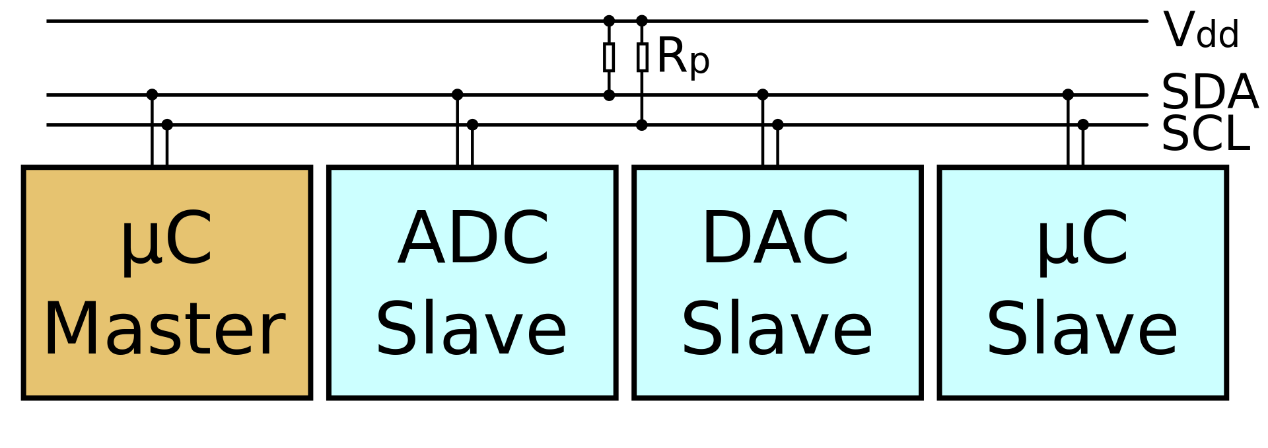
## I2C编程

I2C（Inter-Integrated Circuit BUS）是I2C BUS简称，中文为集成电路总线，是目前应用最广泛的总线之一。和IMX6ULL有些相关的是，刚好该总线是NXP前身的PHILIPS设计。

* 1. I2C协议
     1. **概述**

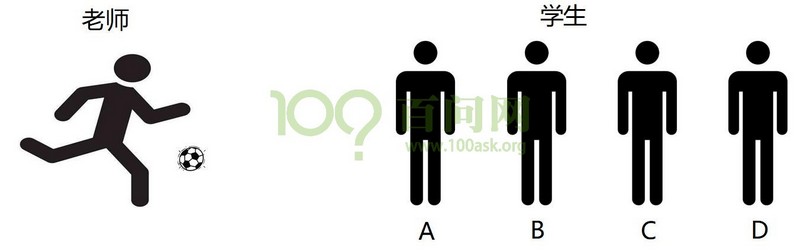
I2C是一种串行通信总线，使用多主从架构，最初设计目的为了让主板、嵌入式系统或手机用来连接低速周边设备。多用于小数据量的场合，有传输距离短，任意时刻只能有一个主机等特性。严格意义上讲，I2C应该是软硬件结合体，所以我们将分物理层和协议层来介绍该总线。

I2C总线结构如下图：



传输数据时，我们需要发数据，从主设备发送到从设备上去；也需要把数据从从设备传送到主设备上去，数据涉及到双向传输。

对于I2C通信的过程，下面使用一个形象的生活例子进行类比。



体育老师：可以把球发给学生，也可以把球从学生中接过来。

① 发球：

1. 老师说：注意了(start)；
2. 老师对A学生说，我要球发给你(A就是地址)；
3. 老师就把球发出去了（传输）；
4. A收到球之后，应该告诉老师一声（回应）；
5. 老师说下课（停止）。

② 接球：

1. 老师说注意了(start)；
2. 老师说：B把球发给我(B是地址)；
3. B就把球发给老师（传输）；
4. 老师收到球之后，给B说一声，表示收到球了（回应）；
5. 老师说下课（停止）。

我们就使用这个简单的例子，来解释一下I2C的传输协议：

① 老师说注意了，表示开始信号(start)

② 老师告诉某个学生，表示发送地址(address)

③ 老师发球/接球，表示数据的传输

④ 老师/学生收到球，回应表示：回应信号(ACK)

⑤ 老师说下课，表示I2C传输接受(P)

* + 1. **物理层**

##### 特性1：半双工（非全双工）

I2C总线中只使用两条线路：SDA、SCL。

**① SDA(串行数据线):**

主芯片通过一根SDA线既可以把数据发给从设备，也可以从SDA上读取数据。在I2C设备内部有两个引脚（发送引脚/接受引脚），它们都连接到外部的SDA线上，具体可以参考下图device端里面的I2Cn\_SDA(output/input)。

**② SCL(串行时钟线)：**

I2C主设备发出时钟，从设备接收时钟。

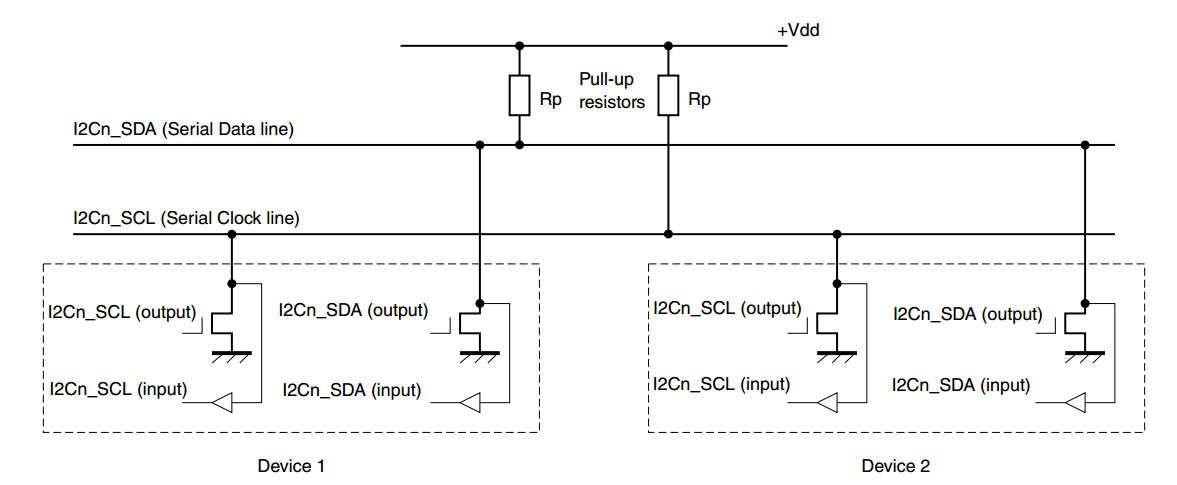
SDA和SCL引脚的内部电路结构一致，引脚的输出驱动与输入缓冲连在一起。其中输出为漏极开路的场效应管、输入缓冲为一只高输入阻抗的同相器。这样结构有如下特性：

a. 由于 SDA、SCL 为漏极开路结构，借助于外部的上拉电阻实现了信号的“线与”逻辑；

b. 引脚在输出信号的同时还作用输入信号供内部进行检测，当输出与输入不一致时，就表示有问题发生了。这为 “时钟同步”和“总线仲裁”提供硬件基础。

SDA和CLK连接线上连有两个上拉电阻，当总线空闲时，两根线均为高电平。连到总线上的任一器件输出的低电平，都将使总线的信号变低。

物理层连接如下图所示：



##### 特性2：地址和角色可配置

每个连接到总线的器件都可以通过唯一的地址和其它器件通信，主机/从机角色和地址可配置，主机可以作为主机发送器和主机接收器。

##### 特性3：多主机

I2C是真正的多主机总线，I2C设备可以在通讯过程转变成主机。如果两个或更多的主机同时请求总线，可以通过冲突检测和仲裁防止总线数据被破坏。

##### 特性4：传输速率

传输速率在标准模式下可以达到100kb/s，快速模式下可以达到400kb/s。

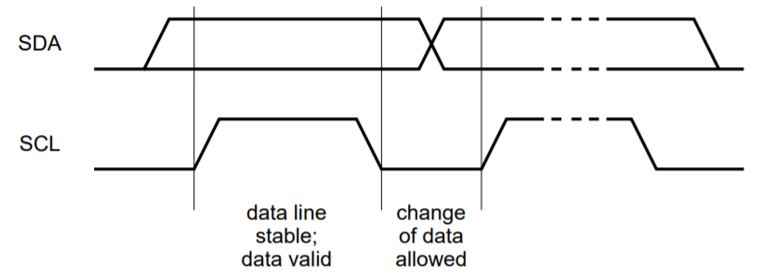
##### 特性5：负载和距离

节点的最大数量受限于地址空间以及总线电容决定，另外总电容也限制了实际通信距离只有几米。

* + 1. **协议层**

##### 数据有效性

I2C协议的数据有效性是靠时钟来保证的，在时钟的高电平周期内，SDA线上的数据必须保持稳定。数据线仅可以在时钟SCL为低电平时改变。

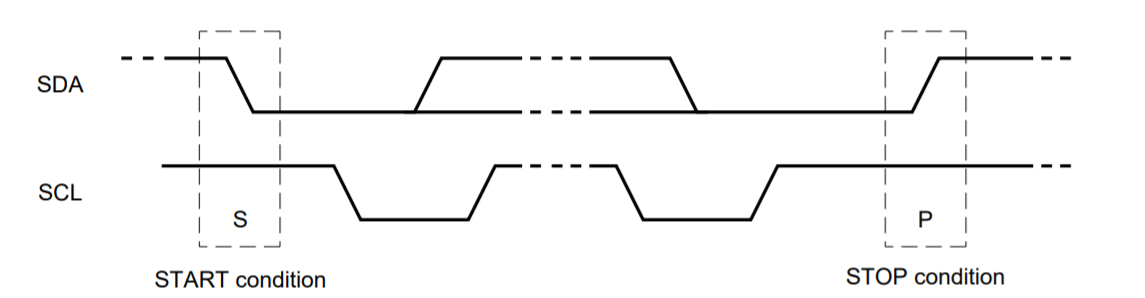


##### 起始和结束条件

**起始条件：**当SCL为高电平的时候，SDA线上由高到低的跳变被定义为起始条件。

**结束条件：**当SCL为高电平的时候，SDA线上由低到高的跳变被定义为停止条件。

要注意起始和终止信号都是由主机发出的，连接到I2C总线上的器件，若具有I2C总线的硬件接口，则很容易检测到起始和终止信号。



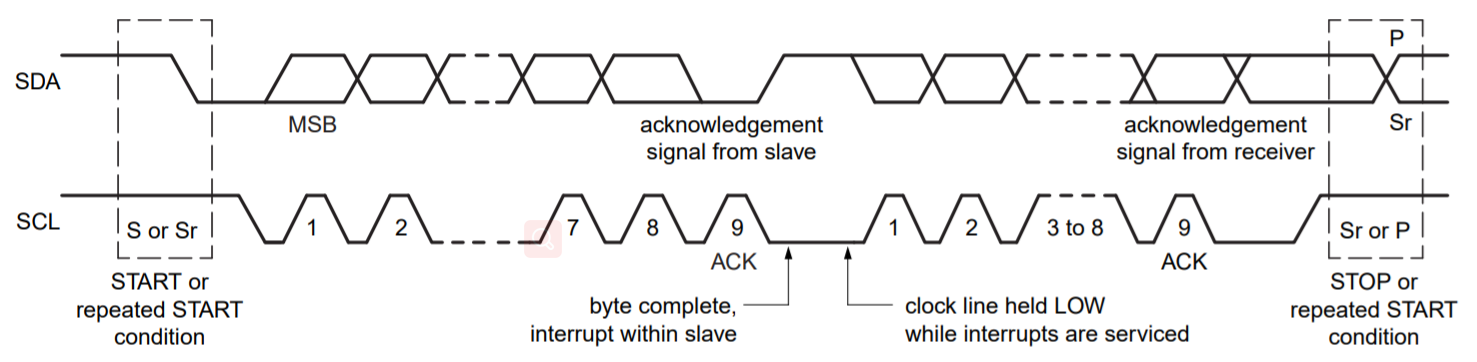
总线在起始条件之后，视为忙状态，在停止条件之后被视为空闲状态。

##### 应答

每当主机向从机发送完一个字节的数据，主机总是需要等待从机给出一个应答信号，以确认从机是否成功接收到了数据，从机应答主机所需要的时钟仍是主机提供的，应答出现在每一次主机完成8个数据位传输后紧跟着的时钟周期，低电平0表示应答，1表示非应答。

##### 数据帧格式

SDA线上每个字节必须是8位长，在每个传输(transfer)中所传输字节数没有限制，每个字节后面必须跟一个ACK。8位数据中，先传输最高有效位（MSB）传输。



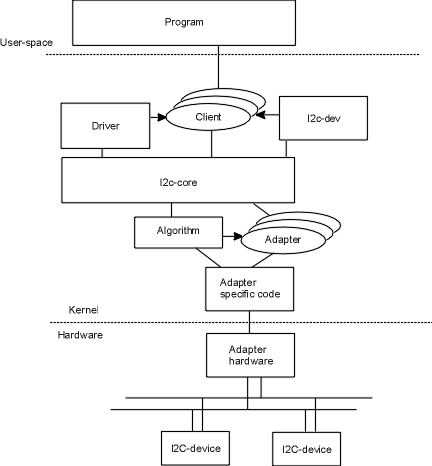
* 1. 在linux系统下操作I2C总线的外设
     1. **概述**

下图是在linux系统环境里操作i2c总线上的外设流程框图。我们按照从下向上的顺序研究一下该流程中各个角色的功能。

在硬件层中，I2C硬件总线只有两条线路，上面可以挂载多个I2C-device，这些I2C-device有的在I2C总线里充当主机的角色，一般情况该主机为板子上的主cpu中的I2C控制器，比如我们用的100ask\_imx6UL板子，这个I2C主机就是imx6中的I2C控制器模块；其他的I2C-device在I2C总线里充当从机的角色，通常这些从机是板子上完成特定功能的传感器外设，只不过该外设与主控cpu的通信方式是只需要两条线路的I2C总线，比如在我们的100ask\_imx6UL板子中就有eeprom和AP3216两个外设，它们在I2C总线中充当的都是I2C从机的角色，它们和主控芯片imx6中的I2C控制器1都是以并联的方式挂在这个I2C总线上。

在内核中，驱动程序对下要完成I2C总线上的I2C通信协议，收集硬件传感器的I2C数据并封装成标准的linux操作接口供用户空间的应用程序操作。对上要实现可以通过linux程序把数据流组织成I2C协议下发到硬件层的相应的外设传感器中。

在用户空间的应用程序中，应用工程师完全可以不必理会I2C协议的详细规定。只需要按照驱动层提供给我们的操作I2C外设的操作接口函数就可以像操作linux中其他普通设备文件那样轻松的操作I2C外设了。



* + 1. **简述I2C的linux驱动**

I2C在linux内核层的驱动框架主要由三部分组成：

##### I2C核心层：

I2C核心提供了I2C总线驱动和设备驱动的注册、注销方法，I2C通信方法(algorithm)的上层部分，并且还提供了一系列与具体硬件平台无关的接口函数以及探测设备，检测设备地址的上层代码等。它位于内核源码目录下的drivers/i2c/i2c-core.c文件中，是I2C总线驱动和设备驱动之间依赖于I2C核心作为纽带。

I2C核心中的主要函数包括：

增加/删除i2c\_adapter

int i2c\_add\_adapter(struct i2c\_adapter \*adap);

int i2c\_del\_adapter(struct i2c\_adapter \*adap);

增加/删除i2c\_driver

int i2c\_register\_driver(struct module \*owner, struct i2c\_driver \*driver);  
int i2c\_del\_driver(struct i2c\_driver \*driver);  
inline int i2c\_add\_driver(struct i2c\_driver \*driver);

i2c\_client依附/脱离

int i2c\_attach\_client(struct i2c\_client \*client);

int i2c\_detach\_client(struct i2c\_client \*client);

i2c传输、发送和接收

int i2c\_transfer(struct i2c\_adapter \* adap, struct i2c\_msg \*msgs, int num); 用于进行I2C适配器和I2C设备之间的一组消息交互。其本身不具备驱动适配器物理硬件完成消息交互的能力，它只是寻找到i2c\_adapter对应的i2c\_algorithm，并使用i2c\_algorithm的master\_xfer()函数真正驱动硬件流程。  
int i2c\_master\_send(struct i2c\_client \*client,const char \*buf ,int count);  
int i2c\_master\_recv(struct i2c\_client \*client, char \*buf ,int count);

i2c\_master\_send()和i2c\_master\_recv()函数内部会调用i2c\_transfer()函数分别完成一条写消息和一条读消息。

1. I2C控制命令分派

下面函数有助于将发给I2C适配器设备文件ioctl的命令分派给对应适配器的algorithm的algo\_control()函数或i2c\_driver的command()函数：  
int i2c\_control(struct i2c\_client \*client, unsigned int cmd, unsigned long arg);  
void i2c\_clients\_command(struct i2c\_adapter \*adap, unsigned int cmd, void \*arg);

##### I2C总线驱动层：

I2C总线驱动是对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部。

它主要完成的功能有：

1. 初始化I2C适配器所使用的硬件资源，申请I/O地址、中断号等。
2. 通过i2c\_add\_adapter()添加i2c\_adapter的数据结构，当然这个i2c\_adapter数据结构的成员已经被xxx适配器的相应函数指针所初始化。
3. 释放I2C适配器所使用的硬件资源，释放I/O地址、中断号等。
4. 通过i2c\_del\_adapter()删除i2c\_adapter的数据结构。

##### I2C总线驱动层：

I2C设备驱动(也称为客户驱动)是对I2C硬件体系结构中设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。I2C设备驱动模块加载函数通用的方法是在I2C设备驱动模块加载函数中完成两件事：通过register\_chrdev()函数将I2C设备注册为一个字符设备。通过I2C核心的i2c\_add\_driver()函数添加i2c\_driver。

* 1. 在linux应用层使用I2C

前面我们讲解了I2C的协议及在linux驱动框架，那么当你拿到开发板或者是从公司的硬件同事拿到一个带有I2C外设的板子，我们应该如何最快速的使用起来这个I2C设备呢？既然我们总是说这个I2C总线在嵌入式开发中被广泛的使用，那么是否有现成的测试工具帮我们完成这个快速使用板子的I2C设备呢？答案是有的，而且这个测试工具的代码还是开源的，它被广泛的应用在linux应用层来快速验证I2C外设是否可用，为我们测试I2C设备提供了很好的捷径。

* + 1. **如何使用I2C tools测试I2C外设**

##### I2C tools概述：

I2C tools包含一套用于Linux应用层测试各种各样I2C功能的工具。它的主要功能包括：总线探测工具、SMBus访问帮助程序、EEPROM解码脚本、EEPROM编程工具和用于SMBus访问的python模块。只要你所使用的内核中包含I2C设备驱动，那么就可以在你的板子中正常使用这个测试工具。

##### 下载I2C tools源码：

前面我们已经说过了这个I2C tools工具是开源的，那么这个源码在哪里可以找到呢？下载方法一：直接在内核的网站https://mirrors.edge.kernel.org/pub/software/utils/i2c-tools/下载I2C tools代码的压缩包，下载方法二：利用git管理工具下载这个I2C tools的源代码，命令为git clone git://git.kernel.org/pub/scm/utils/i2c-tools/i2c-tools.git强烈建议读者采用第二种方法下载这个代码，因为你可以通过git快速地了解这个开源代码的不同版本的功能改进及bug修复，而且使用git开发也是作为一名优秀的开发人员必备的一项技能。

##### 编译I2C tools源码：

进入刚才利用git下载好的iic-tools源码目录，修改编译工具为你当前使用的交叉编译工具：

26 CC ?= arm-linux-gnueabihf-gcc

27 AR ?= arm-linux-gnueabihf-ar

编译源码：如果你想编译静态版本，你可以输入命令：make USE\_STATIC\_LIB=1；如果使用动态库的话，可以直接输入make进行编译。安装命令为：make install,如果你想要让最后生成的二进制文件最小的话，可以在“make install”之前运行“make strip”。但是，这将不能生成任何调试库，也就不能尝试进一步调试。然后将tools目录下的5个可执行文件i2cdetect，i2cdump，i2cget，i2cset和i2ctransfer复制到板子的/usr/sbin/中；将lib目录下的libi2c.so.0.1.1文件复制到板子的/usr/lib/libi2c.so.0。之后别忘了将上面的文件修改为可执行的权限。

##### 介绍I2C tools各功能之—i2cdetect

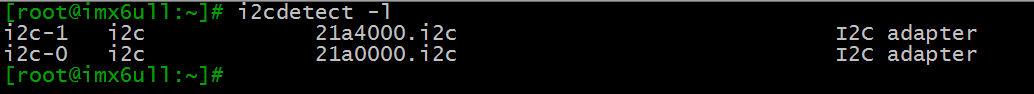
i2cdetect的主要功能就是I2C设备查询，它用于扫描I2C总线上的设备。它输出一个表，其中包含指定总线上检测到的设备的列表。

该命令的常用格式为：i2cdetect [-y] [-a] [-q|-r] i2cbus [first last]。具体参数的含义如下：

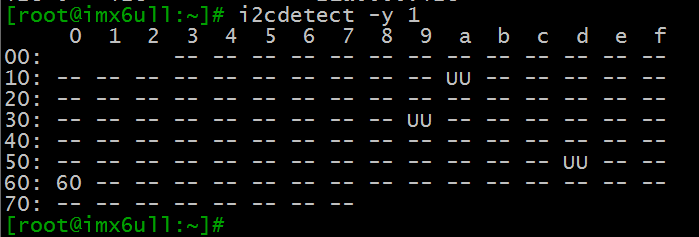
|  |  |
| --- | --- |
| -y | 取消交互模式。默认情况下，i2cdetect将等待用户的确认，当使用此标志时，它将直接执行操作。 |
| -a | 强制扫描非规则地址。一般不推荐。 |
| -q | 使用SMBus“快速写入”命令进行探测。一般不推荐。 |
| -r | 使用SMBus“接收字节”命令进行探测。一般不推荐。 |
| -F | 显示适配器实现的功能列表并退出。 |
| -V | 显示I2C工具的版本并推出。 |
| -l | 显示已经在系统中使用的I2C总线。 |
| i2cbus | 表示要扫描的I2C总线的编号或名称。 |
| first last | 表示要扫描的从设备地址范围。 |

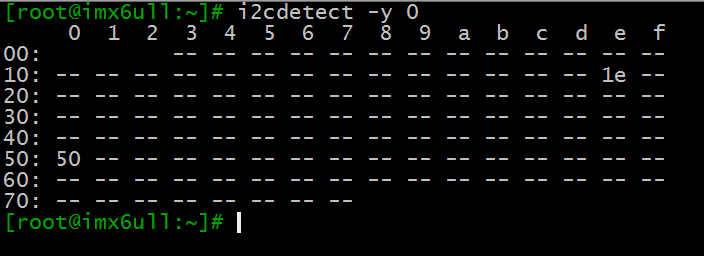
该功能的常用方式：

第一，先通过i2cdetect -l查看当前系统中的I2C的总线情况：

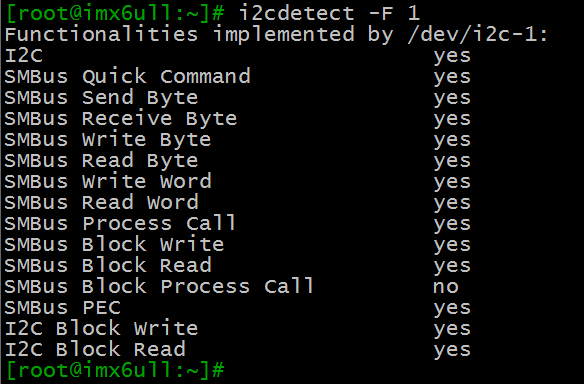


第二，若总线上挂载I2C从设备，可通过i2cdetect扫描某个I2C总线上的所有设备。可通过控制台输入i2cdetect -y 1：（其中"--"表示地址被探测到了，但没有芯片应答； "UU"因为这个地址目前正在被一个驱动程序使用，探测被省略；而16进制的地址号60，1e和50则表示发现了一个外部片选从地址为0x60，0x1e（AP3216）和0x50(eeprom)的外设芯片。





第三，查询I2C总线1 (I2C -1)的功能，命令为i2cdetect -F 1：



##### 介绍I2C tools各功能之—i2cget

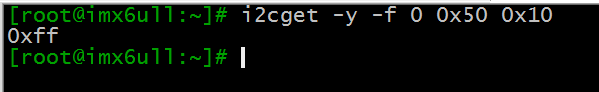
i2cget的主要功能是获取I2C外设某一寄存器的内容。该命令的常用格式为：

i2cget [-f] [-y] [-a] i2cbus chip-address [data-address [mode]]。具体参数的含义如下：

|  |  |
| --- | --- |
| -f | 强制访问设备，即使它已经很忙。  默认情况下，i2cget将拒绝访问已经在内核驱动程序控制下的设备。 |
| -y | 取消交互模式。默认情况下，i2cdetect将等待用户的确认，当使用此标志时，它将直接执行操作。 |
| -a | 允许在0x00 - 0x07和0x78 - 0x7f之间使用地址。一般不推荐。 |
| i2cbus | 表示要扫描的I2C总线的编号或名称。这个数字应该与i2cdetect -l列出的总线之一相对应。 |
| chip-address | 要操作的外设从地址。 |
| data-address | 被查看外设的寄存器地址。 |
| mode | 显示数据的方式：  b (read byte data, default)  w (read word data)  c (write byte/read byte) |

下面是完成读取0总线上从地址为0x50的外设的0x10寄存器的数据，命令为：

i2cget -y -f 0 0x50 0x10



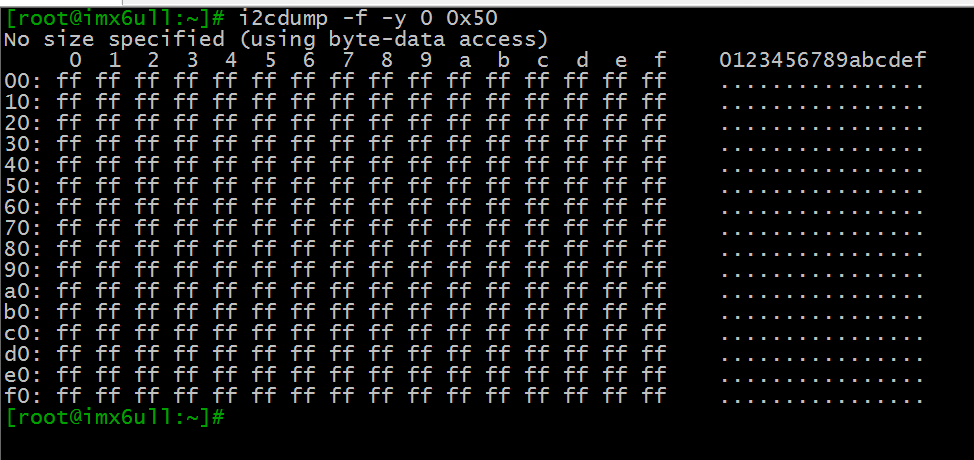
##### 介绍I2C tools各功能之—i2cdump

i2cdump的主要功能查看I2C从设备器件所有寄存器的值。 该命令的常用格式为：i2cdump [-f] [-r first-last] [-y] [-a] i2cbus address [mode [bank [bankreg]]]。具体参数的含义如下：

|  |  |
| --- | --- |
| -f | 强制访问设备，即使它已经很忙。  默认情况下，i2cget将拒绝访问已经在内核驱动程序控制下的设备。 |
| -r | 限制正在访问的寄存器范围。 此选项仅在模式b，w，c和W中可用。对于模式W，first必须是偶数，last必须是奇数。 |
| -y | 取消交互模式。默认情况下，i2cdetect将等待用户的确认，当使用此标志时，它将直接执行操作。 |
| -V | 显示I2C工具的版本并推出。 |
| i2cbus | 表示要扫描的I2C总线的编号或名称。这个数字应该对应于i2cdetect -l列出的总线之一。 |
| first last | 表示要扫描的从设备地址范围。 |
| mode | b: 单个字节  w：16位字  s：SMBus模块  i：I2C模块的读取大小  c: 连续读取所有字节，对于具有地址自动递增功能的芯片（如EEP​​ROM）非常有用。  W与 w类似，只是读命令只能在偶数寄存器地址上发出;这也是主要用于EEPROM的。 |

下面是完成读取0总线上从地址为0x50的eeprom的数据，命令为：

i2cdump -f -y 0 0x50



##### 介绍I2C tools各功能之—i2cset

i2cset的主要功能是通过I2C总线设置设备中某寄存器的值。该命令的常用格式为：

i2cset [-f] [-y] [-m mask] [-r] i2cbus chip-address data-address [value] ...[mode]

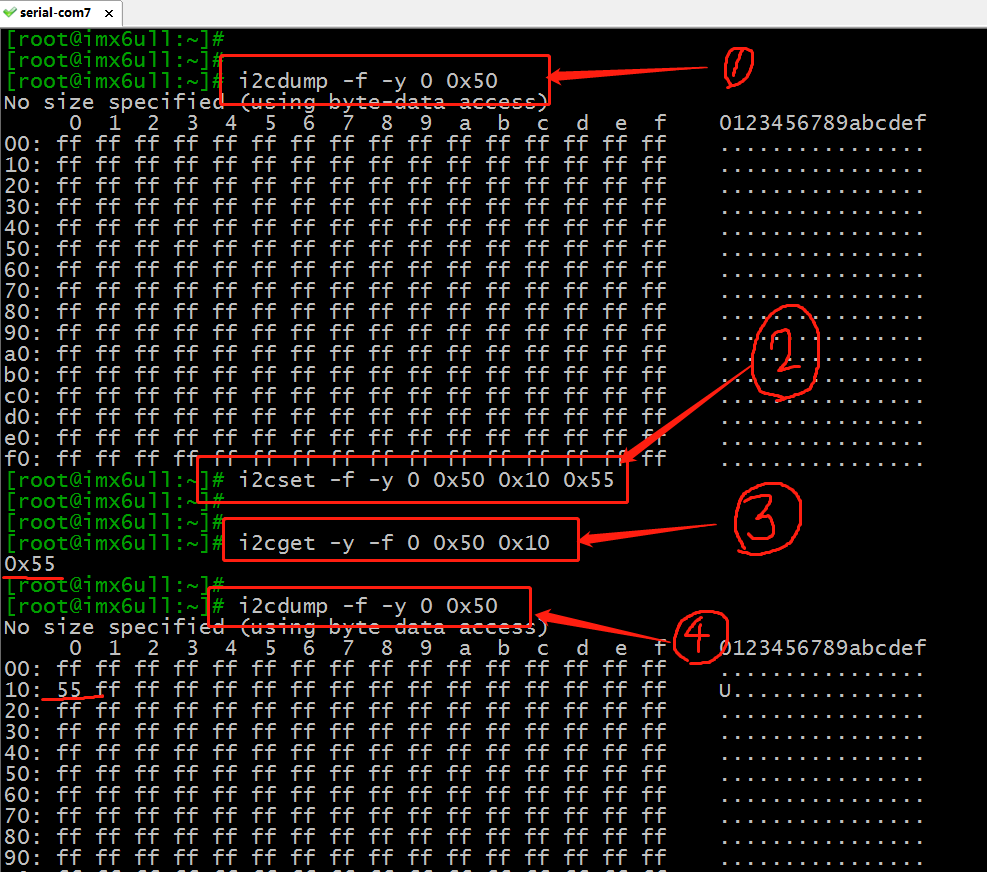
具体参数的含义如下：

|  |  |
| --- | --- |
| -f | 强制访问设备，即使它已经很忙。  默认情况下，i2cget将拒绝访问已经在内核驱动程序控制下的设备。 |
| -r | 在写入值之后立即读取它，并将结果与写入的值进行比较。 |
| -y | 取消交互模式。默认情况下，i2cdetect将等待用户的确认，当使用此标志时，它将直接执行操作。 |
| -V | 显示I2C工具的版本并推出。 |
| i2cbus | 表示要扫描的I2C总线的编号或名称。这个数字应该对应于i2cdetect -l列出的总线之一。 |
| -m mask | 如果指定mask参数，那么描述哪些value位将是实际写入data-addres的。掩码中设置为1的位将从值中取出，而设置为0的位将从数据地址中读取，从而由操作保存。 |
| mode | b: 单个字节  w：16位字  s：SMBus模块  i：I2C模块的读取大小  c: 连续读取所有字节，对于具有地址自动递增功能的芯片（如EEP​​ROM）非常有用。  W与 w类似，只是读命令只能在偶数寄存器地址上发出;这也是主要用于EEPROM的。 |

下面是完成向0总线上从地址为0x50的eeprom的0x10寄存器写入0x55，命令为：

i2cset -y -f 0 0x50 0x10 0x55

然后用i2cget读取0总线上从地址为0x50的eeprom的0x10寄存器的数据，命令为：i2cget -y -f 0 0x50 0x10



##### 介绍I2C tools各功能之—i2ctransfer

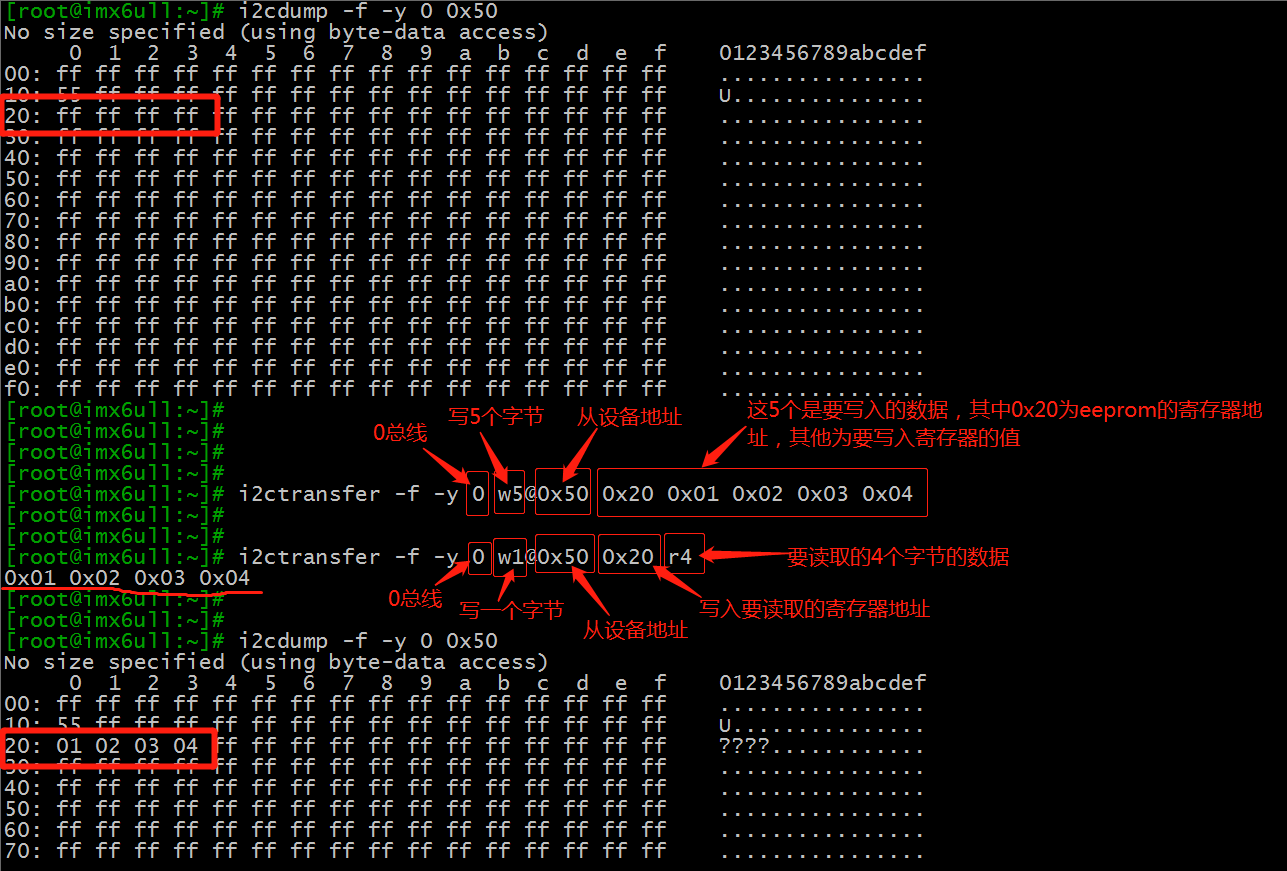
i2ctransfer的主要功能是在一次传输中发送用户定义的I2C消息。i2ctransfer是一个创建I2C消息并将其合并为一个传输发送的程序。对于读消息，接收缓冲区的内容被打印到stdout，每个读消息一行。

该命令的常用格式为：i2ctransfer [-f] [-y] [-v] [-a] i2cbus desc [data] [desc [data]]

具体参数的含义如下：

|  |  |
| --- | --- |
| -f | 强制访问设备，即使它已经很忙。  默认情况下，i2cget将拒绝访问已经在内核驱动程序控制下的设备。 |
| -y | 取消交互模式。默认情况下，i2cdetect将等待用户的确认，当使用此标志时，它将直接执行操作。 |
| -v | 启用详细输出。它将打印所有信息发送，即不仅为读消息，也为写消息。 |
| -V | 显示I2C工具的版本并推出。 |
| -a | 允许在0x00 - 0x02和0x78 - 0x7f之间使用地址。一般不推荐。 |
| i2cbus | 表示要扫描的I2C总线的编号或名称。这个数字应该对应于i2cdetect -l列出的总线之一。 |

下面是完成向0总线上从地址为0x50的eeprom的0x20开始的4个寄存器写入0x01，0x02，0x03，0x04命令为：i2ctransfer -f -y 0 w5@0x50 0x20 0x01 0x02 0x03 0x04然后再通过命令i2ctransfer -f -y 0 w1@0x50 0x20 r4将0x20地址的4个寄存器数据读出来，见下图：



* + 1. **在linux应用程序中读写I2C外设**

首先通过前面的介绍，我们已经知道站在cpu的角度来看，操作I2C外设实际上就是通过控制cpu中挂载该I2C外设的I2C控制器，而这个I2C控制器在linux系统中被称为“I2C适配器”，这个已经在驱动简介中介绍过了。而且众所周知，在linux系统中，每一个设备都是以文件的形式存在的，所以在linux中操作I2C外设就变成了操作I2C适配器设备文件。Linux系统（也就是内核）为每个I2C适配器生成了一个主设备号为89的设备节点（次设备号为0-255），它并没有针对特定的I2C外甥而设计，只是提供了通用的read(),write(),和ioctl()等文件操作接口，在用户空间的应用层就可以借用这些接口访问挂接在适配器上的I2C设备的存储空间或寄存器，并控制I2C设备的工作方式。

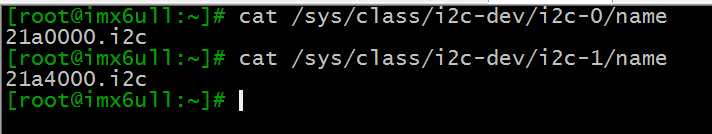
操作流程：

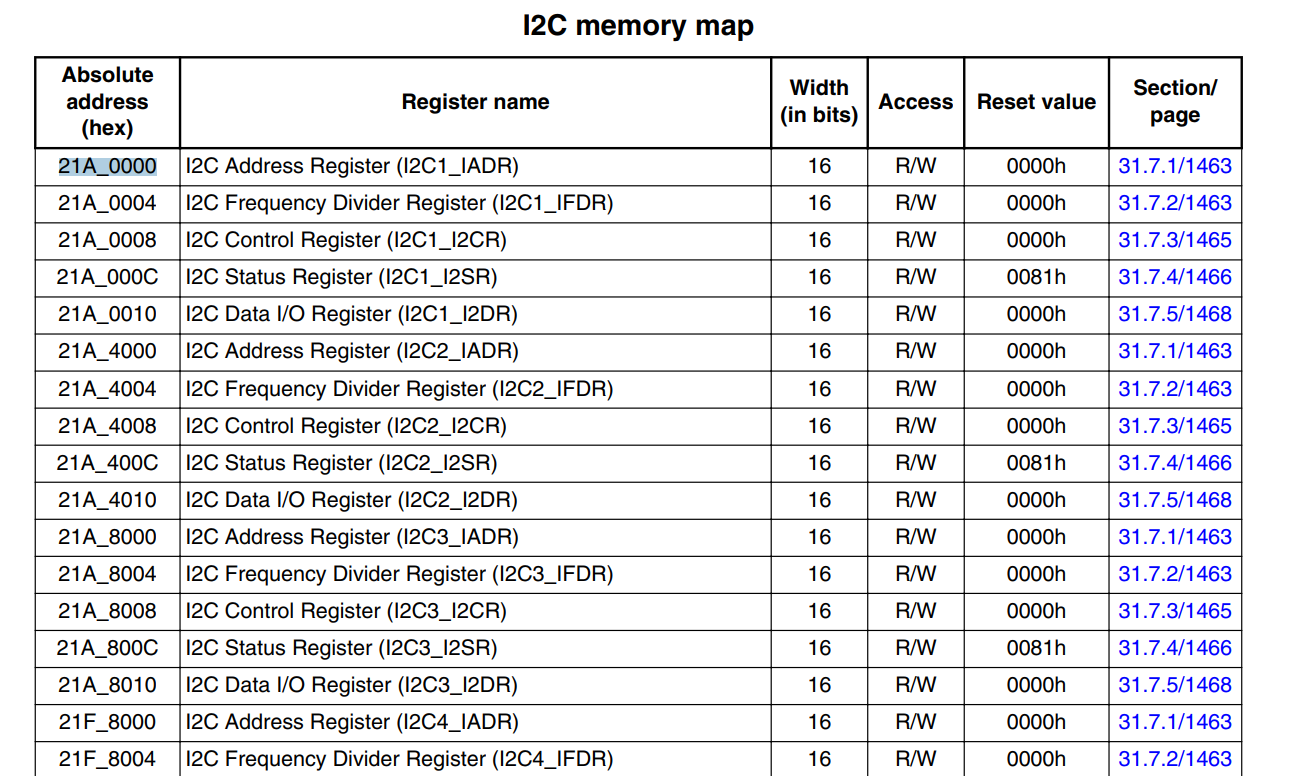
##### 确定I2C适配器的设备文件节点

i2c适配器的设备节点是/dev/i2c-x，其中x是数字。由于适配器编号是动态分配的（和注册次序有关），所以想了解哪一个适配器对应什么编号，可以查看/sys/class/i2c-dev/目录下的文件内容（在这里笔者强烈建议读者好好利用好sys文件系统）：

cat /sys/class/i2c-dev/i2c-0/name

cat /sys/class/i2c-dev/i2c-1/name



然后查看硬件原理图中eeprom是挂在cpu的i2c1控制器中了，然后查看IMX6UL芯片手册中I2C1的寄存器地址为21A\_0000。

比对后，我们就很容易知道eeprom外设对应的I2C控制器的设备节点为：/dev/i2c-0。

##### 打开适配器对应的设备节点

当用户打开适配器设备节点的时候，Kernel中的i2c-dev代码为其建立一个i2c\_client，但是这个i2c\_client并不加到i2c\_adapter的client链表当中。当用户关闭设备节点时，它自动被释放。

##### IOCTL控制

这个可以参考内核源码中的include/linux/i2c-dev.h文件。下面举例说明主要的IOCTL命令：

|  |  |
| --- | --- |
| I2C\_SLAVE\_FORCE | 设置I2C从设备地址（只有在该地址空闲的 情况下成功。） |
| I2C\_SLAVE\_FORCE | 强制设置I2C从设备地址（无论内核中是否已有驱动在使用这个地址都会成功） |
| I2C\_TENBIT | 选择地址位长:  0 表示是7bit地址 ；  不等于0 就是10 bit的地址。只有适配器支持I2C\_FUNC\_10BIT\_ADDR，这个请求才是有效的。 |
| I2C\_FUNCS | 获取适配器支持的功能，详细的可以参考文件include/linux/i2c.h |
| I2C\_RDWR | 设置为可读写 |
| I2C\_RETRIES | 设置收不到ACK时的重试次数 |
| I2C\_TIMEOUT | 设置超时的时限 |

##### 使用I2C协议和设备进行通信

代码为：ioctl(file,I2C\_RDWR,(struct i2c\_rdwr\_ioctl\_data \*)msgset); 它可以进行连续的读写，中间没有间歇。只有当适配器支持I2C\_FUNC\_I2C此命令才有效。参数msgset是一个指针，指向一个i2c\_rdwr\_ioctl\_data类型的结构体，该结构体的功能就是让应用程序可以向内核传递消息，其成员包括：struct i2c\_msg \_\_user \*msgs; 和表示i2c\_msgs 个数的\_\_u32 nmsgs，它也决定了在硬件I2C总线的硬件通信中有多少个开始信号。由于I2C适配器与外设通信是以消息为单位的，所以struct i2c\_msg对我们来说是非常重要的，它可以包含多条消息，而一条消息有可能包含多个数据，比如对于eeprom页写就包含多个数据。下面就介绍一下这个结构体的内容：

|  |  |
| --- | --- |
| \_\_u16 addr; | 从设备地址 |
| \_\_u16 flags; | 标志（读/写） |
| I2C\_M\_TEN | 这是一个10位芯片地址 |
| I2C\_M\_RD | 从设备到适配器读数据 |
| I2C\_M\_NOSTART | 不发送起始位 |
| I2C\_M\_REV\_DIR\_ADDR | 翻转读写标志 |
| I2C\_M\_IGNORE\_NAK | 忽略I2C的NACK信号 |
| I2C\_M\_NO\_RD\_ACK | 读操作的时候不发ACK信号 |
| I2C\_M\_RECV\_LEN | 第一次接收数据的长度 |
| \_\_u16 len; | 写入或者读出数据的个数（字节） |
| \_\_u8 \*buf; | 写入或者读出数据的地址 buf[0]。  注意：千万不要忘记给2c\_rdwr\_ioctl\_data结构体中的最重要的结构i2c\_msg中的buf分配内存。 |

##### 用read和write读写I2C设备

当然你可以使用read()/write()来与I2C设备进行通信，代码如下（以eeprom为例简要概述操作过程）：

第一，打开I2C控制器文件节点: fd =open(“/dev/i2c-0”, O\_RDWR);

第二，设置eeprom的设备地址：ioctl(fd,I2C\_SLAVE, 0x50);

第三，向eeprom写数据:

首先将要操作的eeprom的第一个寄存器地址赋给写buf的第0个元素wr\_buf[0] = 0x10;

然后把要写入的数据写入到后面的buf中for(i=1;i<13;i++) wr\_buf[i]=i;

最后通过write函数完成向eeprom写数据的功能：write(fd, wr\_buf, 13);

最后延迟1秒，让后面的操作与上面的写操作分开。

第四，从eeprom读数据：

首先和写操作一样，将要操作的寄存器首地址0x10发给eeprom：write(fd, wr\_buf, 1);

从0x10寄存器地址处读取12个字节的数据：ret=read(fd, rd\_buf, 12);

你会发现，用read和write一次只能进行一个方向的传输：或者是读外设操作，或者就是写操作传输。

代码如下：

01 #include <stdio.h>

02 #include <sys/ioctl.h>

03 #include <unistd.h>

04 #include <fcntl.h>

05 #include <linux/i2c-dev.h>

06 #include <linux/i2c.h>

07

08 /\* eeprom所对应的I2C控制器的设备节点 \*/

09 #define EEPROM\_DEVICE "/dev/i2c-0"

10

11 /\* eeprom的I2C设备地址 \*/

12 #define EEPROM\_ADDR 0x50

13

14

15 int main()

16 {

17 int fd,i,ret=0;

18 unsigned char w\_add=0x10;

19

20 /\* 将要读取的数据buf\*/

21 unsigned char rd\_buf[13] = {0x10};

22

23 /\* 要写的数据buf，第0个元素是要操作eeprom的寄存器地址\*/

24 unsigned char wr\_buf[13] = {0};

25

26 printf("hello,this is read\_write i2c test \n");

27

28 /\* 打开eeprom对应的I2C控制器文件 \*/

29 fd =open(EEPROM\_DEVICE, O\_RDWR);

30 if (fd< 0)

31 {

32 printf("open"EEPROM\_DEVICE"failed \n");

33 }

34

35 /\*设置eeprom的I2C设备地址\*/

36 if (ioctl(fd,I2C\_SLAVE\_FORCE, EEPROM\_ADDR) < 0)

37 {

38 printf("set slave address failed \n");

39 }

40

41 /\* 将要操作的寄存器首地址赋给wr\_buf[0] \*/

42 wr\_buf[0] = w\_add;

43

44 /\* 把要写入的数据写入到后面的buf中 \*/

45 for(i=1;i<13;i++)

46 wr\_buf[i]=i;

47

48 /\* 通过write函数完成向eeprom写数据的功能 \*/

49 write(fd, wr\_buf, 13);

50

51 /\* 延迟一段时间 \*/

52 sleep(1);

53

54 /\*重新开始下一个操作，先写寄存器的首地址\*/

55 write(fd, wr\_buf, 1);

56

57 /\* 从wr\_buf[0] = w\_add的寄存器地址开始读取12个字节的数据 \*/

58 ret=read(fd, rd\_buf, 12);

59 printf("ret is %d \r\n",ret);

60

61 for(i=0;i<12;i++)

62 {

63 printf("rd\_buf is :%d\n",rd\_buf[i]);

64 }

65

66 /\* 完成操作后，关闭eeprom对应的I2C控制器的设备文件 \*/

67 close(fd);

68

69 return 0;

70 }

##### 用数据包的方式操作I2C设备

构建数据包结构体：

首先是struct i2c\_rdwr\_ioctl\_data data; 应用程序通过该结构体来给内核传递消息。该结构体包含两个成员struct i2c\_msg \_\_user \*msgs;和\_\_u32 nmsgs;其中\*msgs指向表示通信方法传输为消息的结构体。而nmsgs则决定了该数据包有多少个这样的通信消息，在I2C通信协议上来看就代表了有多少个开始信号。

接着就是struct i2c\_msg; 它可以包含多条消息，而一条消息有可能包含多个数据。其成员包括：“代表I2C设备从地址的\_\_u16 addr; 表示本次消息的标志位的\_\_u16 flags; 表示数据长度的\_\_u16 len; 表示数据缓冲区的指针\_\_u8 \*buf”

然后把要和I2C从设备通信的数据与上面两个结构体建立起相应的联系。

最后调用I2C\_RDWR进入驱动程序执行读写组合的I2C数据传输。

代码如下：

01 #include <stdio.h>

02 #include <string.h>

03 #include <sys/ioctl.h>

04 #include <unistd.h>

05 #include <fcntl.h>

06 #include <linux/i2c-dev.h>

07 #include <linux/i2c.h>

08

09 /\* eeprom所对应的I2C控制器的设备节点 \*/

10 #define EEPROM\_DEVICE "/dev/i2c-0"

11

12 /\* eeprom的I2C设备地址 \*/

13 #define EEPROM\_ADDR 0x50

14

15 /\*函数名：eeprom\_write

16 \*\*功能：向eeprom写数据

17 \*\*参数：fd：eeprom对应I2C控制器设备节点的文件名

18 \*\* dev\_addr：eeprom的I2C从设备地址

19 \*\* reg\_addr：eeprom的寄存器地址

20 \*\* data\_buf：要向eeprom写数据的数据buf

21 \*\* len：要写多少个字节。本例中当前最大支持为8个字节

22 \*\*返回值：负数表示操作失败，其他为成功

23 \*/

24 int eeprom\_write(int fd, unsigned char dev\_addr, unsigned char reg\_addr, unsigned char \* data\_buf,int len)

25 {

26 int ret;

27

28 unsigned char msg\_buf[9];

29 struct i2c\_rdwr\_ioctl\_data data;

30

31 struct i2c\_msg messages;

32

33

34 /\* 1. 构建msg\_buf\*/

35 /\* 1.1. 将要操作的寄存器首地址赋给要进行I2C数据通信的首字节数据 \*/

36 msg\_buf[0] = reg\_addr;

37

38 /\* 1.2. 将要向eeprom写数据的数据buf赋在I2C数据通信中eeprom寄存器的后面 \*/

39 if (len < 9) { /\* 本demo最大支持一次向eeprom写一页大小的8个字节数据 \*/

40 memcpy((void \*) &msg\_buf[1], data\_buf, len); //第1位之后是数据

41 } else {

42 printf("This function supports up to 8 bytes at a time !!!\n");

43 return -1;

44 }

45

46 /\* 2. 构建 struct i2c\_msg messages \*/

47 /\* 2.1. 赋值eeprom的I2C从设备地址 \*/

48 messages.addr = dev\_addr;

49

50 /\* 2.2. 赋值flags为本次I2C通信完成写功能 \*/

51 messages.flags = 0;

52

53 /\* 2.3. 赋值len为数据buf的长度 + eeprom寄存器地址的数据长度 \*/

54 messages.len = len+1;

55

56 /\* 2.4. 构建消息包的数据buf\*/

57 messages.buf = msg\_buf;

58

59 /\* 3. 构建struct i2c\_rdwr\_ioctl\_data data \*/

60 /\* 3.1. 将准备好的消息包赋值给i2c\_rdwr\_ioctl\_data中的msgs消息\*/

61 data.msgs = &messages;

62

63 /\* 3.2. 由于本次I2C通信只有写动作，所以消息数为1次 \*/

64 data.nmsgs = 1;

65

66 /\* 4. 调用驱动层的读写组合的I2C数据传输 \*/

67 if(ioctl(fd, I2C\_RDWR, &data) < 0)

68 {

69 printf("I2C\_RDWR err \n");

70 return -1;

71 }

72

73 /\* 5. 等待I2C总线写入完成 \*/

74 sleep(1);

75

76 return 0;

77 }

78

79 /\*函数名：eeprom\_read

80 \*\*功能：从eeprom读数据

81 \*\*参数：fd：eeprom对应I2C控制器设备节点的文件名

82 \*\* dev\_addr：eeprom的I2C从设备地址

83 \*\* reg\_addr：eeprom的寄存器地址

84 \*\* data\_buf：存放从eeprom读数据的buf

85 \*\* len：要读多少个字节。

86 \*\*返回值：负数表示操作失败，其他为成功

87 \*/

88 int eeprom\_read(int fd, unsigned char dev\_addr, unsigned char reg\_addr, unsigned char \* data\_buf,int len)

89 {

90 int ret;

91

92 unsigned char msg\_buf[9];

93 struct i2c\_rdwr\_ioctl\_data data;

94

95 struct i2c\_msg messages[2];

96

97 /\* 1. 构建 struct i2c\_msg messages \*/

98 /\* 1.1. 构建第一条消息 messages[0] \*/

99 /\* 1.1.1. 赋值eeprom的I2C从设备地址 \*/

100 messages[0].addr = dev\_addr;

101

102 /\* 1.1.2. 赋值flags为本次I2C通信完成写动作 \*/

103 messages[0].flags = 0;

104

105 /\* 1.1.3. 赋值len为eeprom寄存器地址的数据长度是1 \*/

106 messages[0].len = 1;

107

108 /\* 1.1.4. 本次写动作的数据是要读取eeprom的寄存器首地址\*/

109 messages[0].buf = &reg\_addr;

110

111 /\* 1.2. 构建第二条消息 messages[1] \*/

112 /\* 1.2.1. 赋值eeprom的I2C从设备地址 \*/

113 messages[1].addr = dev\_addr;

114

115 /\* 1.1.2. 赋值flags为本次I2C通信完成读动作 \*/

116 messages[1].flags = I2C\_M\_RD;

117

118 /\* 1.1.3. 赋值len为要读取eeprom寄存器数据长度len \*/

119 messages[1].len = len;

120

121 /\* 1.1.4. 本次读动作的数据要存放的buf位置\*/

122 messages[1].buf = data\_buf;

123

124 /\* 2. 构建struct i2c\_rdwr\_ioctl\_data data \*/

125 /\* 2.1. 将准备好的消息包赋值给i2c\_rdwr\_ioctl\_data中的msgs消息\*/

126 data.msgs = messages;

127

128 /\* 2.2. 由于本次I2C通信既有写动作也有读动作，所以消息数为2次 \*/

129 data.nmsgs = 2;

130

131 /\* 3. 调用驱动层的读写组合的I2C数据传输 \*/

132 if(ioctl(fd, I2C\_RDWR, &data) < 0)

133 {

134 printf("I2C\_RDWR err \n");

135 return -1;

136 }

137

138 /\* 4. 等待I2C总线读取完成 \*/

139 sleep(1);

140

141 return 0;

142 }

143

144 int main()

145 {

146 int fd,i,ret=0;

147 unsigned char w\_add=0x10;

148

149 /\* 将要读取的数据buf\*/

150 unsigned char rd\_buf[8] = {0};

151

152 /\* 要写的数据buf\*/

153 unsigned char wr\_buf[8] = {0};

154

155 printf("hello,this is I2C\_RDWR i2c test \n");

156

157 /\* 打开eeprom对应的I2C控制器文件 \*/

158 fd =open(EEPROM\_DEVICE, O\_RDWR);

159 if (fd< 0)

160 {

161 printf("open"EEPROM\_DEVICE"failed \n");

162 }

163

164 /\* 把要写入的数据写入到后面的buf中 \*/

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

166 wr\_buf[i]=i;

167

168 /\* 通过I2C\_RDWR完成向eeprom读数据的功能 \*/

169 eeprom\_write(fd,EEPROM\_ADDR,w\_add,wr\_buf,8);

170

171

172 /\* 通过I2C\_RDWR完成向eeprom写数据的功能 \*/

173 eeprom\_read(fd,EEPROM\_ADDR,w\_add,rd\_buf,8);

174

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

176 {

177 printf("rd\_buf is :%d\n",rd\_buf[i]);

178 }

179

180 /\* 完成操作后，关闭eeprom对应的I2C控制器的设备文件 \*/

181 close(fd);

182

183 return 0;

184 }

185

186

* + 1. **简介I2C的调试方式**

##### 概述I2C通信中完成正常通信的常见元素：

第一，先检查I2C总线上的所有设备是否都经上拉电阻到电源，并检查供电是否稳定。

第二，数据线和时钟信号线是否有接反的情况。

第三，I2C的通信速率是否超过了设备所支持的最高速度。

第四，检查外部I2C设备与操作的I2C控制器是否挂在了同一条I2C总线上。

第五，检查操作的I2C外设地址是否正确。

第六，检查I2C总线上是否有多个相同设备地址的从机设备，导致通信冲突。

第七，操作的I2C外设是否处于写保护状态，写保护状态是无法写入数据的。

第八，检查I2C通信时序是否满足I2C通信协议。

第九，检查在没有开始运行I2C通信程序的时候，I2C总线上的电平信号是否干净稳定的保持高电平，是否出现过主机误把SDA拉低的情况，导致I2C总线出现“忙碌”状态。

第十，检查I2C通信过程中是否出现SDA或者SCL被长时间一直拉低的状态。比如I2C外设从机由于异常在发送完ACK信号后没有释放SDA。另一种情况是cpu在做从机的时候，没有及时完成将读取的主机数据进行处理，导致长时间将SCL拉低，破坏了I2C通信流程，因此我们在写I2C通信的时候最好尽快在I2C接收数据中断服务函数中完成数据处理工作并授权I2C控制器让其正常工作。

由于I2C总线的协议特性，如果总线上有任何一个I2C设备将SCL或者SDA的信号拉低，其他的I2C设备都将看到这个低电平，并且都无法拉高他们。这也就是说，如果有设备不释放总线，一直把总线的电平拉低，那么整个I2C总线将会出现暂停挂死的状态，将无法按照I2C协议进行正常通信。

如果负责I2C总线主机cpu的I2C控制器出现上述长时间拉低I2C总线的电平，理论上我们可以通过调试代码找出I2C总线死机的原因，并修改代码重新初始化该I2C控制器来复位它，让其重新进行I2C通信。如果通过调试发现导致I2C总线死机的原因是由I2C外设导致的，那么我们可以复位该外设芯片。但是在实际的项目开发中，可能复位I2C总线上的元件也无法恢复正常的I2C通信，这个时候就要设计I2C总线的主机程序将I2C控制器引脚设置为GPIO功能并模拟I2C协议完成一次完整的I2C通信，再将I2C控制器设置设置为I2C功能。

* 1. 总结I2C在嵌入式项目开发的应用优缺点

优点：只使用两根线，支持多个主控制器和多个从设备，I2C具有非常广泛使用的协议。

缺点：数据传输速率比SPI慢，数据帧的大小限制为8位，实现比SPI更复杂的硬件。而且I2C通信需要注意下面的使用问题：

##### I2C时钟信号（SCL）的同步问题

在I2C总线上传送信息时的时钟同步信号是由挂接在SCL线上的所有器件的逻辑“与”完成的。SCL线上由高电平到低电平的跳变将影响到这些器件，一旦某个器件的时钟信号下跳为低电平，将使SCL线一直保持低电平，使SCL线上的所有器件开始低电平期。此时，低电平周期短的器件的时钟由低至高的跳变并不能影响SCL线的状态，于是这些器件将进入高电平等待的状态。当所有器件的时钟信号都上跳为高电平时，低电平期结束，SCL线被释放返回高电平，即所有的器件都同时开始它们的高电平期。其后，第一个结束高电平期的器件又将SCL线拉成低电平。这样就在SCL线上产生一个同步时钟。可见，时钟低电平时间由时钟低电平期最长的器件确定，而时钟高电平时间由时钟高电平期最短的器件确定。

##### 总线驱动能力

上拉电阻和负载电容决定了总线在某一速率下的稳定性。当输出为高时，电流通过上拉电阻对负载电容充电。上拉越大，电容越大，所需要的时间就越长，如果超过了通信周期的10%，那么这个上升沿就太缓了，相应的建立时间会受到影响，I2C规范的最大负载电容是400pF，快速模式下是100pF。如果输出为低，电流通过上拉电阻被I2C master器件吸取，（注意根据I2C规范，最小只有3毫安的吸取电流）那么这个吸取电流在上拉电阻上的压降就决定了输出低电平能达到的范围，如果不能达到0.3VDD以下，就会有误采样。有人说加大上拉电阻是不妥当的，要具体分析吸取电流、负载电容、上拉电平和通信速率才能决定（普通模式和快速模式是不一样的）。

虽然速度不是特别快，但是信号线上如果有加电容的话，切记不要加大的，一定要小，否则信号还没到从设备呢，就被电容吃了。