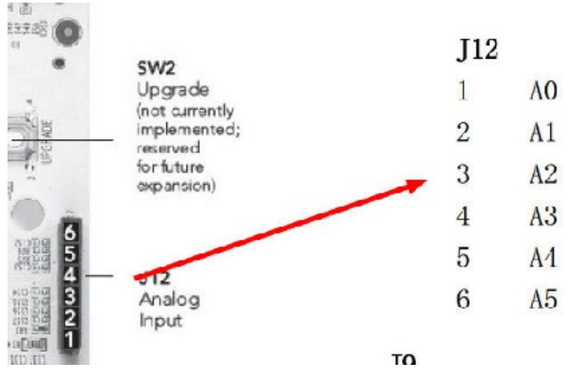
‡ Optional component

# 处理器vs微控制器

- 处理器的用途是手持、桌面或服务器，IO是为人机交互服务的
- 嵌入式微控制器的用途是控制其他设备，IO是为控制（输入/输出）服务的

# 嵌入式系统的GPIO






- GPIO就是可以由CPU直接操纵的引脚，可以写，可以读，可以接通信单元

# 概要

- 如何让一个程序根据开关的动作来点亮一个LED?

- GPIO
  - 基本概念
  - 端口电路
  - 控制寄存器
  - 在C语言中访问硬件寄存器
  - 时钟和复用

- 电路接口
  - 输入
  - 输出

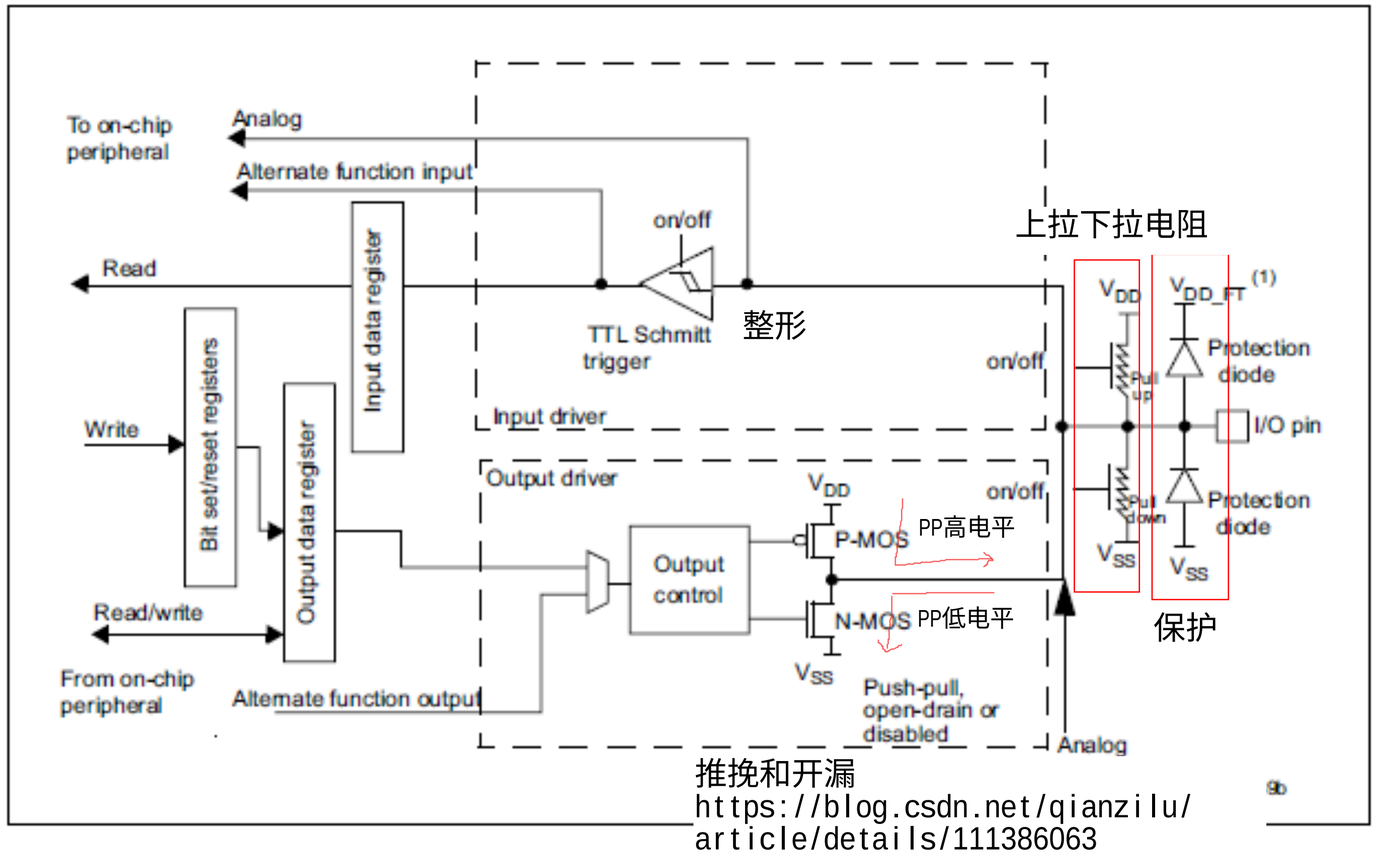
- 其他端口配置

# 基本概念

- **GPIO = 通用输入输出（数字）**
  - 输入：程序可以判断输入信号是1还是0
  - 输出：程序可以设置输出1或0
- **可以以此来接外部器件**
  - 输入：开关/按钮
  - 输出：LED、扬声器

# 单片机中的GPIO端口位电路

- **配置**
  - 方向
  - 复用
  - 模式
  - 速度
- **数据**
  - 输出（不同的访问途径）
  - 输入
  - 模拟
- **锁定**




ESD:防静电

Alternate function(AF)：复用，引脚既可以当做GPIO，也可以用来控制单片机内部设备

# 控制寄存器

- 每个通用I/O端口具有
  - 四个32位配置寄存器 (
    - GPIOx_MODER (输入、输出、AF、模拟)
    - GPIOx_OTYPER (输出类型：推挽或开漏)
    - GPIOx_OSPEEDR(速度)
    - GPIOx_PUPDR(上拉/下拉)
  - 两个32位数据寄存器 (GPIOx_IDR 和GPIOx_ODR)
  - 一个32位置位/清除寄存器 (GPIOx_BSRR)
  - 一个32位锁定寄存器 (GPIOx_LCKR)
  - 两个32位可选功能选择寄存器 (GPIOx_AFRH 和GPIOx_AFRL)

- **每个端口一组控制寄存器（总共10个）**
- **控制寄存器中的每个位对应端口的一个位**
- **所有的寄存器都必须以32位的字来访问**

# GPIO配置寄存器

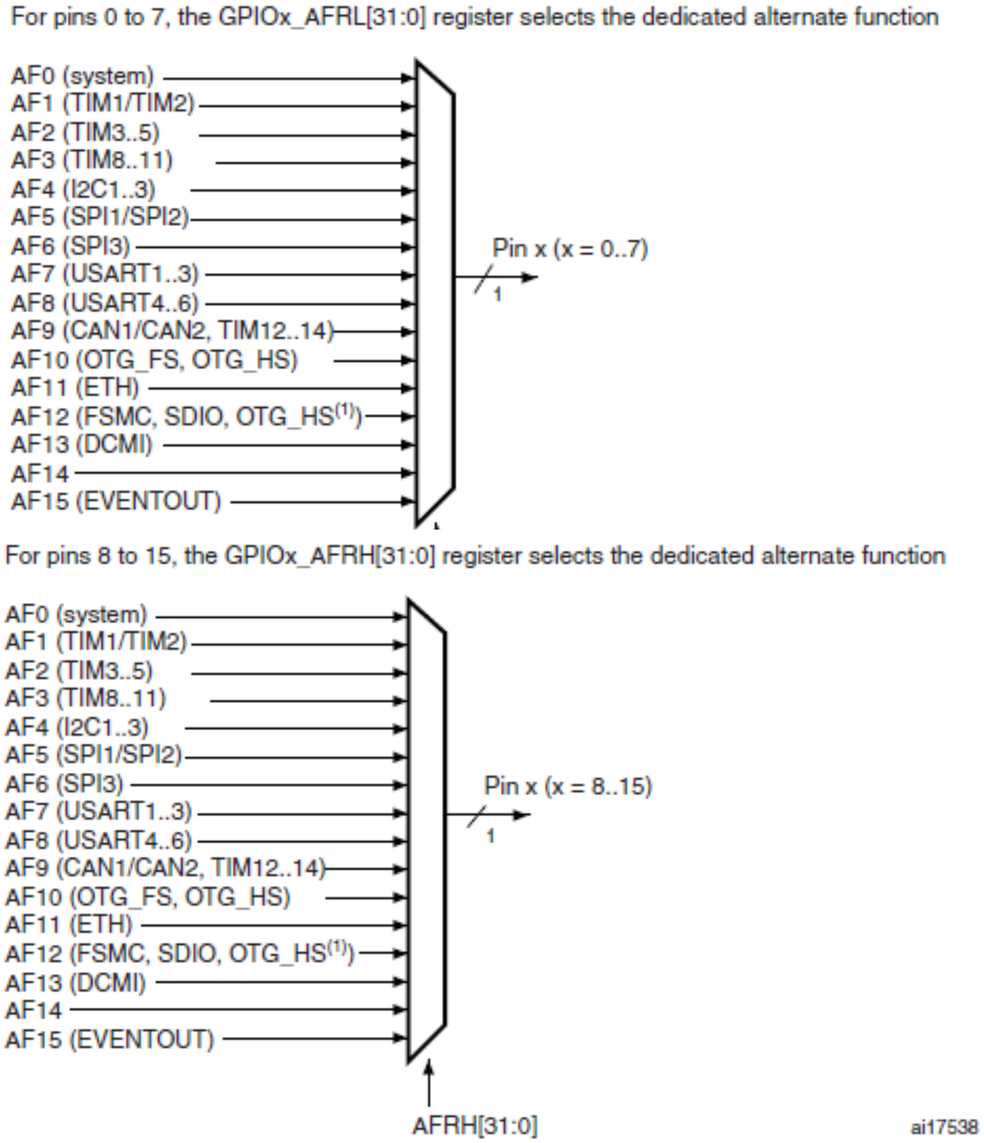
- 每一位可以独立配置
- 启动的时候把每个端口位的方向都清除为0了
- 输出模式：推挽或开漏+上拉/下拉
- 从(GPIOx_ODR)输出数据寄存器输出数据，或使用外设（可选功能输出）
- 输入状态：悬空、上拉/下拉、模拟
- 输入数据到“输入数据寄存器”(GPIOx_IDR)或外设（可选功能输入）

| MODER(i) [1:0] | OTYPER(i) | OSPEEDR(i) [B:A] | | PUPDR(i) [1:0] | | I/O configuration | |
|---|---|---|---|---|---|---|---|
| 01 | 0 | SPEED [B:A] | | 0 | 0 | GP output | PP |
| | 0 | | | 0 | 1 | GP output | PP + PU |
| | 0 | | | 1 | 0 | GP output | PP + PD |
| | 0 | | | 1 | 1 | Reserved | |
| | 1 | | | 0 | 0 | GP output | OD |
| | 1 | | | 0 | 1 | GP output | OD + PU |
| | 1 | | | 1 | 0 | GP output | OD + PD |
| | 1 | | | 1 | 1 | Reserved (GP output OD) | |
| 10 | 0 | SPEED [B:A] | | 0 | 0 | AF | PP |
| | 0 | | | 0 | 1 | AF | PP + PU |
| | 0 | | | 1 | 0 | AF | PP + PD |
| | 0 | | | 1 | 1 | Reserved | |
| | 1 | | | 0 | 0 | AF | OD |
| | 1 | | | 0 | 1 | AF | OD + PU |
| | 1 | | | 1 | 0 | AF | OD + PD |
| | 1 | | | 1 | 1 | Reserved | |
| 00 | x | x | x | 0 | 0 | Input | Floating |
| | x | x | x | 0 | 1 | Input | PU |
| | x | x | x | 1 | 0 | Input | PD |
| | x | x | x | 1 | 1 | Reserved (input floating) | |
| 11 | x | x | x | 0 | 0 | Input/output | Analog |
| | x | x | x | 0 | 1 | Reserved | |
| | x | x | x | 1 | 0 | | |
| | x | x | x | 1 | 1 | | |

1. GP = general-purpose, PP = push-pull, PU = pull-up, PD = pull-down, OD = open-drain, AF = alternate function.

# 可选功能选择寄存器

- **在AF模式，AFRL或AFRH需要配置过才能由特定的外设驱动引脚**
- **可以看作是复用器的选择信号**
- EVENTOUT不能被映射到以下 I/O引脚： PC13、PC14、PC15、PH0、PH1 和PI8.



# 代码风格和按位访问

- **直接用二进制和十六进制的常数容易犯错**
  - “要设置第13和19位，用0000 0000 0000 1000 0010 0000 0000 0000 或 0x00082000”

- **用位移位来产生常数值**

```
n = (1UL << 19) | (1UL << 13);
```

- **为所需的位定义名字**

```
#define POS_0 (13)
#define POS_1 (19)
n = (1UL << POS_0) | (1UL << POS_1);
```

- **建立宏MASK来做移位产生屏蔽字**

```
#define MASK(x) (1UL << (x))
n = MASK(POS_0) | MASK(POS_1);
```

# 使用MASK宏

- 用屏蔽字覆盖已有的值

  n = MASK(foo);

- 置屏蔽字中为1的位为1，其他位不变

  n |= MASK(foo);

- 计算屏蔽字的补码

  ~MASK(foo);

- 清除屏蔽字中为0的位为0，其他位不变

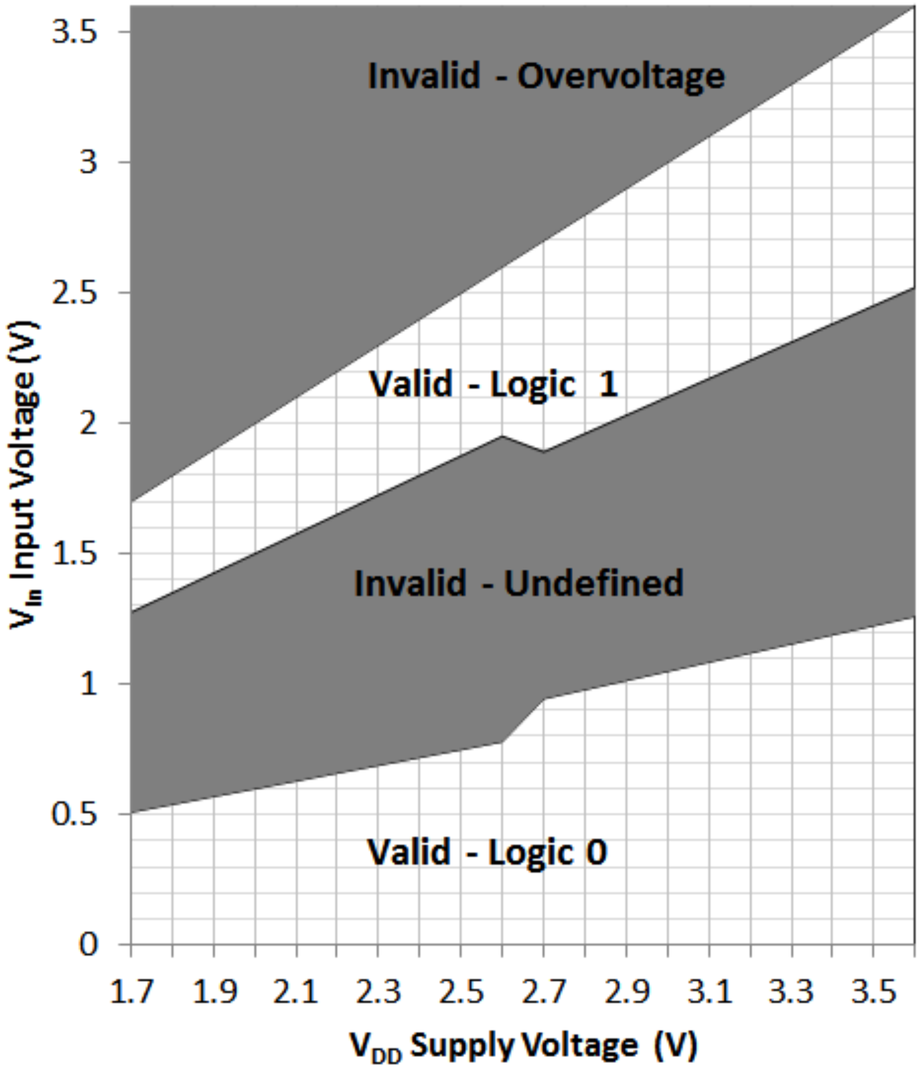
  n &= MASK(foo);

- 清除屏蔽字中为1的位为0，其他位不变

  n &= ~MASK(foo);

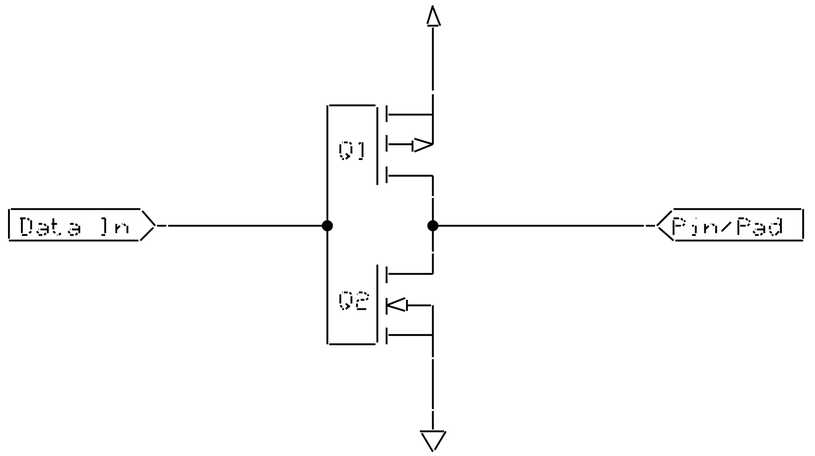
# 输入：什么是1？0呢?

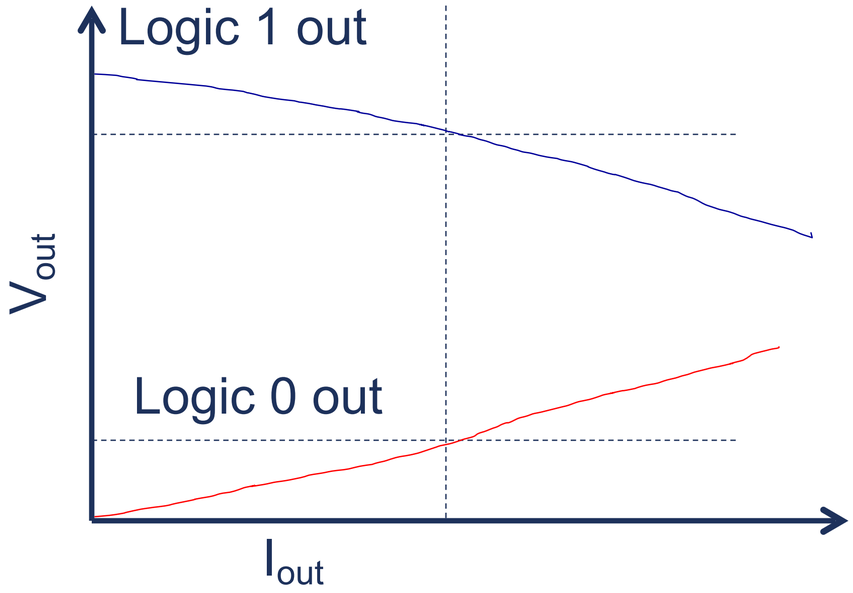
- 输入信号的值是由电压决定的
- 输入阈值电压是由供电电压$V_{DD}$所决定的
- 超过$V_{DD}$ 或 GND 可能会损坏芯片



# 输出：什么是1？0呢?

- **正常输出电压**
  - 1： $V_{DD}$-0.5 V 到 $V_{DD}$
  - 0： 0 to 0.5 V
- **注意：输出电压受引脚上的负载所汲取的电流的影响**
  - 需要考虑晶体管里的源极到漏极的电阻
  - 上述值只是当电流<5mA（对于高驱动能力引脚是18mA）并且$V_{DD}$ > 2.7V时才是有意义的



# GPIO

- 对于一个GPIO引脚，最基本要做的事情是和它是用作输入还是输出有关。如果是输入，要看它所连接的电信号是否有效，而如果是输出，要设置它为有效或无效。
- 更高级的使用方式还可以硬件监视一个输入引脚的状态，当状态变化的时候中断计算机

# 软件中的GPIO

- 允许或禁止一个GPIO引脚；
- 设置何种信号电平为“有效”；
- 决定引脚是输入还是输出；
- 向一个引脚写一个值；
- 从引脚读一个值；
- 设置哪种边沿会产生输入中断；
- 等待中断发生。

# 性能问题

- 信号变化的速度或者说频率。在理想世界里，一个引脚可以达到的最高频率只是由电特性所决定的，就是由GPIO电路的特性和它所驱动的负载或感应的谐振能力所决定的。不过，很可能在信号处理过程中的软件也会实际上造成频率的限制。软件所花的时间会限制每秒能变化的次数，或每秒能检测到的变化。

- ARM使用AHB总线，所以GPIO上的切换速度只有MHz级别

# Linux内核驱动程序

- 驱动程序实现了一套C的函数，内核的其他部分可以用它们来建立使用GPIO的某种具体的硬件设备的驱动程序
- 这个级别的程序在读写一个引脚的时候具有非常低的延迟，但是也受制于系统中断延迟的程度。服务于硬件其他部分的中断响应程序会造成可能的延迟
- http://www.kernel.org/doc/Documentation/gpio.txt

# 应用级别的支持

- 可以通过sysfs目录中特殊的文件来控制GPIO引脚。这些文件不是真实的和磁盘上的数据有关的文件，当读写这些文件的时候，内核中的程序会被调用来实现用户所需的功能。

- 启用、停用、读写、设置何种信号电平为有效以及决定何种边沿会触发中断，是由对这些文件的读和写这样的普通文件操作来实现的。等待一个中断，是由标准的Unix类操作系统一般都支持的“poll”或“select”这样的函数，来通知用户在某个已经打开的文件上发生了任何异常状况的。

# /sys的局限

- 这个级别的程序会受到两种额外的延时和延迟的影响。一种是调度延迟。这实际上是Linux在别的应用程序正在使用处理器时，让另一个应用程序获得处理器所需的时间。另一种是系统负载，当其他更重要的应用程序需要使用处理器时，系统会不让你的程序使用处理器。在这点上还有一些控制可以做，因为你的程序的重要性是可以用“nice”命令来改变的。不过，通常这两种延迟主要都是在等待一个中断处理完毕时发生的。

# /sys/class/gpio

- export：写入引脚编号表明启用
- unexport：写入引脚编号来停用

- 编号和实际CPU引脚的关系：Linux移植者决定

# 启用后

- active_low：0/1；是否低电平有效
- direction：in/out
- edge：none、rising、falling或both；中断触发类型
- value：读当前值；select来等待中断

# 直接访问寄存器

- AHB上的寄存器是挂在总线上的，e.g.每个寄存器有一个32位的地址
- 在用户模式下，MMU的设置使得应用程序无法访问这些寄存器
- 内核程序，如设备驱动程序可以访问
- 裸机程序可以直接访问
- 应用程序经过特殊设置也可以让某块内存映射到AHB上

# Arduino-ish

- An Arduino wiring-like library written in C and released under the GNU LGPLv3 license which is usable from C and C++ and many other languages with suitable wrappers

# blink LED

```
pinMode (0, OUTPUT) ;

for (;;)
{
    digitalWrite (0, 1) ;       // On
    delay (500) ;               // mS
    digitalWrite (0, 0) ;       // Off
    delay (500) ;
  }
```

# APIs

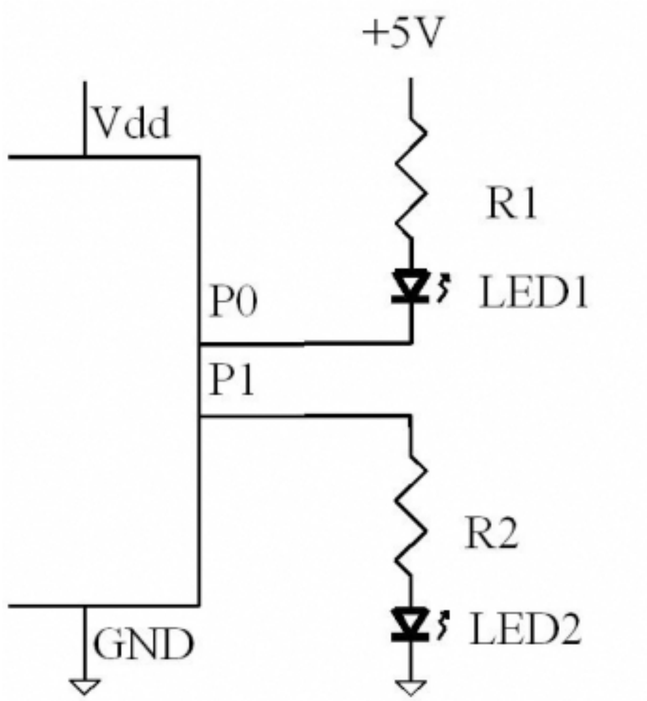
- wiringPiSetup(void) ;
- void pinMode (int pin, int mode) ;
- void digitalWrite (int pin, int value) ;
- void digitalWriteByte (int value) ;
- void pwmWrite (int pin, int value) ;
- int digitalRead (int pin) ;
- void pullUpDnControl (int pin, int pud) ;
- unsigned int millis (void);
- void delay (unsigned int howLong);
- void delayMicroseconds (unsigned int howLong);

# 安装Arduino-ish

- git clone https://github.com/pcduino/c_enviroment
- cd c_enviroment
- make
- #include <core.h>

# 输出HIGH和LOW

- 扇出：输出HIGH时，电流从CPU的Vcc流入，经过引脚输出给外部
- 灌入：输出LOW时，电流从外部的Vcc流入，经过引脚后，直接到地
- 对于某些CPU，使用灌入方式，整体能驱动更大的设备
- 工业界一般采用灌入



# 串行通信

嵌入式系统

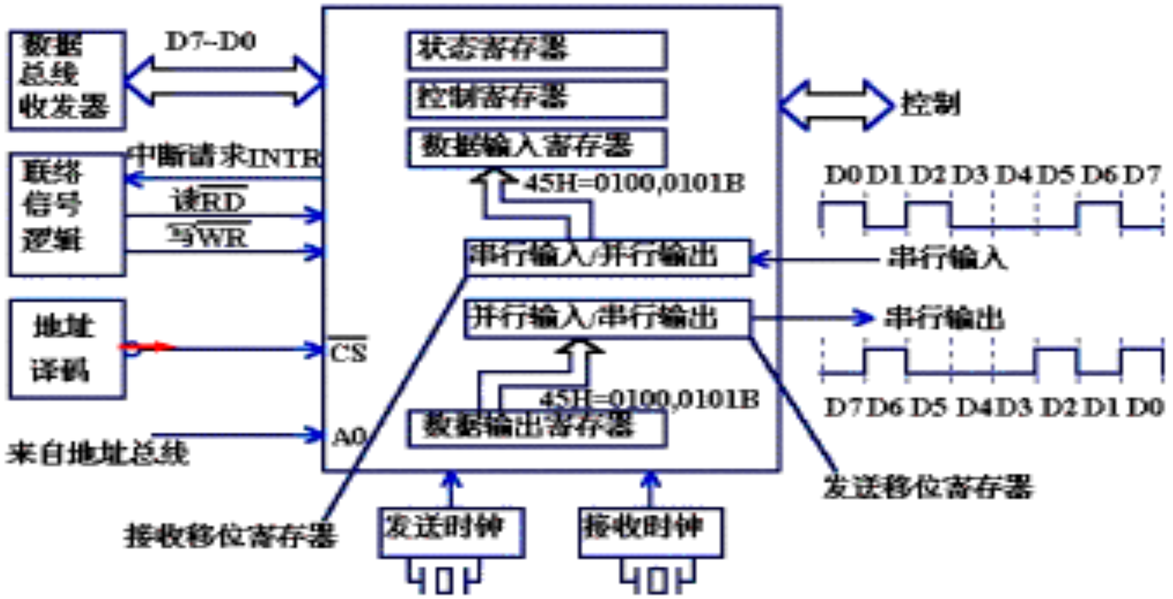
# 通信

- 与外界的信息交换称为通信（讯）。
- 基本的通信方式有并行通信和串行通信两种。
- 一条信息的各位数据被同时传送的通信方式称为并行通信。
- 并行通信的特点是：各数据位同时传送，传送速度快、效率高，但有多少数据位就需多少根数据线，因此传送成本高，且只适用于近距离（相距数米）的通信。

# 串行通信的概念


- 串行通信：一条信息的各位数据被逐位按顺序传送的通信方式称为串行通信。
- 串行通信的特点是：数据位传送，传按位顺序进行，最少只需一根传输线即可完成，
- 成本低但送速度慢。串行通信的距离可以从几米到几千米。
- 根据信息的传送方向，串行通信可以进一步分为单工、半双工和全双工三种。

# 串口通信--接口电路




- 能够完成上述“串<- ->并”转换功能的电路，通常称为“通用异步收发器”（UART：Universal Asynchronous Receiver and Transmitter），
- 典型的芯片有：Intel 8250/8251,16550。

# 异步通信vs同步通信

- 异步通信是指通信的发送与接收设备使用各自的时钟控制数据的发送和接收过程。为使双方的收发协调，要求发送和接收设备的时钟尽可能一致。

# 异步通信

- 异步通信是以字符（构成的帧）为单位进行传输，字符与字符之间的间隙（时间间隔）是任意的，但每个字符中的各位是以固定的时间传送的，即字符之间是异步的（字符之间不一定有“位间隔”的整数倍的关系），但同一字符内的各位是同步的（各位之间的距离均为“位间隔”的整数倍）。

# 异步通信

- 异步通信的特点：不要求收发双方时钟的严格一致，实现容易，设备开销较小，但每个字符要附加2～3位用于起止位，各帧之间还有间隔，因此传输效率不高。

# 同步通信

- 同步通信时要建立发送方时钟对接收方时钟的直接控制，使双方达到完全同步。此时，传输数据的位之间的距离均为“位间隔”的整数倍，同时传送的字符间不留间隙，即保持位同步关系，也保持字符同步关系。

# DTE和DCE

- DTE： Data Terminal Equipment
- DCE： Data Communication Equipment

# 速率

- 比特率是每秒钟传输二进制代码的位数，单位是：位 / 秒（bps）。如每秒钟传送240个字符，而每个字符格式包含10位(1个起始位、1个停止位、8个数据位)，这时的比特率为：
  - 10位×240个/秒 = 2400 bps
- 波特率表示每秒钟调制信号变化的次数，单位是：波特（Baud）。
- 波特率和比特率不总是相同的，对于将数字信号1或0直接用两种不同电压表示的所谓基带传输，比特率和波特率是相同的。所以，我们也经常用波特率表示数据的传输速

# 波特率

| 波特率 (bps) | 1 号电缆传输距离（英尺） AWG#22带屏蔽 | 2 号电缆传输距离（英尺） AWG#22不带屏蔽 |
|---|---|---|
| 110 | 5000 | 3000 |
| 300 | 5000 | 3000 |
| 1200 | 3000 | 3000 |
| 2400 | 1000 | 500 |
| 4800 | 1000 | 250 |
| 9600 | 250 | 250 |

# 速度与距离

- 串行接口或终端直接传送串行信息位流的最大距离与传输速率及传输线的电气特性有关。当传输线使用每0.3m（约1英尺）有50pF电容的非平衡屏蔽双绞线时，传输距离随传输速率的增加而减小。当比特率超过1000 bps 时，最大传输距离迅速下降，如9600 bps 时最大距离下降到只有76m（约250英尺）。

# 流控制

- 解决丢失数据的问题
- 硬件流控制
  - 硬件流控制常用的有RTS/CTS（请求发送/清除发送）流控制和DTR/DSR（数据终端就绪/数据设置就绪）流控制
- 软件流控制
  - 一般通过XON/XOFF来实现软件流控制。

# 奇偶校验

- 奇校验：所有传送的数位（含字符的各数位和校验位）中，“1”的个数为奇数，如：
  - 1 0110，0101
  - 0 0110，0001
- 偶校验：所有传送的数位（含字符的各数位和校验位）中，“1”的个数为偶数，如：
  - 1 0100，0101
  - 0 0100，0001

# RS232

- EIA-RS-232C对电器特性、逻辑电平和各种信号线功能都作了规定。
- 在TxD和RxD上：
  - 逻辑1(MARK) =-3V～-15V
  - 逻辑0(SPACE)=+3～＋15V
- 在RTS、CTS、DSR、DTR和DCD等控制线上：
  - 信号有效（接通，ON状态，正电压）＝+3V～+15V
  - 信号无效（断开，OFF状态，负电压) = -3V～-15V

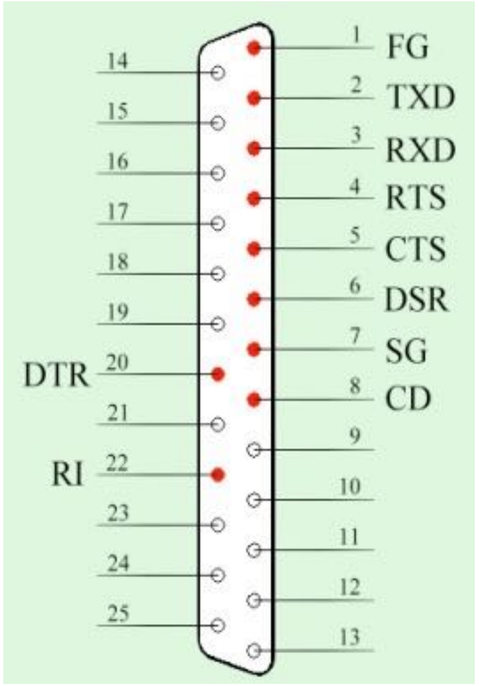
# 232主要信号

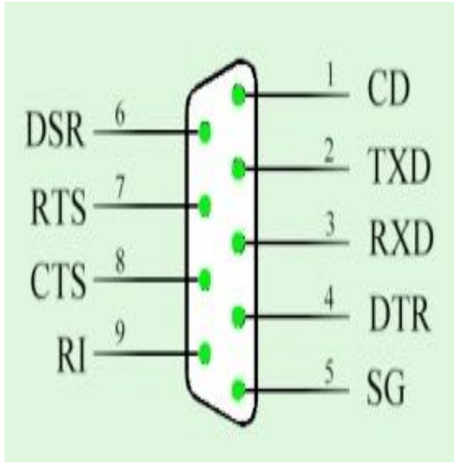
RS-232C 标准接口主要引脚定义

| 插针序号 | 信号名称 | 功能 | 信号方向 |
|---|---|---|---|
| 1 | PGND | 保护接地 | |
| 2（3） | TXD | 发送数据（串行输出） | DTE→DCE |
| 3（2） | RXD | 接收数据（串行输入） | DTE←DCE |
| 4（7） | RTS | 请求发送 | DTE→DCE |
| 5（8） | CTS | 允许发送 | DTE←DCE |
| 6（6） | DSR | DCE 就绪（数据建立就绪） | DTE←DCE |
| 7（5） | SGND | 信号接地 | |
| 8（1） | DCD | 载波检测 | DTE←DCE |
| 20（4） | DTR | DTE 就绪（数据终端准备就绪） | DTE→DCE |
| 22（9） | RI | 振铃指示 | DTE←DCE |

注：插针序号（）内为 9 针非标准连接器的引脚号

# 连接器的机械特性




这里是DTE的标识

# 基本接线方法

| | 9针串口（DB9） | | | 25针串口（DB25） | |
|---|---|---|---|---|---|
| 针号 | 功能说明 | 缩写 | 针号 | 功能说明 | 缩写 |
| 1 | 数据载波检测 | DCD | 8 | 数据载波检测 | DCD |
| 2 | 接收数据 | RXD | 3 | 接收数据 | RXD |
| 3 | 发送数据 | TXD | 2 | 发送数据 | TXD |
| 4 | 数据终端准备 | DTR | 20 | 数据终端准备 | DTR |
| 5 | 信号地 | GND | 7 | 信号地 | GND |
| 6 | 数据设备准备好 | DSR | 6 | 数据准备好 | DSR |
| 7 | 请求发送 | RTS | 4 | 请求发送 | RTS |
| 8 | 清除发送 | CTS | 5 | 清除发送 | CTS |
| 9 | 振铃指示 | DELL | 22 | 振铃指示 | DELL |

# 接口信号DSR DTR

- 数据装置准备好（Data set ready-DSR)——有效时（ON）状态，表明通信装置处于可以使用的状态。
- 数据终端准备好(Data set ready-DTR)——有效时（ON）状态，表明数据终端可以使用。
- 这两个信号有时连到电源上，一上电就立即有效。这两个设备状态信号有效，只表示设备本身可用，并不说明通信链路可以开始进行通信了，能否开始进行通信要由下面的控制信号决定。

# 接口信号RTS CTS

- 请求发送(Request to send-RTS)——用来表示DTE请求DCE发送数据，即当终端要发送数据时，使该信号有效（ON状态），向MODEM请求发送。它用来控制MODEM是否要进入发送状态。
- 允许发送（Clear to send-CTS）——用来表示DCE准备好接收DTE发来的数据，是对请求发送信号RTS的响应信号。当MODEM已准备好接收终端传来的数据，并向前发送时，使该信号有效，通知终端开始沿发送数据线TxD发送数据。
- 这对RTS/CTS请求应答联络信号是用于半双工MODEM系统中发送方式和接收方式之间的切换。在全双工系统中作发送方式和接收方式之间的切换。在全双工系统中，因配置双向通道，故不需要RTS/CTS联络信号，使其变高。

# 单工、半双工和全双工

- 如果在通信过程的任意时刻，信息只能由一方A传到另一方B，则称为单工。
- 如果在任意时刻，信息既可[illegible]，又能由B传A，但[illegible]方向上的传输存在[illegible]只工传输。
- 如果在任意时刻，线路上存在A到B和B到A的双向信号传输，则称为全双工。

小心！无线通信领域的定义与此不同！

# 接口信号DCD RI

- 接收线信号检出(Received Line detection-RLSD)——用来表示DCE已接通通信链路，告知DTE准备接收数据。当本地的MODEM收到由通信链路另一端（远地）的MODEM送来的载波信号时，使RLSD信号有效，通知终端准备接收，并且由MODEM将接收下来的载波信号解调成数字两数据后，沿接收数据线RxD送到终端。此线也叫做数据载波检出(Data Carrier dectection-DCD）线。

- 振铃指示(Ringing-RI)——当MODEM收到交换台送来的振铃呼叫信号时，使该信号有效（ON状态），通知终端，已被呼叫。

# TxD和RxD

- 发送数据(Transmitted data-TxD)——通过TxD终端将串行数据发送到MODEM,(DTE→DCE)。
- 接收数据(Received data-RxD)——通过RxD线终端接收从MODEM发来的串行数据,(DCE→DTE)。

# RS232的不足

- 传输距离短，传输速率低：RS-232C总线标准受电容允许值的约束，使用时传输距离一般不要超过15米（线路条件好时也不超过几十米）。最高传送速率为20Kbps
- 有电平偏移：RS-232C总线标准要求收发双方共地。通信距离较大时，收发双方的地电位差别较大，在信号地上将有比较大的地电流并产生压降。
- 抗干扰能力差：RS-232C在电平转换时采用单端输入输出，在传输过程中当干扰和噪声混在正常的信号中。为了提高信噪比，RS-232C总线标准不得不采用比较大的电压摆幅。

# RS422

- RS-422A输出驱动器为双端平衡驱动器。如果其中一条线为逻辑“1”状态，另一条线就为逻辑“0”，比采用单端不平衡驱动对电压的放大倍数大一倍。差分电路能从地线干扰中拾取有效信号，差分接收器可以分辨200mV以上电位差。若传输过程中混入了干扰和噪声，由于差分放大器的作用，可使干扰和噪声相互抵消。因此可以避免或大大减弱地线干扰和电磁干扰的影响。RS-422A传输速率（90Kbps）时，传输距离可达1200米。

# RS-485

- RS-485是RS-422A的变型：RS-422A用于全双工，而RS-485则用于半双工。RS-485是一种多发送器标准，在通信线路上最多可以使用32 对差分驱动器/接收器。如果在一个网络中连接的设备超过32个，还可以使用中继器。
- RS-485的信号传输采用两线间的电压来表示逻辑1和逻辑0。由于发送方需要两根传输线，接收方也需要两根传输线。传输线采用差动信道，所以它的干扰抑制性极好，又因为它的阻抗低，无接地问题，所以传输距离可达1200米，传输速率可达1Mbps。

# 面向字符的包

- 传送的数据和控制信息都必须由规定的字符集（如ASCII码）中的字符所组成。帧头为1个或2个同步字符SYN（ASCII码为16H）。SOH为序始字符（ASCII码为01H），表示标题的开始，标题中包含源地址、目标地址和路由指示等信息。STX为文始字符（ASCII码为02H），表示传送的数据块开始。数据块是传送的正文内容，由多个字符组成。数据块后面是组终字符ETB（ASCII码为17H）或文终字符ETX（ASCII码为03H）。然后是校验码。

# 面向流的包

- 将数据块看作数据流，并用序列01111110作为开始和结束标志。为了避免在数据流中出现序列01111110时引起的混乱，发送方总是在其发送的数据流中每出现5个连续的1就插入一个附加的0；接收方则每检测到5个连续的1并且其后有一个0时，就删除该0。
- 典型的面向位的同步协议如ISO的高级数据链路控制规程HDLC

# 同步通信

- 同步通信时要建立发送方时钟对接收方时钟的直接控制，使双方达到完全同步。此时，传输数据的位之间的距离均为“位间隔”的整数倍，同时传送的字符间不留间隙，即保持位同步关系，也保持字符同步关系。发送方对接收方的同步可以通过两种方法实现。

# 同步通信

- Send a separate clock line with data
- SPI (serial peripheral interface) protocol
- I2C (or I2C) protocol
- Encode a clock with data so that clock be extracted or data has guaranteed transition density with receiver clock via Phase-Locked-Loop (PLL)
- IEEE Firewire (clock encoded in data)
- USB (data has guaranteed transition density)

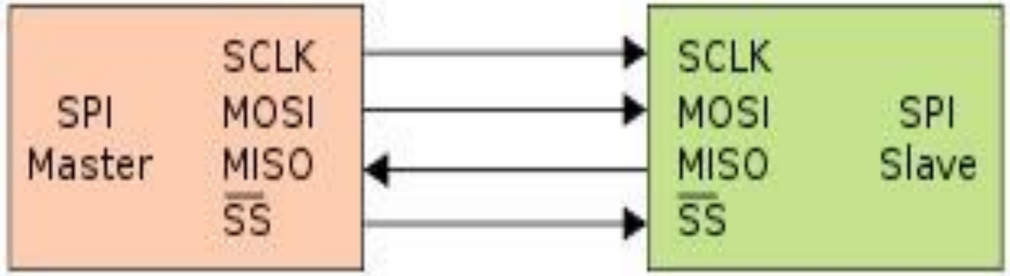
# SPI

- Serial Peripherals Interface
- 应用于外部移位寄存器、D/A、 A/D、串行EEPROM、LED显示驱动器等外部设备进行扩展

# SPI接线

- 4线：
  - MOSI－SPI从输入、主输出
  - MISO－SPI主输入、从输出
  - SPICLK －SPI时钟
  - SPICE－SPI从发送使能

# 典型接线




SPI的通信双方是不平等的

# SPI基础

- A communication protocol using 4 wires
  - Also known as a 4 wire bus
- Used to communicate across small distances
- Multiple Slaves, Single Master
- Synchronized

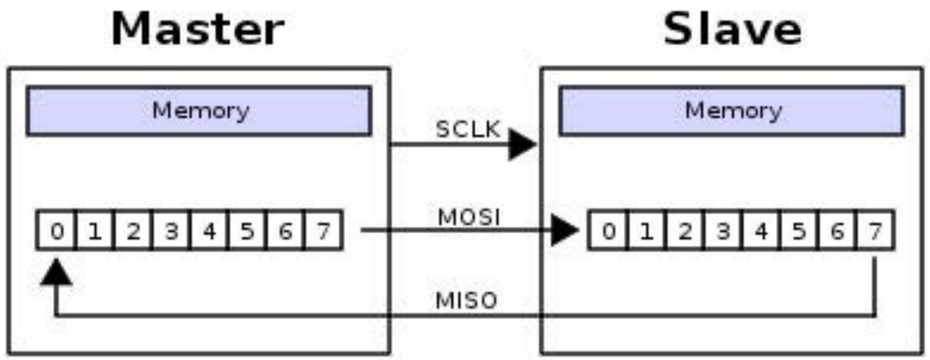
# SPI能力

- Always Full Duplex
  - Communicating in two directions at the same time
  - Transmission need not be meaningful
- Multiple Mbps transmission speed
- Transfers data in 4 to 16 bit characters
- Multiple slaves
- Daisy-chaining possible

# SPI协议

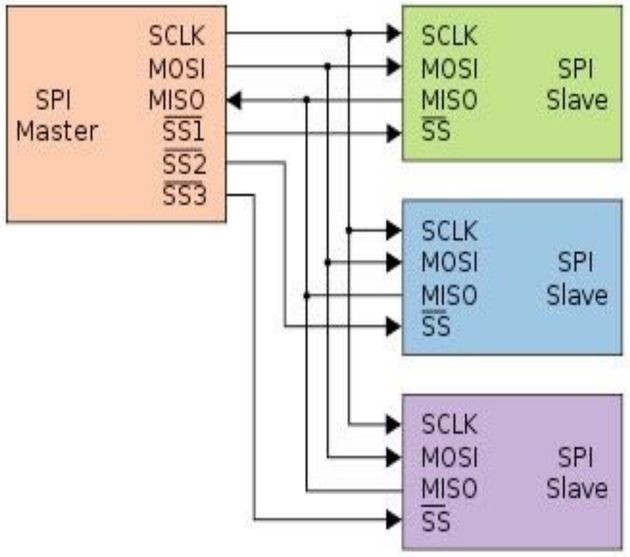
- Wires:
  - Master Out Slave In (MOSI)
  - Master In Slave Out (MISO)
  - System Clock (SCLK)
  - Slave Select 1…N
- Master Set Slave Select low
- Master Generates Clock
- Shift registers shift in and out data

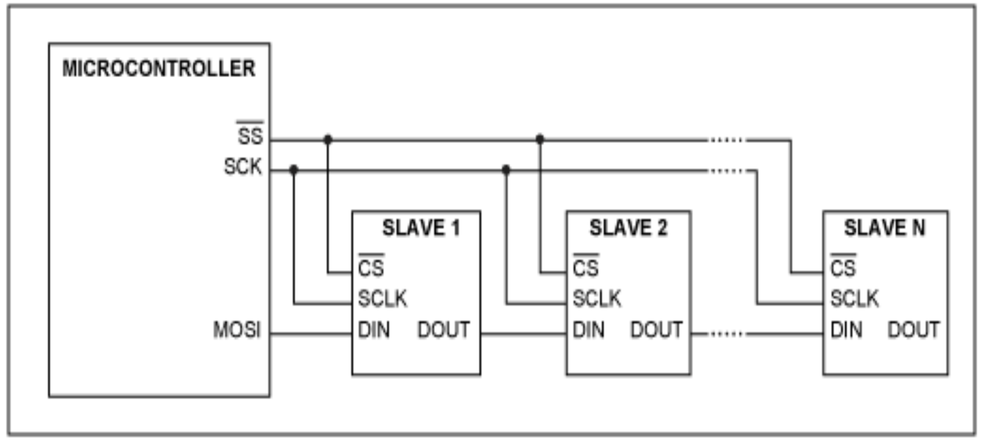
# 协议




- Master shifts out data to Slave, and shift in data from Slave

# 多客户端


SPI
Master
SCLK
MOSI
MISO
SS1
SS2
SS3
SCLK
MOSI
MISO
SS
SPI
Slave
SCLK
MOSI
MISO
SS
SPI
Slave
SCLK
MOSI
MISO
SS
SPI
Slave


MICROCONTROLLER
SS
SCK
MOSI
SLAVE 1
CS
SCLK
DIN
DOUT
SLAVE 2
CS
SCLK
DIN
DOUT
SLAVE N
CS
SCLK
DIN
DOUT

# SPI优点

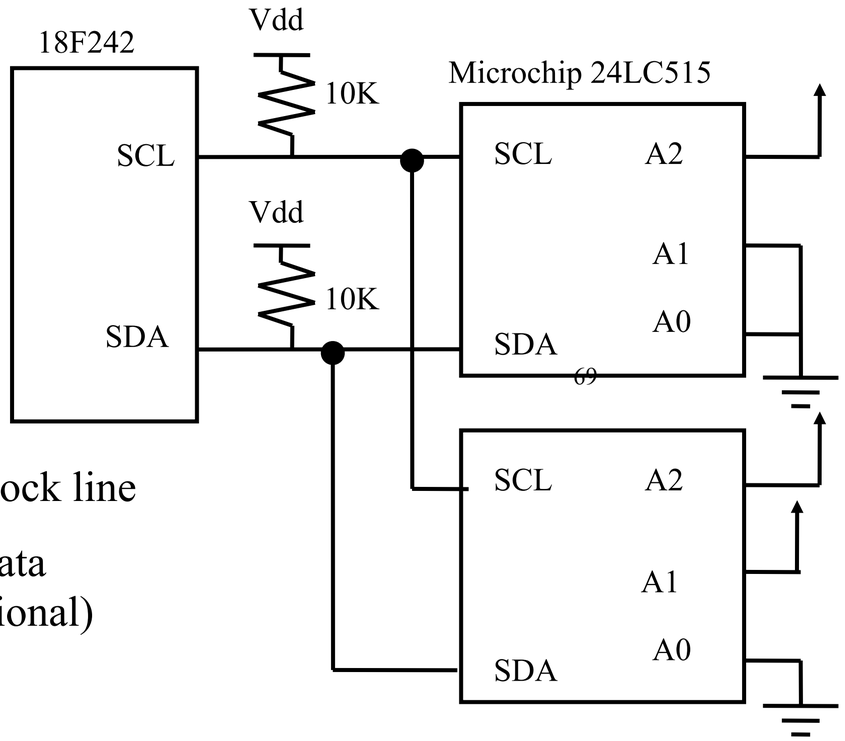
- Fast and easy
  - Fast for point-to-point connections
  - Easily allows streaming/Constant data inflow
  - No addressing/Simple to implement
- Everyone supports it

# SPI缺点

- SS makes multiple slaves very complicated
- No acknowledgement ability
- No inherent arbitration
- No flow control

# I2C总线

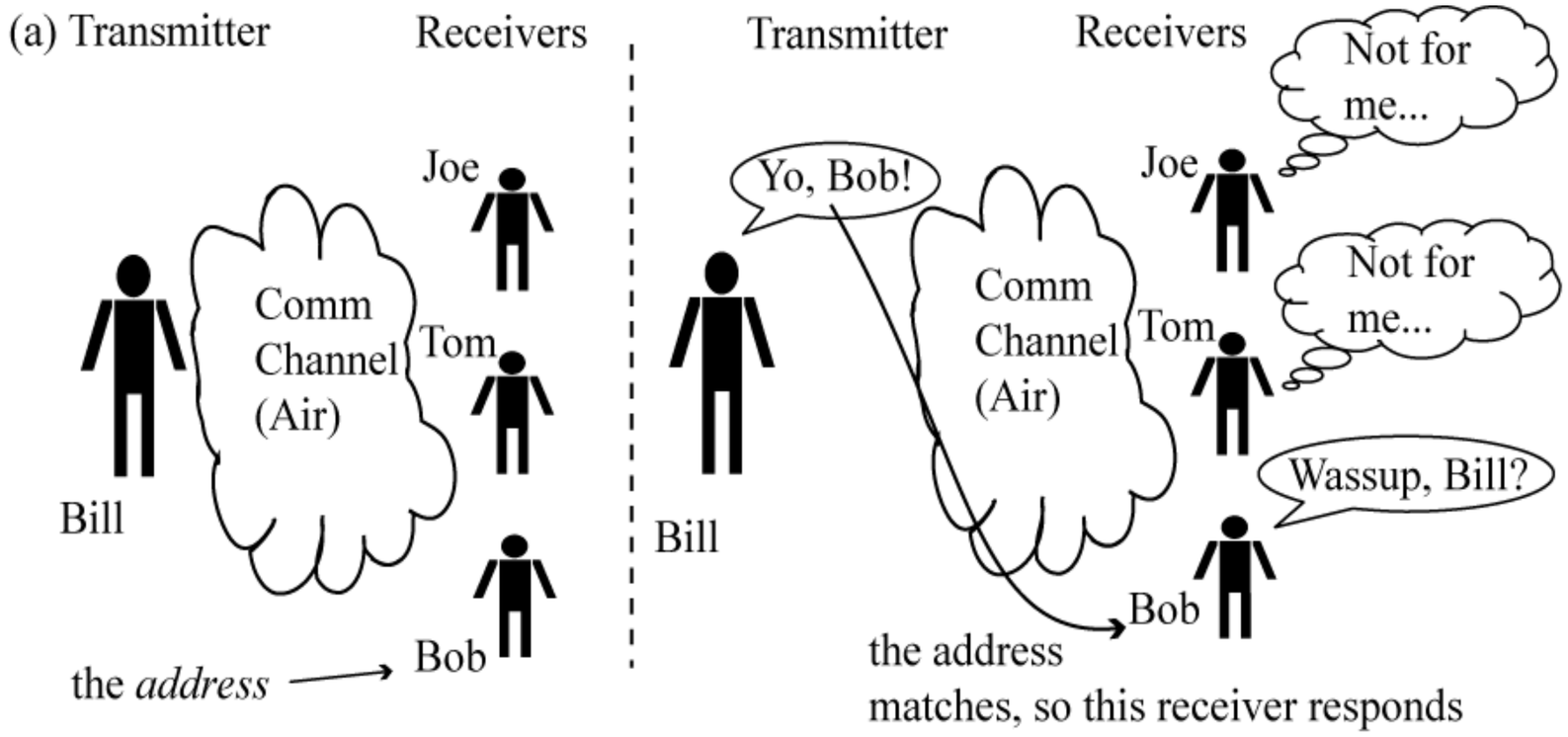
I2C is a two wire serial interface.



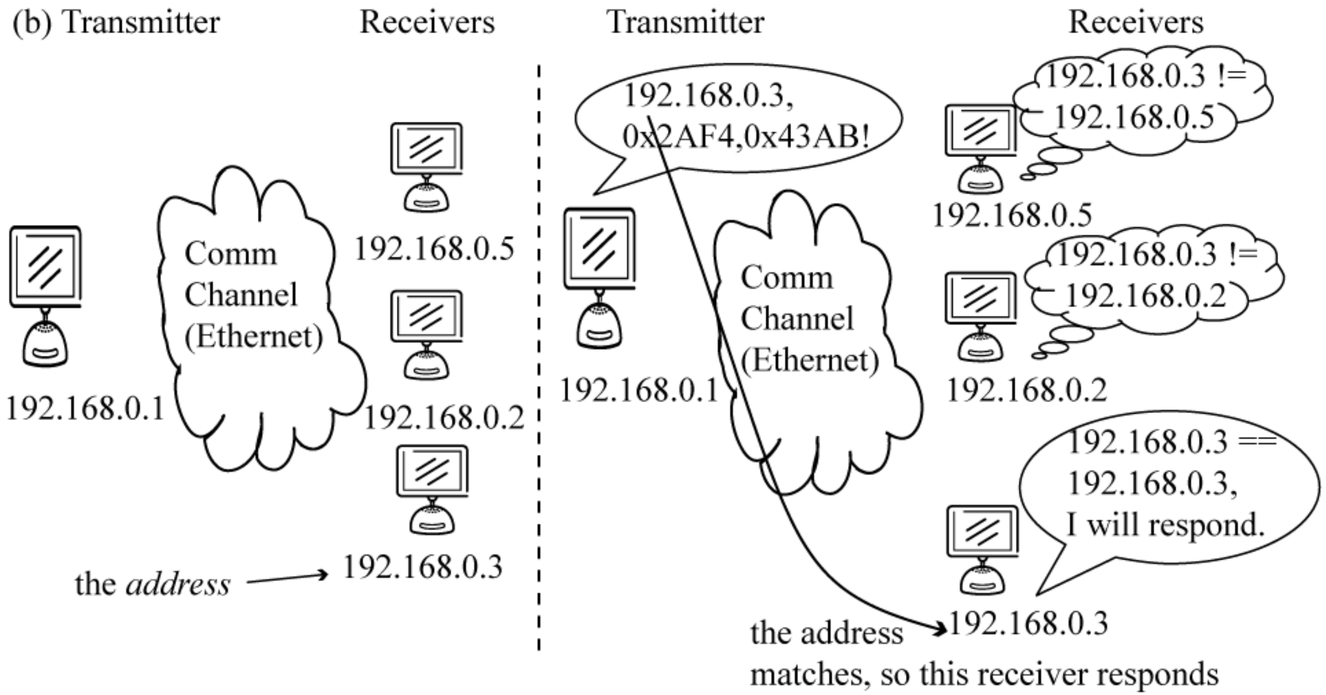

SCL – clock line

SDA – data (bidirectional)

# 总线?



# 以太网总线

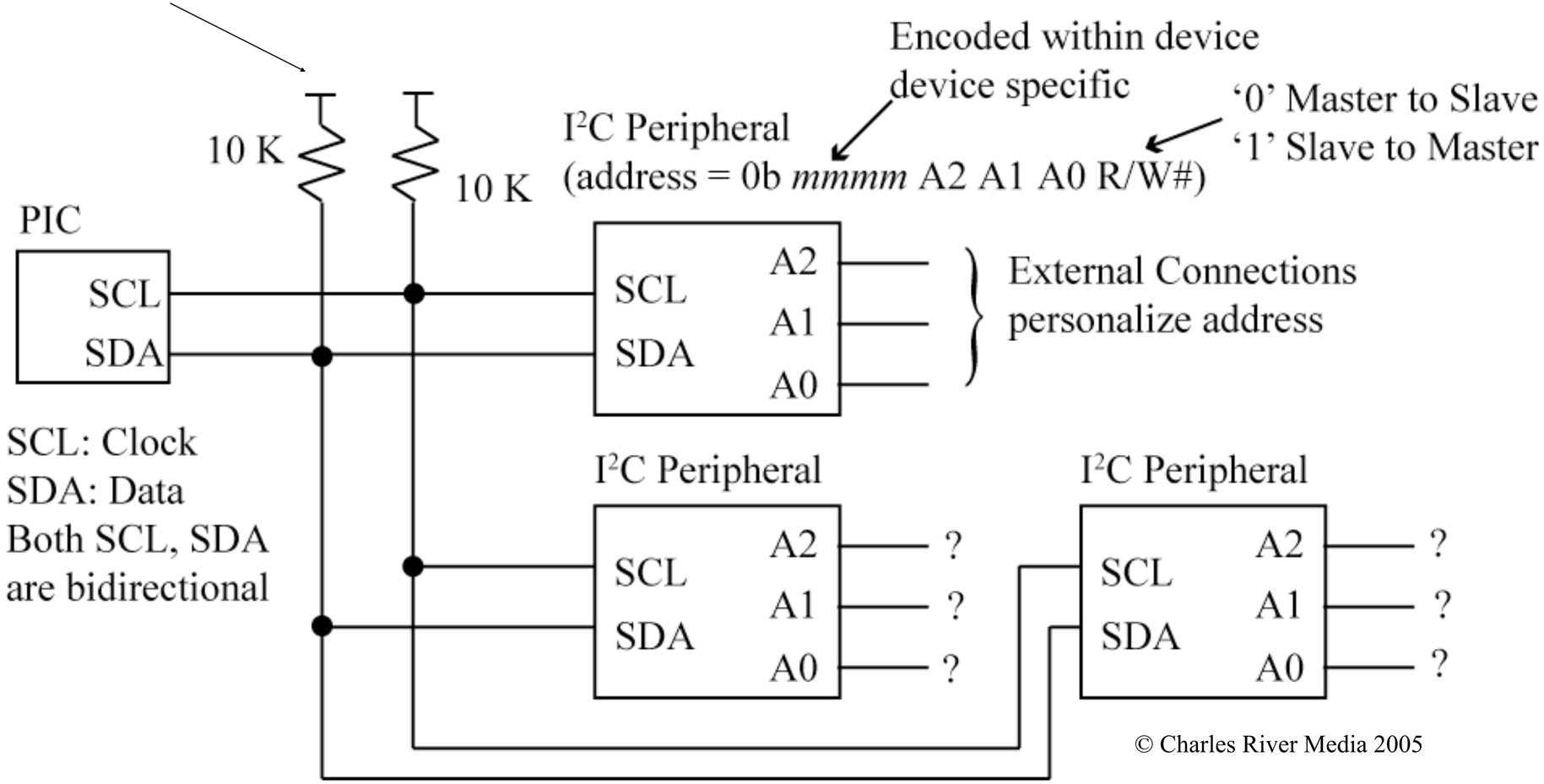


# I2C

- Multiple receivers do not require separate select lines as in SPI
- At  start of each I2C transaction a 7-bit device address is sent
- Each device listens – if device address matches internal address, then device responds
- SDA (data line) is bidirectional, communication is half duplex
- SDA, SCLK are open-drain, require external pullups
- Allows multiple bus masters

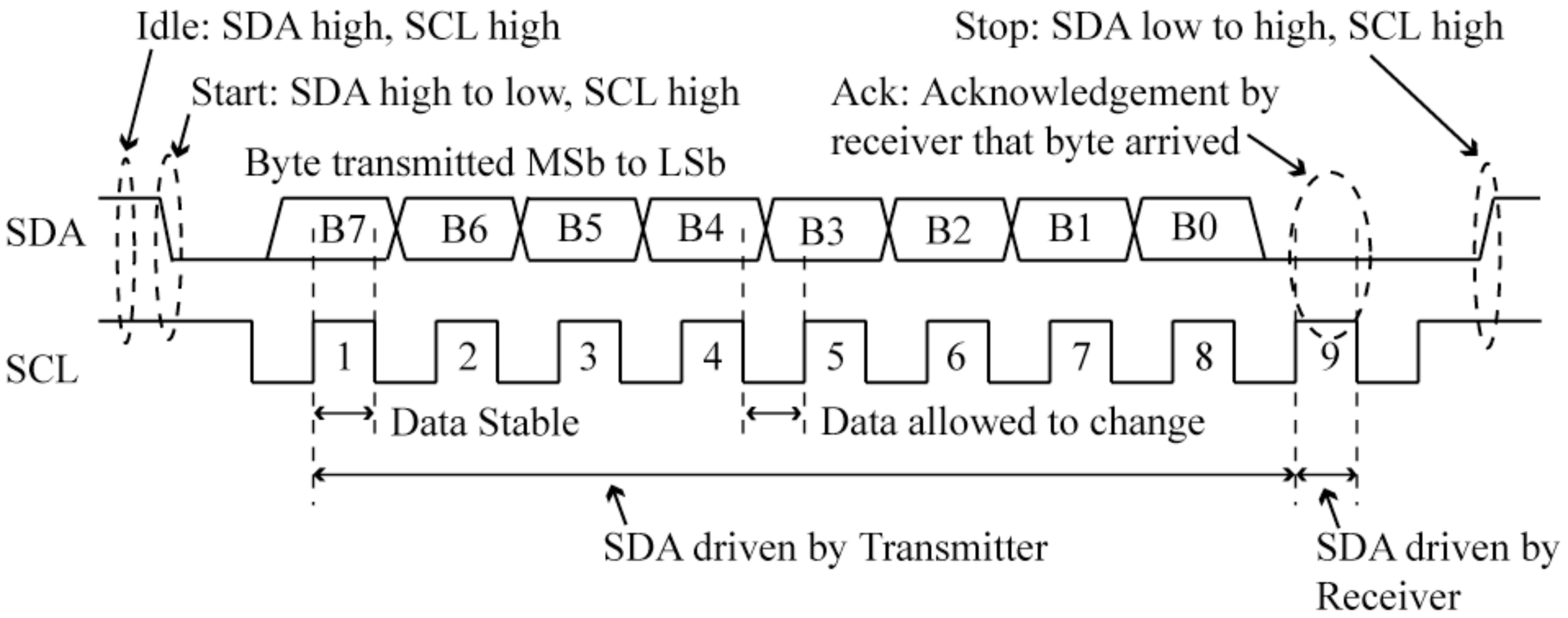
pullups are needed

I2C Bus Addressing



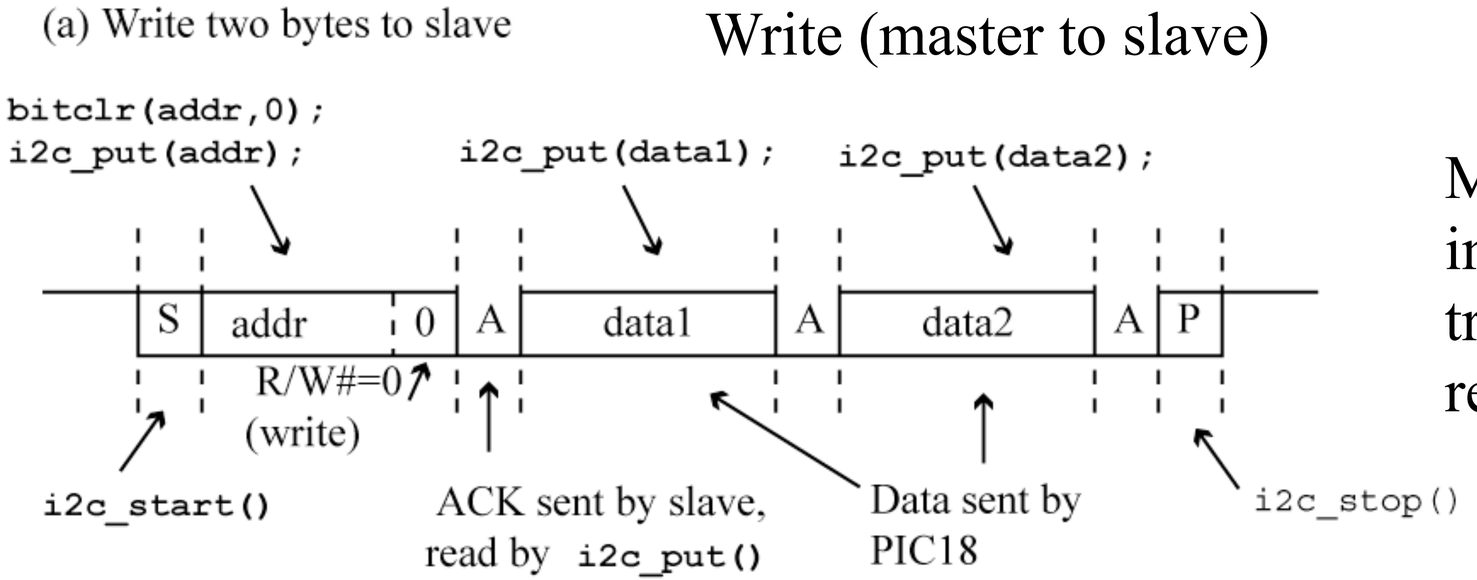

No chip selects needed!!!!!

# I2C Bus Transfer



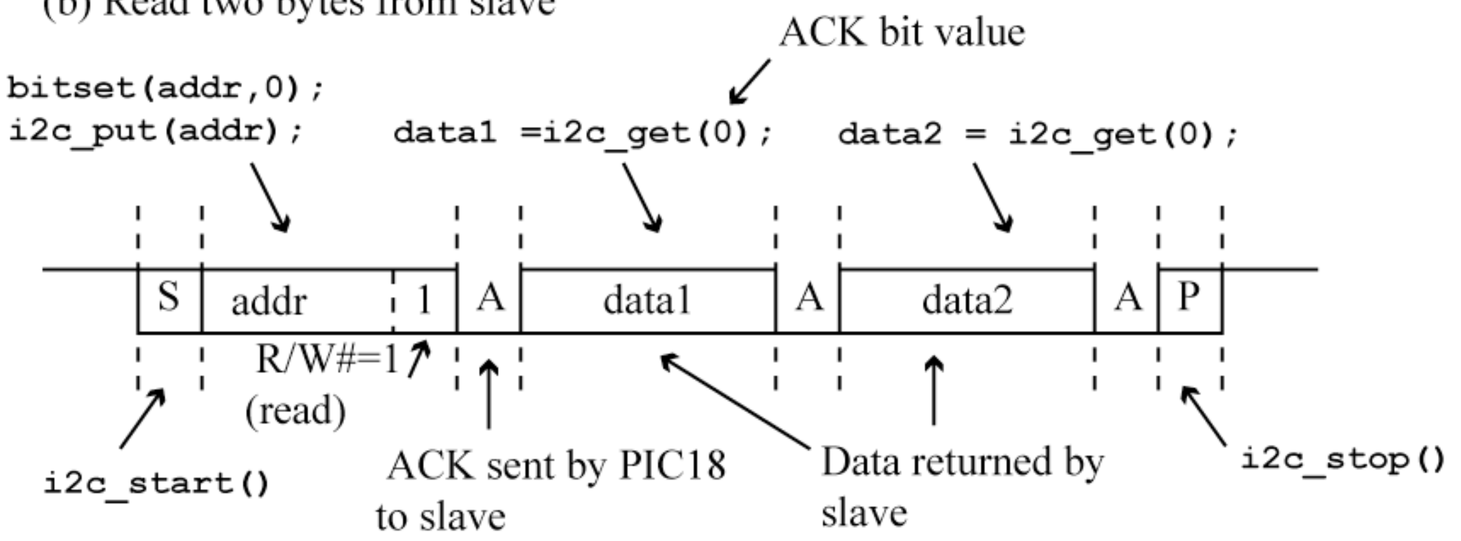

Multiple bytes sent in a transaction; every 8 bits has a 9th bit that is an acknowledge.

## Write (master to slave)




Master initiates all transactions, read or write.

## Read (master from slave)



# Example: I²C Serial EEPROM

Will use the Microchip 24LC515 Serial EEPROM to discuss I²C operation.

The 24LC515 is a 64K x 8 memory. This would require 16 address lines, and 8 data lines if a parallel interface was used, which would exceed the number of available IO pins our PIC18F242!!!

Putting a serial interface on a memory device lowers the required pin count.

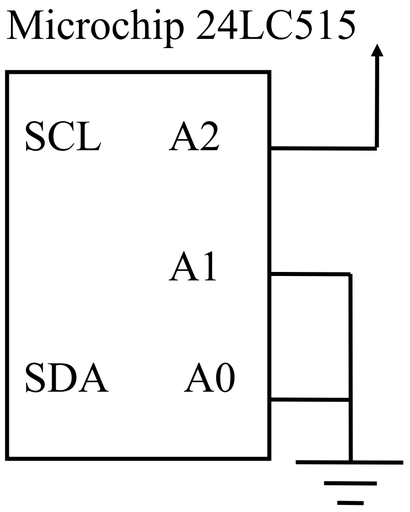
Reduces speed since data has to be sent serially, but now possible to add significant external storage to a low pin-count micro controller.

# I²C Device Addressing

Each I2C device has either a 7-bit or 10-bit device address.

We will use an I2C EEPROM and an I2C DAC (Digital-to-Analog Converter, MAX517) in lab. Both of these devices have a 7-bit address.

Upper four bits are assigned by device manufacturer and are hardcoded in the device. Lower three bits are used in different ways by manufacturer.




LC515 control byte (contains slave address):

| 7 6 5 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| 1 0 1 0 | B0 | A1 | A0 | R/$\overline{W}$ |

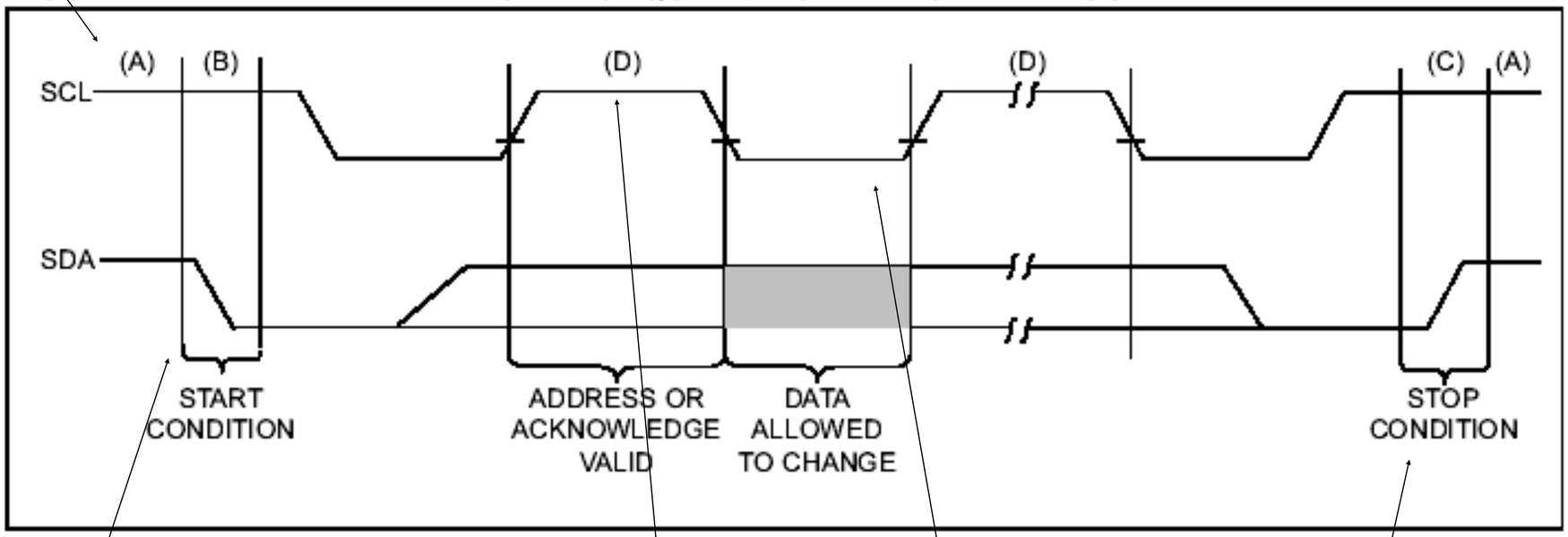
R/$\overline{W}$ = 1 for read, 0 for write.

'B0' is block select (upper/lower 32K). A1, A0 are chip selects, four devices on one bus.

# I2C Transmission

IDLE: SCL, SDA high.

FIGURE 4-1: DATA TRANSFER SEQUENCE ON THE SERIAL BUS




START: high to low transition on SDA while SCL high.

Valid data: While clock is high, data valid and stable.

Data changes while clock is low.

STOP: low to high transition on SDA while SCL high.

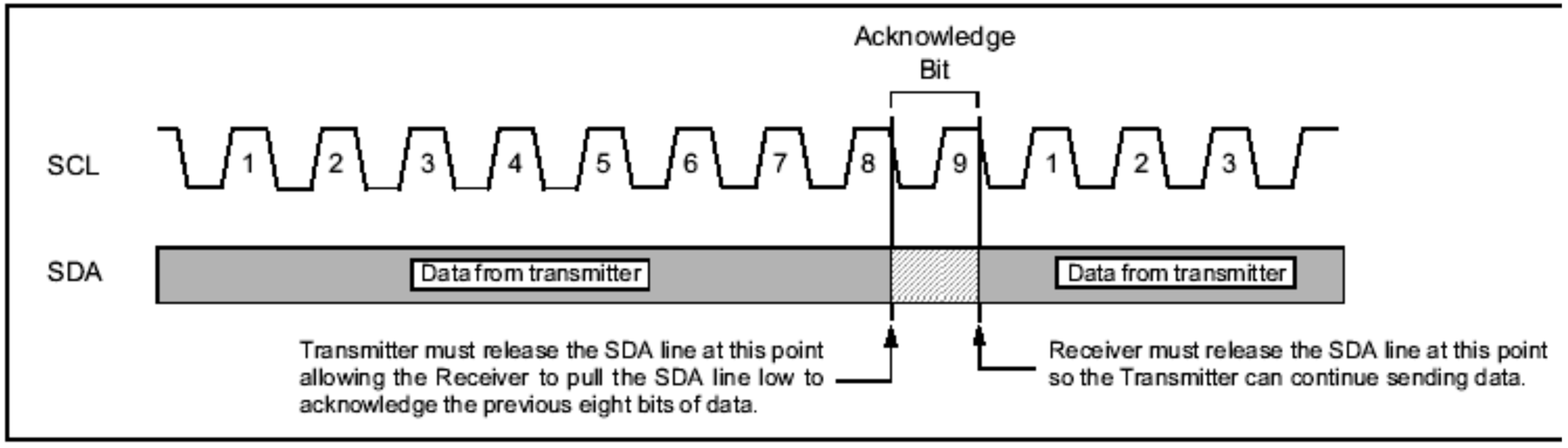
# Acknowledgement

ACK sent by slave after every 8-bits received. Master releases line (stops driving), samples line on next clock.

Slave MUST pull line low. If Master does not detect ACK, then sets error bit. If Slave does not pull line low, the pullup resistors will pull the line low.

Most common cause of ACK error – incorrect device address.

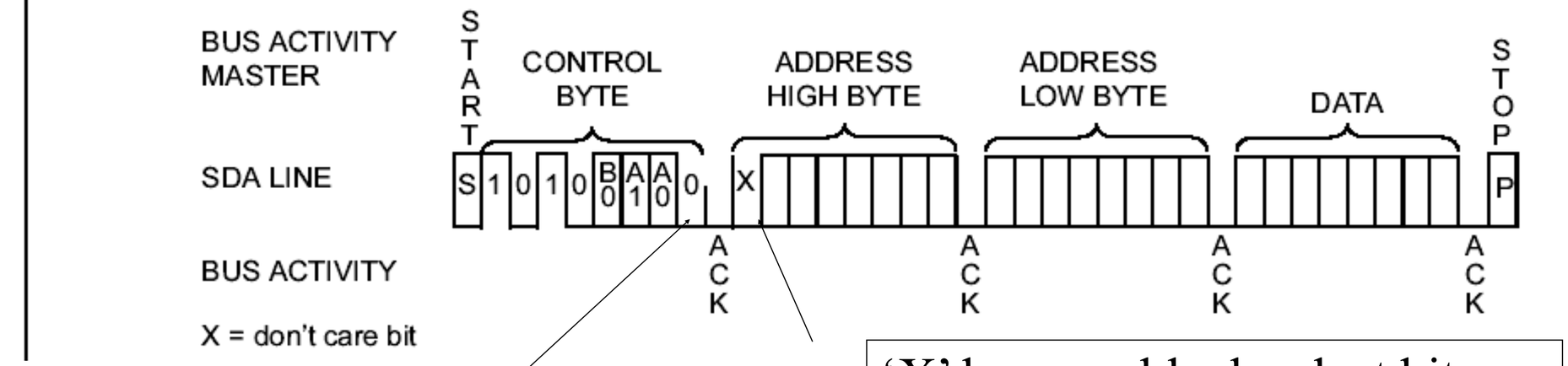
**FIGURE 4-2: ACKNOWLEDGE TIMING**



# Byte Write Operation

- Byte Write: used to write one byte
  - Send Control Byte, High address byte, low address byte, data.
  - After data is sent, takes 5 ms for write to complete
  - SLOOOOOWWWWW....

**FIGURE 6-1: BYTE WRITE**



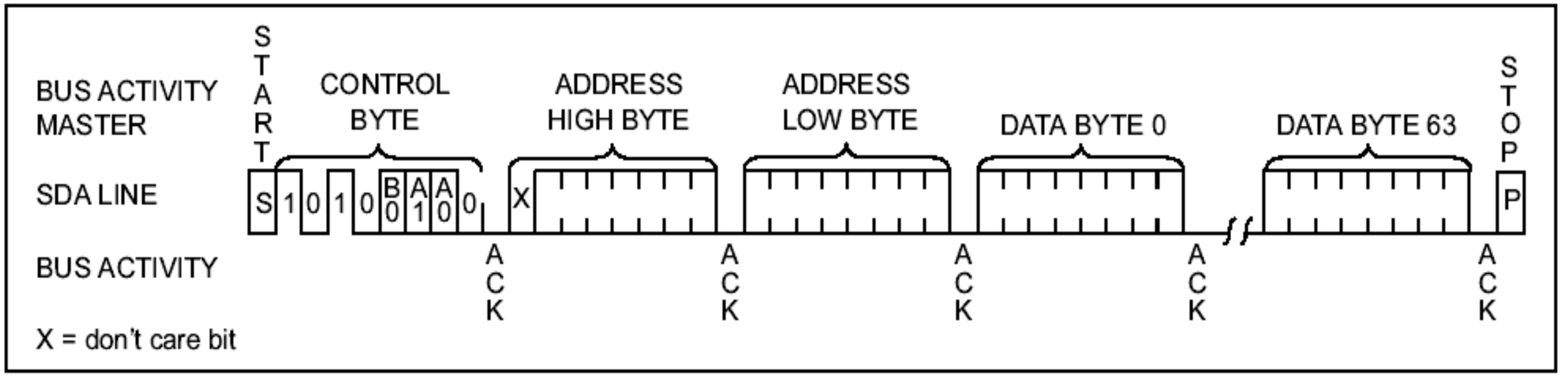

'0' indicates write mode.

'X' because block select bit chooses high or low 32K.

# Page Write Operation

- Send a block of 64 bytes, then perform write
  - Send starting address, followed by 64 bytes
  - After 64 bytes sent, wait 5 ms for write to complete
  - Much faster than individual writes

**FIGURE 6-2: PAGE WRITE**




**Address should be on a page boundary**. For page size = 64 = 0x40, starting address should be a multiple of 64.

# Speed Comparison

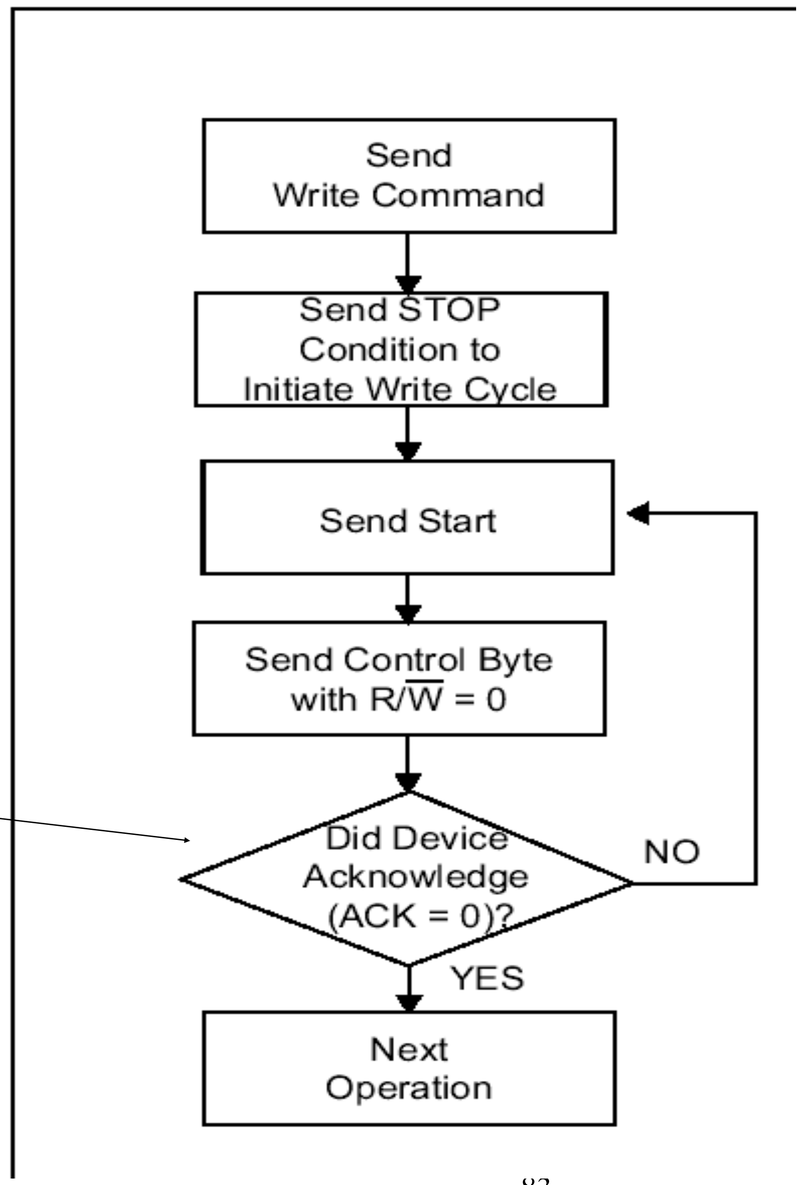
- Assume a 400 Khz I²C bus,  2.5 us clock period (2.5 e-6)
- Random write:
  - 9 bit transmission = 2.5 us * 9 = 22.5 us
  - 5 ms + 22.5 us* 4 (control,addhi,addlo,data) =5.09 ms
  - For 64 bytes =  325 ms approximately, not counting software overhead.
- Page Write
  - 67 bytes total (control, addhi, addlo, data)
  - 5 ms + 67 * 22.5 us =  6.5 ms!!!

# Checking for end-of-write

Timing on write is guaranteed to finish after 5 ms. But can end sooner; to determine if write finished use polling method.

No ACK means device is still busy with last write.

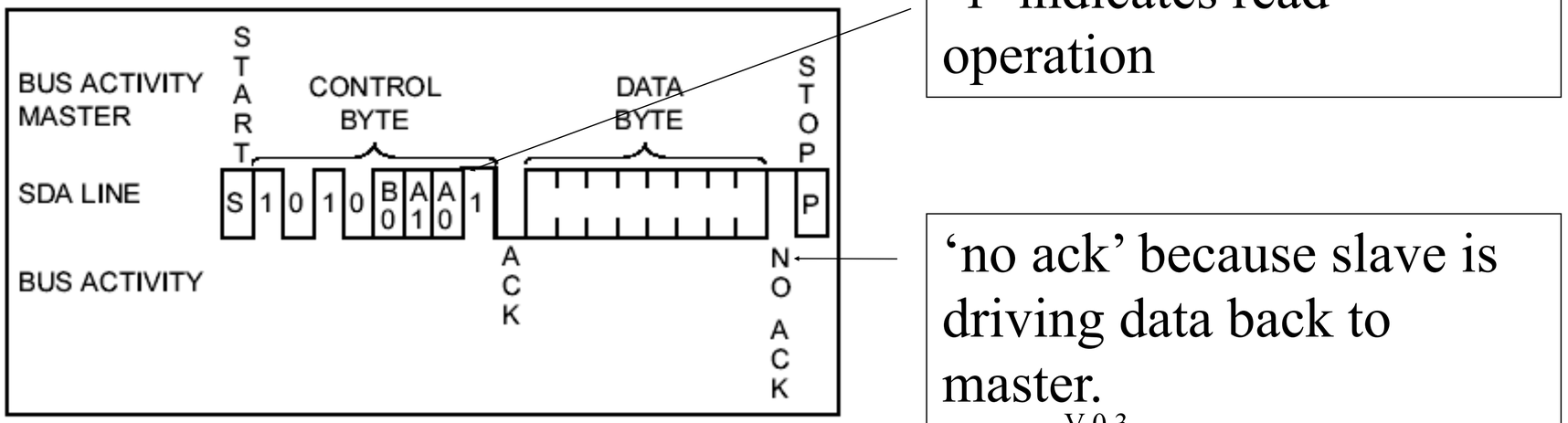
FIGURE 7-1: ACKNOWLEDGE POLLING FLOW



# Read Operation: Current Address

- An internal counter is used to keep track of last address used
- A current address read uses this address, only sends the command byte
  - Internal counter incremented after read operation

**FIGURE 8-1: CURRENT ADDRESS READ**



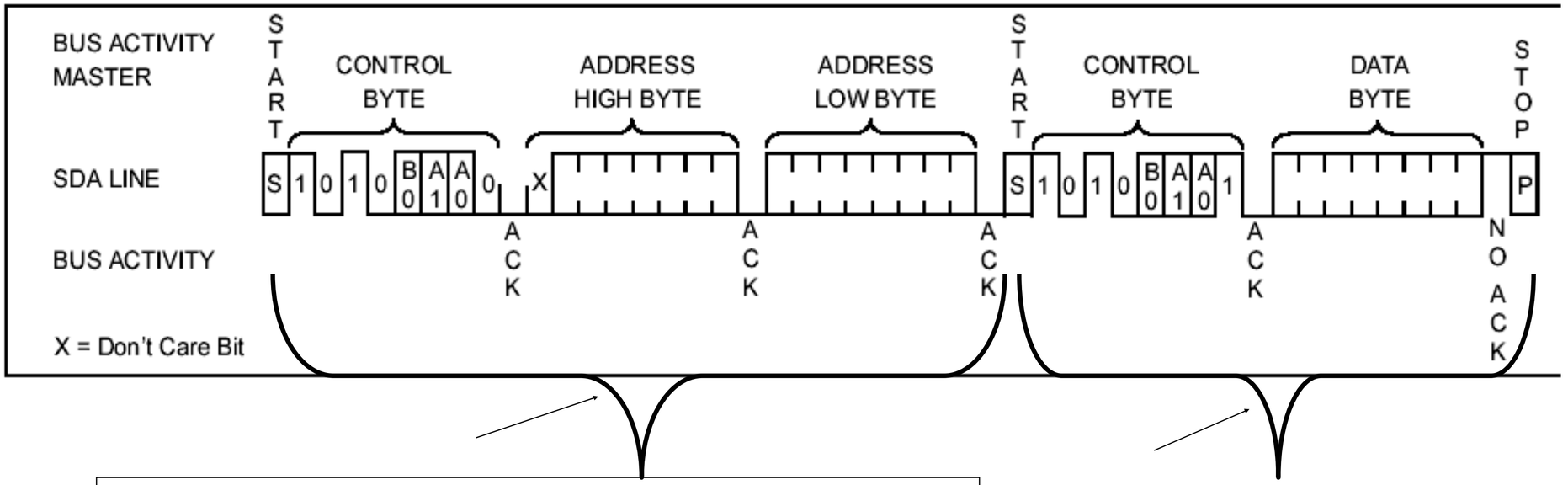

'1' indicates read operation

'no ack' because slave is driving data back to master.

# Random Read Operation

- Must first set the internal address counter by starting a write operation,  but aborting by not sending data byte

**FIGURE 8-2: RANDOM READ**



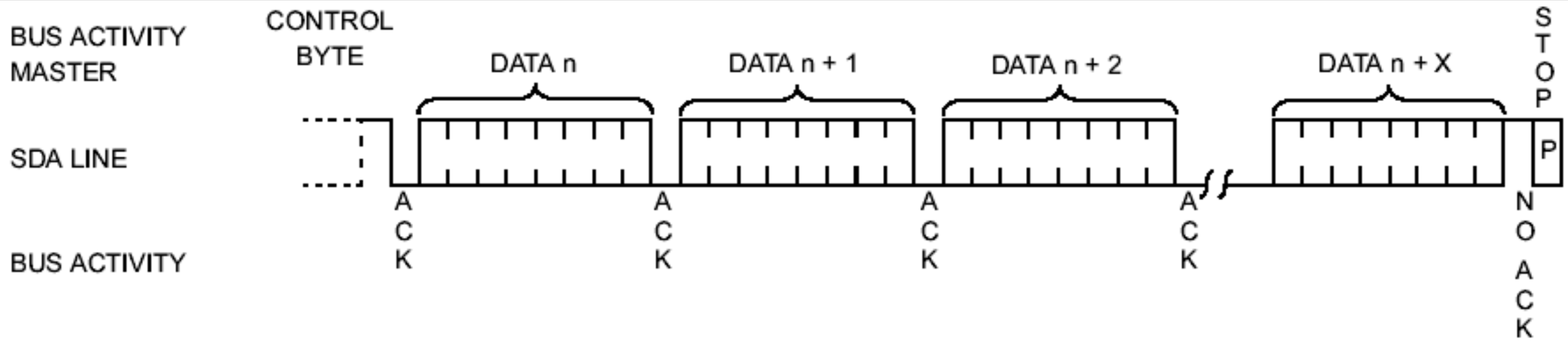

Aborted random write (address only, no data)

Current address read

# Sequential Read

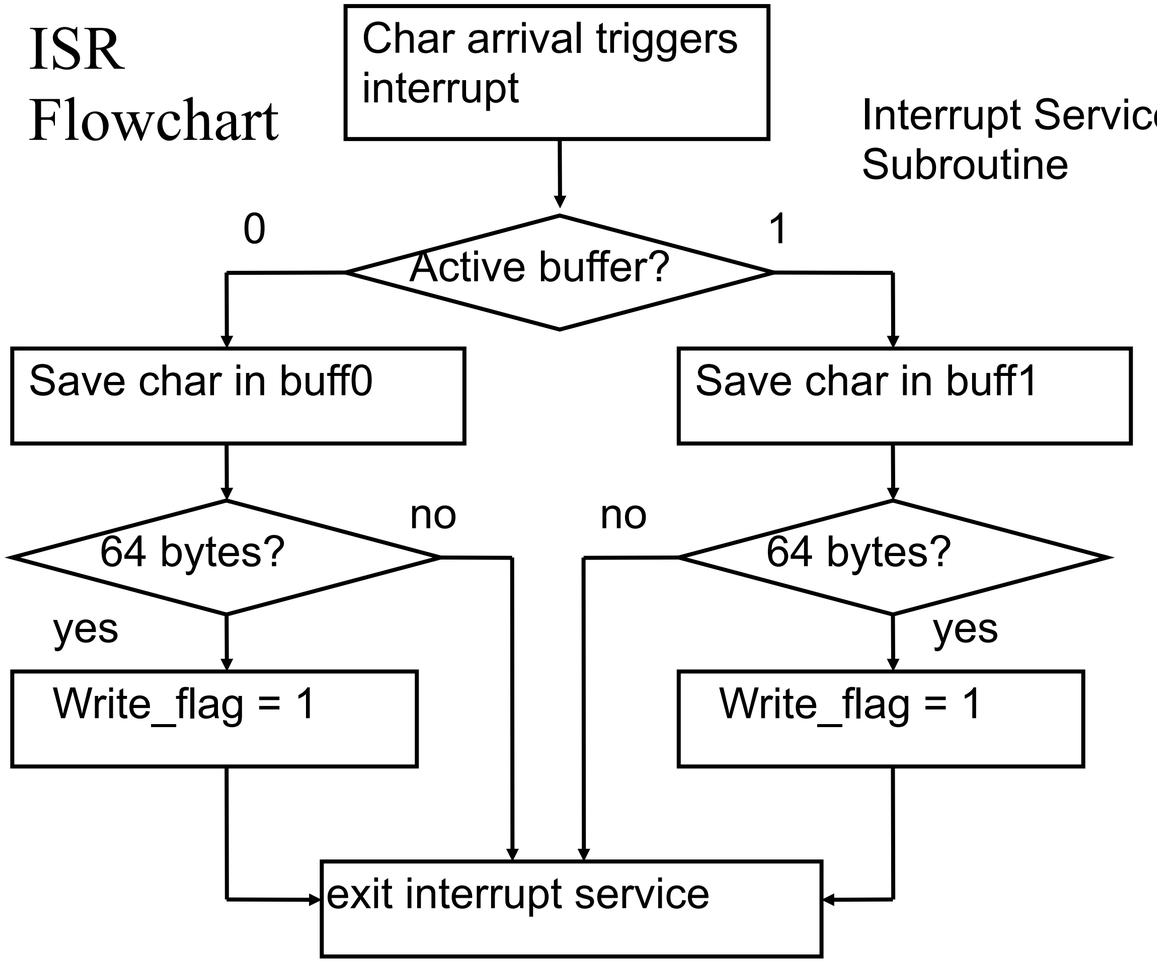
- Like a current address read, but after Slave sends data byte, Master sends ACK instead of STOP
  - Slave then sends next byte
  - Can do this from 0x0000h to 0x7FFF (lower 32K block). When 0x7FFF is reached, address rolls over to 0x0000
  - Upper block goes from 0x8000 to 0xFFFF; at 0xFFFF address rolls over to 0x8000
  - Internal address counter is only 15 bits wide.

**FIGURE 8-3: SEQUENTIAL READ**



# Two Buffers

ISR Flowchart




At least one of the 64 byte buffers MUST be in bank1, not enough room for both in bank0.

While writing data to EEPROM from one buffer, use other buffer to receive incoming data.

ISR sets *write_flag* to indicate to main() that 64 bytes has been received, and that block_mem_write() should be called.

*active_buffer* flag set by main() to indicate what buffer the ISR uses for byte storage.

# Streaming Write Loop: *main()*